A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code

Andrei Arusoaie*, Ștefan Ciobâcă*, Vlad Crăciun*†, Dragoș Gavriluț*†, Dorel Lucanu*
*Faculty of Computer Science, Alexandru Ioan Cuza University, Iaşi
Email: arusoaie.andrei,stefan.ciobaca,dlucanu@info.uaic.ro
†Bitdefender, Iaşi
Email: vcraciun,dgavrilut@bitdefender.com

***Abstract*—We describe work that is part of a research project on static code analysis between the Alexandru Ioan Cuza University and Bitdefender. The goal of the project is to develop customized static analysis tools for detecting potential vulnerabilities in C/C++ code.**

We present the results of benchmarking several existing open source static analysis tools for C/C++ against the Toyota ITC test suite [1] in order to determine which tools are best suited to our purpose. The Toyota ITC test suite is a synthetic benchmark for C/C++ consisting of around 650 test cases organized by defect type and defect subtype and is well-suited to our purpose, since it contains various bugs such as buffer overflows that are common in C/C++ code.

We analyze the open-source static analysis tools according to the existing quality indicators such as detection rate and false positive rate proposed in [1], but we also introduce a new quality metric that we call *robust detection* which also allows us to measure unique detections by tool and by (sub)defect type. We also find several mistakes in the Toyota ITC testsuite that we fix. We publish the harness used to benchmark the static analyzers in order for anyone to be able to reproduce our results.

1. Introduction

This paper describes part of a joint research project between the Alexandru Ioan Cuza University and Bitdefender, which started in October 2016. The main goal of the project is to develop a custom static analysis tool/framework for C/C++ code that is able to detect possible vulnerabilities such as buffer overflows and use of unsanitised data from untrusted sources.

The project aims to improve the code analysis process in terms of productivity and ease of use, while taking into account the increasing complexity of the malware detection process. Therefore the customised solution for code analysis must: • be tailored towards the kind of vulnerabilities the company is interested in; • be fast, in order to increase the productivity of the developers; • be extensible, in the sense that it should be easy to adapt or to configure it for new classes of vulnerabilities; • report only defects that represent possible vulnerabilities the company is interested in, together with their context.

There are several well known problems that can occur when using static code analysis tools: • the output of such tools often include a lot of spurious warnings/errors, which cannot always be switched off through tool configuration options. These make the output difficult to read and it is very probable that developers stop paying attention to it; • the execution of tools takes a long time, which could slow down developer productivity; • the static analysis abstractions used by the tools are not a good match for the abstractions used by the developers, which decreases the precision of the tools.

All these issues are in fact challenges for the design of a customized static code analysis tool. We first compared existing open-source static analysis tools to see how well they perform. This paper reports our experience on the comparison.

We first searched the literature for such a comparison. We found several survey papers on static code analysis tools and several benchmarks for bug detection tools. However, such papers usually concentrate on programming languages other than C/C++ (e.g. [2]), they do not focus on vulnerability detection (e.g. [3], [4]), they only concentrate on certains subsets of vulnerabilities (e.g. [5]) which do not include all vulnerabilities that we are interested in, or the associated artifacts do not seem to be available anymore (e.g. [6]).

The closest match was the Toyota ITC test suite article [1]. In [1], the authors introduce a synthetic test suite for comparing static analyzers of C/C++ code. The test suite contains mostly low-level defects such as numerical overflows, buffer overflows, memory allocation defects, etc., organized by defect type and defect subtype. The test suite was initially created to contain the types of low-level defects that are typical in car code. However, our industrial partner was also interested in these kinds of defects and therefore the test suite is a good match.

However, in [1], only three comercial tools are compared and, even though the test suite is available at https://github.com/regehr/itc-benchmarks/, there is no harness to run analysis tools against the benchmark. Therefore we started to develop our own framework for running and comparing static analysis tools. In addition to the results that we report in the paper, we have also published the harness and the intermediate results so that anyone can reproduce our findings.

For classifying vulnerabilities we had to choose between the classification in the Toyota ITC test suite and the Common Weakness Enumeration (CWE$^{\text{TM}}$) list [7], which was created by the MITRE Corporation to be used as a common taxonomy for software weaknesses. Even though the CWE list offers more flexibility in classifying errors, we decided that the original classification in [1] into defects subtypes and defects types in the ITC suite is sufficient for our purposes. Additionally, we proposed a CWE number to each subdefect type and we added these as comments in the test suite source code, so it would be very easy to use these as well.

The Toyota ITC test suite. The ITC test suite contains 638 test cases. The test cases are categorized into 51 subdefect types and 9 defect types. Each test case comes in two flavours called variations: a "positive" variation in which the bug is present and a "negative" variation which corresponds to one of the positive variations and in which the bug was fixed. Therefore, there are a total of 1276 variations in the test suite. A perfect static analyzer would warn on all 638 variations which contain bugs and would be silent on the 638 variations that do not contain bugs. For each subdefect type, there is one `.c` or `.cpp` file containing one function for each variation of the subdefect type. There is an exception: the 6 cases for testing wrong uses of the "extern" keyword are organized, by necessity, into two `.c` files. Only one subdefect type with 4 test cases, "improper error handling", is C++ specific and the rest of the test cases are for C. We have found a minor mistake in the original testsuite: there is a positive variation that does not have a corresponding negative variation and a negative variation that does not have a corresponding positive variation. We have added both and therefore we have ended up with 639 test cases summarized in Table 1.

