NIST Special Publication
NIST SP 500-341

# SATE VI Report

## *Bug Injection and Collection*

Aurelien Delaitre
Paul E. Black
Damien Cupif
Guillaume Haben
Alex-Kevin Loembe
Vadim Okun
Yann Prono

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.500-341

NIST NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

# NIST Special Publication
# NIST SP 500-341

# SATE VI Report

## *Bug Injection and Collection*

Aurelien Delaitre
Alex-Kevin Loembe
*Prometheus Computing LLC*

Paul E. Black
Vadim Okun
*Software and Systems Division*
*Information Technology Laboratory*

Damien Cupif
*Unaffiliated*

Guillaume Haben
*University of Luxembourg*

Yann Prono
*Mutuelle assurance des instituteurs de France*

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.500-341

June 2023


U.S. Department of Commerce
*Gina M. Raimondo, Secretary*

National Institute of Standards and Technology
*Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology*

NIST SP 500-341
June 2023





**Publication History**
Approved by the NIST Editorial Review Board on 2023-06-07

**How to Cite this NIST Technical Series Publication**
Delaitre A, Black PE, Cupif D, Haben G, Loembe AK, Okun V, Prono Y (2023) SATE VI Report: Bug Injection and Collection. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 500-341. https://doi.org/10.6028/NIST.SP.500-341

**Authors ORCID iDs**
Aurelien Delaitre: 0000-0003-1404-7523
Paul E. Black: 0000-0002-7561-6614
Guillaume Haben: 0000-0002-6290-4959
Vadim Okun: 0000-0003-2391-3681

**Contact Information**
samate@nist.gov

## Abstract

The Static Analysis Tool Exposition (SATE) VI report presents the results of a security-focused bug finding evaluation exercise carried out from 2018 to 2023 on various code bases using static analysis tools. Existing bugs were extracted from bug tracker reports and the National Vulnerability Database (NVD), and additional bugs were injected using automated tools and manual analysis. The results of this exercise showed significant variability across tool effectiveness, depending on the test cases, bug classes, and bug complexity involved. The report discusses the shortcomings and difficulties encountered during the bug injection process, which marginally impeded the efficiency of the evaluation.

The report emphasizes the correlation between high code complexity and tool difficulty in identifying bugs. Recall and discrimination rates were lower for the convoluted C Track than the considerably less complex Java Track. Across all languages and code bases, tools found bugs with lower complexity more readily than bugs with higher complexity. Finding rates varied for different bug classes, in line with the inherent complexity of each bug class (e.g., recall for simpler initialization errors was greater than on more intricate buffer errors).

The report discusses the shortcomings of the bug injection process. Regardless of the test case, injected bugs were not found by tools at the same rate as existing bugs, implying that their quality needs to improve.

The report also includes a summary of the Ockham Sound Analysis Criteria track, which focused on tools that do not report false positives or false negatives.

The SATE VI report concludes that static analysis is a useful technique to find real security bugs in large code bases. The right set of tools, used properly, can help increase code quality and security. Potential users should test a tool or set of tools on their own code base before using them in production. The metrics presented in SATE VI are suitable for assessing tool fitness for such a use case.

## Keywords

Static Analysis; Cybersecurity; Bug Injection; Software Vulnerability; Software Assurance

## Table of Contents

## List of Tables

## List of Figures

## Acknowledgments: Classic Track

We want to thank the SATE VI participants, some of whom have been participating in SATE since 2008. We recognize and appreciate their contributions in the ongoing efforts to improve software assurance. In alphabetical order, the SATE VI participants were: Checkmarx (CxSAST), Clang, Cppcheck, Flawfinder, Gimpel (PC-lint Plus), Grammatech (Code Sonar), Infer, JuliaSoft (Julia), Kiuwan (Code Security), Mathworks (Polyspace Bug Finder), Microfocus (Fortify SCA), Parasoft (C/C++test, Jtest), SpotBugs, Synopsys (Coverity) and Viva64 (PVS-Studio).

We want to especially thank Grammatech for providing us their Bug Injector and working with us to create the SQLite test case for the C Track.

## Acknowledgments: Ockham Criteria Track

We thank AbsInt Angewandte Informatik GmbH for an evaluation license to run Astrée and Dr.-Ing. Jörg Herter, Christoph Mallon, and Dominik Erb for answering our many questions about it. We thank the List Institute, Commissariat à l'énergie atomique et aux énergies alternatives (CEA) for making Frama-C and Eva available and André Maroneze, Florent Kirchner, and David Bühler for answering our many questions about it.

## Caution on Interpreting and Using the SATE Data

SATE VI, as well as its predecessors, taught us many valuable lessons. Most importantly, our analysis should NOT be used as a basis for rating or choosing tools; this was never the goal.

No single metric or set of metrics is considered by the research community to indicate or quantify all aspects of tool performance. We caution readers not to apply unjustified metrics based on the SATE data.

Due to the nature and variety of security weaknesses, defining clear and comprehensive analysis criteria is difficult. While the analysis criteria have been much improved since the first SATE, further refinements are necessary.

The test data and analysis procedure employed have limitations and might not indicate how these tools perform in practice. The results may not generalize to other software because the choice of test cases, as well as the size of test cases, can greatly influence tool performance. Also in the Classic track, we analyzed only the tool warnings that were related to the collected or injected weaknesses.

The procedure that we used for injecting weaknesses in production software has limitations, so the results may not indicate the tools' actual abilities to find important security weaknesses. Specifically, the shortcomings of the injected weaknesses are described in Section 5.2.

Synthetic test cases are much smaller and less complex than production software. Weaknesses may not occur with the same frequency in production software. Additionally, for every synthetic test case with a weakness, there is one test case without a weakness, whereas, in practice, sites with weaknesses appear much less frequently than sites without weaknesses. Due to these limitations, tool results, including false positive rates, on synthetic test cases may differ from results on production software.

The tools were used differently in this exposition from their typical use. We analyzed tool warnings for correctness and looked for related warnings from other tools. Developers, on the other hand, use tools to determine what changes need to be made to the software. Auditors look for evidence of assurance. Also, in practice, users write specific rules, suppress false positives, and write code in certain ways to minimize tool warnings.

We did not consider the tools' user interfaces, integration with the development environment, and many other aspects of the tools, which are important for a user to understand a weakness report efficiently and correctly.

Teams ran their tools against the test sets in 2018 and 2019. The tools continue to progress rapidly, so some observations from the SATE data may already be out of date.

Because of the stated limitations, SATE should not be interpreted as a tool testing exercise. The results should not be used to make conclusions regarding which tools are best for a given application or the general benefit of using static analysis tools.

## 1. Introduction

Concurrently with society's increasing reliance on software, the software complexity and vulnerabilities are growing as well. Hence, software assurance is critically needed.

Software assurance is a set of methods and processes to prevent, mitigate or remove vulnerabilities and ensure that the software functions as intended. Multiple interrelated techniques and tools are used for software assurance [1][2]. One commonly used technique is static analysis, which examines software for weaknesses without executing it [3].

Over the years, the National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tool Evaluation (SAMATE) project has organized six Static Analysis Tool Expositions (SATEs) [4][5][6][7][8], designed to advance research in static analysis tools that find security-relevant weaknesses in source code.

Briefly, NIST provides a set of programs to tool makers, then they run their tools and return tool outputs for analysis. Tool makers and organizers share their experiences and observations at a workshop.

SATE VI included three tracks:

1. Classic track, evaluating tool performance on C and Java test cases produced using bug injection and collection.
2. Ockham Sound Analysis Criteria track, evaluating sound analysis tools.
3. Mobile track, evaluating mobile apps.

The results of the Ockham track are published in [16]. The results of the Mobile track are presented in [31]. We explain the SATE procedure and present the results of SATE VI in this report.

## 1.1. Goals

SATE encourages participation by creating a collaborative, rather than competitive, environment. This broader participation brings more results, on which we build and assess stronger metrics. The SATE metrics provide assessments of tools' features, such as weakness types, the accuracy in detecting such weaknesses, and the rate of missing weaknesses in source code.

Also, SATE provides participating toolmakers with quality feedback, enabling them to assess their tools' strengths and weaknesses. The results produced by their tools are partially reviewed and rated by experts.

Finally, demonstrating the use of tools on production software fosters their adoption by the user community. In fact, several toolmakers informally reported that their current and prospective customers demanded that they participate in SATE.

## 1.2. Scope

SATE focuses on tools capable of finding security flaws. Although its parent project, SAMATE, considers all types of software assurance tools, SATE is only concerned with tools that statically analyze software, i.e., without executing the code.

## 1.3. Target Audience

The target audiences for this report are static analysis toolmakers, security researchers, and tool users.

## 1.4. Terminology

We use the concepts defined in Table 1. Terms specific to the Ockham criteria discussed in Section 6 are defined in Table 2.

**Table 1.** General Glossary of Terms

| Term | Definition |
|---|---|
| Weakness, flaw, bug | Defect in a system that may (or may not) lead to a vulnerability. |
| Vulnerability | A weakness in system security requirements, design, implementation, or operation, that could be accidentally triggered or intentionally exploited and result in a violation of the system's security policy [9]. |
| Site | Conceptual place in a program where an operation is performed. |
| Finding, claim | A definitive statement provided by a tool about a site, e.g., the presence or absence of a weakness. |
| Warning | Claim reporting the presence of a potential weakness. |
| Report | Collection of warnings reported by a tool on a specific test case. |
| Location | A representation of a site, e.g., by file name and line number in source code. |
| Complexity | Code construct encapsulating a site, making the latter more or less difficult to analyze. |
| Synthetic code | Artificial code generated and documented automatically. |
| True positive (TP) | Flawed code correctly reported by a tool. |
| False positive (FP) | Non-flawed code reported by a tool as flawed. |
| False negative (FN) | Flawed code not reported by a tool. |
| Test case | A code base containing weaknesses, or fixes for these weaknesses. |
| Weakness class | A general type of weakness e.g., buffer errors or initialization errors. |
| Ground truth | Knowledge of all weaknesses in a test case, including their location in code and weakness class. |
| Track | An area of focus, such as a programming language (C/C++ and Java), sometimes collectively called "classic tracks", or methodology (Ockham Criteria). |
| Good, fixed, non-buggy code | Code that should not contain any weakness. |
| Bad, flawed, buggy code | Code that contains at least one weakness. |

**Table 2.** Ockham Criteria Glossary of Terms

| Term | Definition |
|---|---|
| Bad function | A function in a Juliet test case that is written to exhibit a weakness. (Sec. 6.1.2.1) |
| Good function | A function in a Juliet test case that is identical to a bad function, except that it does not have the weakness. (Sec. 6.1.2.1) |
| Finding | A definitive statement by a tool about a specific place in code, e.g., the presence or absence of a weakness. (Sec. 6.2) |
| Site | A location in code where a weakness might occur. (Sec. 6.1.2.2) |
| Buggy site | A site that has a bug or weakness. |
| Sound tool | Every finding is correct. (Sec. 6.2.1) |
| Weakness | The property of a piece of code such that execution could lead to a fault. (Sec. 6.1.2.2) |

## 1.5. Metrics

The following metrics address some basic questions about tool performance:

- **Recall** – What proportion of weaknesses can a tool find?

*Recall* is defined by the number of correct findings by a tool compared with the total number of weaknesses present in the code. It is calculated by dividing the number of *True Positives (TP)* by the total number of weaknesses, i.e., the sum of the number of *True Positives (TP)* and the number of *False Negatives (FN)*.

$$Recall = \frac{TP}{TP+FN} \quad (1)$$

- **Precision** – How much can I trust a tool?

*Precision* is the proportion of correct warnings produced by a tool and is calculated by dividing the number of *True Positives (TP)* by the total number of *warnings*. The total number of warnings is the sum of the number of *True Positives (TP)* and the number of *False Positives (FP)*.

$$Precision = \frac{TP}{TP+FP} \quad (2)$$

- **Discrimination Rate** – How smart is a tool?

Buggy and good code often look similar. It is useful to determine whether the tools can differentiate between the two. Although precision captures that aspect of tool efficiency, it is relevant only when good sites dominate buggy sites. When there is parity in the number of good and bad sites, e.g., in some synthetic test suites, a tool could indiscriminately flag both good and bad sites as flawed and still achieve a precision of 50 %. *Discrimination*, however, recognizes a true positive on a specific flawed test case only if a tool did not report a false positive on the corresponding fixed test case [10]. For each weakness instance, a tool is assigned a discrimination of 1 if the tool reports a weakness for a bad site but not for the corresponding good site; otherwise, it is assigned a discrimination of 0. Over a set of test cases, the *Discrimination Rate* is the number of discriminations divided by the total number of weakness instances. A tool that flags all sites (good and bad) indiscriminately would achieve a discrimination rate of 0 %.

- **F1 Score** – Can a tool be measured by a single metric?

The F1 score combines recall and precision in a single metric. In SATE, recall and precision (or alternatively, discrimination) are two independent dimensions of tool behavior. Two tools with different profiles (i.e., different recall and precision) could wind up with the same F1 score. In SATE, we prefer to keep the two metrics as separate axes to better represent tool behavior.

- **Overlap** – Can the findings be confirmed by other tools?

*Overlap* represents the proportion of weaknesses found by more than one tool. This metric identifies which tools behave similarly and which weaknesses are easy or difficult for tools to find. The use of multiple tools would find more weaknesses (higher recall), whereas the use of independent tools would provide a higher confidence that the common warnings are accurate. (Independent tools use different approaches to find bugs, so a bug found by multiple such tools is

more likely to be a real one, in the same way a fire is more likely to be accurately detected by both a smoke detector and a temperature sensor, instead of just one of the two.)

## 1.6. Related Work

Definition and classification of security weaknesses in software are necessary to communicate and analyze security findings. While many classifications have been proposed, Common Weakness Enumeration (CWE) is the most prominent effort [11].

The Bugs Framework (BF) is a structured, complete, orthogonal, and language- and technology-independent classification of software security bugs and weaknesses, which allows precise description of software security vulnerabilities that exploit them [12].

Several studies used synthetic test suites to evaluate tools. Kratkiewicz and Lippmann [13] developed a comprehensive taxonomy of buffer overflows and created 291 test cases, comprised of small C programs, to evaluate tools for detecting buffer overflows. Each test case has three vulnerable versions with buffer overflows just outside, moderately outside, and far outside the buffer, in addition to a fourth, fixed, version. Kratkiewicz's taxonomy [13] lists different attributes, or code complexities, including aliasing, control flow, and loops, which may complicate analysis by the tools.

The largest synthetic test suite in the NIST Software Assurance Reference Dataset (SARD) [14] was created by the U.S. National Security Agency's (NSA) Center for Assured Software (CAS). Juliet 1.0 consists of about 60 000 synthetic test cases, covering 177 CWEs and a wide range of code complexities [10]. CAS ran nine tools on the test suite and found that static analysis tools differed significantly with respect to precision and recall. Also, tools' precision and recall ranking varied for different weaknesses. CAS concluded that sophisticated use of multiple tools would increase the rate of finding weaknesses and decrease the false positive rate.

A newer version of the test suite, Juliet 1.2, correcting several errors and covering a wider range of CWEs and code constructs, was used in SATE V. Juliet 1.3, which increased weakness coverage and corrected many errors in version 1.2 [15], was used in the SATE VI Ockham Sound Analysis Criteria track [16].

Studies also evaluated tools on production software. Rutar et. al. [17] ran five static analysis tools on five open source Java programs, including Apache Tomcat, of varying size and functionality. Due to many tool warnings, Rutar et al. did not categorize every false positive and false negative reported by the tools. Instead, the tool outputs were cross-checked with each other. Additionally, a subset of warnings was examined manually. Previous SATEs also analyzed a subset of tool warnings for production software. One of the conclusions of Rutar et al. was that there was little overlap among warnings from different tools. Another conclusion was that a meta-tool combining and cross-referencing outputs from multiple tools could be used to prioritize warnings.

Several tool evaluation studies identified ground truth in production software. The earliest such effort was by Zitser et al. [18]. At the time of their 2004 publication, sophisticated tools could not handle realistic software, so they extracted source code for model programs. They created fourteen small model programs from three popular, open source, Internet server programs (BIND, Sendmail, and WU-FTP), which contained publicly known, exploitable buffer overflows. The model programs had both vulnerable and patched source code. Complexity of the

model programs related to the buffer overflows was similar to the real programs, while the size was much smaller. Now, many sophisticated tools can handle large software out of the box or with minimal configuration. The study analyzed different characteristics of buffer overflows and evaluated true positive rates, false positive rates, and discrimination counts of static analysis tools.

Li et al. [19] developed VulPecker, an automated vulnerability detection system, based on code similarity analysis. They created a Vulnerability Patch Database, comprised of over 1700 CVEs from nineteen C/C++ open source software. The CVEs are mapped to diff hunks, which are small files tracking the location of a given weakness and changes in source code across versions.

Instead of extracting ground truths, such as CVEs, from software, some studies focused on injecting realistic bugs into software. We review these studies in Section 2.2.

## 1.7. Evolution of SATE

The first SATE [4] used open source, production programs as test cases. We learned that not knowing the locations of weaknesses in the programs complicates the analysis task.

To address this problem, starting in the second SATE [5], we randomly selected a subset of thirty warnings from each tool report, based on weakness category and severity. The selection procedure assigned higher weight to higher severity warnings. We then analyzed the selected warnings for correctness. We also searched for related warnings from other tools, which allowed us to study overlap of warnings between tools.

Over the years, we added other types of test cases. One type, CVE-selected test cases, is based on the Common Vulnerabilities and Exposures (CVE) [20], a database of publicly reported security vulnerabilities. The CVE-selected test cases are pairs of programs: an older vulnerable version with publicly reported vulnerabilities (CVEs) and a fixed version, i.e., a newer version where some or all of the CVEs were fixed. For the CVE-selected test cases, we focused on tool warnings that corresponded to the CVEs.

In SATE IV [7], we introduced a large number of synthetic test cases, the Juliet test suite, which contain precisely characterized weaknesses. Thus, warnings for these weaknesses were amenable to mechanical analysis.

In SATE V [8], we introduced the Ockham Criteria to evaluate sound static analysis tools. Sound tools are designed to never report incorrect findings.

The evolution of SATE is described in detail and summarized in the SATE V report [8], Sec. 1.8.

To address the limitations of the different types of test cases used in SATE V (production, CVE-selected, and synthetic), SATE VI Classic track focused on bug injection and collection.

## 1.8. Mobile Track

Mobile applications are pervasive in the public and private sectors. Enterprises in these sectors should evaluate the mobile applications used within their infrastructures for vulnerabilities to minimize potential risk. The SATE VI Mobile track sought to improve the tools and services used in these evaluations by extending the Static Analysis Tool Exposition (SATE) to include

mobile application tool evaluations. [31] describes NIST's first attempt to carry out that goal and the results that stemmed from the first Mobile SATE track.

## 2. Bug Injection and Collection

The first step in building our test suites was to collect existing bugs in the software we selected for test cases, by browsing bug trackers, bug reports and CVEs. These high-quality bugs were, however, generally too few to achieve statistical significance. We then decided to augment the corpus of existing bugs with injected ones.

### 2.1. Motivation for Bug Injection

The quality of a test suite for static analysis can be articulated around three axes: relevance, ground truth and statistical significance:

- Relevance describes how close to *real* code the test cases are. For example, production-grade software provides the highest relevance, whereas computer-generated test cases do not demonstrate the complexity and development history typical of real software.
- Ground-truth simply means that we sufficiently know the bugs in the test suite to determine if tools found them or not. Bug characteristics of interest are the type of and sequence of operations triggering the bug.
- Statistical significance is gained if the test suite contains enough comparable bugs to draw statistical conclusions.

Test suite types demonstrating two of the three characteristics are readily available:

- Production software offers relevance and statistical significance if the code base is sufficiently large, but no ground truth.
- Known vulnerabilities (as listed in, e.g., CVE/NVD) provide ground truth and relevance but are too few in a single code base to provide statistical significance.
- Synthetic test cases offer ground truth and statistical significance but not the sought-after level of relevance, as their complexity is not comparable to production software's.

One approach to create a better test suite is to inject many synthetic bugs in production software. The numbers will provide statistical significance, and the original software the relevance. The ground truth can be determined during the injection process, as the type and location of the bugs are known. This is the approach we experimented with to create many of the SATE VI Classic Track's test cases.

One benefit of using bug injection is the ability to infer proof of vulnerability (PoV or exploit) to demonstrate that a bug can be triggered and, therefore, matters. PoVs also make tracing bugs easier, which in turn makes matching tool warnings to bugs easier and more accurate.

### 2.2. Related Work on Bug Injection

Several approaches for injecting bugs into production software have been proposed and implemented recently. The Intelligence Advanced Research Projects Activity (IARPA)

attempted to combine all three properties of an ideal test suite in its Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) program [21][22]. IARPA created 7770 test cases by injecting small code snippets, containing weaknesses, into sixteen open-source base programs written in C and Java. Safe and triggering inputs, as well as expected outputs, were also created as part of the test case generation process. Although the base programs were real-world software, the inserted code snippets, or cysts, were unrelated to the control and data flow of the base programs. The resulting weaknesses were not representative of bugs made by real programmers.

EvilCoder [23] extends Joern [24], a tool for robust analysis of C code, to support interprocedural analysis. Joern converts source code into a code property graph - a combinationof the abstract syntax tree, control flow and data flow. Analyzing the code property graph, EvilCoder finds sensitive sinks – security-relevant Application Programing Interface (API) calls such as memcpy. Then, EvilCoder finds data flow connections between the sinks and user-controlled sources, such as files, command line arguments, etc. After that, it traces the control flow from source to sink in order to find the relevant security mechanisms, such as sanitization functions or security checks. Finally, EvilCoder modifies the source code to weaken or remove the security mechanisms or replace secure API calls with insecure ones.

Although EvilCoder produces taint-style bugs, [23] suggests possible extensions to other weakness classes such as race conditions and use-after-free. EvilCoder uses static analysis to insert bugs and does not produce triggering inputs for the bugs, so there is no guarantee that the injected code created real bugs. Additionally, its reliance on static analysis for bug injection may bias static analysis tool evaluation.

