A Comparison of 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,dorel.lucanu@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 have so far benchmarked several existing static analysis tools for C/C++ against the Toyota ITC test suite in order to determine which tools are best suited to our purpose. We discuss and compare several quality indicators such as precision, recall and running time of the tools. We analyze which tools perform best for various categories of potential vulnerabilities such as buffer overflows, integer overflow, etc.

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;
- if several tools are used, then it is likely that each of them uses a different vulnerability taxonomy;
- the vulnerability taxonomy used by the company is different from that used by the tool.

All these issues are in fact challenges for the design of a customized static code analysis tool.

This paper reports our experience on comparing existing static code analysis tools on a set of vulnerabilties that BitDefender is interested in.

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. [1]), they do not focus on vulnerability detection (e.g. [2], [3]), they only concentrate on subsets of vulnerabilities (e.g. [4]) which do not include all vulnerabilities that we are interested in, or the associated artifacts do not seem to be available anymore (e.g. [5]).

Therefore we started to develop our own framework for running and comparing static analysis tools. Since the main language used by the company is C/C++, our focus is on the tools that support C/C++. There are several challenges that we address for this comparison:

1) how to classify vulnerabilities; 2) how to assign vulnerabilities to the appropriate class; 3) how to decide if two vulnerabilities detected by two different tools are in fact same; 4) what kind of statistics are helpful or useful; 5) finding illustrative annotated test benchmarks; 6) finding a unitary reporting system for all tools;

For classifying vulnerabilities we use the Common Weakness Enumeration (CWE$^{\text{TM}}$) list [6], which was created by the MITRE Corporation to be used as a common taxonomy for software weaknesses. The class of weaknesses covered by the CWE list is extensive enough, as it includes flaws, faults, bugs, vulnerabilities, and other errors in software implementation, code, design, or architecture. We focus on vulnerabilities discussed in Section 5.

The CWE list is organized in a hierarchical fashion, each weakness having an ID. For example, the weakness with ID 190 (CWE-190: Integer Overflow or Wraparound) is a child of weakness CWE-682: Incorrect Calculation, which itself is part of several categories, including CWE-189: Numeric Errors and CWE-738: CERT C Secure Coding Section 04 - Integers (INT). The CWE hierachy is therefore organized as a DAG.

We choose to benchmark the static analyis tools against the Toyota ITC test suite [7], which is part of the *Software Assurance Reference Dataset (SARD)* [8]. SARD is itself a component of the Software Assurance Metrics And Tool Evaluation (SAMATE) project, developed by the *National Institute of Standards and Technology (NIST)*. We choose this benchmark since it has references to the CWE taxonomy and covers a large class of C/C++ vulnerabilities.

The Toyota ITC test suite contains several test cases. Each test case consists of a set of C/C++ files that expose various weaknesses. The testcases cover a significant number of weaknesses from the CWE list. The testsuite contains an XML manifest file, which documents the weaknesses of each test case. For each weakness, the manifest file contains the file name, the line number and the CWE id of the weakness. We show a fragment of the manifest file in the next section. Additionally, some C/C++ files are annotated with comments stating that a particular line should not be reported as a vulnerability. Altogether, there are 1915 vulnerabilities spread over 101 test cases.

Our framework runs a set of static analyzers over all test cases. For each test case, we automatically analyze whether the corresponding vulnerabilities in the manifest file are detected by the static analysis tools and we produce a report with various statistics such as the precision (the ratio between the true positives and the total number of defects reported by the tool), recall (the ratio between the true positives and the total number of defects in the source code) and running time for each tool and category of vulnerabilities. The correspondence between the defects reported by the tool and the defects in the ITC test suite is performed based on the line number in the source code. This allows us to handle static analysis tools that use different weakness taxonomies, with a small risk of misclassifying some defects.

In order to increase our confidence in the results, we inspect manually all results for two tools that are of interest to us. The manual inspection shows that our automatic classification is very precise, but it also reveals several bugs in the Toyota ITC test suite and a few imprecisions in the static analysis tools that we describe in this paper and that we will report to the developers.

