2015 IEEE/ACM 1st International Workshop on Complex faUlts and Failures in LargE Software Systems

Evaluating Bug Finders

Test and Measurement of Static Code Analyzers

Aurelien Delaitre
Dept. of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV 26506, USA
aurelien.delaitre@nist.gov

Bertrand Stivalet, Elizabeth Fong, Vadim Okun
Software and Systems Division, IT Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{bertrand.stivalet,efong,vadim.okun}@nist.gov

***Abstract*—Software static analysis is one of many options for finding bugs in software. Like compilers, static analyzers take a program as input. This paper covers tools that examine source codewithout executing itand output bug reports. Static analysis is a complex and generally undecidable problem. Most tools resort to approximation to overcome these obstacles and it sometimes leads to incorrect results. Therefore, tool effectiveness needs to be evaluated. Several characteristics of the tools should be examined. First, what types of bugs can they find? Second, what proportion of bugs do they report? Third, what percentage of findings is correct? These questions can be answered by one or more metrics. But to calculate these, we need test cases having certain characteristics: statistical significance, ground truth, and relevance. Test cases with all three attributes are out of reach, but we can use combinations of only two to calculate the metrics.**

The results in this paper were collected during Static Analysis Tool Exposition (SATE) V, where participants ran 14 static analyzers on the test sets we provided and submitted their reports to us for analysis. Tools had considerably different support for most bug classes. Some tools discovered significantly more bugs than others or generated mostly accurate warnings, while others reported wrong findings more frequently. Using the metrics, an evaluator can compare candidates and select the tool that aligns best with his or her objectives. In addition, our results confirm that the bugs most commonly found by tools are among the most common and important bugs in software. We also observed that code complexity is a major hindrance for static analyzers and detailed which code constructs tools handle well and which impede their analysis.

***Index Terms*—software faults; software assurance; static analysis tools; software vulnerability**

DISCLAIMER

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.

I. INTRODUCTION

Today's large software systems contain many defects[1] that often lead to costly failures and security breaches. Multiple techniques and tools, including testing and static analysis, should be used to reduce the number of defects and improve software assurance [1]. In this paper, we focus on static analysis tools that find security defects in source code. Like compilers, static analysis tools take a program as input. They then examine the code without executing it and produce bug reports as output. Many static analysis tools are currently available, both commercial and open source[2].

A. Overview of Static Analysis Tool Exposition

The National Institute of Standards and Technology (NIST) SAMATE project [2] conducted 5 Static Analysis Tool Expositions (SATEs) to advance research in static analysis tools [3]–[7]. The SATE process is as follows: we (the NIST researchers) provide a set of test cases to the participating tool makers. The tool makers run their tools on the test cases and return the tool outputs to us for analysis. We apply several analysis methods to the tool outputs. This culminates with a workshop organized as a forum for participants to share their findings and experiences.

The first SATE was based on production open source programs as test cases, and we added other types of tests as available and as we perceived a need. Since 2008, SATE accumulated massive amounts of data on static analysis [8].

B. Related Work

Many researchers have studied static analysis tools and collected test sets. Diaz and Bermejo [9] performed an assessment of tools by selecting 9 static analysis tools and executing them against SAMATEs Software Assurance Reference Dataset (SARD) [10] test suites 45 and 46.

Kupsch and Miller [11] evaluated the effectiveness of static analysis tools by comparing their results with the results of an in-depth manual vulnerability assessment. Of the vulnerabilities found by manual assessment, the tools found simple implementation bugs, but did not find any of the vulnerabilities requiring a deep understanding of the code or design.

The U.S. National Security Agencys Center for Assured Software [12] ran 9 tools on about 60 000 synthetic test cases covering 177 Common Weakness Enumeration (CWE) IDs

[1]In this paper, we use the terms defect, bug, and weakness interchangeably.