The first step in our framework is to collect the line number containing errors (in the positive variations) and the line numbers containing fixes (in the negative variations). This is performed by a grep in the source files of the test suite, as each such line is annotated with a comment in a standard format. We expect static analysis tools to throw errors on the line numbers of the positive variations and to remain silent about the line numbers in the negative variations.

Our framework then runs a set of static analyzers over variations. For each variation, we analyze whether the corresponding vulnerabilities in the manifest file are detected by the static analysis tool. We parse the output of the tool and we consider a variation to be detected as a bug when the tool produces an error on a warning on the respective line number.

We produce reports for each tool. We compute the statistics proposed originally in [1], but we also introduce a new measure that we call *robust detection rate*. The new metric allows us to also count *uniques*, that is bugs that can only be detected by one tool and not the others.

We have manually inspected some of the results. The manual inspection reveals some bugs in the Toyota ITC test suite and a few imprecisions in the static analysis tools that we describe.

In the current version of our framework, we have benchmarked the following tools against the test suite: Clang (three versions: one with the default core checkers, one with the less mature alpha checkers, and one with both), Cppcheck, Flawfinder, Flint++, Frama-C, Facebook Infer, Oclint, Sparse, Splint and Uno. In addition, we have also added a non open source static analyzer, which is a builtin part of the compiler used by our industrial partner on their development platform. We refer to this non open source as the "system" analyzer. It is possible to easily add other analysis tools by simply writing a parser for its output.

Contributions. (1) We benchmark several open source static analysis tools on the ITC test suite and rank them by various statistics, including by how well they perform on certain categories of bugs such as numerical defects or buffer overruns. (2) We produce a harness that makes it easy to automatically run a number of static analysis tools on the test suite and compute the relevant statistics, but also to add a new static analysis tool to the harness. This harness is important because without it, it is hardly possible to take advantage of the 639 variations. (3) We make all intermediate output and the harness available for anyone to reproduce our results. (4) We fix a number of small mistakes in the ITC test suite, mostly typos in comments – but comments are crucial metadata since they describe which bugs are expected/not expected in a particular file/at a particular line. More notably, we fix a missing positive variation and a missing negative variation. This results in 2×639 variations for the fixed testsuite in comparison with the 2×638 variation for the original testsuite. (5) We propose a new metric that we call *robust detection* and rank the tools by this new metric as well. The new metric allows us to also measure *uniques*, bugs that are detected by one static analisys tool, but not the others.

Section 2 contains an overview of the main tools and approaches related to our work. Section 3 presents the ITC suite and our framework, including the new metrics that we propose. Section 4 presents the statistics that we have obtained on the open source static analysis tools that we considered. In Section 5 we discuss the findings and possible directions for future work.

Disclaimer. Certain instruments, software tools and their organisations are identified in this paper to specify the exposition adequately. Such identification is not intended to imply recommendation or to imply that the instruments and software tools are necessarily the best available for the purpose.

2. Related Work

We are interested in automated tools for detecting vulnerabilities in software written in C/C++. There are several techinques for such automated analyses, including (bounded) model checking, abstract interpretation, static analysis, runtime monitoring, or combinations thereof, and several commercial and open source tools that implement such techniques. We rule out techniques such as deductive verification since they cannot be easily automated [8].

There are several tools that partially fit our needs, but we concentrate only of the most well known. One of the oldest static analysis tools is Lint; one of its succesors, Splint [9], can check C programs for security vulnerabilities and some other mistakes. The open source CppCheck [10] features several analyses for C++ code such as bounds checking. The UNO analyzer [11] can detect use of uninitialized variables, nulll-pointer dereferencing or out-of-bound array indexing. It works only for C (not C++) code.

Flawfinder [12] is a pattern-matching based tool that searches for uses of C/C++ functions with well-known