As proposed in [25], the injected bugs must:

- Be cheap and plentiful.
- Span the execution lifetime of a program.
- Be embedded in representative control and data flow.
- Come with an input that serves as an existence proof.
- Manifest for a very small fraction of possible inputs.

To address these requirements, large-scale automated vulnerability (LAVA) [25] uses a dynamic taint analysis approach to find locations in code that are relevant for bug injection. Specifically, LAVA identifies two key elements:

- An attack point, that is, a site that can potentially have a bug, and
- User-controlled data that do not determine control flow are available before the attack point on the program trace and could be used at that attack point to trigger the bug.

Then LAVA modifies the program to make the user-controlled data available at the attack point and use them to trigger the vulnerability. Thus, LAVA provides both the triggering input and the bug locations. LAVA can inject thousands of bugs in minutes. However, the tool alters the program data flow in a somewhat unrealistic way. Also, LAVA initially covered only buffer overflows and was extended later [26] to cover a few other types of bugs. With its focus on producing bugs that manifest for a very small fraction of possible inputs, LAVA is well-suited for the evaluation of fuzz testing tools and security competitions such as Capture the Flag.

Apocalypse [27] uses formal techniques - symbolic execution, constraint-based program synthesis and model counting - to automatically inject bugs in large software. Compared to LAVA, Apocalypse improves fairness and depth of bugs. According to [27], a fair bug can be found by practical bug detection techniques, while a deep bug requires a long sequence of data and control flow conditions to be met for it to trigger.

Mutation testing [28] systematically produces mutants, that is, programs with small syntactic changes that correspond to typical programmer errors. Some example changes are replacing a variable with another variable, replacing a relational operator with another relational operator, and deleting a statement. A test set is adequate if it can distinguish the program from each mutant. Mutation testing can be used for both test generation and analysis. Mutation testing does not guarantee that the mutants are real bugs.

GrammaTech developed a bug injector [29] based on the Software Evolution Library[1], independently from the company's static analyzer, Code Sonar. The tool uses instrumentation to discover suitable sites for bug injection, and produces exploits for the bugs it injects. GrammaTech described the tool's process on their blog[2]:

"Bug-Injector takes three inputs: (1) a host program in source format, (2) a set of tests for this program, and (3) a set of bug templates. It attempts to inject bugs from the set of bug templates into the host program and returns multiple different buggy versions of the host program. Each returned buggy program variant contains at least one known bug (the one that was injected), and is associated with a witness—a test input that is known to exercise the injected bug."

## 2.3. Bug Injection and Collection in SATE VI

Bugs can be injected with different degrees of automation. Therein lie challenges, as bug quality tends to degrade with an increase in automation. In SATE VI, we used different degrees of automation with successes and pitfalls.

### 2.3.1. Injected Bug Quality

The first question we asked was what sort of bugs to inject. LAVA publications [25][26] started a reflection on what constitutes a proper injected bug. For our purpose, we expected our bugs to:

- Be embedded in existing control and data flows. Having bugs inserted outside of the program's data flow – like in STONESOUP [21][22] – would not reflect the complexity of real-world bugs.
- Span the execution lifetime of the program. Injecting a bug should not significantly change the behavior of the program. A bug that obviously and consistently breaks the program would have been noticed and remediated by the developer. Arguably, this is not of concern for static analysis, which does not rely on live execution of the target program, but it still introduces bug shadowing issues, which are discussed in Section 2.3.2.

1 https://grammatech.github.io/sel
2 https://blogs.grammatech.com/grammatech-wins-ieee-scam-2019-distinguished-paper-award-for-bug-injector-research

- Trigger on a narrow sequence of inputs. A bug that triggers on a wide variety of inputs would be more likely to be accidentally triggered, noticed, and fixed by the developer and, therefore, would not be realistic.
- Come with a Proof of Vulnerability (PoV) to demonstrate that the bug is real.
- Reflect a human programmer's coding style.

### 2.3.2. Bug Shadowing

When a program contains more than one bug, it is not uncommon that one bug entirely prevents the execution of another. If the second bug in the control flow graph is triggered by a subset of the inputs that trigger the first bug, and all paths leading to the second bug also pass through the first bug, then the first bug will always trigger before the second can be reached, the former shadowing the latter.

This proves problematic for static analyzers, which might be unable to further analysis after encountering the first bug if the state of the program becomes undefined.

To prevent this issue, we focused on injecting bugs in functions deep in the call graph. We listed candidate functions using tools such as Callgrind[3][4] and Flow[5] using this process:

1. Dump the call graph for all available inputs
2. Create a dictionary of all called functions and their inputs
3. Select a function that was rarely called to inject a bug in
4. Delete from the dictionary all functions that used the same inputs as the function in which we injected the bug

For each newly injected bug, we ran the program against the PoVs of already injected bugs. If a PoV triggered the wrong bug, we manually determined if the issue could be easily remediated, otherwise removed the last injected bug, and moved on to the next.

### 2.3.3. Bug Traces

To better match tool warnings to bugs, we spent a considerable amount of time describing the bugs in our test cases. We recorded partial traces leading to each bug, recording and describing each step where tools typically report warnings. Key steps, such as sinks, intermediate bugs, or declaration and initialization of key variables, were given CWEs. These locations were expected to be reported by the tools for us to consider the bug as found.

Traces were collected from PoVs when possible, using call graph tools such as Callgrind and Flow, or manually in last resort. Analysts then curated and annotated the traces manually.

These detailed bug traces could be used for further research and are one of the greatest takeaways of SATE VI.

[3] Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.
4 https://valgrind.org/docs/manual/cl-manual.html
5 http://findtheflow.io

### 2.3.4. Bug Collection and Semi-automated Injection

In SATE VI, we chose to inject specific types of bugs in production software, in addition to collecting existing vulnerabilities. We settled for high-impact, easily testable bug types: buffer errors and pointer issues for the C Track; and Structured Query Language (SQL) injection and Cross-site scripting (XSS) for the Java Track.

Based on the type of bugs we selected, we sought production software that would offer enough suitable sites for injection. For example, we needed software that extensively used a SQL database to allow for SQL injection bugs in sufficient numbers.

#### 2.3.4.1. Bug Collection and Semi-automated Injection in C

For C, we searched for software making pervasive use of pointers and buffers and decided to use one of SATE's classics: Wireshark 1.2.0, a network protocol analyzer. Its large code base, complexity and attack surface made it an interesting candidate for static analysis testing.

We started by searching for all reported bugs in Wireshark's bug tracker and found 49 real-world vulnerabilities. The bug reports oftentimes provided exploits as well, facilitating the retrieval of the bugs traces, sources, and sinks. Most of these bugs are buffer errors, invalid pointer dereferences and a few initialization and calculation issues.

We complemented this list of bugs with our own. To automatically find suitable sites for bug injection in C code bases, we developed a tool called SATESE (for SATE Site Extractor), based on Clang/LLVM and LibTooling/LibASTMatchers. The tool searches for user-defined patterns in the target code base's AST and reports any finding for manual inspection. For SATE VI, we used the following custom patterns:

- Decreasing loops with buffer access for potential buffer underflow
- Array-writing sites
- Buffer-writing functions (see below)
- Pointer casting sites
- Critical functions (see below)
- Loops freeing memory for potential double free bugs
- Any operation increasing integers for integer overflow bugs
- Return statements returning literals
- While loops with null pointer termination condition

The buffer writing functions the tool looked for were: "strcpy", "strncpy", "strncat", "strcat", "fgets", "wcsncpy", "wcscpy", "wcsncat", "wcscat", "wcpcpy", "wcpncpy", "memset", "wmemset", "memcpy", "memmove", "snprintf", "SNPRINTF".

Wireshark 1.2 uses library GLib instead of the C standard library, so we used the critical function matcher to flag functions: "g_free", "g_strdup", "g_malloc", "g_strdup_printf", "g_strlcat", "g_strlcpy".

We used the sites found by the tool to inject bugs falling in three main categories:

- 8 uninitialized pointer dereferences
- 9 incorrect pointer offsets leading to out-of-bound memory access
- 9 other bugs inspired by real bugs found in Wireshark

In total, the test case had 75 bugs, split into 49 existing ones and 26 injected ones.

To prove that these bugs matter, we also collected and created exploits for most of them. The Wireshark bug tracker usually provides triggering inputs for most reported issues, so many of the collected bugs already had a proof of vulnerability (PoV). For the rest, we used guided fuzzing[6] (using AFL/ASAN/Valgrind and adding instrumentation to the code base) with some success. Almost all our bugs have a PoV, many of which are system-dependent.

#### 2.3.4.2. Bug Collection and Semi-automated Bug Injection in Java

For Java, we started with the web repository system DSpace 6.2, which already suffered from XSS vulnerabilities and contained many sites that could accommodate more XSS. DSpace used an Object-Relational Mapping (ORM) paradigm to access its database, making it impractical for SQL injection, so we added Sakai 11.2, a customizable learning management system, accessing its SQL database through generic functions.

We used tools such as Flow to navigate control and data flows in the two programs and help find suitable locations for bug injection in their code bases. Selecting locations based on the control and data flows also helped with crafting PoV, as the input vector was known in advance.

In DSpace, we collected 18 existing XSS bugs (1 reflected, 17 stored), and complemented the set by injecting 12 more bugs (5 reflected, 7 stored). These 30 vulnerabilities span a large portion of the code base.

We were not aware of any vulnerability in Sakai, so we seeded 30 SQL injection bugs by transforming existing prepared statements into unfiltered SQL queries, or by grafting new unfiltered SQL queries at select locations in the code base and using available tainted data flow.

We created the PoVs manually, based on the input vectors and data flows previously collected, and used to inject the bugs. Most PoVs are as simple as entering JavaScript code or malformed SQL queries for XSS and SQL injection, respectively.

### 2.3.5. Mostly-automated Bug Injection

In SATE VI, we used an automated bug injection tool, designed by GrammaTech[7], to inject buffer errors in SQLite 3.21, a relational database management system written in C. The bug injector is described in Section 2.2.

[6] Fuzzing was "guided" by carefully selecting the initial input data to reach the vicinity of the target bug, then by adding "abort()" calls along the control flow graph leading to the bug to drive the fuzzer toward the bug. When one such call was hit, the fuzzer recorded it as a crash and saved the input. We then removed the call to "abort()" that was hit and started fuzzing again with the new input data, until the bug was reached at last. For such application, the fuzzer was not used to discover unknown bugs, but to generate a proof of vulnerability for an already known bug.
[7] Although we used the GrammaTech bug injector to create one of the test cases, it did not particularly benefit or hinder their static analysis tool, Code Sonar, which is completely independent.

GrammaTech provided us with about 10 000 inputs for SQLite to drive the instrumented analysis of the software. Using the data collected, we ran the injection tool multiple times on the code base and collected as many new buggy versions of SQLite. We then carefully selected a set of buggy versions, avoiding bug shadowing, and consolidated 30 buffer error bugs into a single code base. Bug shadowing is discussed in Section 2.3.2. Briefly, we added the bugs one by one to the code base. After each injection, we ran the PoVs of all bugs present in the code base, to check if the newest bug prevented another from triggering, or if a previously present bug prevented the newest bug from triggering. In such instance, we discarded the shallowest of the two interacting bugs and continued the process.

### 2.3.6. Manual Buggy Application Building

In 2016, Defense Advanced Research Projects Agency (DARPA) launched the Cyber Grand Challenge[8] (CGC), a competition to create automatic defensive systems capable of reasoning about flaws, formulating patches and deploying them on a network in real time. For the purpose, DARPA created an extensive test suite of custom-made, buggy programs that were later ported to i386 architecture by Trail of Bits. From the test suite's GitHub[9]:

"The DARPA Challenge Binaries (CBs) are custom-made programs specifically designed to contain vulnerabilities that represent a wide variety of crashing software flaws. They are more than simple test cases, they approximate real software with enough complexity to stress both manual and automated vulnerability discovery. The CBs come with extensive functionality tests, triggers for introduced bugs, patches, and performance monitoring tools, enabling benchmarking of patching tools and bug mitigation strategies."

"Porting work was completed by Kareem El-Faramawi and Loren Maggiore, with help from Artem Dinaburg, Peter Goodman, Ryan Stortz, and Jay Little. Challenges were originally created by NARF Industries, Kaprica Security, Chris Eagle, Lunge Technology, Cromulence, West Point Military Academy, Thought Networks, and Air Force Research Labs while under contract for the DARPA Cyber Grand Challenge."

This test suite is a treasure trove of relatively complex applications containing known bugs, which comes close to the definition of a perfect test suite (see Section 2.1) with a few caveats discussed in Section 5.2.5.

[8] https://www.darpa.mil/program/cyber-grand-challenge
[9] https://github.com/trailofbits/cb-multios

## 2.4. SATE VI Test Suite Summary

The full test suite for the SATE VI Classic Track is summarized in Table 3.

**Table 3.** SATE VI Classic Track Test Cases.

| | Test Case | Ver. | Real Bugs | Inj. Bugs | Inj. type | Bug type | App type |
|---|---|---|---|---|---|---|---|
| **C** | Wireshark | 1.2.0 | 49 | 26 | Semi-auto | Pointer, Buffer, Calc. | Net. traffic analyzer |
| | SQLite | 3.21 | 0 | 30 | Mostly-auto | Buffer | Database engine |
| | CGC | n/a | 0 | 370+ | Manual | Pointer, Buffer, Calc. | Multiple |
| **Java** | DSpace | 6.2 | 9 | 4 | Semi-auto | XSS | Repository system |
| | Sakai | 11.2 | 0 | 30 | Semi-auto | SQL | Learning mngmt. |

# 3. Overall Procedure

SATE follows the Text REtrieval Conference (TREC) model [30] and is divided into tracks. Toolmakers are free to participate in any track and to analyze any test case. SATE VI included three tracks:

- Classic track evaluating tool performance on C and Java test cases produced using bug injection and collection,
- Ockham Sound Analysis Criteria track evaluating sound analysis tools, and
- Mobile track, evaluating mobile apps.

The results of the Ockham track are published in [16]. The procedure and results for the mobile track are presented in [31].

## 3.1. Changes Since SATE V

SATE VI brings several significant changes since SATE V. The most substantive change from SATE V was the focus on bug injection and collection, which is described in Section 2.

To facilitate the participants' tasks, the test cases for SATE VI were shipped as Docker containers. These are lightweight containers that include all dependencies necessary to compile the test cases. They were preconfigured with proper compilation options.

To simplify analysis, teams converted their output to a common Extensible Markup Language (XML) output format. As an alternative to the simple SATE output format [32], in SATE VI we also accepted the powerful Static Analysis Results Interchange Format (SARIF) [44].

## 3.2. Steps / Organization

SATE uses the following steps:

1. Preparation: NIST researchers prepare the test data.
2. Meanwhile, toolmakers are invited to sign up.
3. Kickoff: Test cases are released, and each team starts its analysis.
4. Submission: Each team sends its tool's outputs back to us.
5. Analysis: We analyze tool outputs, using methods specific to each test case type.
6. Workshop: Teams, NIST researchers, and others from industry and academia gather to share their experiences.
7. Publication: We release the SATE report, summarizing SATE VI results.

The Classic track test suites were produced using bug injection and collection; they are summarized in Section 2.4.

The Ockham Sound Analysis Criteria track participants used the Juliet 1.3 C test suite [15].

## 3.3. Participation

The Classic track had the following 15 participants[10]:

- Checkmarx CxSAST
- Clang
- Cppcheck
- Flawfinder
- Gimpel PC-lint Plus
- Grammatech Code Sonar
- Infer
- JuliaSoft Julia
- Kiuwan Code Security
- Mathworks Polyspace Bug Finder
- Microfocus Fortify SCA
- Parasoft C/C++test and Jtest
- SpotBugs
- Synopsys Coverity
- Viva64 PVS-Studio

We ran three of the open source tools (Clang, Infer, and SpotBugs) and converted their output to the SATE output format ourselves.
Two tool makers, Astrée and Frama-C, participated in the Ockham Sound Analysis Criteria track.

Participation in SATE VI was the highest of all SATE events with 17 participants (Table 4). In the Classic track, some teams ran their tool on both C and Java test cases, while others chose one programming language only.

[10] Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

**Table 4.** Overall Participation per Track / Language over SATEs

| SATE | C/C++ | Java | PHP | Ockham | Unique Participants |
|---|---|---|---|---|---|
| **2008** | 4 | 7 | | | 9 |
| **2009** | 5 | 5 | | | 8 |
| **2010** | 8 | 4 | | | 10 |
| **IV** | 7 | 3 | 0 | | 8 |
| **V** | 11 | 6 | 1 | 1 | 14 |
| **VI** | 11 | 7 | | 2 | 17 |

## 3.4. Data Anonymization

SATE is not a competition. To prevent endorsement and protect the intellectual property of toolmakers, randomly assigned aliases will be used to identify their products from this point on. Tools will be referred to as Tools A through R consistently throughout the report.

# 4. Results Summary

This section compiles and draws observations from the results detailed in Section 5.

Injected bugs in SQLite and Sakai suffered from shortcomings, summarized in Section 5.2.6, which may have affected the results for these two test cases.

On the C track (Tables 5, 60 and 70), Tool C performed best on Wireshark and SQLite, but comparatively not as capably on CGC. Tool H performed well on all C test cases. Although less so, Tools B and E performed adequately on all C test cases. Tools D, F, G and K performed variably, depending on the test case. Tools A, I and J did not report any true positive (TP). Excluding tools that did not report any TP, discrimination rate ranged from 8 % to 20 % on Wireshark and from 10 % to 23 % with an outlier at 60 % on SQLite. Recall ranged from 1 % to 13 % on CGC.

On the Java track (Tables 91 and 101), Tool O performed best on DSpace and Sakai. Tool R was on par with Tool O on Sakai but did not support DSpace's XSS bugs. Tools Q and N showed good discrimination rate on both Java test cases, though trailing Tool O. Tools M and P correctly reported several TPs but suffered from poor discrimination on DSpace. Tool L did not report any TP. Excluding tools that did not report any TP, discrimination rate ranged from 0 % to 87 % on DSpace and from 40 % to 67 % on Sakai.

Finding rates varied for different bug types. Buffer errors in Wireshark (Table 15) and CGC (Table 73) had a recall ranging from 2 % to 14 %. Pointer errors in Wireshark (Table 25) had a recall between 21 % and 33 % with an outlier at 3 %, while in CGC (Table 85) tools found almost no pointer errors. Initialization errors were found at a rate of 9 % to 55 % in Wireshark (Table 45) and 14 % to 29 % in CGC (Table 88). Buffer errors tend to be inherently more complex than, e.g., initialization errors, which could factor in these discrepancies.

Across all languages, test cases and bug types, when bug complexity increased, tools found fewer bugs (fewer TPs) and discrimination rate decreased (more FPs per TP). Tools were better

at finding simpler bugs and were adversely affected by increases in bug complexity (Tables 11, 65, 72, and 97).

Ultimately, tool recall and discrimination vary wildly across tools and test cases. Testing tools on relevant code bases is, therefore, necessary to make informed decisions when integrating static analysis in a development pipeline. The metrics used in SATE help measure tool effectiveness and should be part of the process of selecting a static analysis tool suite. Users will seek high recall to maximize the number of bugs reported by the tool, and high discrimination rate to minimize tool confusion between good and bad code.

Additionally, users can combine multiple tools advantageously. To maximize the number of reported bugs, tools with high recall and low overlap can be used together. The union of their reports could add up to more TPs and support of more bug types. Likewise, using independent tools with high overlap could increase the discrimination rate, by focusing on the intersection of the tool reports. Bugs that are reported by both tools have a higher probability of being TPs.

Regarding the quality of injected bugs, shortcomings have been listed in Section 5.2.6. Tables 7 to 10 and 93 to 96 indicate that tools tended to find injected bugs more frequently and with higher discrimination than existing bugs. This result can be partially explained by the low average complexity of injected bugs compared to existing bugs, as detailed in Tables 24 and 100. Considering these observations, injected bugs in SATE VI were discernible from existing bugs, evidence that the injection process fell somewhat short of producing realistic bugs.

# 5. Results

## 5.1. Procedure

The procedure to analyze tool warnings remained essentially the same as in SATE V for the CVE-based test cases ([8], Sec. 3.2.2).

### 5.1.1. Rating of Tool Warnings

All injected bug traces (Sec. 2.3.3) and tool reports were converted to the SARIF format and imported in the SATE database. On the SATE web application, our analysts followed each bug trace to determine if any of its key step was reported by tools. If so, the warnings were analyzed to assess if they appropriately described the weakness. Depending on the assessment, the analyst would rate the warning as a miss, a hint, or a partial, alternate or exact match:

- Miss: a tool did not report any relevant warning for a specific bug trace.
- Hint: a tool reported a warning that could indirectly clue a reviewer to find the bug.
- Partial: a tool reported a related warning on one of the key steps of the trace.
- Alternate: a tool reported the same bug but on a different branch.
- Match: a tool reported the full trace leading to the bug.

In SATE VI, we considered as true positives (TPs) warnings rated as partial, alternate or match for the buggy version of the test cases. For the fixed version of the test cases, warnings rated as partial, alternate or match were considered false positives (FP).

### 5.1.2. Metrics

SATE VI used a set of metrics to measure various aspects of tool effectiveness and test case properties.

#### 5.1.2.1. Tool Effectiveness

SATE VI predominantly used two metrics to measure the effectiveness of the tools: recall and discrimination rate. Recall is the number of TPs divided by the number of bugs. Discrimination rate is the number of discriminated TPs divided by the number of bugs. A discriminated TP is granted if a tool reported a TP in the buggy test case, for which the tool did not report a FP for the same (fixed) bug in the fixed version of the test case. This shows that the tool did indeed understand the bug and did not report blanket warnings for code smells.

Discrimination rate is a more discerning metric than recall, therefore we used it to sort the result tables. Secondary sorting considered the quality of the TPs, so exact matches were prioritized over alternate ones, over partial ones.