[2]http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

978-1-4673-7034-9 2015

DOI 10.1109/COUFLESS.2015.10

[13] and found that static analysis tools differed significantly in precision and recall. Also, tools precision and recall ordering varied for different weaknesses. One of the conclusions in [12] was that sophisticated use of multiple tools would increase the rate of finding weaknesses and decrease the false-positive rate. The Juliet 1.0 test cases, used in SATE IV, are derived, with minor changes, from the set analyzed in [12].

C. Static Analysis Ttools

While static analysis can be applied to different artifacts, including requirements or design, our paper is focused on tools that take a program as input and produce warnings as output. The warnings include a weakness name, a code location or trace, and other useful information, such as an explanation of the weakness and suggested mitigations. Many important aspects of static analysis tools, including defect tracking, user interface, and integration into the development environment, are out of the scope of this paper.

Multiple static analysis tools generally do not find the same weaknesses [14]. This is due to the differences in design philosophies, specialization, scope, and technical details. Tools differ in the programming languages that they process and types of weaknesses that they look for. They also differ in the type of analysis performed, ranging from simple pattern matching to sophisticated control and data flow analysis capabilities.

Static analysis is a generally undecidable problem. This may lead to false-positives, i.e., reporting a weakness for correct code, and false-negatives, i.e., missing a weakness. Therefore, tool effectiveness needs to be evaluated.

This paper will describe our methodology, including metrics and test cases, for evaluating static analysis tools and present our findings from applying this methodology to a portion of the SATE V test suite.

II. Measuring of the Effectiveness of Tools

The science of measuring software quality and security is at an early stage of development. There is no commonly accepted formula that can define the state or behavior of software. Black proposed some concepts that may be a basis for bug metrics [15]. Despite the lack of foundational knowledge, using the methodology described in this paper helps better understand and compare tool effectiveness.

A. Evaluation Metrics

Over the five SATEs, we gradually clarified the questions that would be useful and the metrics addressing those questions.

The first question often asked is what types of weaknesses a tool can find. Different defects require different detection schemes, and tools implement only a subset of these checkers. Some weaknesses are even impossible for tools to find, e.g. design weaknesses, as they require extra context outside of the tool's scope. Coverage measures the types of weaknesses detected by tools.

Since most tools use heuristics to detect weaknesses, they may miss some defects. Sometimes the complexity of the target software is such that tools have to make approximations in order to scale. This high complexity can cause a tool to miss defects whereas the same weaknesses would have been found in a simpler construct. It is then essential to measure the proportion of defects tools report. Recall is the number of correct findings compared to the total number of defects present in the code.

Due to the same approximations, tools sometimes also falsely report bugs. False-positives hinder remediation work and even detection, as annoyed assessors may turn off noisy rules. The true-positive rate, called precision, is the proportion of correct warnings to the total number of warnings produced by a tool. It gives a sense of the trustworthiness of tool findings.

A complementary measurement is discrimination. It reflects whether a tool can detect a weakness when there is one, but remain silent when a similar code construct is used safely. For example, a tool reporting all occurrences of calls to a dangerous function, whether it is used correctly or not, could still achieve a decent score if solely based on recall and precision. Discrimination helps differentiate basic tools from smarter ones that can determine if a code construct is safe or not, even if it contains potentially hazardous elements. Discrimination was introduced in [16] section 2.3.2.

Lastly, we want to measure the overlap, i.e. the proportion of weaknesses found by more than one tool. This metric helps determine to what extent tool outputs show statistical independence. A set of tools having a large overlap may be used to increase confidence that reported weaknesses are true-positives. On the other hand, independent tools can increase the total number of defects found, or recall.

Table I summarizes the questions one may ask and the metrics answering them.

TABLE I
Questions and Metrics

Question	Metrics
What proportion of defects can a tool find?	Recall and coverage
How noisy is a tool?	Precision and discrimination
How similar are unrelated tools?	Overlap

B. Design of Test Cases

Over the past SATEs, we distilled three test case characteristics required to calculate these metrics: statistical *significance*, *ground truth* and *relevance* [8]. The first is obtained through the size and diversity of the code base. A small test program will have a limited amount and diversity of defects. If a tool overlooks a weakness that is unique in such a small test case, it will rank poorly even if it is usually very efficient at finding this type of weakness. Therefore, we need test cases large enough to contain many occurrences of the same defect types and possibly wide weakness-type diversity.