Defect Subtype	Defect Type	Positive Variations	Negative Variations
Dead lock	Concurrency defects	5	5
Double lock	Concurrency defects	4	4
Double release	Concurrency defects	6	6
Live lock	Concurrency defects	1	1
Locked but never unlock	Concurrency defects	9	9
Long lock	Concurrency defects	3	3
Race condition	Concurrency defects	9	9
Unlock without lock	Concurrency defects	8	8
Assign small buffer for structure	Dynamic memory defects	11	11
Deletion of data structure sentinel	Dynamic memory defects	3	3
Dynamic buffer overrun	Dynamic memory defects	32	32
Dynamic buffer underrun	Dynamic memory defects	39	39
Memory copy at overlapping areas	Dynamic memory defects	2	2
Contradict conditions	Inappropriate code	10	10
Dead code	Inappropriate code	13	13
Improper error handling	Inappropriate code	4	4
Improper termination of block	Inappropriate code	4	4
Redundant conditions	Inappropriate code	14	14
Return value of function never checked	Inappropriate code	16	16
Unused variable	Inappropriate code	7	7
Bad extern type for global variable	Misc defects	6	6
Non void function does not return value	Misc defects	4	4
Uninitialized variable	Misc defects	15	15
Unintentional end less loop	Misc defects	9	9
Useless assignment	Misc defects	1	1
Bit shift bigger than integral type or negative	Numerical defects	17	17
Data overflow	Numerical defects	25	25
Data underflow	Numerical defects	12	12
Division by zero	Numerical defects	16	16
Integer precision lost because of cast	Numerical defects	19	19
Integer sign lost because of unsigned cast	Numerical defects	19	19
Power related errors	Numerical defects	29	29
Bad cast of a function pointer	Pointer related defects	15	15
Comparison NULL with function pointer	Pointer related defects	2	2
Dereferencing a NULL pointer	Pointer related defects	17	17
Free NULL pointer	Pointer related defects	14	14
Incorrect pointer arithmetic	Pointer related defects	2	2
Uninitialized pointer	Pointer related defects	16	16
Wrong arguments passed to a function pointer	Pointer related defects	18	18
Double free	Resource management defects	12	12
Free non dynamically allocated memory	Resource management defects	16	16
Invalid memory access to already freed area	Resource management defects	17	17
Memory allocation failure	Resource management defects	16	16
Memory leakage	Resource management defects	18	18
Return of a pointer to a local variable	Resource management defects	2	2
Uninitialized memory access	Resource management defects	15	15
Cross thread stack access	Stack related defects	6	6
Stack overflow	Stack related defects	7	7
Stack underrun	Stack related defects	7	7
Static buffer overrun	Static memory defects	54	54
Static buffer underrun	Static memory defects	13	13
Total		639	639

TABLE 1. A SUMMARY OF THE ITC TEST SUITE THAT WE USED. THE ONLY DIFFERENCES ARE THAT SOME TYPOS ARE FIXED AND THAT WE HAVE 9 TEST CASES FOR RACE CONDITIONS INSTEAD OF 8 IN THE ORIGINAL PAPER.

problems such as possible buffer overflows when the wrong arguments are used. Oclint [13] is an AST-based static analyzer that looks for suspicious patterns in the source code. Sparse [14] is a static analysis tool written initially for the Linux kernel. Flint++ [15] is a cross-platform port of the flint program.

On the commercial side, we have looked at several tools. Astrée [16] is a static analysis tool based on abstract interpretation [17] for a subset of the C language that aims to be sound and which is especially useful for finding buffer overflows and/or undefined results. Grammatech produces the tool CodeSonar [18], an interprocedural analyzer that can find buffer overflows, memory leaks and others. Coverity (and in particular Coverity Security Advisor) [19] provides similar capabilities. PVS-Studio is a static analyzer featuring incremental analysis which integrates with Visual Studio. Of the commercial tools above, it is currently not possible to obtain an evaluation version of Coverity. We have decided not to benchmark the commercial tools, since they have already been benchmarked in [1].

Of the open source tools, Clang [20] features a static analyzer based on interprocedural data-flow analysis. Frama-C [21] is a framework for C program analysis that is sound and which features several plugins for static analysis or verification. One such plugin is the builtin value analysis plugin, which can be used to test for buffer overflows, pointer aliasing, etc. In addition to analyzing C code, it is possible to use Frama-C to analyze C++ code with the early stage framac-clang plugin. This plugin works by translating C++ into C, using the clang frontend, but it is not as mature as Frama-C itself.

Several tools are available from Microsoft, including the `/analyze` command-line option in Visual C++ (however, it requires code annotations to work effectively). Microsoft's Static Driver Verifier/SLAM Project [22] implements counter-example guided abstraction refinement for checking API usage in C code, and has been used to verify several correctness properties in drivers. HP provides Fortify for C/C++. An interesting approach is taken by Infer [23] from Facebook, which uses bi-abduction to perform interprocedural analysis. Static analysis tools such as Cppdepend [24] concentrate on code metrics and visualizations.

The Toyota ITC benchmark [25] provides a set of C programs annotated with defects such as static and/or dynamic buffer overflow/underflow, dereferencing of NULL pointers, etc. The article [1] compares CodeSonar (GrammaTech) and Code Prover/Bug Finder (MathWorks). The BugBench benchmark paper [4] compares three runtime analysis tools against a 17 open-source C/C++ applications with known bugs. Other runtime bug finding tools, such as RV-Match [26], were benchmarked against the Toyota ITC tests. The OWASP project contains the OWASP benchmark [27], which is a test suite with vulnerabilities in Java code.