To determine if tools handled some type of bugs better than others, we also sliced the results per common bug type, e.g., buffer / pointer / calculation errors, and calculated the recall and discrimination rate on these subsets.

Similarly, we calculated recall and discrimination rate for each tool depending on bug complexity. Each bug was rated as simple, medium, high or extreme complexity depending on guidelines discussed in [8], Sec. 3.2.3.3.

#### 5.1.2.2. Tool Correlation

Tool result overlap was used in SATE VI to determine if tools were independent or positively correlated. Independent (uncorrelated or negatively correlated) tools report different bugs, thus can be used in concert to increase recall. Positively correlated tools, provided they use independent underlying engines, should report a sizeable number of common bugs, which increases the confidence that these warnings are true positives.

In SATE VI, overlap was measured for each bug individually, i.e., we counted the number of bugs found by both Tools X and Y. To calculate the overlap of Tool X over Tool Y, we divided the result by the number of TPs reported by Tool Y. This number is the proportion of Tool Y's TPs found by Tool X.

#### 5.1.2.3. Injected Bug Quality

One way to check if injected bugs measured up to existing bugs is to compare tools effectiveness on injected vs. existing bugs. Throughout the results section, recall and discrimination rates are presented separately for each of the two bug categories separately, enabling direct comparison.

## 5.2. Shortcomings

During the analysis, issues with some injected bugs came to light. This section describes these shortcomings and possible remediations.

### 5.2.1. Cheap but Unhelpful Bugs

A common pattern used to introduce pointer errors in Wireshark is loosely based on existing bug 8C32D803[11]. It consisted of adding a large, context-relevant offset to a valid pointer, throwing the pointer out of range. For example, to create bug 80FA3989, the following valid snippet of code:

```
memcmp(tvb_get_ptr(tvb, offset, 4), "YPNS", 4)
```

was modified by adding `tvb_length(tvb)*10` to the original pointer returned by function `tvb_get_ptr`, as follows:

```
memcmp(tvb_get_ptr(tvb, offset, 4) + tvb_length(tvb)*10, "YPNS", 4)
```

While this is indeed a valid bug, it is nearly impossible to statically differentiate a valid pointer address from an invalid one, especially in the context of Wireshark's many complex memory allocation schemes.

Out of 33 pointer errors, 9 bugs (27 %) were based on this pattern: 8EBE37FF, 80FA3989, 4E251C0D, 256C7C53, 61CF9E42, 299E59EB, C75CCA7F, 3723B848, D5F4E690. These bugs were all rated as having extreme complexity, so the results obtained on low-, medium- and high-complexity bugs remained unaffected.

This type of bugs serves little purpose in a static analysis tool evaluation and should not be used in the future.

### 5.2.2. Asymmetrical Bug/Fix Pairs

Another pattern used in all SQL injection errors consists of turning prepared statements such as:

```
query = "INSERT INTO FORUM_SATE (ID, TITLE, BODY) VALUES(?, ?, ?)";
statement = connection.prepareStatement(query);
statement.setString(1, uuid);
statement.setString(2, title);
statement.setString(3, "body");
statement.executeUpdate();
```

Into unfiltered SQL statements such as:

```
query = "INSERT INTO FORUM_SATE (ID, TITLE, BODY) \
                    VALUES('" + uuid + "', '" + title + "', 'body')";
statement = connection.createStatement();
int i = statement.executeUpdate(query);
```

The original, non-buggy snippet used SQL injection-proof method `prepareStatement` while the buggy snippet used unsafe method `createStatement` with unfiltered inputs. To better test discrimination, the fixed code should have also used method `createStatement`, but with

[11] Bug IDs are used to flag bug traces directly in the Wireshark source code available at: https://samate.nist.gov/SATE6/wireshark-1.2-sate6.tar.xz

comprehensively filtered inputs. This would have assessed whether the tools understood input validation in the context of SQL injection.

### 5.2.3. Automatically Injected Bugs Issues

The GrammaTech Bug Injector brought significant improvements over existing bug injection tools [29]. One such improvement consists of using existing data flows within the target program as sources for the bugs. In SQLite, 10 out of 30 bugs we injected used a global variable named `Sqlite3PendingByte`. This variable is set once to `0x40000000`, and an attacker could never gain control over it. In conjunction with the bug patterns, we used with the Bug Injector in SATE VI, this shortcoming caused several unforeseen effects:

1. 26 bugs depended on variable `Sqlite3PendingByte`, which could not be user-controlled, making these bugs somewhat less complex and realistic.
2. 10 of these bugs, expected to be integer overflow to buffer overflow chains, turned out to be zero-sized memory allocation bugs. The integer operation would overflow exactly to zero and, depending on the underlying implementation, the memory allocation function would return a zero-sized buffer or a null pointer. In the first case, the buffer overflow could still occur, but in the second, the bug was a null pointer dereference. In both cases, the sinks were not immediately discernable.
3. Four bugs depending on variable `Sqlite3PendingByte` used always-true conditions, as the value of the variable cannot change.
4. Similarly, 14 fixes depending on variable `Sqlite3PendingByte` used always-false conditions, turning the sink into dead code, which could be ignored by some tools and, therefore, no different from removing the offending code entirely.

In the future, improved quality control should enforce the diversity and user control of source variables. The sinks should also be identified.

### 5.2.4. Sink Separation

Four bugs in SQLite used the same sink in function `set_i`. Some tools correctly reported one or more bugs inside the function, but without specifying the full trace. It was, therefore, impossible to attribute the true-positive to a specific bug.

In such future occurrence, the sink function should be duplicated for each bug. With separate sinks, it will be possible to differentiate which bug was found even if only the sink line is reported by a tool.

### 5.2.5. CGC Specificities

The CGC test suite was created with vulnerability exploitability in mind. Its programs use customized memory allocation schemes based on the “mmap” Portable Operating System Interface **(**POSIX) system call, and customized buffer and string operation functions. These functions tend to follow the C standard library conventions, but use different names and can behave differently, which can hinder tool analysis. Tools can usually be configured to handle

custom functions, but SATE VI participants may not have had enough time and resources to analyze these functions and set up their tool accordingly. The source code of the functions was, however, included in the test suite, so fine-tuning tools might not have been strictly necessary.

CGC was also designed to run on an Intel 32-bit architecture, which has become unusual nowadays, with 64-bit architectures being prevalent. Most bugs should nonetheless trigger similarly on both architectures.

### 5.2.6. Shortcomings Summary

Although problems altered some bugs in SATE VI, the issues remained relatively contained. Affected bugs tended to be less useful and realistic than initially planned, but not fundamentally wrong.

We advise the reader to consider the affected results with a critical eye. These issues largely afflicted SQLite and the results obtained on this test case are unlikely to represent the effectiveness of tools on real software. In Sakai, discrimination was very high and might be the product of the asymmetry described in Section 5.2.2. In Wireshark, overall recall and discrimination rate in Table 5 would be 13.6 % higher if we disregarded the bugs described in Section 5.2.1, and 39 % higher for pointer-specific Table 25.

## 5.3. C

### 5.3.1. Wireshark: Existing and Semi-automatically Injected Bugs

Wireshark contained 75 bugs with different properties. In the following sections, we first present the general results on all bugs in Section 5.3.1.1, then slice the results along the various bug properties to offer more specific insights on tool strengths and weaknesses and on bug quality. Each bug can be a chain of different weaknesses and fall in more than one category in Sections 5.3.1.2 through 5.3.1.5.

#### 5.3.1.1. Overall Analysis

This section presents the tool results on all 75 bugs present in Wireshark. Table 5 offers an overview of the number of bugs found by each tool, the accuracy of the findings, and the main two metrics used in SATE VI: recall and discrimination rate. Excluding tools without valid findings, the average recall reached 16 % and discrimination rate 14 %. The very small difference between these two numbers supports the conjecture that the tools correctly grasped the workings of the bugs they reported.

Table 6 offers a view of tool warning overlap. For example, Tool C had the highest recall (see Table 5) and found 67 % of the bugs correctly reported by Tools H, E and F. Reciprocally, Tools H, E and F found 47 % of the bugs reported by Tool C. Some tools, such as H and E (83 %), had even higher overlap. Using these two tools together would not significantly increase the number of bugs found, but would increase confidence that bugs reported by both tools are TPs (assuming the tools are independent). Using Tool G along with Tool H, E or F would provide little benefit, as bugs reported by Tool G were always reported by the latter tools. Overall, tool warning

overlap is significant. Barring Tool K, which had low recall, tools overlapped usually by over 40 %. Half of the tools overlapped by over 60 %.

**Table 5.** Overall Recall and Discrimination in Wireshark

| Findings | Tool C | Tool E | Tool H | Tool F | Tool B | Tool G | Tool D | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 55 | 62 | 62 | 61 | 56 | 66 | 50 | 66 | 75 | 75 | 75 |
| Hint | 3 | 1 | 1 | 2 | 6 | 1 | 12 | 1 | 0 | 0 | 0 |
| Partial | 5 | 3 | 3 | 11 | 11 | 6 | 5 | 3 | 0 | 0 | 0 |
| Alternate | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 11 | 9 | 9 | 1 | 2 | 2 | 8 | 5 | 0 | 0 | 0 |
| TP | 17 | 12 | 12 | 12 | 13 | 8 | 13 | 8 | 0 | 0 | 0 |
| Disc. TP | 15 | 12 | 12 | 12 | 10 | 8 | 7 | 6 | 0 | 0 | 0 |
| Bugs | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 | 75 |
| Recall | 23 % | 16 % | 16 % | 16 % | 17 % | 11 % | 17 % | 11 % | 0 % | 0 % | 0 % |
| Disc. Rate | 20 % | 16 % | 16 % | 16 % | 13 % | 11 % | 9 % | 8 % | 0 % | 0 % | 0 % |

**Table 6.** Overall Tool Warning Overlap in Wireshark

| | Tool C | Tool H | Tool E | Tool F | Tool B | Tool G | Tool D | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool C | N/A | 67 % | 67 % | 67 % | 46 % | 88 % | 62 % | 50 % | N/A | N/A | N/A |
| Tool H | 47 % | N/A | 83 % | 75 % | 62 % | 100 % | 54 % | 38 % | N/A | N/A | N/A |
| Tool E | 47 % | 83 % | N/A | 75 % | 46 % | 100 % | 54 % | 50 % | N/A | N/A | N/A |
| Tool F | 47 % | 75 % | 75 % | N/A | 38 % | 100 % | 54 % | 50 % | N/A | N/A | N/A |
| Tool B | 35 % | 67 % | 50 % | 42 % | N/A | 50 % | 46 % | 25 % | N/A | N/A | N/A |
| Tool G | 41 % | 67 % | 67 % | 67 % | 31 % | N/A | 46 % | 13 % | N/A | N/A | N/A |
| Tool D | 47 % | 58 % | 58 % | 58 % | 46 % | 75 % | N/A | 25 % | N/A | N/A | N/A |
| Tool K | 24 % | 25 % | 33 % | 33 % | 15 % | 13 % | 15 % | N/A | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |

Tables 7, 8, 9 and 10 summarize recall and discrimination rate for all existing vs. injected bugs in Wireshark.

Table 7 shows that injected bugs were found significantly more frequently on average (29 % excluding nil findings) than existing bugs (9 % excluding nil findings), and Table 8 that the discrimination rate followed the same trend. Tools C, B and K found a balanced mix of existing and injected bugs, while tools D, E, H, F and G primarily reported injected bugs. No tool had a higher recall on existing bugs than on injected bugs.

Table 9 shows an average drop (excluding nil findings) of 36 % from recall to discrimination rate on existing bugs, while Table 10 shows an average drop of only 7 % (excluding nil findings) on injected bugs.

Overall, injected bugs were found significantly more frequently and with higher discrimination than existing bugs. We can then speculate that the bugs we injected in Wireshark were, on average, less complex than existing bugs from a static analysis perspective.

**Table 7.** Overall Recall for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool C | Tool B | Tool K | Tool D | Tool E | Tool H | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 49 | 18 % | 16 % | 10 % | 10 % | 6 % | 6 % | 2 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 26 | 31 % | 19 % | 12 % | 31 % | 35 % | 35 % | 42 % | 31 % | 0 % | 0 % | 0 % |

**Table 8.** Overall Discrimination Rate for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool C | Tool B | Tool H | Tool E | Tool K | Tool D | Tool F | Tool G | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 49 | 14 % | 12 % | 6 % | 6 % | 6 % | 4 % | 2 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 26 | 31 % | 15 % | 35 % | 35 % | 12 % | 19 % | 42 % | 31 % | 0 % | 0 % | 0 % |

**Table 9.** Overall Recall and Discrimination for Existing Bugs in Wireshark

| | Tool C | Tool B | Tool K | Tool D | Tool E | Tool H | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 18 % | 16 % | 10 % | 10 % | 6 % | 6 % | 2 % | 0 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 14 % | 12 % | 6 % | 4 % | 6 % | 6 % | 2 % | 0 % | 0 % | 0 % | 0 % |

**Table 10.** Overall Recall and Discrimination for Injected Bugs in Wireshark

| | Tool F | Tool K | Tool D | Tool C | Tool G | Tool E | Tool B | Tool H | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 42 % | 35 % | 35 % | 31 % | 31 % | 31 % | 19 % | 12 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 42 % | 35 % | 35 % | 31 % | 31 % | 19 % | 15 % | 12 % | 0 % | 0 % | 0 % |

The SAMATE team classified the 75 bugs in Wireshark according to their complexity from a static analysis standpoint. For example, a bug completely contained within a single function would typically be rated as having low complexity. An interprocedural bug would be rated medium or higher. Undecidable bugs were rated as extreme. The team followed guidelines and homogenized the ratings. However, the classification was not always self-evident, and some ratings could be subjective.

Tables 11 and 12 show an inverse correlation between the proportion of bugs found by tools and bug complexity. Table 13 shows the percentage drop between undiscriminated recall (i.e., regular recall) and discriminated recall (i.e., discrimination rate). As complexity increased, tools were not only less able to find bugs, but also less able to distinguish bugs and their respective fixes.

**Table 11.** Recall per Bug Complexity in Wireshark

| Complexity | Bugs | Tool C | Tool B | Tool H | Tool F | Tool E | Tool D | Tool K | Tool G | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 16 | 56 % | 38 % | 56 % | 69 % | 63 % | 44 % | 25 % | 50 % | 0 % | 0 % | 0 % |
| Medium | 33 | 18 % | 15 % | 6 % | 0 % | 6 % | 12 % | 12 % | 0 % | 0 % | 0 % | 0 % |
| High | 14 | 14 % | 14 % | 7 % | 7 % | 0 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 12 | 0 % | 0 % | 0 % | 0 % | 0 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 12.** Discrimination per Bug Complexity in Wireshark

| Complexity | Bugs | Tool C | Tool B | Tool H | Tool F | Tool E | Tool D | Tool K | Tool G | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 16 | 56 % | 31 % | 56 % | 69 % | 63 % | 31 % | 25 % | 50 % | 0 % | 0 % | 0 % |
| Medium | 33 | 15 % | 12 % | 6 % | 0 % | 6 % | 6 % | 6 % | 0 % | 0 % | 0 % | 0 % |
| High | 14 | 7 % | 7 % | 7 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 12 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 13.** Effect of Bug Complexity on Discrimination in Wireshark

| Complexity | Bugs | Tool C | Tool B | Tool H | Tool F | Tool E | Tool D | Tool K | Tool G | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 16 | 0 % | -17 % | 0 % | 0 % | 0 % | -29 % | 0 % | 0 % | N/A | N/A | N/A |
| Medium | 33 | -17 % | -20 % | 0 % | N/A | 0 % | -50 % | -50 % | N/A | N/A | N/A | N/A |
| High | 14 | -50 % | -50 % | 0 % | 0 % | N/A | -100 % | N/A | N/A | N/A | N/A | N/A |
| Extreme | 12 | N/A | N/A | N/A | N/A | N/A | -100 % | N/A | N/A | N/A | N/A | N/A |

Table 14 breaks down the number of existing and injected bugs across the different levels of bug complexity in Wireshark. Existing bugs' complexity was clustered in the medium-high range while injected bugs were mostly simple or extreme. (See Section 5.2.1 for a description of the extreme injected bugs.)

**Table 14.** Breakdown of Bug Count per Bug Properties in Wireshark

| Complexity | Existing | Injected |
|---|---|---|
| Low | 6 | 10 |
| Medium | 29 | 4 |
| High | 12 | 2 |
| Extreme | 2 | 10 |

### 5.3.1.2. Buffer Errors

This section narrows down the results to the 44 buffer error bugs contained in Wireshark. Table 15, as a focused version of Table 5, presents the number of buffer errors found by each tool, the accuracy of the findings, and the recall and discrimination rate. Excluding tools with nil findings,

the average recall reached 10 % and discrimination rate 7 %. These metrics are significantly lower than the overall recall and discrimination rate (Section 5.3.1.1), showing that tools found these bugs with more difficulty than average. However, recall and discrimination rate remain close, evidence that the tools mostly understood the bugs they reported.

Interestingly, Tool K is the only tool that performed worse on the overall analysis (11 % recall) than on buffer errors (14 % recall), which appears to be one of its strengths.

Table 16 presents the tool warning overlap for buffer errors, which represented a majority of 59 % of the bugs in Wireshark. Consequently, the overlap for buffer errors was similar to the overall overlap reported in Table 6.

**Table 15.** Recall and Discrimination on Buffer Errors in Wireshark

| Findings | Tool H | Tool C | Tool K | Tool E | Tool B | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 38 | 35 | 38 | 39 | 33 | 28 | 41 | 43 | 44 | 44 | 44 |
| Hint | 1 | 3 | 0 | 1 | 5 | 12 | 1 | 0 | 0 | 0 | 0 |
| Partial | 2 | 2 | 3 | 0 | 6 | 3 | 2 | 1 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 3 | 4 | 3 | 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| TP | 5 | 6 | 6 | 4 | 6 | 4 | 2 | 1 | 0 | 0 | 0 |
| Disc. TP | 5 | 4 | 4 | 4 | 3 | 2 | 2 | 1 | 0 | 0 | 0 |
| Bugs | 44 | 44 | 44 | 44 | 44 | 44 | 44 | 44 | 44 | 44 | 44 |
| Recall | 11 % | 14 % | 14 % | 9 % | 14 % | 9 % | 5 % | 2 % | 0 % | 0 % | 0 % |
| Disc. Rate | 11 % | 9 % | 9 % | 9 % | 7 % | 5 % | 5 % | 2 % | 0 % | 0 % | 0 % |

**Table 16.** Tool Warning Overlap on Buffer Errors in Wireshark

| | Tool H | Tool C | Tool K | Tool E | Tool B | Tool D | Tool F | Tool G | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool H | N/A | 33 % | 50 % | 75 % | 67 % | 50 % | 100 % | 100 % | N/A | N/A | N/A |
| Tool C | 40 % | N/A | 50 % | 25 % | 17 % | 25 % | 50 % | 100 % | N/A | N/A | N/A |
| Tool K | 60 % | 50 % | N/A | 100 % | 33 % | 50 % | 100 % | 100 % | N/A | N/A | N/A |
| Tool E | 60 % | 17 % | 67 % | N/A | 33 % | 50 % | 100 % | 100 % | N/A | N/A | N/A |
| Tool B | 80 % | 17 % | 33 % | 50 % | N/A | 25 % | 50 % | 0 % | N/A | N/A | N/A |
| Tool D | 40 % | 17 % | 33 % | 50 % | 17 % | N/A | 100 % | 100 % | N/A | N/A | N/A |
| Tool F | 40 % | 17 % | 33 % | 50 % | 17 % | 50 % | N/A | 100 % | N/A | N/A | N/A |
| Tool G | 20 % | 17 % | 17 % | 25 % | 0 % | 25 % | 50 % | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |

Tables 17, 18, 19 and 20 summarize recall and discrimination rate for all existing vs. injected buffer errors in Wireshark.

Table 17 shows that existing and injected buffer errors were found at similar rates on average, but the proportion of existing vs. injected bugs found depended on the tool. Tools C and B favored existing bugs, while Tools E, D, F and G were more apt with injected bugs. Tools K and H were roughly equally competent on both bug types. Table 18 shows that injected bugs were understood on average at a higher rate than existing bugs, characterized by the sharp drop between recall and discrimination rate on existing bugs for Tools C, B and K (Table 19). Except for Tools B and D, there was no discrimination drop for injected bugs (Table 20).

**Table 17.** Recall on Buffer Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool C | Tool B | Tool K | Tool H | Tool E | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 28 | 18 % | 18 % | 14 % | 11 % | 7 % | 4 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 16 | 6 % | 6 % | 13 % | 13 % | 13 % | 19 % | 13 % | 6 % | 0 % | 0 % | 0 % |

**Table 18.** Discrimination on Buffer Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool H | Tool C | Tool B | Tool K | Tool E | Tool F | Tool D | Tool G | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 28 | 11 % | 11 % | 11 % | 7 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 16 | 13 % | 6 % | 0 % | 13 % | 13 % | 13 % | 13 % | 6 % | 0 % | 0 % | 0 % |

**Table 19.** Recall and Discrimination on Buffer Errors for Existing Bugs in Wireshark

| | Tool C | Tool B | Tool H | Tool K | Tool E | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 18 % | 18 % | 11 % | 14 % | 7 % | 4 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 11 % | 11 % | 11 % | 7 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 20.** Recall and Discrimination on Buffer Errors for Injected Bugs in Wireshark

| | Tool D | Tool K | Tool H | Tool E | Tool F | Tool C | Tool G | Tool B | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 19 % | 13 % | 13 % | 13 % | 13 % | 6 % | 6 % | 6 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 13 % | 13 % | 13 % | 13 % | 13 % | 6 % | 6 % | 0 % | 0 % | 0 % | 0 % |

Tables 21, 22 and 23 present buffer error results sliced by bug complexity. To a higher degree than in Tables 11 and 12, each increase in bug complexity reduced the number of tools that found buffer errors. Recall and discrimination rate were generally higher at lower complexity, but remarkably Tools H, C and B performed better on high-complexity bugs than on medium-complexity bugs. Table 23 highlights the same phenomenon observed in Table 13, i.e., higher complexity correlated with a larger decrease in discrimination rate compared to recall.