Ideally, we also need to know where all defects are located in our test cases, a.k.a. ground truth. It provides much higher assurance and makes assessing tool warnings easier:

by subtraction, we can determine the number of overlooked weaknesses.

But the test cases must also be representative of real source code. Ultimately, static analysis tools run on production software, and our test cases should be close to that used in industry. Real software tends to be more complex than crafted test cases so a tool performing well on the latter might perform poorly on the former. Hence the metrics would be meaningless in a production context if the test cases lack this relevance.

In summary, the perfect test cases are a set of large production software, developed according to typical industry practices and whose defects are all identified. Unfortunately, such cases do not exist and creating them would consume immense resources. However, test cases exhibiting any two of the three characteristics are readily available.

Actual production software is relevant by definition. If large enough, it also provides statistical significance but not ground-truth. We could examine publicly reported vulnerabilities in production software (Common Vulnerabilities and Exposures, abbreviated as CVE) [17] but these lack in number, so we trade off statistical significance for ground truth. The last option consists of generating a large number of small programs containing planted vulnerabilities, so as to have ground truth while gaining statistical significance. Unfortunately, these synthetic tests are not representative of real code.

Despite the absence of "perfect" test cases, these three types of test cases allow us to measure all aforementioned metrics. Coverage can be determined by using production software, but these test cases can lack in variety of vulnerabilities. However, synthetic test cases contain, by design, an extensive set of vulnerability types. Recallthe rate of discovered defectscan be calculated from CVEs, but as they usually lack in number, synthetic test cases are preferred. Precisionthe proportion of correct findingscan be established over real software as well as on synthetic test cases. Discrimination, or "smartness", can be determined by running tools on software containing CVEs and on a later version of the same program correcting the problem. If a tool reports a defect only in the vulnerable version, it discriminated properly. Alternatively, synthetic test cases come in pairs, one containing a weakness and its counterpart the remediation. Running the tools on both easily establishes whether these differentiate good from bad code. Lastly, overlap can be calculated on all test sets by having several tools analyze the same test cases and comparing their findings.

Table II summarizes the types of cases one can use to calculate the metrics shown in Table I. Label *Applicable* indicates that the metric can be computed, *Limited* states that there are some limitations with the calculation, and *N/A* (not applicable) means that metric computation is not possible.

While it seems that synthetic test cases are ideal, one must keep in mind that they may not be representative of real production code. Therefore, the derived metrics might not accurately describe the behavior of tools in real conditions.

Once the tester has determined the metrics of interest, he or she can deduce the types of test case he or she needs and build the test set. Most users will run the tools on their own code base. In this case, CVEs can be replaced by previously identified bugs. Otherwise, the following aspects should be considered for choosing a solid "production" test case:

TABLE II
METRICS TO TEST CASE TYPE MAP

	Production Software	Software w/ CVEs	Synthetic Cases
Coverage	Limited	Limited	Applicable
Recall	N/A	Limited	Applicable
Precision	Applicable	N/A	Applicable
Discrimination	N/A	Limited	Applicable
Overlap	Applicable	Applicable	Applicable

- Attack surface: the software should be designed to face a hostile environment (e.g. ,a web server on the Internet).
- Size: the software should be comparable in size to the target code base.
- CVEs: the software should ideally contain a collection of diverse known vulnerabilities.
- Language: the software should be written in the same language as the target code base.
- Compilation: dependencies and other complications can lead to considerable overhead.

Quality synthetic test suites exist in several languages [10]. Some important aspects to consider are as follows:

- Coverage: the suite should have test cases for most of the targeted defects.
- Complexity: the suite should have several instances of the same defect using different code structures.

Once the tester has established the test case set, it is time to run the tools and to calculate the metrics based on the reports produced.