The BegBunch [6] benchmark consists of several bug kernels (small pieces of code designed to capture the essence of bugs occuring in real code). The bug kernels were extracted by the BegBunch team from real applications such as OpenSolaris and MySql, but also contains some bug kernels from BugBench [4] or Zitser [5]. Unfortunately, the benchmark does not seem to be available for download anymore. Most of the existing comparisons use various criteria (e.g., performance of the tools based on annotated code versus tools that do not need annotations [28], detection ratio [29], etc.). Others (e.g., [30], [31]) include extensive studies on the functionality provided by the tools, or exhaustive analyses and statistics over large test suites.

3. Comparing Static Analyzers

The Toyota ITC suite. The ITC suite contains 638 test cases. The test cases are categorized into 51 defect subtypes and 9 defect types. Each test case comes in two flavours called variations: a "positive" variation in which the bug is present and a "negative" variation which corresponds to one of the positive variations and in which the bug was fixed. The test cases are summarized in Table 1. Here is the positive variation of the second test case in the "Dynamic buffer overrun" category:

```
void dynamic_buffer_overrun_002()
{
  short*buf=(short*)calloc(5,sizeof(short));
  if(buf!=NULL)
  {
    *(buf+5)=1;/*ERROR:Buffer overrun*/
    free(buf);
  }
}
```

and here is the negative variation:

```
void dynamic_buffer_overrun_002 ()
{
  short*buf=(short*)calloc(5,sizeof(short));
  if(buf!=NULL)
  {
    *(buf+4)=1;/*No ERROR:Buffer overrun*/
    free(buf);
  }
}
```

A good static analysis tool will identify an error in the first function (preferably at the line annotated with the "ERROR" comment), but not in the second (at the line annotated with the "NO ERROR" comment).

There are a total of 1276 variations in the test suite. A perfect static analyzer would warn on all 638 variations that contain bugs and would be silent on the 638 variations that do not contain bugs. For each defect subtype, there is one `.c` or `.cpp` file containing one function for each variation of the subdefect type. There is an exception: the 6 cases for testing wrong uses of the "extern" keyword are organized, by necessity, into two `.c` files. Only one subdefect type with 4 test cases, "improper error handling", is C++ specific and the rest of the test cases are for C.

The original test suite contains a number of small typos: missing "ERROR" comments, missing "NO ERROR" comments, typos in the comments, typos in the defect (sub)types. We have fixed these and we have grep-ed the source code to produce a set of file–line number pairs where the static analysis tools are expected to produce warnings (the positive variations) and a set of file-line number pairs where the static analysis tools are not expected to produce warnings. We make this file available and we use it to perform the statistics described later on.

We have also found that in the original testsuite, there is a positive variation that does not have a corresponding negative variation and a negative variation that does not have a corresponding positive variation. We have added both and therefore we have ended up with 639 test cases summarized in Table 1 instead of the 638 test cases in the original test suite.

In the original ITC suite paper [1], three statistics are proposed for each tool:

- detection rate
$$DR = \frac{\textit{number of positive variations detected}}{\textit{number of positive variations}};$$
- false positive rate
$$FPR = \frac{\textit{number of negative variations detected}}{\textit{number of negative variations}};$$
- productivity
$$PR = \sqrt{DR \times (100 - FPR)}.$$

Each of the three measures can be computed over the entire test suite, over a defect type, or over a defect subtype, resulting in rankings of tools by defect type/subtype/overall. There is an additional measure based on price, but since we are comparing open source tools, this measure does not make sense for our purposes. We compute the statistics described above and the running times of the tools and present them in Section 4.

Additional statistics. We propose a new metric that is to our knowledge original. We call it *robust detection rate* and it is computed as explained next.

Definition 3.1. We say that a tool *robustly handles a test case* if it detects the positive variation of the test case but not the negative variation of the test case.

The robust detection rate is defined as follows:

- robust detection rate

$$RDR = \frac{\textit{number of test cases robustly handled}}{\textit{number of test cases}}.$$

The robust detection rate can again be computed over the entire suite, over just one defect type or over a defect subtype, yielding rankings of tools in the respective category. The intuition behind the measure is that a static analysis tool is better whenever it accurately distinguishes between a situation where a defect occurs and a (syntactically or semantically) similar situation where the defect does not occur. This metric will be particularly punishing for AST based static analyzers, since they often lack the context necessary to distinguish between false positives and true positives.

Uniques. The most important advantage of the RDR is that it allows us to compute a notion of *uniques*, that is bugs that can only be detected by a certain tool. In particular, we compute for each tool the number of test cases that are handled robustly by that tool, but not by any other tool. Again, the uniques can be computed over the entire test suite or just over a defect (sub)type. A tool that has a high number of uniques is intuitively a tool that is better in the sense of being able to find more advanced bugs (bugs that are missed by other tools).

4. Experiments

As announced earlier, we have run the following tools against the test suite: Clang (three versions: one with the default core checkers, one with the less mature alpha checkers, and one with both), the system analyzer, Cppcheck, Flawfinder, Flint++, Frama-C, Facebook Infer, Oclint, Sparse, Splint and Uno. In this section, we present the statistics that we obtained.