**Table 21.** Recall per Bug Complexity on Buffer Errors in Wireshark

| Complexity | Bugs | Tool H | Tool C | Tool B | Tool K | Tool E | Tool F | Tool D | Tool G | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 3 | 67 % | 33 % | 33 % | 67 % | 67 % | 67 % | 67 % | 33 % | 0 % | 0 % | 0 % |
| Medium | 23 | 9 % | 13 % | 13 % | 17 % | 9 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 7 | 14 % | 29 % | 29 % | 0 % | 0 % | 0 % | 14 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 11 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 9 % | 0 % | 0 % | 0 % | 0 % |

**Table 22.** Discrimination per Bug Complexity on Buffer Errors in Wireshark

| Complexity | Bugs | Tool H | Tool C | Tool B | Tool K | Tool E | Tool F | Tool D | Tool G | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 3 | 67 % | 33 % | 0 % | 67 % | 67 % | 67 % | 67 % | 33 % | 0 % | 0 % | 0 % |
| Medium | 23 | 9 % | 9 % | 9 % | 9 % | 9 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 7 | 14 % | 14 % | 14 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 11 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 23.** Effect of Bug Complexity on Discrimination for Buffer Errors in Wireshark

| Complexity | Bugs | Tool H | Tool C | Tool B | Tool K | Tool E | Tool F | Tool D | Tool G | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 3 | 0 % | 0 % | -100 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Medium | 23 | 0 % | -33 % | -33 % | -50 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| High | 7 | 0 % | -50 % | -50 % | N/A | N/A | N/A | -100 % | N/A | N/A | N/A | N/A |
| Extreme | 11 | N/A | N/A | N/A | N/A | N/A | N/A | -100 % | N/A | N/A | N/A | N/A |

Table 24 breaks down the number of existing and injected buffer errors across the different levels of bug complexity in Wireshark. Existing bugs' complexity was clustered in the medium range while injected bugs were mostly extreme.

**Table 24.** Breakdown of Bug Count per Bug Properties on Buffer Errors in Wireshark

| Complexity | Existing | Injected |
|---|---|---|
| Low | 1 | 2 |
| Medium | 20 | 3 |
| High | 6 | 1 |
| Extreme | 1 | 10 |

#### 5.3.1.3. Pointer Errors

This section narrows down the results to the 33 pointer error bugs contained in Wireshark. Table 25, as a focused version of Table 5, presents the number of pointer errors found by each tool, the accuracy of the findings, and the recall and discrimination rate. Excluding tools with nil findings,

the average recall reached 24 % and discrimination rate 22 %. These metrics are significantly higher than the overall recall and discrimination rate (Section 5.3.1.1), showing that tools found these bugs with more facility than average. Furthermore, recall and discrimination rate remain close, evidence that the tools mostly understood the bugs they reported.

In this case, Tool K continued to underperform by reporting a single bug, while the other tools reported close to 9 on average (excluding Tools A, J and I that did not report any bug).

9 of the injected bugs were likely impossible for tools to find, as described in Section 5.2.1. Recall and discrimination rate would be 37.5 % higher if these bugs were not counted.

Table 26 presents the tool warning overlap for pointer errors, which represented 44 % of the bugs in Wireshark. Consequently, the overlap for pointer errors was similar to the overall overlap reported in Table 6, except for Tool K for which low recall produced inconclusive results.

**Table 25.** Recall and Discrimination on Pointer Errors in Wireshark

| Findings | Tool C | Tool E | Tool F | Tool H | Tool B | Tool G | Tool D | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 22 | 24 | 23 | 25 | 22 | 25 | 17 | 31 | 33 | 33 | 33 |
| Hint | 1 | 0 | 1 | 0 | 3 | 1 | 5 | 1 | 0 | 0 | 0 |
| Partial | 3 | 3 | 8 | 1 | 6 | 5 | 4 | 1 | 0 | 0 | 0 |
| Alternate | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 6 | 6 | 1 | 7 | 2 | 2 | 7 | 0 | 0 | 0 | 0 |
| TP | 10 | 9 | 9 | 8 | 8 | 7 | 11 | 1 | 0 | 0 | 0 |
| Disc. TP | 10 | 9 | 9 | 8 | 7 | 7 | 6 | 1 | 0 | 0 | 0 |
| Bugs | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 | 33 |
| Recall | 30 % | 27 % | 27 % | 24 % | 24 % | 21 % | 33 % | 3 % | 0 % | 0 % | 0 % |
| Disc. Rate | 30 % | 27 % | 27 % | 24 % | 21 % | 21 % | 18 % | 3 % | 0 % | 0 % | 0 % |

**Table 26.** Tool Warning Overlap on Pointer Errors in Wireshark

| | Tool C | Tool F | Tool E | Tool H | Tool B | Tool G | Tool D | Tool K | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Tool C** | N/A | 67 % | 78 % | 75 % | 63 % | 86 % | 64 % | 0 % | N/A | N/A | N/A |
| **Tool F** | 60 % | N/A | 89 % | 100 % | 63 % | 100 % | 55 % | 100 % | N/A | N/A | N/A |
| **Tool E** | 70 % | 89 % | N/A | 100 % | 63 % | 100 % | 55 % | 100 % | N/A | N/A | N/A |
| **Tool H** | 60 % | 89 % | 89 % | N/A | 63 % | 100 % | 55 % | 100 % | N/A | N/A | N/A |
| **Tool B** | 50 % | 56 % | 56 % | 63 % | N/A | 57 % | 55 % | 100 % | N/A | N/A | N/A |
| **Tool G** | 60 % | 78 % | 78 % | 88 % | 50 % | N/A | 45 % | 0 % | N/A | N/A | N/A |
| **Tool D** | 70 % | 67 % | 67 % | 75 % | 75 % | 71 % | N/A | 100 % | N/A | N/A | N/A |
| **Tool K** | 0 % | 11 % | 11 % | 13 % | 13 % | 0 % | 9 % | N/A | N/A | N/A | N/A |
| **Tool J** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| **Tool I** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| **Tool A** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |

Tables 27, 28, 29 and 30 summarize recall and discrimination rate for all existing vs. injected pointer errors in Wireshark.

Tables 27 and 28 show that existing and injected pointer errors were found at similar rates and with similar understanding by Tools D, C and B, but Tools E, F, H, G and K almost exclusively found injected bugs. Tables 29 and 30 show little loss in discrimination vs. recall on both injected and existing bugs, with the exception of Tool D, which did not properly understand close to half the bugs it reported.

**Table 27.** Recall on Pointer Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool D | Tool C | Tool B | Tool E | Tool F | Tool H | Tool G | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 13 | 31 % | 31 % | 23 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 20 | 35 % | 30 % | 25 % | 40 % | 45 % | 40 % | 35 % | 5 % | 0 % | 0 % | 0 % |

**Table 28.** Discrimination on Pointer Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool C | Tool B | Tool D | Tool E | Tool F | Tool H | Tool G | Tool K | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 13 | 31 % | 23 % | 15 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 20 | 30 % | 20 % | 20 % | 40 % | 45 % | 40 % | 35 % | 5 % | 0 % | 0 % | 0 % |

**Table 29.** Recall and Discrimination on Pointer Errors for Existing Bugs in Wireshark

| | Tool C | Tool B | Tool D | Tool E | Tool F | Tool H | Tool G | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 31 % | 23 % | 31 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 31 % | 23 % | 15 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 30.** Recall and Discrimination on Pointer Errors for Injected Bugs in Wireshark

| | Tool F | Tool H | Tool E | Tool G | Tool C | Tool D | Tool B | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 45 % | 40 % | 40 % | 35 % | 30 % | 35 % | 25 % | 5 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 45 % | 40 % | 40 % | 35 % | 30 % | 20 % | 20 % | 5 % | 0 % | 0 % | 0 % |

Tables 31, 32 and 33 present pointer error results sliced by bug complexity. Similarly to Tables 11 and 12, each increase in bug complexity reduced the number of tools that found pointer errors. Recall and discrimination rate were generally higher at lower complexity. Table 33 differs significantly from Table 13. Indeed, with the exception of Tool D, the tools understood pointer errors much better than other bug types and showed little drop from recall to discrimination rate.

**Table 31.** Recall per Bug Complexity on Pointer Errors in Wireshark

| Complexity | Bugs | Tool F | Tool C | Tool B | Tool D | Tool E | Tool H | Tool G | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 11 | 73 % | 64 % | 55 % | 55 % | 82 % | 73 % | 64 % | 9 % | 0 % | 0 % | 0 % |
| **Medium** | 10 | 0 % | 30 % | 20 % | 40 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 2 | 50 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Extreme** | 10 | 0 % | 0 % | 0 % | 10 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 32.** Discrimination per Bug Complexity on Pointer Errors in Wireshark

| Complexity | Bugs | Tool F | Tool C | Tool B | Tool D | Tool E | Tool H | Tool G | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 11 | 73 % | 64 % | 45 % | 36 % | 82 % | 73 % | 64 % | 9 % | 0 % | 0 % | 0 % |
| **Medium** | 10 | 0 % | 30 % | 20 % | 20 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 2 | 50 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Extreme** | 10 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 33.** Effect of Bug Complexity on Discrimination for Pointer Errors in Wireshark

| Complexity | Bugs | Tool F | Tool C | Tool B | Tool D | Tool E | Tool H | Tool G | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 11 | 0 % | 0 % | -17 % | -33 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| **Medium** | 10 | N/A | 0 % | 0 % | -50 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **High** | 2 | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Extreme** | 10 | N/A | N/A | N/A | -100 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

Table 34 breaks down the number of existing and injected pointer errors across the different levels of bug complexity in Wireshark. Existing bugs' complexity was clustered in the medium range while injected bugs were mostly of low or extreme complexity.

**Table 34.** Breakdown of Bug Count per Bug Properties on Pointer Errors in Wireshark

| Complexity | Existing | Injected |
|---|---|---|
| Low | 3 | 8 |
| Medium | 9 | 1 |
| High | 1 | 1 |
| Extreme | 0 | 10 |

#### 5.3.1.4. Calculation Errors

This section narrows down the results to the 12 calculation error bugs contained in Wireshark. Table 35, as a focused version of Table 5, presents the number of calculation errors found by each tool, the accuracy of the findings, and the recall and discrimination rate. Excluding tools with nil findings, the average recall reached 20 % and discrimination rate 17 %. These metrics are slightly higher than the overall recall and discrimination rate (Section 5.3.1.1), showing that tools found these bugs more readily than average. Furthermore, recall and discrimination rate remain close, evidence that the tools mostly understood the bugs they reported.

Table 36 presents the tool warning overlap for calculation errors, which represented 16 % of the bugs in Wireshark. The low number of calculation bugs and lower number of bugs found by tools did not allow to draw strong conclusions here. We could still determine that one bug was found by Tools B, H, K and E and that Tools B and H found the same three bugs. Eight bugs remained completely undiscovered.

**Table 35.** Recall and Discrimination on Calculation Errors in Wireshark

| Findings | Tool B | Tool H | Tool C | Tool E | Tool K | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 6 | 8 | 9 | 11 | 11 | 8 | 11 | 12 | 12 | 12 | 12 |
| Hint | 2 | 1 | 0 | 0 | 0 | 4 | 1 | 0 | 0 | 0 | 0 |
| Partial | 4 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 0 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TP | 4 | 3 | 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Disc. TP | 3 | 3 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| Recall | 33 % | 25 % | 25 % | 8 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Disc. Rate | 25 % | 25 % | 17 % | 8 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 36.** Tool Warning Overlap on Calculation Errors in Wireshark

| | Tool B | Tool H | Tool C | Tool K | Tool E | Tool J | Tool G | Tool F | Tool I | Tool D | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool B | N/A | 100 % | 33 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool H | 75 % | N/A | 33 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool C | 25 % | 33 % | N/A | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool K | 25 % | 33 % | 0 % | N/A | 100 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool E | 25 % | 33 % | 0 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool G | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool F | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool D | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |

Tables 37, 38, 39 and 40 summarize recall and discrimination rate for all existing vs. injected calculation errors in Wireshark.

Tables 37 and 38 show that existing calculation errors were found by several tools, while injected bugs remained completely undiscovered. This may indicate that our injected bugs were not realistic or too scarce. Table 39 reveals little decrease in discrimination vs. recall on existing bugs, demonstrating that tools understood the bugs they reported. Table 40 is shown for consistency but provides no meaningful information, since no injected bugs were found.

**Table 37.** Recall on Calculation Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool B | Tool C | Tool H | Tool E | Tool K | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 8 | 50 % | 38 % | 38 % | 13 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 4 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 38.** Discrimination on Calculation Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool H | Tool B | Tool C | Tool K | Tool E | Tool J | Tool G | Tool F | Tool I | Tool D | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 8 | 38 % | 38 % | 25 % | 13 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 4 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 39.** Discrimination on Calculation Errors for Existing Bugs in Wireshark

| | Tool B | Tool H | Tool C | Tool E | Tool K | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 50 % | 38 % | 38 % | 13 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 38 % | 38 % | 25 % | 13 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 40.** Recall and Discrimination on Calculation Errors for Injected Bugs in Wireshark

| | Tool B | Tool H | Tool C | Tool E | Tool K | Tool D | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

Tables 41, 42 and 43 present calculation error results sliced by bug complexity. Similarly to Tables 11 and 12, each increase in bug complexity reduced the number of tools that found calculation errors. Recall and discrimination rate were more evenly distributed across bug complexities than for any other bug type. Table 43 has meaningful information for medium- and high-complexity bugs only and suggests a larger decrease in discrimination compared with recall for the high-complexity bugs.

**Table 41.** Recall per Bug Complexity on Calculation Errors in Wireshark

| Complexity | Bugs | Tool H | Tool B | Tool C | Tool K | Tool E | Tool A | Tool D | Tool F | Tool G | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Medium** | 7 | 29 % | 43 % | 14 % | 14 % | 14 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 4 | 25 % | 25 % | 50 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Extreme** | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 42.** Discrimination per Bug Complexity on Calculation Errors in Wireshark

| Complexity | Bugs | Tool H | Tool B | Tool C | Tool K | Tool E | Tool A | Tool D | Tool F | Tool G | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Medium | 7 | 29 % | 29 % | 14 % | 14 % | 14 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 4 | 25 % | 25 % | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 43.** Effect of Bug Complexity on Discrimination for Calculation Errors in Wireshark

| Complexity | Bugs | Tool H | Tool B | Tool C | Tool K | Tool E | Tool A | Tool D | Tool F | Tool G | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Medium | 7 | 0 % | -33 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| High | 4 | 0 % | 0 % | -50 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Extreme | 1 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

Table 44 breaks down the number of existing and injected calculation errors across the different levels of bug complexity in Wireshark. Existing bugs' complexity was clustered in the medium to high range while injected bugs were few and more evenly distributed.

**Table 44.** Breakdown of Bug Count per Bug Properties on Calculation Errors in Wireshark

| Complexity | Existing | Injected |
|---|---|---|
| Low | 0 | 0 |
| Medium | 5 | 2 |
| High | 3 | 1 |
| Extreme | 0 | 1 |

### 5.3.1.5. Initialization Errors

This section narrows down the results to the 11 initialization error bugs contained in Wireshark. Table 45, as a focused version of Table 5, presents the number of initialization errors found by each tool, the accuracy of the findings, and the recall and discrimination rate. Excluding tools with nil findings, the average recall reached 38 % and discrimination rate 33 %. These metrics are markedly higher than the overall recall and discrimination rate (Section 5.3.1.1), showing that tools found these bugs with more success than average. Furthermore, recall and discrimination rate remain close, evidence that most tools understood the bugs they reported. Tools D and B were the only tools with less than perfect discrimination on our initialization bugs.

Table 46 presents the tool warning overlap for initialization errors, which represented 15 % of the bugs in Wireshark. Despite the lower number of bugs of this type, we observe very high overlap across the board, showing that the tools found essentially the same bugs.

**Table 45.** Recall and Discrimination on Initialization Errors in Wireshark

| Findings | Tool E | Tool H | Tool F | Tool C | Tool G | Tool D | Tool B | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 6 | 6 | 6 | 7 | 7 | 5 | 7 | 10 | 11 | 11 | 11 |
| Hint | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Partial | 1 | 1 | 4 | 2 | 3 | 2 | 3 | 1 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 4 | 4 | 1 | 1 | 1 | 4 | 0 | 0 | 0 | 0 | 0 |
| TP | 5 | 5 | 5 | 4 | 4 | 6 | 3 | 1 | 0 | 0 | 0 |
| Disc. TP | 5 | 5 | 5 | 4 | 4 | 3 | 2 | 1 | 0 | 0 | 0 |
| Bugs | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| Recall | 45 % | 45 % | 45 % | 36 % | 36 % | 55 % | 27 % | 9 % | 0 % | 0 % | 0 % |
| Disc. Rate | 45 % | 45 % | 45 % | 36 % | 36 % | 27 % | 18 % | 9 % | 0 | 0 | 0 |

**Table 46.** Tool Warning Overlap on Initialization Errors in Wireshark

| | Tool H | Tool F | Tool E | Tool G | Tool C | Tool D | Tool B | Tool K | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool H | N/A | 100 % | 100 % | 100 % | 75 % | 83 % | 100 % | 100 % | N/A | N/A | N/A |
| Tool F | 100 % | N/A | 100 % | 100 % | 75 % | 83 % | 100 % | 100 % | N/A | N/A | N/A |
| Tool E | 100 % | 100 % | N/A | 100 % | 75 % | 83 % | 100 % | 100 % | N/A | N/A | N/A |
| Tool G | 80 % | 80 % | 80 % | N/A | 75 % | 67 % | 67 % | 0 % | N/A | N/A | N/A |
| Tool C | 60 % | 60 % | 60 % | 75 % | N/A | 67 % | 33 % | 0 % | N/A | N/A | N/A |
| Tool D | 100 % | 100 % | 100 % | 100 % | 100 % | N/A | 100 % | 100 % | N/A | N/A | N/A |
| Tool B | 60 % | 60 % | 60 % | 50 % | 25 % | 50 % | N/A | 100 % | N/A | N/A | N/A |
| Tool K | 20 % | 20 % | 20 % | 0 % | 0 % | 17 % | 33 % | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |

Tables 47, 48, 49 and 50 summarize recall and discrimination rate for all existing vs. injected initialization errors in Wireshark.

Tables 47 and 48 show that existing initialization errors bugs remained largely undiscovered while injected bugs were found by most tools. This may indicate that our injected bugs were too simple to be realistic. 49 is shown for consistency but provides little meaningful information. 50 reveals little loss in discrimination vs. recall on injected bugs, demonstrating that most tools understood the bugs they reported.

**Table 47.** Recall on Initialization Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool C | Tool D | Tool E | Tool H | Tool F | Tool G | Tool B | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 4 | 25 % | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 7 | 43 % | 71 % | 71 % | 71 % | 71 % | 57 % | 43 % | 14 % | 0 % | 0 % | 0 % |

**Table 48.** Discrimination on Initialization Errors for Existing vs. Injected Bugs in Wireshark

| | Bugs | Tool C | Tool H | Tool F | Tool E | Tool G | Tool D | Tool B | Tool K | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Existing** | 4 | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Injected** | 7 | 43 % | 71 % | 71 % | 71 % | 57 % | 43 % | 29 % | 14 % | 0 % | 0 % | 0 % |

**Table 49.** Recall and Discrimination on Initialization Errors for Existing Bugs in Wireshark

| | Tool C | Tool D | Tool E | Tool H | Tool F | Tool G | Tool B | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 25 % | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 50.** Discrimination on Initialization Errors for Injected Bugs in Wireshark

| | Tool E | Tool H | Tool F | Tool G | Tool D | Tool C | Tool B | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 71 % | 71 % | 71 % | 57 % | 71 % | 43 % | 43 % | 14 % | 0 % | 0 % | 0 % |
| **Disc. Rate** | 71 % | 71 % | 71 % | 57 % | 43 % | 43 % | 29 % | 14 % | 0 % | 0 % | 0 % |

Tables 51, 52 and 53 present initialization error results sliced by bug complexity. To a higher degree than in Tables 11 and 12, each increase in bug complexity reduced the number of tools that found initialization errors. No tool found any of the high- or extreme-complexity bugs. Recall and discrimination rate were generally higher at lower complexity. Table 53 highlights the same phenomenon observed in Table 13, i.e., higher complexity correlated with a higher drop in discrimination rate compared to recall.

**Table 51.** Recall per Bug Complexity on Initialization Errors in Wireshark

| Complexity | Bugs | Tool C | Tool E | Tool F | Tool H | Tool G | Tool D | Tool B | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 5 | 60 % | 100 % | 100 % | 100 % | 80 % | 100 % | 60 % | 20 % | 0 % | 0 % | 0 % |
| **Medium** | 3 | 33 % | 0 % | 0 % | 0 % | 0 % | 33 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 2 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Extreme** | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 52.** Discrimination per Bug Complexity on Initialization Errors in Wireshark

| Complexity | Bugs | Tool C | Tool E | Tool F | Tool H | Tool G | Tool D | Tool B | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 5 | 60 % | 100 % | 100 % | 100 % | 80 % | 60 % | 40 % | 20 % | 0 % | 0 % | 0 % |
| Medium | 3 | 33 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 2 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 53.** Effect of Bug Complexity on Discrimination for Initialization Errors in Wireshark

| Complexity | Bugs | Tool C | Tool E | Tool F | Tool H | Tool G | Tool D | Tool B | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 5 | 0 % | 0 % | 0 % | 0 % | 0 % | -40 % | -33 % | 0 % | N/A | N/A | N/A |
| Medium | 3 | 0 % | N/A | N/A | N/A | N/A | -100 % | N/A | N/A | N/A | N/A | N/A |
| High | 2 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Extreme | 1 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

Table 54 breaks down the number of existing and injected initialization errors across the different levels of bug complexity in Wireshark. Existing bugs' complexity was clustered in the medium to high range while injected bugs were mostly of low complexity.