III. TOOL EFFECTIVENESS MEASUREMENT

The following results were collected during SATE V [7]. Participating teams ran 14 tools on the test sets that we provided, and they submitted their results to us for analysis. The following sections describe our findings using the methods explained above. For simplicity, we focus mainly on the synthetic Java test suite and the 4 tools that analyzed it. The complete results will be published in [7].

This test suite was computer-generated by combining short code templates written by experts. The templates contain seeded weaknesses that are vulnerable on at least one execution path. Different templates implement different code complexities, so the same seeded weakness is included in various code constructs to create different test cases. The full list of defects contained in the test suite is described in [18].

A. General Considerations

Tools report defects using their own vocabulary. Although most map to CWE, the latter proves sometimes unsuitable to evaluate static analyzers. For example, a tool could report a buffer overflow under CWE 121 (Stack-based Buffer Overflow), but its parent CWE 787 (Out-of-bounds Write) would also be legitimate. We overcome this hierarchy problem by

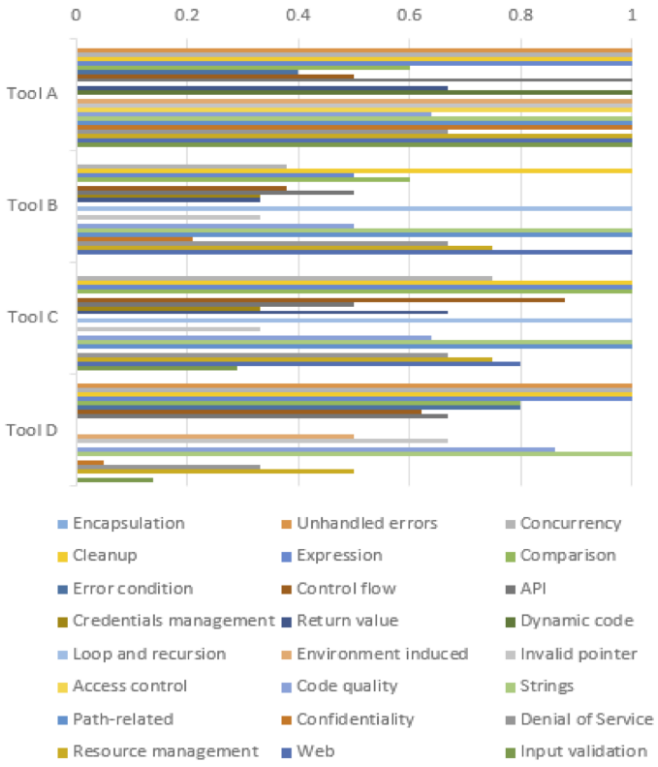

Fig. 1. Coverage spectrum per tool for Synthetic Java

grouping CWEs into 24 generic categories better aligned with static analyzer findings. Fig. 1 lists these groups in its legend.

B. Defects Found by Tools

The data we collected reveal varying coverage spectra for different tools. Fig. 1 shows the shape of these spectra for visual comparison[3]. Each bar represents one of the 24 weakness categories. We consider the test cases belonging to each category to determine their coverage by a tool. If the tool reports at least one true-positive for a test case with a given CWE, we consider that CWE covered. The overall score for a category is the proportion of CWEs of that category covered by a tool. For example, a tool reports true-positives on 2 CWEs in a category made of 5 CWEs, so the coverage is 40 %.

Although some categories, like *comparison* defects, are uniformly detected, the support for most other weakness classes differs considerably. For example, tool D's spectrum is sparse, meaning it can detect only a few types of defects, while tool A's is much denser.

Does this mean that tool A is better than the others? Not necessarily. Some vulnerability types, like *credential management*, are found by tools B and C but not by tool A. If other tools support all the weaknesses you are interested in, they might be solid candidates as well. Now that we know which defects a tool is able to report, we have to determine how well it can find them.

[3]For detailed numbers, refer to [7].

Fig. 2 shows the proportion of weaknesses each tool reported. Tool A discovered significantly more weaknesses than the others. Note that the horizontal axis ends at 70 %.