In Table 2, we summarize the performance of the tools over the entire test suite. We computed for each tool the Detection Rate (column DR), False Positive Rate (column FPR), Productivity (column PR), Robust Detection Rate (column RDR), Uniques (columnU) and Time (column T, in seconds). The statistics are over the entire test suite. The results are sorted by PR.

Tool	DR	FPR	PR	RDR	U	Time
Clang	35.84	10.95	56.49	25.67	59	13.55
Frama-C	27.86	5.79	51.23	22.38	52	14.43
Oclint	44.13	52.74	45.67	5.01	0	23.95
System	21.75	4.23	45.64	18.15	6	42.47
Cppcheck	20.81	0.78	45.44	20.03	1	2.83
Splint	23.63	16.12	44.52	9.39	7	2.18
Infer	9.70	1.41	30.92	8.29	4	47.30
Uno	5.48	0.16	23.39	5.32	3	50.80
Flawfinder	2.97	2.97	16.98	0.16	1	1.15
Sparse	1.56	0.00	12.49	1.56	0	3.97
Flint++	1.10	1.10	10.43	0.16	0	0.27

TABLE 2. OVERVIEW OF RESULTS.

All tools except UNO and the system analyzer were tested on a fairly recent MacBook Pro computer [1], UNO was tested on a Linux server machine and the system analyzer on a virtual machine. Therefore the running times of these two tools should not be compared aginst the rest or between themselves.

The system analyzer and Frama-C do not support `pthread.h` (at least not out of the box) and we had to supply our own replacement. This means that they have a disadvantage in finding the concurrency errors in the test suite. In addition, Frama-C stops with a fatal error when reaching the test case that exercises variables declared with different types in different translation units. Therefore, we excluded this test case for Frama-C (even though Frama-C correctly detects it).

For Clang, we enabled both the out-of-the-box checkers in its "core" package, as well as the experimental checkers in the "alpha" package, which are not enabled by default. We also present the summary findings where the two checker packages are separated in Table 3.

Tool	DR	FPR	PR	RDR	U	Time
Frama-C	27.86	5.79	51.23	22.38	52	14.43
Clang (alpha)	28.17	10.33	50.26	18.94	46	13.29
Oclint	44.13	52.74	45.67	5.01	0	23.95
System	21.75	4.23	45.64	18.15	6	42.47
Cppcheck	20.81	0.78	45.44	20.03	1	2.83
Splint	23.63	16.12	44.52	9.39	7	2.18
Clang (core)	15.34	0.63	39.04	14.87	9	6.42
Infer	9.70	1.41	30.92	8.29	4	47.30
Uno	5.48	0.16	23.39	5.32	3	50.80
Flawfinder	2.97	2.97	16.98	0.16	1	1.15
Sparse	1.56	0.00	12.49	1.56	0	3.97
Flint++	1.10	1.10	10.43	0.16	0	0.27

TABLE 3. OVERVIEW OF RESULTS (CLANG ALPHA CHECKERS SEPARATED FROM CLANG CORE CHECKERS).

We can see that RDR generally correlates well with PR, with only a few exceptions. Given that Oclint produces a good number of false positives, we feel that its productivity score is too high, and perhaps RDR would be a better indicator of overall quality. We can also see that the top tools detect many unique bugs. This partly explains a well-known rule of thumb in the static analysis community, that the more tools you run on a code base, the more bugs you find.

4.1. Results by Defect Type

In order to report the results of tools by defect type, we use the following abbreviations for the 9 defect types:

D1	Concurrency defects
D2	Dynamic memory defects
D3	Inappropriate code
D4	Misc defects
D5	Numerical defects
D6	Pointer related defects
D7	Resource management defects
D8	Stack related defects
D9	Static memory defects

1. The tests have been run on an computer equipped with 16GB RAM and a 3.1 GHz i5 processor with 2 cores and hyperthreading.

All percentages presented in this subsection are rounded to the nearest integer in order to save space. We first describe de detection rate of each tool for a particular defect type in Table 4.

Tool	D1	D2	D3	D4	D5	D6	D7	D8	D9
System	0	39	7	34	5	25	34	0	40
Clang	47	15	35	40	36	43	16	5	82
Cppcheck	0	6	0	11	31	20	23	0	64
Flawfinder	0	2	1	3	0	5	8	15	0
Flint++	0	0	6	0	0	4	0	0	0
Frama-C	2	83	10	26	31	19	32	0	0
Infer	0	1	0	0	0	18	48	0	0
Oclint	71	22	51	40	26	50	82	50	22
Sparse	0	0	0	0	7	0	0	0	0
Splint	0	6	6	40	40	25	45	20	7
Uno	0	1	0	11	0	7	1	0	34

TABLE 4. DETECTION RATE FOR EACH TOOL AND DEFECT TYPE.

However, the detection rate needs to be balanced against the false positive rate, which is described for each tool and defect type in Table 5.