**Table 54.** Breakdown of Bug Count per Bug Properties on Initialization Errors in Wireshark

| Complexity | Existing | Injected |
|---|---|---|
| Low | 0 | 5 |
| Medium | 2 | 1 |
| High | 2 | 0 |
| Extreme | 0 | 1 |

#### 5.3.1.6. Summary

This section summarizes the results obtained from Wireshark on the different bug types.

Table 55 presents the discrimination rate for each tool on the different bug types. Tools (except Tools A, J and I) found and understood initialization errors with an average discrimination rate of 33 %, pointer errors with an average of 22 %, calculation errors with an average of 10 % and buffer errors with an average of 7 %.

Tables 56, 57, 58 and 59 are presented in the same order to ease their comparison by the reader. They expose the differences in how tools handled existing vs. injected bugs.

Generally, tools understood the bugs they reported for all bug types, as demonstrated by the similarities in Tables 56 and 57 for existing bugs and in Tables 58 and 59 for injected bugs.

Comparing Tables 56 and 58 or Tables 57 and 59 shows that tools behaved very differently on existing bugs than on injected bugs. This could imply that the bugs we injected in Wireshark are not representative of real bugs. Ignoring Tools A, J and I, overall discrimination rate is much higher for injected bugs (27 %) than for existing bugs (6 %). This leads us to conclude that injected bugs were much easier for tools to find than existing ones.

**Table 55.** Discrimination Rate for All Bug Categories in Wireshark

| | Bugs | Tool C | Tool H | Tool E | Tool F | Tool B | Tool G | Tool D | Tool K | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Buffer** | 44 | 9 % | 11 % | 9 % | 5 % | 7 % | 2 % | 5 % | 9 % | 0 % | 0 % | 0 % |
| **Calculations** | 12 | 17 % | 25 % | 8 % | 0 % | 25 % | 0 % | 0 % | 8 % | 0 % | 0 % | 0 % |
| **Pointers** | 33 | 30 % | 24 % | 27 % | 27 % | 21 % | 21 % | 18 % | 3 % | 0 % | 0 % | 0 % |
| **Initialization** | 11 | 36 % | 45 % | 45 % | 45 % | 18 % | 36 % | 27 % | 9 % | 0 % | 0 % | 0 % |
| **Overall** | 75 | 20 % | 16 % | 16 % | 16 % | 13 % | 11 % | 9 % | 8 % | 0 % | 0 % | 0 % |

**Table 56.** Recall on Existing Bugs for All Bug Categories in Wireshark

| | Bugs | Tool K | Tool B | Tool C | Tool H | Tool D | Tool E | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Buffer** | 28 | 14 % | 18 % | 18 % | 11 % | 4 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Calculations** | 8 | 13 % | 50 % | 38 % | 38 % | 0 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Pointers** | 13 | 0 % | 23 % | 31 % | 0 % | 31 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Initialization** | 4 | 0 % | 0 % | 25 % | 0 % | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Overall** | 49 | 10 % | 16 % | 18 % | 6 % | 10 % | 6 % | 2 % | 0 % | 0 % | 0 % | 0 % |

**Table 57.** Discrimination on Existing Bugs for All Bug Categories in Wireshark

| | Bugs | Tool K | Tool B | Tool C | Tool H | Tool D | Tool E | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Buffer** | 28 | 7 % | 11 % | 11 % | 11 % | 0 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Calculations** | 8 | 13 % | 38 % | 25 % | 38 % | 0 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Pointers** | 13 | 0 % | 23 % | 31 % | 0 % | 15 % | 8 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Initialization** | 4 | 0 % | 0 % | 25 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Overall** | 49 | 6 % | 12 % | 14 % | 6 % | 4 % | 6 % | 2 % | 0 % | 0 % | 0 % | 0 % |

**Table 58.** Recall on Injected Bugs for All Bug Categories in Wireshark

| | Bugs | Tool K | Tool B | Tool C | Tool H | Tool D | Tool E | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Buffer** | 16 | 13 % | 6 % | 6 % | 13 % | 19 % | 13 % | 13 % | 6 % | 0 % | 0 % | 0 % |
| **Calculations** | 4 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Pointers** | 20 | 5 % | 25 % | 30 % | 40 % | 35 % | 40 % | 45 % | 35 % | 0 % | 0 % | 0 % |
| **Initialization** | 7 | 14 % | 43 % | 43 % | 71 % | 71 % | 71 % | 71 % | 57 % | 0 % | 0 % | 0 % |
| **Overall** | 26 | 12 % | 19 % | 31 % | 35 % | 31 % | 35 % | 42 % | 31 % | 0 % | 0 % | 0 % |

**Table 59.** Discrimination on Injected Bugs for All Bug Categories in Wireshark

| | Bugs | Tool K | Tool B | Tool C | Tool H | Tool D | Tool E | Tool F | Tool G | Tool A | Tool J | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Buffer** | 16 | 13 % | 0 % | 6 % | 13 % | 13 % | 13 % | 13 % | 6 % | 0 % | 0 % | 0 % |
| **Calculations** | 4 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Pointers** | 20 | 5 % | 20 % | 30 % | 40 % | 20 % | 40 % | 45 % | 35 % | 0 % | 0 % | 0 % |
| **Initialization** | 7 | 14 % | 29 % | 43 % | 71 % | 43 % | 71 % | 71 % | 57 % | 0 % | 0 % | 0 % |
| **Overall** | 26 | 12 % | 15 % | 31 % | 35 % | 19 % | 35 % | 42 % | 31 % | 0 % | 0 % | 0 % |

### 5.3.2. SQLite: Mostly-automatically Injected Bugs

SQLite contained 30 injected bugs, including 20 buffer errors and 10 calculation errors. In this section, we present general results and results per bug category.

The bugs injected in SQLite had many shortcomings, as described in Section 5.2.3. Therefore, the reader should avoid drawing conclusions from these results.

Table 60 offers an overview of the number of bugs found by each tool, the accuracy of the findings, and recall and discrimination rate. Excluding tools with nil findings (Tools J, F, I and A), the average recall reached 36 % and discrimination rate 21 %. Tools B and K experienced a significant drop between recall and discrimination rate, but the other tools retained the same discrimination rate as their recall, supporting the idea that they understood the bugs they reported. However, the issues mentioned in Section 5.2.3 Item 4 keep this result from being conclusive.

Table 61 narrows down the results to the 10 calculation error bugs contained in SQLite. It presents the number of such bugs found by each tool, the accuracy of the findings, and the recall and discrimination rate. Excluding tools with nil findings, the average recall reached 90 % and discrimination rate 33 %. Out of 7 tools that reported bugs in SQLite, only 3 reported calculation errors. Furthermore, Tool C understood the bugs it reported, but Tools K and B did not properly discriminate bugs from fixes, despite a high recall.

Table 62 narrows down the results to the 20 buffer error bugs contained in SQLite. It presents the number of such bugs found by each tool, the accuracy of the findings, and the recall and discrimination rate. Excluding tools with nil findings, the average recall reached 34 % and discrimination rate 25 %. All tools, except for Tool K, had perfect discrimination, suggesting that they understood the bugs they reported.

Table 63 summarizes the results obtained from SQLite on the two bug types. Tool C performed best with a high discrimination rate on both calculation and buffer errors. Tools H, D, G, B, E and K only found buffer errors with a discrimination rate ranging from 15 % to 35 %. Tools J, F, I and A did not find any of the bugs.

**Table 60.** Overall Recall and Discrimination in SQLite

| Findings | Tool C | Tool H | Tool D | Tool B | Tool G | Tool E | Tool K | Tool J | Tool F | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 12 | 23 | 25 | 8 | 26 | 26 | 4 | 30 | 30 | 30 | 30 |
| Hint | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partial | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 18 | 7 | 5 | 10 | 4 | 4 | 26 | 0 | 0 | 0 | 0 |
| TP | 18 | 7 | 5 | 11 | 4 | 4 | 26 | 0 | 0 | 0 | 0 |
| Disc. TP | 18 | 7 | 5 | 4 | 4 | 4 | 3 | 0 | 0 | 0 | 0 |
| Bugs | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| Recall | 60 % | 23 % | 17 % | 37 % | 13 % | 13 % | 87 % | 0 % | 0 % | 0 % | 0 % |
| Disc. Rate | 60 % | 23 % | 17 % | 13 % | 13 % | 13 % | 10 % | 0 % | 0 % | 0 % | 0 % |

**Table 61.** Recall and Discrimination on Calculation Errors in SQLite

| Findings | Tool C | Tool K | Tool B | Tool H | Tool D | Tool G | Tool E | Tool F | Tool J | Tool A | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 0 | 0 | 2 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Hint | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partial | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 10 | 10 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TP | 10 | 10 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Disc. TP | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Recall | 100 % | 100 % | 70 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Disc. Rate | 100 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 62.** Recall and Discrimination on Buffer Errors in SQLite

| Findings | Tool C | Tool H | Tool D | Tool B | Tool G | Tool E | Tool K | Tool F | Tool J | Tool A | Tool I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 12 | 13 | 15 | 6 | 16 | 16 | 4 | 20 | 20 | 20 | 20 |
| Hint | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partial | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 8 | 7 | 5 | 3 | 4 | 4 | 16 | 0 | 0 | 0 | 0 |
| TP | 8 | 7 | 5 | 4 | 4 | 4 | 16 | 0 | 0 | 0 | 0 |
| Disc. TP | 8 | 7 | 5 | 4 | 4 | 4 | 3 | 0 | 0 | 0 | 0 |
| Bugs | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| Recall | 40 % | 35 % | 25 % | 20 % | 20 % | 20 % | 80 % | 0 % | 0 % | 0 % | 0 % |
| Disc. Rate | 40 % | 35 % | 25 % | 20 % | 20 % | 20 % | 15 % | 0 % | 0 % | 0 % | 0 % |

**Table 63.** Discrimination Rate for All Bug Categories in SQLite

| | Bugs | Tool C | Tool H | Tool D | Tool G | Tool B | Tool E | Tool K | Tool J | Tool F | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Calc. | 10 | 100 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Buffer | 20 | 40 % | 35 % | 25 % | 20 % | 20 % | 20 % | 15 % | 0 % | 0 % | 0 % | 0 % |
| Overall | 30 | 60 % | 23 % | 17 % | 13 % | 13 % | 13 % | 10 % | 0 % | 0 % | 0 % | 0 % |

Table 64 offers a view of tool warning overlap. Tool C's high recall of 60 % (Table 60) affected its full overlap of Tools H, D, B, G and E and a high overlap of Tool K. The limited diversity of bugs injected in SQLite could be a contributing factor in the high overlap across the board. Tools can usually find simple patterns anywhere in a code base, so the application of the same templates for multiple bugs could allow a tool that can find a particular bug, to find other bugs using the same pattern.

**Table 64.** Overall Tool Warning Overlap in SQLite

| | Tool C | Tool H | Tool D | Tool B | Tool G | Tool E | Tool K | Tool J | Tool F | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Tool C** | N/A | 100 % | 100 % | 100 % | 100 % | 100 % | 54 % | N/A | N/A | N/A | N/A |
| **Tool H** | 39 % | N/A | 100 % | 27 % | 100 % | 100 % | 15 % | N/A | N/A | N/A | N/A |
| **Tool D** | 28 % | 71 % | N/A | 27 % | 100 % | 100 % | 15 % | N/A | N/A | N/A | N/A |
| **Tool B** | 61 % | 43 % | 60 % | N/A | 75 % | 75 % | 38 % | N/A | N/A | N/A | N/A |
| **Tool G** | 22 % | 57 % | 80 % | 27 % | N/A | 100 % | 15 % | N/A | N/A | N/A | N/A |
| **Tool E** | 22 % | 57 % | 80 % | 27 % | 100 % | N/A | 15 % | N/A | N/A | N/A | N/A |
| **Tool K** | 78 % | 57 % | 80 % | 91 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A |
| **Tool J** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| **Tool F** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| **Tool I** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| **Tool A** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |

The SAMATE team classified the 30 bugs in SQLite according to their complexity from a static analysis standpoint. Please refer to Table 11's comments for further explanations and examples. SQLite contained 14 low-, 16 medium- and no high- or extreme-complexity bugs.

Tables 65, 66 and 67 present recall and discrimination rate sliced by bug complexity. Increase in bug complexity reduced the number of tools that found bugs. Ignoring Tools A, F, I and J, the average recall was 55 % for low-complexity bugs but only 19 % for medium-complexity bugs. Table 67 shows the percentage drop between undiscriminated recall (i.e., regular recall) and discriminated recall (i.e., discrimination rate). In this case, the results are inconclusive and the table is presented for consistency.

**Table 65.** Recall per Bug Complexity in SQLite

| Complexity | Bugs | Tool C | Tool H | Tool D | Tool B | Tool E | Tool G | Tool K | Tool A | Tool F | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 14 | 100 % | 29 % | 29 % | 71 % | 29 % | 29 % | 100 % | 0 % | 0 % | 0 % | 0 % |
| **Medium** | 16 | 25 % | 19 % | 6 % | 6 % | 0 % | 0 % | 75 % | 0 % | 0 % | 0 % | 0 % |

**Table 66.** Discrimination per Bug Complexity in SQLite

| Complexity | Bugs | Tool C | Tool H | Tool D | Tool B | Tool E | Tool G | Tool K | Tool A | Tool F | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 14 | 100 % | 29 % | 29 % | 21 % | 29 % | 29 % | 21 % | 0 % | 0 % | 0 % | 0 % |
| **Medium** | 16 | 25 % | 19 % | 6 % | 6 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 67.** Effect of Bug Complexity on Discrimination in SQLite

| Complexity | Bugs | Tool C | Tool H | Tool D | Tool G | Tool E | Tool B | Tool K | Tool J | Tool F | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 14 | 0 % | 0 % | 0 % | -70 % | 0 % | 0 % | -79 % | N/A | N/A | N/A | N/A |
| Medium | 16 | 0 % | 0 % | 0 % | 0 % | N/A | N/A | -100 % | N/A | N/A | N/A | N/A |

Tables 68 and 69 present recall and discrimination rate for low- and medium-complexity bugs, respectively. Tools C, G, H, D and E showed perfect discrimination for low-complexity bugs, but the issues mentioned in Section 5.2.3 Item 4 might have skewed the results. Tools K and B experienced a large drop between recall and discrimination. Similarly, on medium-complexity bugs, Tools C, H, B and D showed perfect discrimination, but Tool K suffered from a significant drop.

**Table 68.** Recall and Discrimination on Low Complexity Bugs in SQLite

| | Tool C | Tool G | Tool H | Tool D | Tool E | Tool K | Tool B | Tool J | Tool F | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 100 % | 29 % | 29 % | 29 % | 29 % | 100 % | 71 % | 0 % | 0 % | 0 % | 0 % |
| **Disc.Rate** | 100 % | 29 % | 29 % | 29 % | 29 % | 21 % | 21 % | 0 % | 0 % | 0 % | 0 % |

**Table 69.** Recall and Discrimination on Medium Complexity Bugs in SQLite

| | Tool C | Tool H | Tool B | Tool D | Tool K | Tool J | Tool G | Tool F | Tool I | Tool A | Tool E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Recall** | 25 % | 19 % | 6 % | 6 % | 75 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Disc.Rate** | 25 % | 19 % | 6 % | 6 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

Overall, the results from SQLite offered limited insights on tool behavior and efficiency, but raised multiple issues regarding the bug injection process. These observations will be taken into account to improve the bug injection process in future SATEs.

### 5.3.3. CGC: Manually Built Test Suite

For this report, we sampled 111 of the more than 350 bugs contained in the CGC test suite. In the following sections, we first present the general results on all bugs in Section 5.3.3.1, then slice the results along the various bug properties to offer more specific insights on tool strengths and weaknesses and on bug quality. Each bug can be a chain of different weaknesses and fall in more than one category in Sections 5.3.3.2 through 5.3.3.7.

CGC used custom memory management and string/buffer handling functions, which were included in the test suite. This may have increased code complexity, compared to using standard APIs.

The fixed version of the test suite was not provided to participants as part of SATE VI, so discrimination rates could not be calculated.

#### 5.3.3.1. Overall Analysis

This section presents the tool results on all 111 bugs sampled from CGC. Table 70 offers an overview of the number of bugs found by each tool, the accuracy of the findings, and recall. Excluding tools without relevant findings, recall varied from 1 % to 13 %. Note that most bugs in the sample related to buffer errors, which could explain the congruence between this score range and the results observed on buffer errors in Wireshark (Table 15).

Table 71 offers a view of tool warning overlap. Tools F, D and G had lower recall and a higher overlap than most other tools. The majority of bugs they found were reported by other tools, suggesting that these bugs were easier to find than average.

Tool C stands out with a very low overlap, so although its recall was on the lower end of the spectrum, it reported bugs that other tools missed. It could be used to increase recall in complement to one of the other tools.

The SAMATE team classified the 111 bugs in CGC according to their complexity from a static analysis standpoint. Please refer to Table 11's comments for further explanations and examples. CGC contained 28 low-, 64 medium-, 17 high- and 2 extreme-complexity bugs.

Table 72 presents recall sliced by bug complexity. Increase in bug complexity reduced the number of tools that found bugs. Ignoring Tools A, I and J, the average recall was 17 % for low-complexity bugs, 4 % for medium-complexity bugs, 1 % for high-complexity bugs and 0 % for extreme-complexity bugs.

**Table 70.** Overall Recall in CGC

| Findings | Tool E | Tool H | Tool B | Tool F | Tool D | Tool C | Tool G | Tool K | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 95 | 97 | 96 | 103 | 102 | 105 | 108 | 110 | 111 | 111 | 111 |
| Hint | 2 | 0 | 4 | 1 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| Partial | 3 | 5 | 6 | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 11 | 9 | 5 | 5 | 5 | 4 | 2 | 0 | 0 | 0 | 0 |
| TP | 14 | 14 | 11 | 7 | 6 | 5 | 3 | 1 | 0 | 0 | 0 |
| Bugs | 111 | 111 | 111 | 111 | 111 | 111 | 111 | 111 | 111 | 111 | 111 |
| Recall | 13 % | 13 % | 10 % | 6 % | 5 % | 5 % | 3 % | 1 % | 0 % | 0 % | 0 % |

**Table 71.** Overall Tool Warning Overlap in CGC

| | Tool H | Tool E | Tool B | Tool F | Tool D | Tool C | Tool G | Tool K | Tool A | Tool I | Tool J |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Tool H** | N/A | 57 % | 18 % | 57 % | 50 % | 20 % | 100 % | 0 % | N/A | N/A | N/A |
| **Tool E** | 57 % | N/A | 45 % | 86 % | 67 % | 40 % | 100 % | 0 % | N/A | N/A | N/A |
| **Tool B** | 14 % | 36 % | N/A | 71 % | 83 % | 40 % | 67 % | 0 % | N/A | N/A | N/A |
| **Tool F** | 29 % | 43 % | 45 % | N/A | 83 % | 40 % | 100 % | 0 % | N/A | N/A | N/A |
| **Tool D** | 21 % | 29 % | 45 % | 71 % | N/A | 40 % | 67 % | 0 % | N/A | N/A | N/A |
| **Tool C** | 7 % | 14 % | 18 % | 29 % | 33 % | N/A | 0 % | 0 % | N/A | N/A | N/A |
| **Tool G** | 21 % | 21 % | 18 % | 43 % | 33 % | 0 % | N/A | 0 % | N/A | N/A | N/A |
| **Tool K** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| **Tool J** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| **Tool I** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |
| **Tool A** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A |

**Table 72.** Overall Recall per Bug Complexity in CGC

| Complexity | Bugs | Tool H | Tool E | Tool B | Tool F | Tool D | Tool C | Tool G | Tool K | Tool J | Tool I | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 28 | 29 % | 32 % | 18 % | 21 % | 14 % | 11 % | 11 % | 4 % | 0 % | 0 % | 0 % |
| **Medium** | 64 | 8 % | 8 % | 9 % | 2 % | 3 % | 3 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 17 | 6 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Extreme** | 2 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

### 5.3.3.2. All Buffer Errors

This section narrows down the results to the 84 buffer error bugs contained in the CGC sample. It combines heap-based (51 %), stack-based (38 %) and data-based (11 %) buffer errors, which are individually covered in Sections 5.3.3.3, 5.3.3.4 and 5.3.3.5, respectively. Buffer errors accounted for 65 % of all bugs in the CGC sample, inducing a high correlation between the overall results in Section 5.3.3.1 and these results, which are offered as a more precise view of tool effectiveness on buffer errors.

Table 73 breaks down the number of bugs found by each tool, the accuracy of the findings, and recall. Excluding tools with nil findings, recall ranged from 2 % to 14 % with an average of 7 %, in line with the results presented in Table 70. Similarly, tool warning overlap as detailed in Table 74 follows the same trend as Table 71.

Table 75 presents buffer error results sliced by bug complexity. Following the trend of Table 72, each increase in bug complexity reduced the number of tools that found buffer errors. Recall was on par with overall results.