C. Tool Noisiness

Tools can indeed report a lot of informationnot all of it useful. Users want to minimize human review as much as possible, so tool noisiness is an important aspect of tool testing.

Fig. 3 shows which proportion of tool warnings are actually useful. Tools B, C and D have precision of about 90 %, meaning almost all reports they produce should be carefully considered. Tool A slightly lags behind in that regard.

Real software usually contains a number of similar sites[4] of which only a small fraction is vulnerable. To achieve good precision on such code, a tool must be able to differentiate defective from safe sites. When using CVEs or synthetic test cases to test tools, there is nearly an equal number of vulnerable and fixed sites. Therefore a tool reporting all sites would still achieve a precision of 50 %, as it would be right half of the time. The problem can be mitigated by introducing the discrimination rate, which accounts for true-positives only if the tool did not incorrectly report the same defect at the safe site (false-positive).

In our case, Fig. 4 shows that the discrimination rate does not differ significantly from precision, simply because all these tools are fairly sophisticated and do not generate many uncalled-for warnings.

D. Combination of Tool Metrics

Recall and precision can be combined in a measure called F- Score [16]. The metric can be calculated with different biases whether one wants to favor recall (finding more defects) or precision (reporting more useful information). We use the harmonic mean giving equal weight to both.

This combination leaves us with three measures: F-Score, coverage and discrimination. Fig. 5 shows that according to these criteria, tools B, C and D behave similarly. Tool A is in a separate class, sporting a higher F-Score and coverage while missing out on discrimination. Tool A would be a good candidate to find as many defects as possible while tools B, C and D could help findings fewer bugs but in a shorter time.

These metrics regard only more technical aspects of tools. Other fundamentals like user interface, integration and support should also be considered.

E. Legitimacy of Results on Synthetic Test Cases

All these metrics can be calculated using synthetic test cases, but can we generalize these results to production software? The answer is not simple. Fig. 6 presents the precision measured on two production test cases (Openfire[5] and JSPwiki[6]) against the precision obtained on the synthetic test suite. First, we observe that tools perform better on

[4]A site is an abstract location where a bug can happen, e.g. a buffer access.
[5]http://www.igniterealtime.org/projects/openfire
[6]https://jspwiki.apache.org