Tool	D1	D2	D3	D4	D5	D6	D7	D8	D9
System	0	11	0	0	0	8	10	0	0
Clang	11	9	15	3	15	10	7	5	15
Cppcheck	0	0	0	0	2	0	2	0	0
Flawfinder	0	3	1	3	0	5	8	10	0
Flint++	0	0	6	0	0	2	1	0	0
Frama-C	0	23	10	9	1	2	4	0	0
Infer	0	1	0	0	0	1	7	0	0
Oclint	87	29	50	43	42	65	86	75	19
Sparse	0	0	0	0	0	0	0	0	0
Splint	0	9	1	17	37	10	21	20	7
Uno	0	0	0	0	0	1	0	0	0

TABLE 5. FALSE POSITIVE RATE FOR EACH TOOL AND DEFECT TYPE.

In [1], the PR metric is used to compare tools ($PR = \sqrt{DR \times (100 - FPR)}$). This metric rewards tools that have a good DR, but punishes at the same time tools that have a high FPR. The PR of each tool by defect type is summarized in Table 6. We can see that Frama-C has a good performance for defect type D2 (Dynamic memory defects) and that Clang has good performance for defect type D9 (Static memory defects). Infer has the best performance for defect type D7 (Resource management defects).

Tool	D1	D2	D3	D4	D5	D6	D7	D8	D9
System	0	59	27	59	23	48	55	0	63
Clang	64	37	55	62	56	62	38	22	84
Cppcheck	0	24	0	34	55	45	47	0	80
Flawfinder	0	15	12	17	0	21	28	37	0
Flint++	0	0	24	0	0	19	0	0	0
Frama-C	15	80	30	48	55	43	56	0	0
Infer	0	11	0	0	0	42	67	0	0
Oclint	31	39	51	48	39	42	33	35	42
Sparse	0	0	0	0	27	0	0	0	0
Splint	0	23	24	58	50	48	60	40	26
Uno	0	11	0	34	0	27	10	0	59

TABLE 6. PRODUCTIVITY FOR EACH TOOL AND DEFECT TYPE.

We propose a new metric called robust detection. The robust detection rate of each tool for a particular defect type is presented in Table 7. Note that the winning tools for each defect type are unaffected, and therefore there is strong correlation with the productivity metric.

Tool	D1	D2	D3	D4	D5	D6	D7	D8	D9
System	0	29	7	34	5	20	24	0	40
Clang	36	7	21	40	23	35	8	5	67
Cppcheck	0	6	0	11	28	20	21	0	64
Flawfinder	0	0	0	0	0	0	0	5	0
Flint++	0	0	0	0	0	1	0	0	0
Frama-C	2	61	0	17	30	18	28	0	0
Infer	0	0	0	0	0	17	41	0	0
Oclint	0	1	6	11	8	1	0	0	16
Sparse	0	0	0	0	7	0	0	0	0
Splint	0	0	4	31	3	20	26	0	0
Uno	0	1	0	11	0	6	1	0	34

TABLE 7. ROBUST DETECTION RATE FOR EACH TOOL AND DEFECT TYPE.

However, the main advantage of robust detection is that it allows to compute unique detections by tool. In Table 8, we present the *number* of robustly handled test cases by tool and defect type. Note that this table reinforces the common knowledge that running more static analysis tools reveals more bugs. Note that this table represents counts, not percentages, as the other tables in this subsection.

Tool	D1	D2	D3	D4	D5	D6	D7	D8	D9
System	0	0	5	0	0	0	0	0	1
Clang	15	1	9	1	9	8	2	1	13
Cppcheck	0	0	0	0	0	0	1	0	0
Flawfinder	0	0	0	0	0	0	0	1	0
Flint++	0	0	0	0	0	0	0	0	0
Frama-C	0	28	0	6	8	9	1	0	0
Infer	0	0	0	0	0	1	3	0	0
Oclint	0	0	0	0	0	0	0	0	0
Sparse	0	0	0	0	0	0	0	0	0
Splint	0	0	2	0	3	1	1	0	0
Uno	0	0	0	0	0	1	0	0	2

TABLE 8. UNIQUES FOR EACH TOOL AND DEFECT TYPE.

4.2. Results by Defect Subtype

We also compute the best tool by robust detection rate for each defect subtype. Whenever the RDR is zero for a defect subtype, it means that no tool was able to detect robustly any defect of that subtype and therefore there is no winner. We present these findings in Table 9.

The following table summarizes how many defect subtype categories have been won (according to robust detection rate) by each tool:

Clang	17	Cppcheck	1
Frama-C	9	Uno	1
Splint	4	Oclint	1
Infer	3	No winner	12
System	3		
Total			51

5. Discussion and Future Work

The statistics in the previous section are obtained automatically by a set of scripts.