**Table 73.** Recall on All Buffer Errors in CGC

| Findings | Tool H | Tool E | Tool B | Tool F | Tool G | Tool D | Tool C | Tool J | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 72 | 72 | 75 | 79 | 81 | 79 | 81 | 84 | 84 | 84 | 84 |
| Hint | 0 | 1 | 2 | 1 | 0 | 2 | 1 | 0 | 0 | 0 | 0 |
| Partial | 3 | 2 | 5 | 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 9 | 9 | 2 | 2 | 2 | 2 | 1 | 0 | 0 | 0 | 0 |
| TP | 12 | 11 | 7 | 4 | 3 | 3 | 2 | 0 | 0 | 0 | 0 |
| Bugs | 84 | 84 | 84 | 84 | 84 | 84 | 84 | 84 | 84 | 84 | 84 |
| Recall | 14 % | 13 % | 8 % | 5 % | 4 % | 4 % | 2 % | 0 % | 0 % | 0 % | 0 % |

**Table 74.** Tool Warning Overlap on All Buffer Errors in CGC

| | Tool H | Tool E | Tool B | Tool F | Tool G | Tool D | Tool C | Tool J | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool H | N/A | 73 % | 29 % | 100 % | 100 % | 100 % | 50 % | N/A | N/A | N/A | N/A |
| Tool E | 67 % | N/A | 29 % | 100 % | 100 % | 67 % | 50 % | N/A | N/A | N/A | N/A |
| Tool B | 17 % | 18 % | N/A | 50 % | 67 % | 67 % | 0 % | N/A | N/A | N/A | N/A |
| Tool F | 33 % | 36 % | 29 % | N/A | 100 % | 67 % | 0 % | N/A | N/A | N/A | N/A |
| Tool G | 25 % | 27 % | 29 % | 75 % | N/A | 67 % | 0 % | N/A | N/A | N/A | N/A |
| Tool D | 25 % | 18 % | 29 % | 50 % | 67 % | N/A | 0 % | N/A | N/A | N/A | N/A |
| Tool C | 8 % | 9 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| Tool K | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A |

**Table 75.** Recall per Bug Complexity on All Buffer Errors in CGC

| Complexity | Bugs | Tool H | Tool E | Tool B | Tool F | Tool G | Tool D | Tool C | Tool J | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 23 | 35 % | 30 % | 13 % | 17 % | 13 % | 9 % | 9 % | 0 % | 0 % | 0 % | 0 % |
| Medium | 48 | 8 % | 8 % | 8 % | 0 % | 0 % | 2 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 12 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

#### 5.3.3.3. Heap-based Buffer Errors

This section narrows down the results to the 43 heap-based buffer error bugs contained in the CGC sample. These errors accounted for 40 % of all bugs in the sample.

Table 76 breaks down the number of bugs found by each tool, the accuracy of the findings, and recall. Excluding tools with nil findings, recall remained in a narrow 7 % to 12 % range. The high tool warning overlap detailed in Table 77 shows that tools largely found the same few bugs. Moreover, tools clustered their findings around simple and some medium complexity bugs, as demonstrated by the break down in Table 78.

As a reminder, CGC used a custom memory management framework, which could have impacted these results.

**Table 76.** Recall on Heap-based Buffer Errors in CGC

| Findings | Tool B | Tool E | Tool H | Tool D | Tool G | Tool F | Tool C | Tool J | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 36 | 39 | 39 | 39 | 40 | 40 | 42 | 43 | 43 | 43 | 43 |
| Hint | 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Partial | 3 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 2 | 4 | 3 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 0 |
| TP | 5 | 4 | 4 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 43 | 43 | 43 | 43 | 43 | 43 | 43 | 43 | 43 | 43 | 43 |
| Recall | 12 % | 9 % | 9 % | 7 % | 7 % | 7 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 77.** Tool Warning Overlap on Heap-based Buffer Errors in CGC

| | Tool B | Tool H | Tool E | Tool G | Tool F | Tool D | Tool J | Tool C | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool B | N/A | 50 % | 50 % | 67 % | 67 % | 67 % | N/A | N/A | N/A | N/A | N/A |
| Tool H | 40 % | N/A | 75 % | 100 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A |
| Tool E | 40 % | 75 % | N/A | 100 % | 100 % | 67 % | N/A | N/A | N/A | N/A | N/A |
| Tool G | 40 % | 75 % | 75 % | N/A | 100 % | 67 % | N/A | N/A | N/A | N/A | N/A |
| Tool F | 40 % | 75 % | 75 % | 100 % | N/A | 67 % | N/A | N/A | N/A | N/A | N/A |
| Tool D | 40 % | 75 % | 50 % | 67 % | 67 % | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A |
| Tool C | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A |
| Tool K | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A |

**Table 78.** Recall per Bug Complexity on Heap-based Buffer Errors in CGC

| Complexity | Bugs | Tool B | Tool H | Tool E | Tool G | Tool F | Tool D | Tool J | Tool C | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 11 | 18 % | 27 % | 27 % | 27 % | 27 % | 18 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Medium | 25 | 12 % | 4 % | 4 % | 0 % | 0 % | 4 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 6 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

### 5.3.3.4. Stack-based Buffer Errors

This section narrows down the results to the 32 stack-based buffer error bugs contained in the CGC sample. These errors accounted for 29 % of all bugs in the sample.

Table 79 breaks down the number of bugs found by each tool, the accuracy of the findings, and recall. Excluding tools with nil findings, recall was clustered in two groups, one in 19-22 % range and one in the 3-6 % range. Indeed, Tools E and H significantly outperformed other tools on stack-based buffer errors.

Tools E and H found essentially the same bugs, as demonstrated by their high overlap in Table 80. They also both found the bugs reported by Tools C and F. On the other hand, Tool B's findings were not reported by any other tool.

Table 81 shows that 5 tools found low-complexity bugs while only 3 for medium-complexity. High-complexity bugs remained undiscovered. Recall was significantly higher for low-complexity bugs than for their medium-complexity counterparts.

Unlike Section 5.3.3.3, the custom memory management framework used by CGC should have interfered less with the results on stack-based buffer errors, which do not rely directly on heap memory allocation schemes.

**Table 79.** Recall on Stack-based Buffer Errors in CGC

| Findings | Tool E | Tool H | Tool B | Tool C | Tool F | Tool D | Tool J | Tool G | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 24 | 26 | 30 | 31 | 31 | 31 | 32 | 32 | 32 | 32 | 32 |
| Hint | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Partial | 2 | 1 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 5 | 5 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TP | 7 | 6 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 |
| Recall | 22 % | 19 % | 6 % | 3 % | 3 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 80.** Tool Warning Overlap on Stack-based Buffer Errors in CGC

| | Tool E | Tool H | Tool B | Tool C | Tool F | Tool D | Tool J | Tool G | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Tool E** | N/A | 83 % | 0 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool H** | 71 % | N/A | 0 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool B** | 0 % | 0 % | N/A | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool C** | 14 % | 17 % | 0 % | N/A | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool F** | 14 % | 17 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool D** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool J** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool G** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool I** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool K** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool A** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A |

**Table 81.** Recall per Bug Complexity on Stack-based Buffer Errors in CGC

| Complexity | Bugs | Tool E | Tool H | Tool B | Tool C | Tool F | Tool J | Tool G | Tool I | Tool K | Tool D | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 10 | 40 % | 40 % | 10 % | 10 % | 10 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Medium** | 16 | 19 % | 13 % | 6 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 6 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Extreme** | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

### 5.3.3.5. Data-based Buffer Errors

This section narrows down the results to the 9 data-based buffer error bugs contained in the CGC sample. This category combines all buffer errors that are neither stack- nor heap-based, e.g., overflows of static global variables. These errors accounted for 8 % of all bugs in the CGC sample.

Table 82 breaks down the number of bugs found by each tool, the accuracy of the findings, and recall. Tools H and C were the only tools to report valid findings. Table 83 shows that they targeted different bugs (there was no overlap) and Table 84 that they each found one of the two low-complexity bugs, and one medium-complexity bug for Tool H. However, the small size of the sample limits the relevance of these results.

**Table 82.** Recall on Data-based Buffer Errors in CGC

| Findings | Tool H | Tool C | Tool F | Tool J | Tool G | Tool B | Tool I | Tool K | Tool D | Tool A | Tool E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 7 | 8 | 8 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| Hint | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partial | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TP | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| Recall | 22 % | 11 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 83.** Tool Warning Overlap on Data-based Buffer Errors in CGC

| | Tool H | Tool C | Tool A | Tool B | Tool D | Tool E | Tool F | Tool G | Tool I | Tool J | Tool K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool H | N/A | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool C | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool B | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool D | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool E | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool F | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool G | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool K | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

**Table 84.** Recall per Bug Complexity on Data-based Buffer Errors in CGC

| | Bugs | Tool H | Tool C | Tool J | Tool G | Tool F | Tool B | Tool I | Tool K | Tool D | Tool A | Tool E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 2 | 50 % | 50 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Medium | 7 | 14 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Extreme | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

### 5.3.3.6. Pointer Errors

This section narrows down the results to the 11 pointer error bugs contained in the CGC sample. These errors accounted for 10 % of all bugs in the sample.

Table 85 breaks down the number of bugs found by each tool, the accuracy of the findings, and recall. Only two tools correctly reported one bug each, with a recall of 9 %. Table 86 confirms that the two bugs are different and, according to Table 87, of medium complexity. The small size of the sample limits the relevance of these results.

**Table 85.** Recall on Pointer Errors in CGC

| Findings | Tool C | Tool H | Tool B | Tool D | Tool E | Tool J | Tool G | Tool F | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 10 | 10 | 10 | 10 | 10 | 11 | 11 | 11 | 11 | 11 | 11 |
| Hint | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partial | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| Recall | 9 % | 9 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 86.** Tool Warning Overlap on Pointer Errors in CGC

| | Tool C | Tool H | Tool A | Tool B | Tool D | Tool E | Tool F | Tool G | Tool I | Tool J | Tool K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool C | N/A | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool H | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool A | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool B | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool D | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool E | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool F | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool G | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool I | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool J | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Tool K | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

**Table 87.** Recall per Bug Complexity on Pointer Errors in CGC

| Complexity | Bugs | Tool H | Tool C | Tool J | Tool G | Tool F | Tool B | Tool I | Tool K | Tool D | Tool A | Tool E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Low | 1 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Medium | 8 | 13 % | 13 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| High | 2 | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| Extreme | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

### 5.3.3.7. Initialization Errors

This section narrows down the results to the 7 initialization error bugs contained in the CGC sample. These errors accounted for 6 % of all bugs in the sample.

Table 88 breaks down the number of bugs found by each tool, the accuracy of the findings, and recall. Excluding tools with nil findings, recall varied from 14 % to 29 %, i.e. 1 to 2 true positives for Tools B, E, F and D.

Table 89 shows that these four tools found the same two bugs. One bug was found by all four tools and the other by two tools. According to Table 90, the first bug was of low-complexity and the second of medium complexity.

**Table 88.** Recall on Initialization Errors in CGC

| Findings | Tool B | Tool E | Tool F | Tool D | Tool J | Tool G | Tool H | Tool C | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Miss | 5 | 5 | 6 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| Hint | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Partial | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Alternate | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TP | 2 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bugs | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| Recall | 29 % | 29 % | 14 % | 14 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |

**Table 89.** Tool Warning Overlap on Initialization Errors in CGC

| | Tool B | Tool E | Tool D | Tool F | Tool A | Tool C | Tool G | Tool H | Tool I | Tool J | Tool K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Tool B** | N/A | 100 % | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool E** | 100 % | N/A | 100 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool D** | 50 % | 50 % | N/A | 100 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool F** | 50 % | 50 % | 100 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool A** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool C** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool G** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool H** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool I** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool J** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Tool K** | 0 % | 0 % | 0 % | 0 % | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

**Table 90.** Recall per Bug Complexity on Initialization Errors in CGC

| Complexity | Bugs | Tool B | Tool E | Tool F | Tool D | Tool J | Tool G | Tool H | Tool C | Tool I | Tool K | Tool A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Low** | 2 | 50 % | 50 % | 50 % | 50 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **Medium** | 5 | 20 % | 20 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % |
| **High** | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| **Extreme** | 0 | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

## 5.4. Java

### 5.4.1. DSpace: Existing and Semi-automatically Injected Bugs

DSpace contained 18 existing and 12 injected XSS bugs.

Table 91 offers an overview of the number of bugs found by each tool, the accuracy of the findings, and recall and discrimination rate. Most tools correctly reported bugs, except Tools L and R, which did not support XSS. Recall ranged from 7 % to 100 % with an average of 60 % and discrimination rate from 13 % to 87 %. Tool O did not report false positives on the fixes corresponding to the bugs it correctly reported, so its recall and discrimination rate are the same. Other tools had high discrimination, except Tools P and M, which reported nearly as many false-positives as true positives.

The high recall induced a high overlap between tools in Table 92 and consequently, most bugs were found by more than one tool.

Tables 93, 94, 95 and 96 compare tool results on existing vs. injected bugs. Table 93 shows that recall was analogous for both types of bugs, but discrimination rate had more pronounced

variation for existing vs. injected bugs, as described in Table 94. Tables 95 and 96 reinforce this observation: most tools' discrimination was less than stellar on existing bugs, but it was substantially higher on injected bugs.

Tables 97, 98 and 99 break down the results by bug complexity. Table 97 shows some decrease in recall for medium-complexity bugs, compared to low-complexity bugs. However, the effect is much more contained than what we witnessed on the C test cases. Some tools even found more medium-complexity than low-complexity bugs. Discrimination, on the other hand, was more affected by increase in complexity, as can be seen in Tables 98 and 99. The discrimination rate for medium-complexity bugs was much lower than for low-complexity bugs, except for Tool O, which had perfect discrimination.

Table 100 breaks down existing and injected bugs across bug complexities. Injected bugs were all of low-complexity and existing bugs split between low- and medium-complexity.

**Table 91.** Recall and Discrimination on XSS in DSpace

| Findings | Tool O | Tool Q | Tool N | Tool P | Tool M | Tool L | Tool R |
|---|---|---|---|---|---|---|---|
| **Miss** | 2 | 9 | 16 | 0 | 21 | 30 | 30 |
| **Hint** | 2 | 2 | 1 | 0 | 7 | 0 | 0 |
| **Partial** | 12 | 12 | 0 | 30 | 2 | 0 | 0 |
| **Alternate** | 13 | 4 | 6 | 0 | 0 | 0 | 0 |
| **Match** | 1 | 3 | 7 | 0 | 0 | 0 | 0 |
| **TP** | 26 | 19 | 13 | 30 | 2 | 0 | 0 |
| **Disc. TP** | 26 | 13 | 10 | 4 | 0 | 0 | 0 |
| **Bugs** | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| **Recall** | 87 % | 63 % | 43 % | 100 % | 7 % | 0 % | 0 % |
| **Disc. Rate** | 87 % | 43 % | 33 % | 13 % | 0 % | 0 % | 0 % |

**Table 92.** Tool Warning Overlap on XSS in DSpace

| | Tool P | Tool O | Tool Q | Tool N | Tool M | Tool L | Tool R |
|---|---|---|---|---|---|---|---|
| **Tool P** | N/A | 100 % | 100 % | 100 % | 100 % | N/A | N/A |
| **Tool O** | 87 % | N/A | 89 % | 92 % | 100 % | N/A | N/A |
| **Tool Q** | 63 % | 65 % | N/A | 85 % | 50 % | N/A | N/A |
| **Tool N** | 43 % | 46 % | 58 % | N/A | 0 % | N/A | N/A |
| **Tool M** | 7 % | 8 % | 5 % | 0 % | N/A | N/A | N/A |
| **Tool L** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A |
| **Tool R** | 0 % | 0 % | 0 % | 0 % | 0 % | N/A | N/A |

**Table 93.** Recall for Existing vs. Injected Bugs in DSpace

| | Bugs | Tool P | Tool O | Tool Q | Tool N | Tool M | Tool L | Tool R |
|---|---|---|---|---|---|---|---|---|
| **Existing** | 18 | 100 % | 83 % | 61 % | 22 % | 6 % | 0 % | 0 % |
| **Injected** | 12 | 100 % | 92 % | 67 % | 75 % | 8 % | 0 % | 0 % |

**Table 94.** Discrimination Rate for Existing vs. Injected Bugs in DSpace

| | Bugs | Tool O | Tool Q | Tool P | Tool N | Tool L | Tool M | Tool R |
|---|---|---|---|---|---|---|---|---|
| **Existing** | 18 | 83 % | 33 % | 17 % | 11 % | 0 % | 0 % | 0 % |
| **Injected** | 12 | 92 % | 58 % | 8 % | 67 % | 0 % | 0 % | 0 % |

**Table 95.** Discrimination on Existing Bugs in DSpace

| | Tool O | Tool Q | Tool P | Tool N | Tool M | Tool L | Tool R |
|---|---|---|---|---|---|---|---|
| **Recall** | 83 % | 61 % | 100 % | 22 % | 6 % | 0 % | 0 % |
| **Disc. Rate** | 83 % | 33 % | 17 % | 11 % | 0 % | 0 % | 0 % |

**Table 96.** Discrimination on Injected Bugs in DSpace

| | Tool O | Tool N | Tool Q | Tool P | Tool M | Tool L | Tool R |
|---|---|---|---|---|---|---|---|
| **Recall** | 92 % | 75 % | 67 % | 100 % | 8 % | 0 % | 0 % |
| **Disc. Rate** | 92 % | 67 % | 58 % | 8 % | 0 % | 0 % | 0 % |

**Table 97.** Recall per Bug Complexity in DSpace

| Complexity | Bugs | Tool O | Tool Q | Tool P | Tool N | Tool L | Tool M | Tool R |
|---|---|---|---|---|---|---|---|---|
| **Low** | 18 | 83 % | 72 % | 100 % | 61 % | 0 % | 6 % | 0 % |
| **Medium** | 12 | 92 % | 50 % | 100 % | 17 % | 0 % | 8 % | 0 % |

**Table 98.** Discrimination per Bug Complexity in DSpace

| Complexity | Bugs | Tool O | Tool Q | Tool P | Tool N | Tool L | Tool M | Tool R |
|---|---|---|---|---|---|---|---|---|
| **Low** | 18 | 83 % | 61 % | 17 % | 56 % | 0 % | 0 % | 0 % |
| **Medium** | 12 | 92 % | 17 % | 8 % | 0 % | 0 % | 0 % | 0 % |

**Table 99.** Effect of Bug Complexity on Discrimination in DSpace

| Complexity | Bugs | Tool O | Tool Q | Tool P | Tool N | Tool L | Tool M | Tool R |
|---|---|---|---|---|---|---|---|---|
| **Low** | 18 | 0 % | -15 % | -83 % | -9 % | N/A | -100 % | N/A |
| **Medium** | 12 | 0 % | -67 % | -92 % | -100 % | N/A | -100 % | N/A |

**Table 100.** Breakdown of Bug Count per Bug Properties in DSpace

| Complexity | Existing | Injected |
|---|---|---|
| Low | 6 | 12 |
| Medium | 12 | 0 |
| High | 0 | 0 |
| Extreme | 0 | 0 |

### 5.4.2. Sakai: Semi-automatically Injected Bugs

Sakai contained 30 SQL injection bugs. The bugs were all injected and of medium complexity, limiting the number of ways we can split the results.

Table 101 offers an overview of the number of bugs found by each tool, the accuracy of the findings, and recall and discrimination rate. Excluding Tool L, which did not support SQL injection, the recall ranged from 53 % to 67 % with an average of 58 % and discrimination rate from 40 % to 67 % with an average of 55 %. Discrimination was very high for all tools, but the shortcomings described in Section 5.2.2 render this particular result inconclusive.

Table 102 shows high overlap across tools, which can be partially explained by the high recall. Tool R reported all bugs found by Tools N, Q, P and M. Tool O, on the other hand, found several bugs that were missed by other tools. Tools R and O together found all but a single bug.

**Table 101.** Recall and Discrimination on SQL Injection in Sakai

| Findings | Tool O | Tool R | Tool Q | Tool N | Tool M | Tool P | Tool L |
|---|---|---|---|---|---|---|---|
| Miss | 10 | 9 | 13 | 13 | 15 | 14 | 30 |
| Hint | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Partial | 7 | 20 | 14 | 17 | 15 | 16 | 0 |
| Alternate | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| Match | 8 | 0 | 3 | 0 | 0 | 0 | 0 |
| **TP** | 20 | 20 | 17 | 17 | 15 | 16 | 0 |
| **Disc. TP** | 20 | 20 | 17 | 17 | 13 | 12 | 0 |
| **Bugs** | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| **Recall** | 67 % | 67 % | 57 % | 57 % | 50 % | 53 % | 0 % |
| **Disc. Rate** | 67 % | 67 % | 57 % | 57 % | 43 % | 40 % | 0 % |

**Table 102.** Tool Warning Overlap on SQL Injections in Sakai

| | Tool O | Tool R | Tool N | Tool Q | Tool P | Tool M | Tool L |
|---|---|---|---|---|---|---|---|
| **Tool O** | N/A | 57 % | 71 % | 71 % | 56 % | 73 % | N/A |
| **Tool R** | 60 % | N/A | 100 % | 100 % | 100 % | 100 % | N/A |
| **Tool N** | 60 % | 81 % | N/A | 100 % | 81 % | 100 % | N/A |
| **Tool Q** | 60 % | 81 % | 100 % | N/A | 81 % | 100 % | N/A |
| **Tool P** | 45 % | 76 % | 76 % | 76 % | N/A | 73 % | N/A |
| **Tool M** | 55 % | 71 % | 88 % | 88 % | 69 % | N/A | N/A |
| **Tool L** | 0 % | 0 % | 0 % | 0 % | 0 % | 0 % | N/A |

## 6. Ockham Criteria

The Ockham criteria track used specific terms described in Section 1.4.

### 6.1. Background

In SATE V [8], the SAMATE team introduced the Ockham Sound Analysis Criteria, a track for static analyzers whose analysis is logically sound. Tools that are not "bug-finders" can satisfy the Ockham Criteria, too. A tool that reports that pieces of code are certainly bug free is welcome.

We check that tools satisfy the SATE VI Ockham Sound Analysis Criteria to show that they are reliable and worth the effort to use. Beyond that, our test material and approach should help others investigate what assurance a tool provides for their own code in their own development process.

The rest of this section gives additional background and explains changes between the previous Ockham Sound Analysis Criteria evaluation and the current one. Section 6.2 explains the Criteria in detail. It also presents the general procedure we used to evaluate a tool by the Criteria. Section 6.3 explains details of the evaluation for Astrée, Section 6.3.1, and Frama-C, Section 6.3.2. Section 6.4 lists what we found and our conclusions.