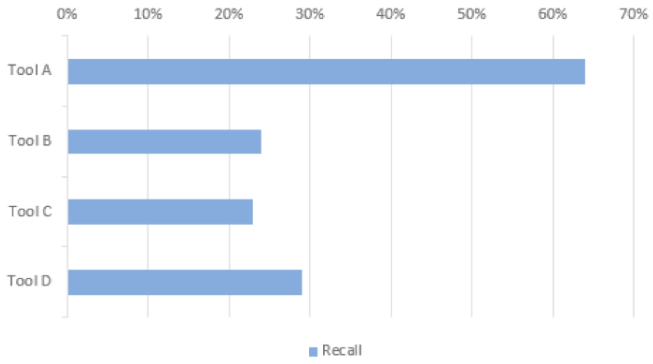

Fig. 2. Recall per tool for Synthetic Java

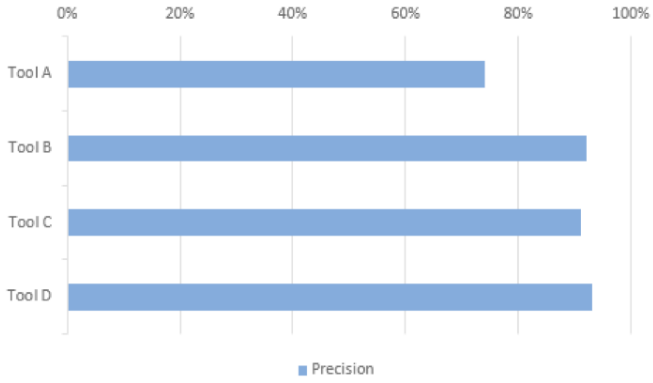

Fig. 3. Precision per tool for Synthetic Java

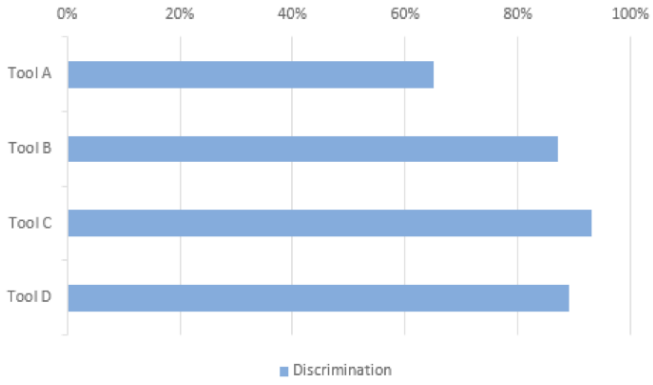

Fig. 4. Discrimination per tool for Synthetic Java

Openfire than on JSPwiki, emphasizing their sensitivity to the code base they analyze. Second, the precision they yield from synthetic test cases is overall higher than their precision on production software. Using solely synthetic test cases to assess a tool might therefore give an incomplete view of a tool's capabilities.

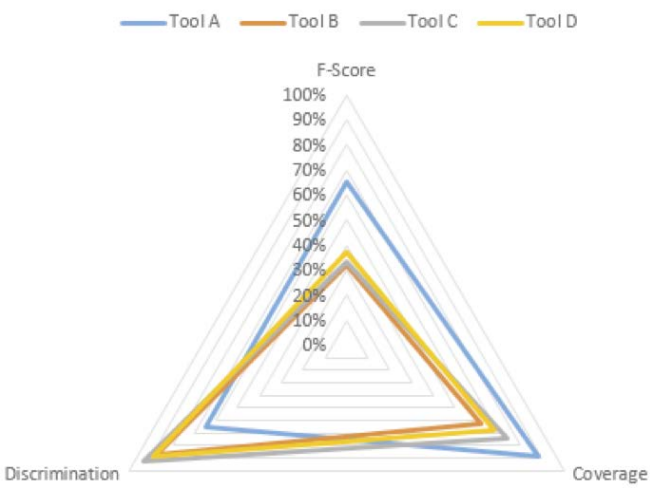

Fig. 5. Combination of Tool Metrics

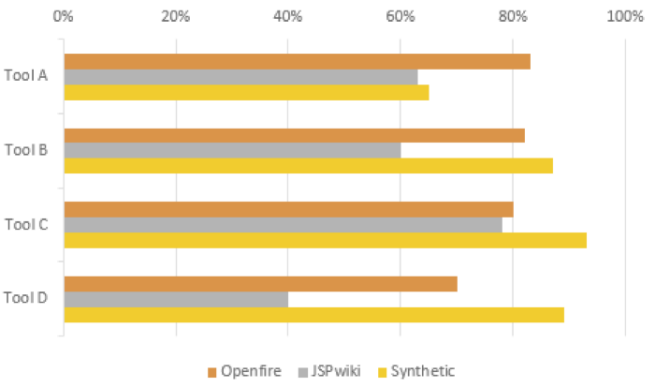

Fig. 6. Precision per tool on production software vs. synthetic cases for Java

IV. On Bugs

A. Code Complexity

In addition to testing tools for a wide variety of weaknesses, the synthetic test suite allows us to test the tool's handling of different kinds of complexities encountered in code. The synthetic test cases have complexities in control and data flow constructs. A control flow structure is added by inserting statements or variables into the source code, whereas a data flow structure will incorporate multiple data types, function calls, data structures, and memory attributes. Both make the software more complex from a tool's point of view.

Statement complexity is introduced by using control flow statements like *if*, *switch*, *while*, *for* or *goto* in the code, whereas a flow controlled by a global variable value adds *Variable* complexity.