The current version of our framework runs the following static analysis tools: Code Sonar [9], Clang Static Analyzer [10] (core and alpha checkers), CppCheck [11], Facebook Infer [12], Splint [13] (standard and weak). The core checkers of Clang are the stable checkers enabled by default in the tool. The alpha checkers are unstable and can be enabled using command line options. Splint has a standard mode, with all the warnings enabled, and a weak mode where the checking parameters are set such that it reports less warnings. It is possible to easily add other static code analysis tools and run them on the same benchmarks, but the output needs manual inspection, which could be costly.

Section 2 contains an overview the main tools and approaches related to our work. Section 3 presents the current status of our framework. Section 4 shows statistics that we generate with the proposed framework, which automatically performs the correspondence between the defects reported by the tool and the defects in the ITC test suite. Section 5 presents the results of the manual inspection of the result for two of the tools that are of particular interest to us. Section 6 points out the main features that we will investigate and implement in 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 [14].

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 [13], can check C programs for security vulnerabilities and some other mistakes. The open source CppCheck [11] features several analyses for C++ code such as bounds checking.

On the commercial side, we have looked at several tools. Astrée [15] is a static analysis tool based on abstract interpretation [16] 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 [9], an interprocedural analyzer that can find buffer overflows, memory leaks and others. Coverity (and in particular Coverity Security Advisor) [17] 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.

Of the open source tools, Clang [10] features a static analyzer based on interprocedural data-flow analysis. Frama-C [18] 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. It is possible to use Frama-C to analyze C++ code with the early stage framac-clang plugin, which works by translating C++ into C, using the clang frontend.

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 [19] 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 [12] from Facebook, which uses bi-abduction to perform interprocedural analysis. Static analysis tools such as Cppdepend [20] concentrate on code metrics and visualizations.

The Toyota ITC benchmark [21] 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 [7] compares CodeSonar (GrammaTech) and Code Prover/Bug Finder (MathWorks). The BugBench benchmark paper [3] 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 [22], were benchmarked against the Toyota ITC tests. The OWASP project contains the OWASP benchmark [23], which is a test suite with vulnerabilities in Java code.

The BegBunch [5] 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 [3] or Zitser [4]. Unfortunately, the benchmark does not seem to be available for download anymore. The main differences to BegBunch are that we concentrate on vulnerabilities (and not general bugs) and we produce reports that are more fine grained. Moreover, we manually inspect the output of the tools in order to increase our confidence in the reports. The manual inspection reveals some bugs in the Toyota ITC benchmark and some imprecisions in some of the static analysis tools.

3. Our Framework

One of the tasks in our joint research project between Bitdefender and the Alexandru Ioan Cuza University is to compare the existing static code analysis tools. Besides the ad-hoc comparisons that we can find on the internet, there are also several research papers that address the same topic. Most of them do comparisons using various criteria (e.g., comparing the performance of the tools based on annotated code vs. tools that do not need annotations [24], detection ratio [25], etc.). Others (e.g., [26], [27]) include extensive studies on the functionality provided by the tools, or exhaustive analyses and statistics over large test suites. Although these results are indeed useful to us, we have to consider different criteria driven by the needs of the company. An example would be to find which analysis tools perform better when it comes about detecting exploitable vulnerabilities (e.g., buffer overflow), and, at the same time, finish the analysis faster. Thus, we need a comparison of the existing static analysis tools.

To achieve this, we developed a framework that can be configured to run various static analysers over the Toyota ITC test suite. It takes as input a list of static analysis tools and a test suite and runs the tools on the test suite; it outputs the results in human readable format.

To avoid reinventing the wheel, we opted for the SARD standard format for test suites. Each test suite includes source files and an XML *manifest* which contains information about the test cases. A sample manifest entry for the 199231 test case (chosen arbitrarily) is shown below, where we use "`...`" to omit irrelevant details:

```
<testcase id="199231"
          type="Source Code"
          instruction="gcc -Wall ... "  ...>
  <description>
        ...
        Defect Code to identify bit shift errors
  </description>
    ...
  <file path="000/199/231/bit_shift_main.c"
          language="C" ... />
  <file path="000/199/231/bit_shift.c"
         language="C" ... >
     <flaw line="236"
             name="CWE-189: Numeric Errors"/>
      ...
   </file>
    ...
</testcase>
```

The relevant lines from `bit_shift.c` are:

```
233: void bit_shift_017 ()
234: {
235:   int ret;
236:   ret = 1 << 32; /*ERROR:Bit shift error*/
237:   sink = ret;
238: }
```

For each test case, the manifest specifies the files involved, the command line to use for compiling the test case, a short description, and some other information[1]. Also, for each file, the manifest includes the numbers of the lines containing errors/warnings, together with references to the CWE list. This information is useful for classifying errors (e.g., to determine which of them are exploitable vulnerabilities).