#### 6.1.1. Using Sound Static Analyzers

Our evaluation of tools against the Ockham Sound Analysis Criteria only reflects one aspect of using a sound static analyzer in a production software development process. Adding almost any tool to a software development process takes work. Even to evaluate as we have, there is a particular learning curve to effectively use static analysis tools.

To be precise, such tools use a detailed description of the actual compilation and execution environments of the software being analyzed. Is an int 32 or 64 bits on the target computer? Does the code rely on the compiler laying out memory for a struct in a certain order with no padding? Do you want warnings of unsigned short integer overflow if your code does a lot of masking and shifting, e.g., for hashes or encryption? Is the high-order bit propagated when a signed integer shifted right? The C11 standard [33] allows different definitions of main() and different

behaviors of bitwise operators. The term "implementation-defined" occurs almost 200 times in the C11 standard.

In addition, the tools are elaborate systems with many abilities. As an analogy, consider that the word "vehicle" includes bicycles, dump trucks, and buses. All have wheels, can be steered, and transport something, but their design and uses are very different. Similarly, we evaluate only a small part of the tools' capabilities for the SATE VI Ockham Criteria. Astrée [34] has sophisticated graphical user interfaces to select hundreds of options, including checking Motor Industry Software Reliability Association (MISRA) guidelines, controlling software checking, and displaying violations found in context. Frama-C [35] is an open-source suite of tools to analyze software written in C, such as code slicing, dependency analysis, and enabling proofs that code satisfies functional specifications. Another tool, not in SATE VI Ockham, Kestrel Technology's CodeHawk-C, exposes the validity requirements—proof obligations (PO)—of every code fragment reporting that each PO is satisfied, violated, or cannot be proved.

Consider that Thales defines many levels of using formal verification for software assurance [36]. These levels are Stone—adhering to the SPARK [37] programming language— then Bronze—proving variable initialization and clear data flow—then Silver, Gold, and finally Platinum—proving that software meets its fully- and formally-specified requirements. Similarly, a knowledgeable user will "tune" the use of sophisticated tools, for instance, specify depth of recursion, number of loops to unroll, and analysis options so the tool produces the most useful result. It took us three or four full days of experimenting, reading, and guidance from tool makers to get tools reporting the errors that we were interested in. Even then we do not claim that our choices were optimal for a software production environment.

### 6.1.2. Differences Between SATE V and SATE VI Ockham Exercises

In this subsection, we examine differences between the SATE V Ockham evaluation procedure and that of SATE VI Ockham. For details of the SATE V evaluation, see [9].

We only evaluated Frama-C in the SATE V Ockham Sound Analysis Criteria. For this SATE VI, we evaluated two tools: Astrée [34] and a new version of Frama-C [35] with its Evolved Value Analysis (Eva) plug-in.

#### 6.1.2.1. Known Bugs

SATE VI Ockham used Juliet Version 1.3 test cases. Juliet cases are small, synthetic programs with deliberate bugs. Juliet was originally developed by National Security Agency's Center for Assured Software. It is available from the Software Assurance Reference Dataset (SARD) [14]. It includes a list of known bugs in a *manifest* file.

#### 6.1.2.2. Determining Sites

To explain sites, we first define "weakness". A piece of code has a *weakness* when some execution could lead to a fault. In contrast, a *vulnerability* in a system could be accidentally triggered or intentionally exploited to cause a failure [9]. Every vulnerability is one or more weaknesses. A weakness is not a vulnerability if it is guarded by code or has other mitigations anywhere in the broader system. For example, suppose an analyst is considering a dozen lines of

code and sees that that piece of code has no protection from an SQL Injection attack. The code has an SQL Injection weakness. However, if the broader system context filters out any possible string with SQL Injection attacks, there is no vulnerability.

To be precise, we used the concept of *site*. A site "is a location in code where a weakness might occur." [38]. See [38] for further exposition of what constitutes a site. In the current Ockham, SATE VI, we simply checked that all buggy sites were included in warnings, that is, $B \subseteq W$, see Figure 1.

Not determining sites means that we cannot calculate that Ockham Criterion 2, explained in Section 6.2.2, is satisfied.

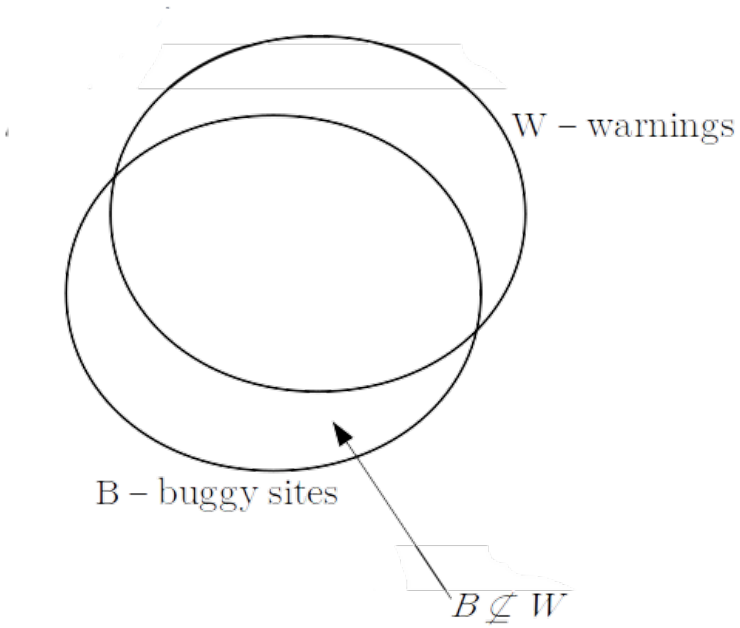


**Figure 1.** Relation between warnings reported, W, and known buggy sites, B, for SATE VI Ockham. Ockham Criterion 3 is satisfied if B⊆W.

### 6.1.2.3. Bug or Weakness Classes

The Juliet test cases are grouped by Common Weakness Enumeration (CWE) [11]. For SATE VI Ockham, we defined our own classes to be orthogonal (no overlaps) and precise. Our classes are listed and defined in Section 6.2.4.

We still had significant difficulty assigning tool warnings to these new classes. Part of the problem was understanding *exactly* what the tool warning covered or what it did not cover. Part of the problem was that our classes made distinctions that tools did not and vice versa. Our own classes were easier to use than CWEs but did not come close to being reasonable universal classes.

## 6.2. The Criteria

This section has the details of the Ockham Criteria themselves and includes explanation and discussion. Much of this section comes from Section 2 of the SATE V Ockham report [38].

The Criteria are named for William of Ockham, best known for Occam's Razor. Since the details of the Criteria will likely change in the future, the name includes a time reference: SATE VI Ockham Sound Analysis Criteria. The criteria were:

1. The tool is claimed to be sound.
2. For at least one weakness class and one test case the tool produces findings for a minimum of 75 % of appropriate sites.
3. Even one incorrect finding disqualifies a tool for this SATE.

An implicit criterion is that the tool is useful, not merely a toy.

A *finding* is a definitive report about a site, which is a specific place in code. In other words, the tool reports that the site has a specific weakness (is buggy) or that the site does not have that weakness.

No manual editing of the tool output was allowed. No automated filtering specialized to a test case or to SATE VI was allowed, either. The tool's settings and options may be selected to produce the best result, as alluded to in Section 6.1.1. Such setting should be reported.

### 6.2.1. Criterion 1: "Sound" (and "Complete") Analysis

Criterion 1 is "The tool is claimed to be sound."

We use the term *sound* to mean that every finding is correct. In other words, "Sound analysis means that the [tool] never asserts a property to be true when it is not true." ([39], FM.1.6.2). The tool need not produce a finding for every site; that is completeness.

A tool may have settings that allow unsound analysis. The tool still qualifies if it is "mostly sound, with specific, well-identified unsound choices" [40], that is, it is *soundy* as defined by Livshits, et. al. For a more detailed exposition of uses of the terms "sound" and "complete" applied to static analysis, see [38], Sec. 2.3.

### 6.2.2. Criterion 2: Tools Produce Findings for Most Sites

Criterion 2 balances usefulness with theoretical limits: the tool produces findings for a minimum of 75 % of sites.

A sound tool reports one of three findings about a site: it definitely has a certain weakness, it definitely does not have a certain weakness, or the tool cannot determine.

As explained previously, a *site* "is a location in code where a weakness might occur." ([38], Sec. 2.2).

A *finding* may be that a site is buggy or that a site does not have a particular bug. Either type of statement (or both!) is acceptable. For instance, a tool may use conservative approximations and

sometimes produce warnings about (possible) bugs at sites that are actually bug-free. If it never misses a bug, then any site without a warning is sure to be correct.

Because we did not determine sites for weakness classes, we cannot calculate that Criterion 2 is satisfied.

### 6.2.3. Criterion 3: Determining That All Findings Are Correct

Criterion 3 is "Even one incorrect finding disqualifies a tool for this SATE." This section describes the general procedure we followed to confirm that a tool satisfied Criterion 3.

Initial comparison between findings and the Juliet manifest almost always produced thousands of mismatches.

One reason for mismatches is that reasoning is based on models, assumptions, definitions, etc. (collectively, "models"). Mismatches that result from model differences do not automatically disqualify a tool. To satisfy the SATE VI Ockham Criteria, any such differences are publicly reported in the full report.

We performed the bulk of the analysis with automated scripts and custom programs. Some exclusions and special handling were built into the code. These are mentioned in relevant sections in the full report. The steps are listed below and are given in Figure 2:

1. Distill bugs from the list of known bugs in the test cases.
2. Run the tool on the test cases.
3. Extract findings from the tool results.
4. Check that all bugs are in the findings. If so, Criterion 3 is satisfied.

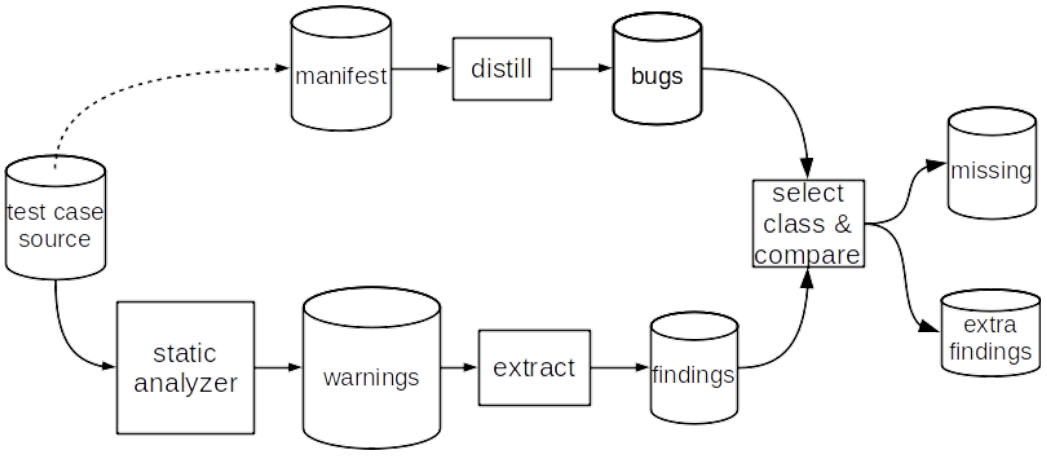


**Figure 2.** General flow to confirm that a tool satisfied the SATE VI Ockham Sound Analysis Criterion 3. Distill bugs from the known-bugs manifest. Run the tool on the test cases. Extract findings from the tool output. Compare bugs and findings.

When we distill bugs and extract findings, we encode them as classes that we created for a common representation. The classes are listed and explained in Section 6.2.4.

When a bug from the known list is missing from findings extracted from the tool result, we resolve the discrepancy. See the full report for details.

### 6.2.4. Ockham Bug Classes

We defined our own classes to organize our examination of warnings. We tried to precisely and rigorously define clear classes. The classes must not overlap. That is, a bug must be in one class or another. In many instances, our classes did not correspond well to the tool warnings.

This section briefly explains each class. Complete explanations are in the full report. We treat closely related classes or subclasses in a single subsection.

#### 6.2.4.1. ARC—Arithmetic or Conversion Fault Classes

We begin with the Arithmetic or Conversion Fault, or ARC, class. We define ARC as:

Software produces a faulty result due to conversions between primitive types, range violations, or domain violations.

We subdivided ARC into several classes: Truncate, Overflow, Underflow, Distort, and Undefined.

ARC/Truncate is when the value to be stored is too big for the destination.

ARC/Overflow is when the types match, but the result is too big for the destination.

ARC/Underflow is when the types match, but the result is too negative for the destination.

ARC/Distort is when the result is otherwise distorted.

To contrast the four preceding subclasses, truncate is when the type of the source is larger than the type of the destination. Overflow and Underflow are when the destination is at least as big as the source, but the result of an arithmetic operation will not fit in the destination type. Distort is when the destination is at least as big as the source and the result would fit, but it is changed for another reason.

ARC/Undefined is a domain violation, such as divide by zero or negative shift.

#### 6.2.4.2. ARG/Memcpy—Incorrect Argument for memcpy()

We define ARG as:

Software calls a function with incorrect arguments.

#### 6.2.4.3. BOF/Read and BOF/Write—Read or Write Outside Buffer

We define BOF as:

Software accesses through an array a memory location that is outside the boundaries of that array.

Because there are so many BOF bugs in Juliet, we divided this into BOF/Read and BOF/Write classes.

#### 6.2.4.4. DEP—Dereference Erroneous Pointer Classes

We define DEP as:

Software dereferences an invalid pointer.

The invalid pointer may be NULL, refer to memory that was freed (see MAL below), be produced by incorrect pointer arithmetic (see PAR below), or be completely arbitrary.

Because Juliet has particular cases for incorrect pointer arithmetic, we separate them into their own class, DEP/Incorrectly Computed Pointer (DEP/ICP).

#### 6.2.4.5. PAR—Pointer Arithmetic

We define PAR as:

Software produces a faulty value because of incorrect pointer arithmetic.

DEP is when an incorrect pointer is dereferenced. PAR is when the result of pointer arithmetic is used as a number or when pointers are compared incorrectly.

#### 6.2.4.6. ILP—Infinite Loop

We define ILP as:

Software never terminates execution.

#### 6.2.4.7. INI—Initialization Fault

We define INI as:

Software uses a faulty value because an entity was not properly initialized.

“Entity” includes a variable, a member of a structure or record, parts of an array, a pointer or reference, etc. “Not properly” covers both not initialized at all and not correctly initialized. Not initialized correctly includes initialized with a value that leads to a security concern.

#### 6.2.4.8. MAL—Memory Allocation and Deallocation

We define MAL as:

Software improperly allocates or deallocates memory.

#### 6.2.4.9. UCE—Unchecked Error

We define UCE as:

Software does not check, checks incorrectly, or checks but does not take action on a possible error condition.

## 6.3. SATE VI Evaluation

We evaluated several tools by the SATE VI Ockham Sound Analysis Criteria. This section has one subsection for each tool.

All of the scripts and files are available in a tar file with xz compression [41] at DOI http://dx.doi.org/10.18434/M32187 or https://nist-sate-ockham-sound-analysis-criteria-evaluationmaterial.s3.amazonaws.com/ockham-sate-VI-2020/ockhamCriteriaSATEVI data2020.tar.xz. The README is available at https://nist-sate-ockham-sound-analysiscriteria-evaluation-material.s3.amazonaws.com/ockham-sate-VI-2020/README.

### 6.3.1. Astrée

"Astrée is a static code analyzer that proves the absence of run-time errors and invalid concurrent behavior in safety-critical software written or generated in C. [...] Astrée is sound for floating-point computations and handles them precisely and safely. [...] Astrée offers powerful annotation mechanisms for supplying external knowledge and fine-tuning the analysis precision for individual loops or data structures. [...] This allows for analyses with very few or even zero false alarms." [34]

AbsInt Angewandte Informatik GmbH granted NIST an evaluation license to run Astrée for C on a Linux 64-bit platform. In June 2018 we installed $a^3$ for C 18.04 (2819232) and began analyzing the Juliet test cases.

By its own definition, Astrée claimed to be sound: "Astrée is sound — that is, if no errors are signaled, the absence of errors has been proved." [34]. This satisfies Criterion 1.

Since Astrée produced thousands of warnings, we believe it satisfies Criterion 2.

#### 6.3.1.1. Performing the Evaluation

What we refer to elsewhere in this report as warnings, Astrée refers to as alarms. We used the classes listed in Section 6.2.4 to organize our examination of Astrée alarms.

AbsInt supplied the initial wrappers and stub code, declaration and configuration (.dax) files, and Astrée Annotation Language (.aal) files. AbsInt notes that the stubs are only example implementations and should always be checked to determine what enhancements, if any, are necessary to match the libraries in use and the properties of importance.

Astrée allows users to enable or disable rules and checks individually or as groups.

#### 6.3.1.2. Common Considerations

This section explains some considerations that applied to all the classes. These are details of Astrée operation or steps we took to match Astrée alarms to our notion of classes, locations, and warnings.

In many cases the line that Astrée produces and the line in the manifest are different, but both are reasonable. See the full report for examples and resolutions.

Sometimes Astrée reports an alarm as in a utility file, not in the calling function, where it would be convenient for us. With highly-automated processing, we added some context sensitive checks.

Another consideration is that Astrée did not support C++, so we excluded all .cpp cases[12]. Astrée needs library stubs and type definitions to support Windows-specific code. Since we did not have them, we excluded _w32_ and _wchar_t_ test cases. Astrée followed C99, not C11, semantics.

The following are one subsection for each weakness class.

#### 6.3.1.3. ARC/Overflow—Arithmetic Overflow

Anomalies, Observations, and Interpretations:

Analyzing this class showed that the manifest was missing ARC/Overflow bugs for

CWE680 cases.

Results: 3666 buggy sites.

#### 6.3.1.4. ARC/Underflow—Arithmetic Underflow

Anomalies, Observations, and Interpretations:

The vast majority of the unmatched findings (4412 of 4792) are uses of the RAND32 macro. The rest (380) are code with something-1, which underflows for some types.

We found poor code in “good” functions in CWE191. We also found that “bad” functions in CWE680 had extraneous errors.

Results: 2622 buggy sites.

#### 6.3.1.5. ARC/Undefined—Divide by Zero

Results: 684 buggy sites.

#### 6.3.1.6. ARC/Distort—Result Distortion

Results: 1824 buggy sites.

#### 6.3.1.7. ARC/Truncate—Result Truncation

Anomalies, Observations, and Interpretations:

We had many ARC/Truncate alarms from cases under CWE367 TOC TOU and CWE404 Improper Resource Shutdown. We posit that open() is modeled as returning a “signed long long”, which would be truncated (ARC/Truncate) to fit in int. However, [42] says open() returns int, so the Juliet code is not buggy. Many alarms for cases under CWE369 Divide by Zero are

[12] C++ support has been added and is scheduled to be available to all users with the 20.04 release.

valid, but not interesting to us. We did not have a .dax files for CWE681 Incorrect Conversion Between Numeric Types, so we did not run Astrée on those cases.

Initial comparison showed 26 buggy sites not in the findings.

Results: 684 buggy sites.

#### 6.3.1.8. ARG/Memcpy—Incorrect Argument for memcpy()

Results: 18 buggy sites.

#### 6.3.1.9. BOF/Read—Read Outside Buffer

Anomalies, Observations, and Interpretations:

Reconciling (the lack of) Astrée alarms found 72 "fossil" errors in the manifest. The Juliet code had been corrected in Juliet 1.3, see [15], Sec. 2.3, but the errors had not been removed from manifest.

We found 1450 unmatched alarms in code calling printLine(). These are valid, but we did not add these to the manifest since these are caused by the out-of-bounds access, which is already listed.

Results: 1188 buggy sites.

#### 6.3.1.10. BOF/Write—Write Outside Buffer

Anomalies, Observations, and Interpretations:

Astrée produced "invalid dereference" alarms for the call to strlen().

The manifest has warnings for the memcpy() in cases such as SARD case 231444. We believe that Astrée gives no alarm for these because the data copied stay within the structure, although they go outside the buffer. We filter out these unmatched manifest warnings.

Astrée models allopca() as possibly returning NULL. Because of Astrée's model, it gives an alarm of possible NULL pointer dereference (DEP) and stops analysis. We removed these alloca() cases from the buggy set to match.

Analyzing Astrée alarms led us to discover that the manifest had incorrect locations for many CWE665 Improper Initialization cases. We corrected those in the manifest.

Results: 5192 buggy sites.

#### 6.3.1.11. DEP—Dereference Erroneous Pointer

Anomalies, Observations, and Interpretations:

Some cases under CWE476 NULL Pointer Dereference are intended to find out if a tool reports a useless NULL check after a dereference. Since Astrée behaves reasonably, we skip these cases.

As with BOF/Write, Section 6.3.1.10, the model for alloca() led to many unmatched alarms, which we consider spurious.

Results: 1166 buggy sites.

#### 6.3.1.12. DEP/ICP—Incorrect Pointer Arithmetic

Anomalies, Observations, and Interpretations:

It appears that Astrée does not check for CWE188 at all. We, therefore, did not examine any CWE188 cases.

Results: 52 buggy sites.

#### 6.3.1.13. PAR—Pointer Arithmetic

Results: 18 buggy sites.

#### 6.3.1.14. ILP—Infinite Loop

Anomalies, Observations, and Interpretations:

We found a difference in reporting locations in some cases. In these cases the extractor patches the extracted alarm to correspond with the manifest.

Results: 6 buggy sites.

#### 6.3.1.15. INI—Initialization Fault

Anomalies, Observations, and Interpretations:

We found a difference in reporting locations from some code.

Results: 776 buggy sites.

#### 6.3.1.16. MAL—Memory Deallocation

Anomalies, Observations, and Interpretations:

Evaluating Astrée lead us to discover that the manifest was wrong for some code. Resolving the discrepancy also showed that the manifest was wrong for other code.

Results: 536 buggy sites.