Path complexity is concerned with how the data are passed between functions in the same source file or in different files. We consider a copy of data, the use of two pointers to the same value, accessing the same data with two methods and a reference to data, as *Data* complexity. *Structure* complexity

is introduced when a variable is accessed through an array, a pointer, a structure, a vector, a list, a hash map or a class. *Memory* complexity involves passing the data as an argument to a virtual function called via a reference or a pointer, or passing the data to a class constructor or destructor by declaring the class object either on the stack or on the heap.

Fig. 7 presents the recall per code complexity type. Note that the horizontal axis ends at 20 %.

On average, tools find about 20 % of weaknesses in the basic test cases (no complexity added). When a control flow construct (i.e., *Statement* or *Variable*) is inserted into the code, the ratio decreases by one fifth to about 16 % of findings. A complicated data flow construct (i.e. *Path*, *Data*, *Structure* or *Memory*) cuts the chance of success in half to 9 % of findings. As shown in Fig. 7, adding statements does not degrade tool effectiveness significantly. Tools handle *Statement* complexity relatively well, but have more difficulty with *Memory* complexity. Our main observation here is that the simpler the control and data flow structure of the software is, the more effective the tools will be at finding weaknesses. Generally, design and coding clarity helps improve software assurance [19].

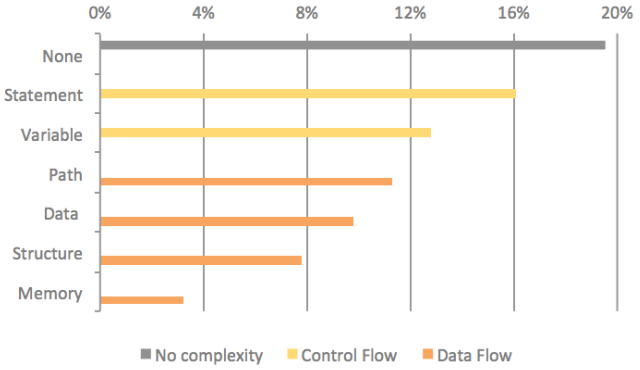

Fig. 7. Recall per complexity for Synthetic C

B. Bugs that are Easy vs. Hard to Find

Tools do not detect all types of weaknesses with the same efficiency. Among other things, this depends on tools' goals and the type of analysis performed. Fig. 8 shows overlap of tools' findings.

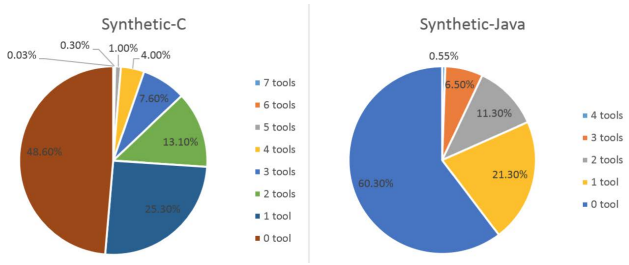

Fig. 8. Findings' overlap by tools

For the synthetic C test suite, almost half of all weaknesses were not detected by any of the 8 tools we evaluated. Only 16 weaknesses out of the total 61 387 (0.03 %) were detected by 7 tools. For the Java test suite, only 0.55 % (141 of 25 477) of the weaknesses were detected by all 4 tools used in our study, while 60 % of the weaknesses were not detected by any tool. This is consistent with observations from earlier SATEs [14].

Considering the above results, and in order to better understand the behavior of tools for different languages, we ranked weakness groupsas listed in Fig. 1s legendby tool recall. Our ranking shows *buffer operation*, *input validation* and *memory release* are the groups of weaknesses that have the highest detection rate by the tools in C/C++. Tools produce the fewest findings for *encapsulation* or *confidentiality* bugs. For the Java test suite, *input validation*, *web*, and *path-related* groups are found the most by the tools, while *encapsulation* and *loop and recursion* groups have the lowest detection rate. Precise numbers will be presented in [7]. The top ranked categories are similar to the most common weaknesses found in software, e.g. , [20]. It is reasonable that tool makers put their resources in the detection of the most common and important defects.