Defect subtype	Tool	RDR
Assign small buffer for st...	-	0.00
Bad cast of a function poi...	Frama-C	100.00
Bad extern type for global...	-	0.00
Bit shift bigger than inte...	Frama-C	94.12
Comparison NULL with funct...	Clang	100.00
Contradict conditions ...	System	50.00
Cross thread stack access ...	Clang	16.67
Data overflow	Frama-C	64.00
Data underflow	Frama-C	50.00
Dead code	-	0.00
Dead lock	-	0.00
Deletion of data structure...	-	0.00
Dereferencing a NULL point...	Infer	70.59
Division by zero	Clang	75.00
Double free	Infer	91.67
Double lock	Clang	75.00
Double release	Clang	83.33
Dynamic buffer overrun ...	Frama-C	75.00
Dynamic buffer underrun ...	Frama-C	74.36
Free NULL pointer	Clang	7.14
Free non dynamically alloc...	Frama-C	93.75
Improper error handling ...	-	0.00
Improper termination of bl...	Splint	50.00
Incorrect pointer arithmet...	Oclint	50.00
Integer precision lost bec...	Frama-C	15.79
Integer sign lost because ...	Clang	21.05
Invalid memory access to a...	System	58.82
Live lock	Clang	100.00
Locked but never unlock ...	-	0.00
Long lock	Clang	100.00
Memory allocation failure ...	Infer	12.50
Memory copy at overlapping...	-	0.00
Memory leakage	Splint	22.22
Non void function does not...	System	100.00
Power related errors ...	Splint	3.45
Race condition	Clang	22.22
Redundant conditions ...	Clang	92.86
Return of a pointer to a l...	Cppcheck	100.00
Return value of function n...	Splint	6.25
Stack overflow	-	0.00
Stack underrun	-	0.00
Static buffer overrun ...	Clang	72.22
Static buffer underrun ...	Uno	53.85
Uninitialized memory acces...	Clang	20.00
Uninitialized pointer ...	Clang	50.00
Uninitialized variable ...	Clang	66.67
Unintentional end less loo...	Frama-C	66.67
Unlock without lock ...	Clang	25.00
Unused variable	-	0.00
Useless assignment	-	0.00
Wrong arguments passed to ...	Clang	16.67

TABLE 9. BEST TOOL BY ROBUST DETECTION RATE FOR EACH DEFECT SUBTYPE.

All files needed to reproduce out results, as well as the files generated for our intermediary steps, can be found at the following urls:

- https://github.com/andreiarusoaie/itc-benchmarks for the test suite. We started with the test suite at https://github.com/regehr/itc-benchmarks/ and corrected the small mistakes described in Section 3.
- https://github.com/andreiarusoaie/itc-testing-tools for the scripts.

The test harness first parses the source code of the test suite and gathers from the annotations the line numbers at which positive variations and negative variations occur. These line numbers are stored in a set of `.csv` files for later use.

The test harness then runs each tool on the set of test cases. There is a custom parser for each static analysis tool that extracts the line numbers/file names at which the tool produces warnings. These line numbers/file names associations are also stored in several `.csv` files for later processing.

Our harness considers that a tool detects a variation when it produces a warning at the line number gathered from the test suite. The various statistics are computed from the set of variations detected by the tools as explained earlier.

Summary of findings. We rank existing open-source static analysis tools by productivity, as defined in the original ITC paper [1]. Productivity aims to balance the detection rate with the false positive rate in order to penalize tools with a high false positive rate. We propose a new metric, called robust detection rate, which correlates with productivity, but which in our experience seems better at ranking tools that have an annoyingly high false positive rate. The new metric also provides a way to count the number of unique bugs detected by a single tool. Counting uniques allows us to confirm a fact known in the community, namely that the more static analysis tools are used on a codebase, the more bugs are found. We have also confirmed that certain tools are better at detecting certain defect types than others, as per the results in Table 7.

We have presented these results to the industrial partner and, based on the results, we have chosen to customize the Clang Static Analyzer for the needs of the company. Another finding was that most static analysis tools are very difficult to install/compile, because they rely on older versions of various libraries or compilers. Finally, a great pain point is that many static analysis tools have difficulty parsing the system headers and certain workarounds must be provided (e.g., by adding `#defines` in the command line). Glancing over the results of the tools, we find many interesting issues, including small bugs in the ITC suite or various strengths and weaknesses of the tools. Unfortunately, there is not enough space to describe these findings here. Our work also identifies the defect subtypes for which no good static analyzer exists (see for example Table 9). This is an open space where static analysis tool vendors or researchers could step in.

Possible threats to validity. A possible weakness in our approach is that the output of the static analysis tools is not inspected manually. In particular, if a static analysis tool produces a wrong warning, but the line number and file name matches the line number and file name of a positive variation, we consider the variation detected. In our experience of glancing over the results, this weakness does not affect the results greatly.

Another possible weakness in our analysis is that a tool could produce a valid warning for a positive variation, but not at the exact line number annotated in the test suite.