#### 6.3.1.17. UCE—Unchecked Error

Anomalies, Observations, and Interpretations:

Astrée did not check the handling of error returns from functions like fread(), putchar(), and scanf(), which are under CWE253. We, therefore, did not check any CWE253 cases. We did not run Astrée on any cases under CWE390 or CWE391.

Results: 540 buggy sites.

#### 6.3.1.18. Summary of Evaluation

Astrée reported all buggy sites and satisfied SATE VI Ockham Criteria 3.

Alarms from Astrée led us to find and fix thousands of mistakes in the Juliet known-bug list, manifest.xml.

Because Astrée analyzes code very precisely and we checked meticulously, details of modeling that otherwise would be inconsequential showed up and were resolved.

In the test cases of the 28 sets used from the Juliet 1.3 C test suite, we considered 18 954 buggy sites for Astrée. We processed a total of 36 316 Astrée alarms.

**Astrée satisfied the SATE VI Ockham Sound Analysis Criteria.**

### 6.3.2. Frama-C

"Frama-C is a suite of tools dedicated to the analysis of the source code of software written in C." [35] "Frama-C allows [you] to verify that the source code complies with a provided formal specification. Functional specifications can be written in a dedicated language, ANSI/ISO C Specification Language (ACSL). The specifications can be partial, concentrating on one aspect of the analyzed program at a time." [43] It is free software licensed under the GNU Lesser General Public License (LGPL) v2.1 license[13].

We began evaluation in June 2019 and used Frama-C 'Argon' 18.0 Version.

By its own definition, Frama-C claimed to be sound: "it aims at being *correct*, that is, never to remain silent for a location in the source code where an error can happen at run-time" [35]. This satisfies Criterion 1.

Since Frama-C with the Evolved Value Analysis (Eva) plug-in produced thousands of warnings, we believe it would satisfy Criterion 2.

#### 6.3.2.1. Performing the Evaluation

To produce all the warnings we were interested in, we eventually ran Frama-C and Eva on each test case with four different sets of options. See the full report for details.

We used the classes listed in Section 6.2.4 to organize our examination of Frama-C/Eva warnings. We examined warnings generally class by class.

#### 6.3.2.2. Common Considerations

This section explains some considerations that applied to all the classes.

Frama-C halts analysis when it reaches an invalid state. These are often reported as nonterminating states. Undefined code leads to a state where anything can happen. Following that, no sound analysis makes sense, so Frama-C performs no further analysis.

Because of our misunderstanding, we unnecessarily skipped wchar_t cases in many classes.

[13] http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

Following is one subsection for each weakness class. See the full report for more.

#### 6.3.2.3. ARC/Overflow—Arithmetic Overflow

Anomalies, Observations, and Interpretations:

The good functions in 38 test cases named CWE190_Integer_Overflow__unsigned_ int_max_square_ are written wrong. We excused these test cases from analysis.

Results: 3628 buggy sites.

#### 6.3.2.4. ARC/Underflow—Arithmetic Underflow

Results: 2622 buggy sites.

#### 6.3.2.5. ARC/Undefined—Divide by Zero

Results: 684 buggy sites.

#### 6.3.2.6. ARC/Distort—Result Distortion

Results: 1824 buggy sites.

#### 6.3.2.7. ARC/Truncate—Result Truncation

Results: 684 buggy sites.

#### 6.3.2.8. ARG/Memcpy—Incorrect Argument for memcpy()

Results: 36 buggy sites.

#### 6.3.2.9. BOF/Read—Read Outside Buffer

Results: 1358 buggy sites.

#### 6.3.2.10. BOF/Write—Write Outside Buffer

Anomalies, Observations, and Interpretations:

The manifest has extra warnings for 74 CWE665 test cases. Frama-C warns about the initialization problem but does not give any BOF/Write warning. That is reasonable behavior. For automated checking, we have the analysis change all 74 of the warnings from Frama-C for these test cases.

The manifest incorrectly has different lines for 26 CWE122 test cases. For no particular reason, instead of changing the manifest lines or having the Frama-C extractor patch the warnings, we created a file with the 26 excused warnings. The evaluation script includes this file when checking for mismatches.

Test cases under CWE242 purposely use gets(), even though gets() cannot be used safely. Since the library we used for Frama-C does not correctly model gets(), we excluded all CWE242 from the analysis.

Results: 5156 buggy sites.

#### 6.3.2.11. DEP—Dereference Erroneous Pointer

In several instances, Frama-C produces the same, perfectly useful warning for classes of bugs that we distinguish. We used the names of the test cases to distinguish them and encode them into our comparison classes.

Anomalies, Observations, and Interpretations:

In 114 cases under CWE690, the manifest lists two flaws. As explained in Section 6.3.2.2, Frama-C warns about the first assignment when data are NULL, which is undefined behavior, then stops further analysis. We accommodated this by adding to the file of excused warnings.

Results: 1204 buggy sites.

#### 6.3.2.12. DEP/ICP—Incorrect Pointer Arithmetic

Anomalies, Observations, and Interpretations:

Frama-C did not warn about cases of implementation-dependent code under CWE188. We skipped analysis of CWE188 test cases due to time constraints.

Results: 34 buggy sites.

#### 6.3.2.13. PAR—Pointer Arithmetic

Results: 36 buggy sites.

#### 6.3.2.14. INI—Initialization Fault

Anomalies, Observations, and Interpretations:

In 96 cases under CWE457, the manifest lists two flaws. After warning, Frama-C enters a “non-terminating state” and stops further analysis, as explained in Section 6.3.2.2. We accommodated this in automated analysis by adding to the file of excused warnings.

Results: 776 buggy sites.

#### 6.3.2.15. MAL—Memory Deallocation

Results: 784 buggy sites.

#### 6.3.2.16. Summary of Evaluation

Warnings from Frama-C led us to discover 38 test cases with incorrect "good" code in CWE190 (Section 6.3.2.3).

Frama-C always warns about buggy sites but may warn about sites without bugs.

In many instances, there are minor differences between the location of flaws given in the manifest and locations reported by Frama-C (Section 6.3.2.10).

Frama-C lacks a sufficiently detailed model for the function gets(). The function gets() is inherently dangerous and should not be used anyway (Section 6.3.2.10).

In the test cases of the 24 sets used from the Juliet 1.3 C test suite, we considered 18 826 buggy sites for Frama-C. We processed a total of 42 056 Frama-C warnings.

**Frama-C with Eva satisfied the SATE VI Ockham Sound Analysis Criteria.**

## 6.4. Observations and Conclusions

### 6.4.1. New Errors Found in Juliet 1.3 and its Manifest

While evaluating Frama-C, Astrée, and another tool, we found several previously unknown systematic problems in Juliet 1.3 and thousands of problems in its manifest of known errors. For a summary, see the full report.

### 6.4.2. Weakness Classes

Although the SATE VI Ockham Sound Analysis Criteria used the term "weakness classes", no classes are specified. For evaluation we defined our own classes, see Section 6.2.4. We tried to be clear and logical in our choice of classes, but they still did not always correspond well to the warnings that the tools used.

It may have been easier to evaluate the warnings as produced by the tools.

Without understanding the exact definition of the class of weakness the tool was considering, we could not decide whether a known bug corresponded to a tool warning: perhaps there was just a difference in choice of which line number to report. If the tool was intended to report that class, then a missing warning indicated an error. If the tool in actuality is not considering a particular class of warning, such as integer overflow of types smaller than int, then a known bug should be ignored.

In retrospect, there is little need to have DEP/Incorrectly Computed faults as a separate class. It has less than 5 % of all DEP cases.

### 6.4.3. Summary

We processed a total of 78 372 warnings over 29 sets from the Juliet 1.3 C test suite.

Both Astrée and Frama-C with Eva satisfied the SATE VI Ockham Sound Analysis Criteria.

# 7. Workshop Outcome

On September 19, 2019, we welcomed participating tool makers, tool users, and security researchers to the SATE VI Workshop [45]. While the organizers presented initial results, tool makers shared their experiences in participating in SATE.

Several tool makers noted that they used SATE feedback to improve their tools. There was a general agreement for doing SATE VII in the future.

Tool makers found several errors in the SATE test cases. For example, Andre Maroneze and Julien Signoles reported finding unintentional weaknesses in the Juliet test suite. Some tool makers also noted that tool outputs may not always map to expected tool outputs, e.g., due to differences in how sinks and sources are defined.

Eric Schulte presented the Bug Injector, which was used in SATE VI. In particular, he described the injection methodology, bug templates, and evaluation results from [29].

Workshop participants supported using bug injection for SATE VII, but with a few conditions. First, the types of injected bugs should be prioritized based on community input. Second, we must be careful about introducing test case bias and favoring some tools. Finally, the base program needs to be analyzed prior to bug injection.

Paul Anderson noted that the bugs injected in SQLite were based on just five different scions and recommended using more varied bugs in the future. He emphasized the importance of tool customization and stated that small examples of real programs are far better than micro tests. Paul Anderson also suggested varying the application domain of the test cases, for example, using an embedded application.

Igor Matlin stated that SQLite and DSpace have the optimal size for SATE, considering the limited resources. He observed that tools need to be customized and also expressed hope that SATE can automate matching tool outputs to expected outputs.

It was proposed to choose a popular project on GitHub and analyze its history to mine for CVEs. This is like our use of CVE-selected test cases in previous SATEs. Some participants proposed using publicly available bug collections such as BugZoo [46] and ManyBugs [47].

Workshop participants thought that the choice of programming languages for SATE should depend on the language popularity, number of static analysis tools for the language, and other parameters. Matt Rhodes noted that these parameters are fluid, as the Internet of Things may change the language relevance.

Most workshop participants supported the introduction of JavaScript test cases in the next SATE, although some participants noted that additional languages can be too much of a burden for the organizers.

Inclusion of other types of tools in SATE was opposed by most participants to avoid diluting the effort.

All tool makers present at the workshop said that their tools either already supported SARIF [44] or will support it soon. Accordingly, it was agreed that SATE VII should use SARIF only.

A process consisting of two rounds was proposed for SATE VII. In the first round, to iron out any issues with the test cases, run the tools out of the box, that is, without customization. In the second round, run the tools with customization, for best results.

Matt Rhodes stated that there is a potential synergy between sound and unsound tools. Sound tool can clean up unsound tool output, e.g., by reducing false positives. Unsound tool can add details to sound tool findings, e.g., by corroborating exploitability. Matt Rhodes also emphasized that static analysis does not eliminate the need for software testing.

The workshop information and presentations are available in [45].

## 8. Conclusion

In SATE VI, we used existing and injected bugs to measure tool effectiveness (Sec. 2). Existing bugs were mined out of bug tracker reports and the CVE/NVD database. To complement the limited number of bugs thus obtained, we injected additional bugs using a combination of automated tools and manual analysis. The bug injection process encountered many difficulties, such as bug shadowing (Sec. 2.3.2), bug tracing laboriousness (Sec. 2.3.3) or tooling complications leading to significant, but limited shortcomings (Sec. 5.2) and delays in running SATE VI and publishing this report. This undertaking taught us many lessons that will help better position ourselves for a potential SATE VII.

The SATE VI results (Sec. 4 and 5) show considerable variability across tools, test cases, bug classes and bug complexity. Some tools did not support some of the bug classes at all.

Overall, low code complexity was the prominent driver for tool success in identifying bugs. Test cases of the C Track were on average more complex than test cases of the Java Track, and indeed recall and discrimination rates were lower for the C Track than the Java Track. On the C Track, Wireshark had a much higher code complexity than SQLite. Recall and discrimination rate for buffer errors on Wireshark averaged 10 % and 7 %, respectively (Sec. 5.3.1.2), while on SQLite they averaged a higher 34 % and 25 %, respectively (Sec. 5.3.2).

The same observation was made on individual bugs' complexity throughout the report. Across all languages and code bases, tools found bugs with lower complexity more readily than bugs with higher complexity. Table 72 illustrates this effect on CGC. Code complexity remained the curse of static analysis.

Calculation of tool warning overlap showed tool results ranging from highly correlated to highly independent. This metric is particularly useful when working with a set of tools. Common findings by independent tools can lead to increased confidence in their accuracy. Besides, using multiple tools can increase overall recall.

Regardless of the test case, injected bugs were not found by tools at the same rate as existing bugs, implying that the quality of injected bugs needs to improve. Ideally, injected bugs should be indiscernible from existing ones.

We encourage the reader to start with the concise summary of all results in Section 4, which includes references to Section 5 for further details.

The Ockham Criteria track [16] is briefly discussed in Section 6.

SATE VI showed that static analysis is a useful technique to find real security bugs in large code bases. Tools are not perfect, but the right set of tools, used properly, can help increase code quality and security. A potential user should test a tool or a set of tools on his own code base before using them in production. The metrics presented in SATE VI are suitable to assess tool

fitness for such a use case. For further gains, static analysis tools can also be complemented by other types of tools, such as dynamic analyzers (e.g., fuzzers).

## 8.1. Future Work

SATE VI's foray in the realm of bug injection taught us valuable lessons. The comparison between injected and existing bugs clearly showed a need to improve injected bug quality if the avenue is to be further pursued. Armed with this new wisdom, we may be able to improve our injection process to create better test cases for a potential SATE VII.

The advent of large language models (LLM) could prove an interesting complement to bug injection. LLMs can generate small, but more realistic programs than template-generated test suites like Juliet [15]. They can also generate functionally-identical alternatives of the same program, offering a promising way to explore code complexity variations in the context of static analysis tool evaluation.

## References

[1] Wheeler DA, Henninger AE (2016) State-of-the-art resources (SOAR) for software vulnerability detection, test, and evaluation 2016 (Institute for Defense Analyses), P-8005. https://apps.dtic.mil/sti/pdfs/AD1106086.pdf. (Accessed 13 August 2022)

[2] Black, P., Okun, V. and Guttman, B. (2021), Guidelines on Minimum Standards for Developer Verification of Software, NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.IR.8397, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=933350 (Accessed 13 August, 2022)

[3] SAMATE, Source code security analyzers (SAMATE list of static analysis tools). https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers (Accessed 13 August 2022)

[4] Okun, Vadim, Romain Gaucher, and Paul E. Black, editors, "Static Analysis Tool Exposition (SATE) 2008", NIST Special Publication 500-279, June 2009, https://samate.nist.gov/docs/NIST_Special_Publication_500-279.pdf

[5] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, "The Second Static Analysis Tool Exposition (SATE) 2009", NIST Special Publication 500-287, June 2010, https://samate.nist.gov/docs/NIST_Special_Publication_500-287.pdf

[6] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, editors, "Report on the Third Static Analysis Tool Exposition (SATE) 2010", NIST Special Publication 500-283, October 2011, https://dx.doi.org/10.6028/NIST.SP.500-283

[7] Okun, Vadim, Aurelien Delaitre, and Paul E. Black, "Report on the Static Analysis Tool Exposition (SATE) IV", NIST Special Publication 500-297, January 2013, https://dx.doi.org/10.6028/NIST.SP.500-297

[8] Delaitre, Aurelien, Bertrand Stivalet, Paul E. Black, Vadim Okun, Terry S. Cohen, and Athos Ribeiro, (2018), SATE V Report: Ten Years of Static Analysis Tool Expositions, NIST Special Publication 500-326, National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.SP.500-326 (Accessed August 13, 2022)

[9] Stoneburner, G., Hayden, C., Feringa, A. (2004, June) "Engineering principles for information technology security (a baseline for achieving security), revision A: recommendations of the National Institute of Standards and Technology", National Institute of Standards and Technology Special Publication 800-27 Rev A, https://dx.doi.org/10.6028/NIST.SP.800-27rA

[10] Center for Assured Software, U.S. National Security Agency, "CAS Static Analysis Tool Study – Methodology", December 2011, http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf

[11] MITRE, "Common weakness enumeration (CWE)," https://cwe.mitre.org

[12] Bojanova, I., Galhardo, C. and Moshtari, S. (2021), Input/Output Check Bugs Taxonomy: Injection Errors in Spotlight, 2021 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE), Wuhan, CN, [online], https://doi.org/10.1109/ISSREW53611.2021.00052, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=933193

[13] Kratkiewicz, K., and Lippmann, R., “Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools”, Workshop on the Evaluation of Software Defect Tools, 2005.
[14] SAMATE, “Software Assurance Reference Dataset (SARD),” https://samate.nist.gov/SARD/
[15] Black, Paul E., “Juliet 1.3 Test Suite: Changes From 1.2”, June 2018, NIST Technical Note (TN) 1995, https://dx.doi.org/10.6028/NIST.TN.1995
[16] Paul E. Black and Kanwardeep Singh Walia, SATE VI Ockham Sound Analysis Criteria, April 2020, NIST Internal Report (IR) 8304, DOI 10.6028/NIST.IR.8304
[17] Rutar, Nick, Christian B. Almazan, and Jeffrey. S. Foster, “A Comparison of Bug Finding Tools for Java”, 15th IEEE Int. Symp. on Software Reliability Eng. (ISSRE’04), France, November 2004, http://dx.doi.org/10.1109/ISSRE.2004.1
[18] Zitser, M., Lippmann, R., Leek, T., “Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code”, IGSOGT Software Engineering Notes, 29(6):97-106, ACM Press, New York (2004), http://dx.doi.org/10.1145/1041685.1029911
[19] Li, Z., Zou, D., Xu, S, Jin, H., Qi, H., & Hu, J. (2016). VulPecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 201-213. https://dx.doi.org/10.1145/2991079.2991102
[20] MITRE, "Common vulnerabilities and exposures (CVE)," https://cve.mitre.org/
[21] De Oliveira, C. and Boland, F. (2015). Real world software assurance test suite: STONESOUP (Presentation). IEEE 27th Software Technology Conference (STC’2015) October 12-15, 2015.
[22] De Oliveira, C. D., Fong, E., and Black, P. E. (2017, February). Impact of code complexity on software analysis. NISTIR 8165. https://dx.doi.org/10.6028/NIST.IR.8165.
[23] Pewny, Jannik, and Thorsten Holz, EvilCoder: automated bug insertion. In Proceedings of the 32nd Annual Conference on Computer Security Applications (2016), ACM, pp. 214–225.
[24] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," 2014 IEEE Symposium on Security and Privacy, 2014, pp. 590-604.
[25] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, Ryan Whelan. LAVA: Large-scale Automated Vulnerability Addition. Proceedings of the IEEE Symposium on Security and Privacy (Oakland), May 2016.
[26] Patrick Hulin, Andy Davis, Rahul Sridhar, Andrew Fasano, Cody Gallagher, Aaron Sedlacek, Tim Leek, and Brendan Dolan-Gavitt. AutoCTF: Creating Diverse Pwnables via Automated Bug Injection. USENIX Workshop on Offensive Technologies (WOOT), August 2017.
[27] Awanish Pandey, Yu Hu, Brendan Dolan-Gavitt, and Subhajit Roy. Realistic Bug Synthesis for Testing Bug-Finding Tools. ESEC/FSE 2018.
[28] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer, 11(4):34-41, April 1978.
[29] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords, Shih An Pan, Julien Henry, David Melski, and Eric Schulte, "Automated Customized Bug-Benchmark Generation," 2019 19th International Working Conference on Source Code Analysis and

Manipulation (SCAM), Cleveland, OH, USA, 2019, pp. 103-114, doi: 10.1109/SCAM.2019.00020.
[30] Text REtrieval Conference (TREC), https://trec.nist.gov/ (Accessed 30 September 2022).
[31] Michael Ogata, "Static Analysis Tool Exposition (SATE) VI: Mobile Track Report", NIST Internal Report, https://doi.org/10.6028/NIST.IR.8462.
[32] SAMATE, SATE VI: Classic Track, https://www.nist.gov/itl/ssd/software-quality-group/sate-vi-classic-track (Accessed 2 October 2022).
[33] ISO/IEC 9899:2011 programming languages - C, Committee Draft — April 12, 2011 N1570 (The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) Joint Technical Committee JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces, Working Group WG 14 - C), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.
[34] Fast and sound runtime error analysis, 2019, https://www.absint.com/Astrée/ (Accessed 26 August 2019).
[35] What is Frama-C. Available at http://frama-c.com/what is.html.
[36] AdaCore and Thales (2018) Implementation guidance for the adoption of SPARK, https://www.adacore.com/uploads/books/pdf/ePDF-ImplementationGuidanceSPARK.pdf.
[37] AdaCore, Ltd AU (2019) SPARK 2014 user's guide, Available at http://docs.adacore.com/spark2014-docs/html/ug/. (Accessed 21 February 2020)
[38] Black PE, Ribeiro A (2017) SATE V Ockham sound analysis criteria (National Institute of Standards and Technology), IR 8113, https://doi.org/10.6028/NIST.IR.8113.
[39] RTCA (2011) Formal Methods Supplement to DO-178C and DO-178A. DO-333.
[40] Livshits B, et. al., *In Defense of Soundiness: A Manifesto*, CACM, 58(2):44-46, February 2015, doi:10.1145/2644805, http://yanniss.github.io/Soundiness-CACM.pdf.
[41] XZ Utils, http://tukaani.org/xz/.
[42] Open(2). Accessed 30 August 2019. Available at http://man7.org/linux/ man-pages/man2/open.2.html.
[43] Proving formal properties for critical software, 2019, http://frama-c.com/features.html.
[44] OASIS Static Analysis Results Interchange Format (SARIF) TC, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif (Accessed 2 October 2022).
[45] SAMATE, Static Analysis Tool Exposition (SATE) VI Workshop, Co-located with the Software and Supply Chain Assurance (SSCA) Forum, in McLean, VA, 19 September, 2019, https://www.nist.gov/itl/ssd/software-quality-group/static-analysis-tool-exposition-sate-vi-workshop.
[46] C. Timperley, S. Stepney and C. Goues, "Poster: BugZoo – A Platform for Studying Software Bugs," in 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Gothenburg, Sweden, 2018 pp. 446-447.
[47] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer, The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs, IEEE Transactions on Software Engineering (TSE), vol. 41, no. 12, December 2015, pp. 1236-1256, DOI: 10.1109/TSE.2015.2454513.