V. Conclusion

In this paper, we presented methods and metrics for evaluating static analyzers' effectiveness. We explained different aspects of importance when assessing tools and how to design or select tests to measure these characteristics. Armed with such metrics, an evaluator can compare tools and select the most suitable for his or her needs. We focused solely on the technical dimension. Other dimensions, like usability and integration, should also be considered.

Overall, our results show that tools report substantially different defects. Also, no tool prevails in all regards, so tool users should consider trade offs when choosing tools for their objectives.

We shared observations on bugs we encountered in code during SATE V. The ability of tools to find defects varies with code complexity. Bugs embedded in simple programs were reported by more tools than those located in convoluted software, advancing the case for writing simpler code.

We are still falling short on test cases. We need large software with known vulnerabilities. Intelligence Advanced Research Projects Activity (IARPA) made progress in that respect with the STONESOUP project[7], by seeding faults in production software. We will take advantage of this new test set to further improve our methodology and our understanding of tools.

References

[1] D. Wheeler, and R. S. Moorthy, "State-of-the-art resources (SOAR) for software vulnerability detection, test, and evaluation," Institute of Defense Analyses IDA Paper P-5061, July 2014.

[2] National Institute of Standards and Technology, "SAMATE - software assurance metrics and tool evaluation" Available from: http://samate.nist.gov

[3] V. Okun, R. Gaucher, and P.E. Black, "Static analysis tool exposition (SATE) 2008, NIST Special Publication 500-279, June 2009.

[7] http://www.iarpa.gov/index.php/research-programs/stonesoup

[4] V. Okun, A. Delaitre, and P. E. Black, "The second static analysis tool exposition (SATE) 2009," NIST Special Publication 500-287, June 2010.

[5] V. Okun, A. Delaitre, and P. E. Black, "Report on the third static analysis tool exposition (SATE) 2010," NIST Special Publication 500-283, October 2011.

[6] V. Okun, A. Delaitre, and P. E. Black, "Report on the static analysis tool exposition (SATE) IV," NIST Special Publication 500-297, January 2013, http://dx.doi.org/10.6028/NIST.SP.500-297

[7] A. Delaitre et al., "SATE V report," NIST Special Publication, unpublished.

[8] A. Delaitre, V. Okun, and E. Fong, "Of massive static analysis data," Proceeding of Software Security and Reliability (SERE 2013), June 2013.

[9] G. Diaz and J. R. Bermejo, "Static analysis of source code security: assessment of tools against SAMATE tests," Information and software technology 55 (2013) 1462-1476. (Also available at http://dx.doi.org/10.10.16/j.infsof.2013.02.005)

[10] Software Assurance Reference Dataset (SARD) project web site. http://samate.nist.gov/SARD

[11] J. A. Kupsch and B. P. Miller, "Manual vs automated vulnerability assessment: A case study," First international workshop on managing insider security threats (MIST 2009), West Lafayette, IN, June 2009.

[12] C. Willis, "CAS static analysis tool study overview," In proc. eleventh annual high confidence software and systems conference, page 86, National Security Agency, 2011, http://hcss.csp.org

[13] Common Weakness Enumeration (CWE). The MITRE corporation, http://cwe.mitre.org

[14] Paul E. Black, "Static analyzers: seat belts for your code," May-June 2012, IEEE Security & Privacy, 10(3):48-52.

[15] Paul E. Black, "Counting bugs is harder than you think," in 11th IEEE Int'l Working conference on source Code Analysis and Manipulation (SCAM2011), Williamsburg, VA, September 2011.

[16] "CAS static analysis tool study methodology," Center for Assured Software, NSA, 2011, http://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf

[17] Common Vulnerabilities and Exposures (CVE). The MITRE corporation, http://cve.mitre.org

[18] "Juliet test suite v1.2 user guide," Appendix A, Center for Assured Software, NSA, December 2012, http://samate.nist.gov/SARD/testsuite.php

[19] Gerard J. Holzmann, "Conquering complexity," Computer 40 (12): 111-113, Dec. 2007.

[20] 2011 CWE/SANS Top 25 Most Dangerous Software Errors, http://cwe.mitre.org/top25/