Our framework processes each test case as follows: firstly, it generates a temporary directory containing all the required source files; secondly, it runs the static analyzers over the test case files and stores the results into internal data structures; thirdly, it traverses the data structures and outputs a report.

Since the analyzers use different formats for their outputs, our tool converts the outputs of the analyzers into a unique format. In order to add a new analyzer, such a conversion component must be added to the framework.

By default, the tool generates a report containing statistics for each test-case and each CWE vulnerability: the total number of errors included in the manifest file, the number of errors from the manifest file detected by each analyzer, and the number of errors not included in the manifest file but detected by each tool (these "extra" errors could be false positives or simply true errors that are not specified in the manifest file).

4. Experiments

To exhibit the functionality of the tool, we have chosen for comparison the following static analysis tools: CppCheck, two versions of the Clang Static Analyzer (one with only the *core* checkers enabled and one with the less mature *alpha* checkers), GrammaTech CodeSonar, two versions of Splint (*weak* mode and *standard* mode) and Facebook Infer. We are currently in the process of integrating other static analysis tools as well.

1. Readers can consult the manifest file for the test suite available at [21] for details.

4.1. Summary

We summarize the results in Table 1. For each analyzer, we list the tool version, the number of detected errors, the number of “extra” errors, and the analysis time for the entire testsuite (101 testcases). Currently, we do not weigh errors by importance.

The total number of bugs included in the manifest is 1915. The second line in Table 1 contains the number of bugs detected by the corresponding tool. To decide whether a particular weakness is detected by a tool, we currently rely just on the line number. That is, if the manifest specifies that a weakness is present at line L and the tool reports an error at line L, we assume that it found the right one. This could lead to misclassification because the tool could miss the real error specified in the manifest and output another one at the same line. Our manual inspection process in Section 5 suggests that the number of such misclassified bugs is minimal.

The second line in Table 1 contains the number of errors reported by the tool which do not appear in the manifest. We call these “extra” errors. Given the maturity of the benchmark, we expect that most “extra” errors are false positives, but this has to be manually checked. The tool does not yet detect automatically if the error reported by a static analyzer, at a given line, falls in the same category as the corresponding error in the manifest. However, it can generate a spreadsheet containing all errors detected by a tool, together with the corresponding errors in the manifest (if they exist) and the source code of the test case, which facilitates manual inspection of the tool output.

We have also timed the execution of the static analysis tools on the entire benchmark. The running times appear in the fourth line in Table 1. The lightweight tools (CppCheck, Splint) are quite fast. On the other side, the tools based on symbolic execution (Clang, CodeSonar) are the slowest, followed by Facebook Infer which is a little faster.

All tests have been performed on an computer equipped with 8GB RAM and an 2.9 GHz i3 processor with 2 cores and hyperthreading.

4.2. Results by Category

We have also analyzed how the tools perform on the various categories of vulnerabilities in the CWE list. As the CWE list contains thousands of vulnerabilities, we have chosen some *views* of the CWE list, which contain vulnerabilities that we are interested in. A CWE *view* is simply a subset of the DAG-like CWE hierarchy, which includes several categories of weaknesses based on some criteria. An example of such a view is the *CWE/SANS Top 25 Most Dangerous Software Errors*[2]. This view covers critical software errors that can lead to serious vulnerabilities in software.

The results of running the analysis tools over *CWE/SANS Top 25 Most Dangerous Software Errors* are shown in Table 2. For each weakness category and for each tool we show how many weaknesses included in that category were detected by the tool w.r.t. the total number of weaknesses in that category reported in the manifest. For example, CodeSonar finds 267 weaknesses (out of 652 reported in the manifest) in the *CWE-867:2011 Top 25 - Weaknesses On the Cusp* category. Splint standard detects 69 out of 89 weaknesses in category *CWE-681: Incorrect Conversion between Numeric Types*, while its weak version finds only 45.