For example, a tool could report a memory leak at the end of the function where the memory was allocated, while the test suite contains an annotation for the memory leak at the allocation site. These differences in the "philosophy" of the tool versus the "philosophy" of the test suite can be overcome by a manual inspection of the results, but this can prove too costly (for example, the tools generate around 15000 warnings altogether; the warnings are distributed evenly among the tools). In our experience of glancing over the results, this is not an issue at all in some defect types (such as numerical errors), but more of an issue in more complicated defect types, such as resource management.

Another instance of this possible issue is represented by data races, where the tool could report any of the (two or more) sites at which the data race occurs. Currently, we annotate only the first place in the file where the data race occurs and therefore tools that report the second occurence are disadvantaged.

Future work. We propose for future work the following open problem: how to automatically determine whether a tool finds a variation or not in a more robust way. It is currently possible to easily add static analysis tools to the harness, by simply writing a parser for the tool output, and therefore a possible direction for future work is to compare other tools, including commercial tools. A weakness of the ITC suite is that there are just 4 test cases for exercising C++ code. Therefore, an obvious direction for future work is to expand the suite with C++-specific tests. Also, it should be possible to map defect (sub)types in the ITC suite with CWE error numbers in order to present the results in a more standardized manner. Of course, the main open problem is to improve the existing analyzers or write a new analyzer that outperforms existing analyzers.

Acknowledgment

We thank to the anonymous reviewers for their meaningful comments and for suggesting the use of the uniqueness metric. They helped significantly improve the quality of the paper. This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI UEFISCDI, project number PN-III-P2-2.1-BG-2016-0394, within PNCDI III.

References

[1] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *ISSREW 2015*. IEEE Computer Society, 2015, pp. 12–15.

[2] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Syst. J.*, vol. 46, no. 2, pp. 265–288, Apr. 2007.

[3] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[4] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.

[5] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 97–106, Oct. 2004.

[6] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "Begbunch: Benchmarking for c bug detection tools," in *International Workshop on Defects in Large Software Systems*. New York, NY, USA: ACM, 2009, pp. 16–20.

[7] CWE, "Common weakness enumeration," 2017. [Online]. Available: https://cwe.mitre.org/

[8] J.-C. Filliâtre, "Deductive software verification," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, p. 397, Aug 2011. [Online]. Available: https://doi.org/10.1007/s10009-011-0211-0

[9] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, "Lclint: A tool for using specifications to check code," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 5, pp. 87–96, Dec. 1994.

[10] D. Marjamäki, "Cppcheck - a tool for static c/c++ code analysis," 2017. [Online]. Available: http://cppcheck.wiki.sourceforge.net/

[11] G. J. Holzmann, "Uno: Static source code checking for user defined properties," in *In 6th World Conf. on Integrated Design and Process Technology, IDPT '02*, 2002.

[12] D. Wheeler, "Flawfinder," 2017. [Online]. Available: https://www.dwheeler.com/flawfinder/

[13] O. Contributors, "Oclint," 2017. [Online]. Available: http://oclint.org/

[14] C. Li, "Sparse," 2017. [Online]. Available: https://sparse.wiki.kernel.org/index.php/Main_Page/

[15] F. Contributors, "Flint++," 2017. [Online]. Available: https://github.com/L2Program/FlintPlusPlus

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *PLDI 2003*. ACM, 2003, pp. 196–207.

[17] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL 1977*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[18] GrammaTech, "Codesonar," 2017, (retrieved March, 2017). [Online]. Available: https://www.grammatech.com/products/codesonar

[19] Synopsys, "Coverity," 2017. [Online]. Available: https://scan.coverity.com/

[20] Clang., "Clang Static Analyzer," 2017, (retrieved March, 2017). [Online]. Available: https://clang-analyzer.llvm.org/

[21] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, 2015.

[22] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *IFM 2004*. Springer Berlin Heidelberg, 2004, pp. 1–20.

[23] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NFM 2015*. Springer International Publishing, 2015, pp. 3–11.

[24] CoderGears, "CppDepend," 2017. [Online]. Available: http://www.cppdepend.com/

[25] ITC, "Static Analysis benchmark," 2016. [Online]. Available: https://samate.nist.gov/SARD/view.php?tsID=104

[26] D. Guth, C. Hathhorn, M. Saxena, and G. Rosu, "Rv-match: Practical semantics-based program analysis," in *CAV 2016*, ser. LNCS, vol. 9779. Springer, July 2016, pp. 447–453.

[27] OWASP, "Owasp benchmark," 2017. [Online]. Available: https://www.owasp.org/index.php/Benchmark

[28] M. Mantere, I. Uusitalo, and J. Röning, "Comparison of static code analysis tools," in *SECURWARE 2009*. IEEE Computer Society, 2009, pp. 15–22.

[29] H. K. Brar and P. J. Kaur, "Comparing detection ratio of three static analysis tools," *International Journal of Computer Applications*, vol. 124, no. 13, pp. 35–40, August 2015.

[30] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electr. Notes Theor. Comput. Sci.*, vol. 217, pp. 5–21, 2008.

[31] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders - test and measurement of static code analyzers," in *COUFLESS 2015*. IEEE Press, 2015, pp. 14–20.