Table 2 consists only of the categories in the view *CWE/SANS Top 25 Most Dangerous Software Errors* that have at least an instance in the ITC benchmark.

Two other views that we are interested in are the *CERT C* and *CERT C++* Secure Coding Standards views. These standards contain rules for secure coding in the C/C++ programming languages. The rules serve as guidelines for developers and at the same time they define a set of best practices. The corresponding CWE view includes weaknesses addressed by rules in the CERT C/C++ Secure Coding Standard. We analysed the Toyota ITC benchmark with respect to weaknesses occuring in these views and we show the results in Table 3.

Note that Tables 2 and 3 only show a partial picture of the problem, since we do not currently calculate the number of “extra” errors reported by the tools. This calculation would require inspecting the error messages of the tool and determining if it fits a particular category and it seems difficult to perform automatically. We report on our manual inspection of the “extra” errors in Section 5.

5. Discussion

The data in Tables 1, 2 and 3 are obtained automatically by our scripts. We count the number of weaknesses in the testsuite manifest that are detected by a tool by analyzing the output of the tool and comparing line numbers: if the manifest states that a weakness is present at line L and the tool reports a potential bug at line L, then we consider that the tool found the particular weakness. In particular, we do not correlate in any way the weakness name from the manifest with the error string of the tool. Obviously, this is potentially imprecise, because the tool could report, say, a division by zero error at line L and the manifest could state that line L contains a buffer overflow.

As the goal of our project is to produce a customized static analysis solution for the partner company, we have decided to manually inspect the output of the CppCheck tool and of the Clang Static Analyzer, which seem to be the most promising among the open source tools, having reasonable recall and precision ratios, based on results in Table 1.

The other open source tools (Splint weak, Splint standard and Clang alpha) seem to find very few bugs, but also report many extra errors (likely false positives).

Therefore, we have manually analyzed the output of CppCheck and of the Clang Static Analyzer (core) to determine whether misclassification errors occured, but also to better understand their behavior. We have also manually checked all errors reported by these two tools at line numbers for which the manifest does not contain any information. We initially expected that these extra errors reported by the tool fall in the following category:

1) false positives;
2) irrelevant to our case study because they do not represent vulnerabilities;

2. We use here the latest version of this view which was released in 2011.

	CppCheck	**Splint (weak)**	**Splint (standard)**	**Clang (core)**	**Clang (alpha)**	**CodeSonar**	**Facebook Infer**
Tool version	1.72	3.1.1	3.1.1	5.0.0	5.0.0	4.4p0	v0.12.0
Detected	132/1915	79/1915	344/1915	125/1915	326/1915	651/1915	149/1915
Extra	42	229	2809	125	1406	649	39
Total analysis time (seconds)	5.80	2.33	1.88	293.62	580.84	1346.86	159.52

TABLE 1. SUMMARY OF FINDINGS.

Category	**CppCheck**	**Clang core**	**Clang alpha**	**CodeSonar**	**Facebook Infer**	**Splint standard**	**Splint weak**
CWE-867:2011 Top 25 - Weaknesses On the Cusp	32/652	41/652	77/652	267/652	129/652	194/652	57/652
CWE-476:NULL Pointer Dereference	12/353	40/353	36/353	195/353	108/353	99/353	7/353
CWE-681:Incorrect Conversion between Numeric Types	0/89	1/89	27/89	35/89	0/89	69/89	45/89
CWE-865:2011 Top 25 - Risky Resource Management	10/85	3/85	12/85	12/85	0/85	8/85	4/85
CWE-190:Integer Overflow or Wraparound	10/63	1/63	12/63	4/63	0/63	8/63	4/63
CWE-825:Expired Pointer Dereference	11/58	0/58	5/58	24/58	12/58	7/58	0/58
CWE-772:Missing Release of Resource after Effective Lifetime	9/53	0/53	1/53	2/53	9/53	14/53	2/53
CWE-770:Allocation of Resources Without Limits or Throttling	0/33	0/33	4/33	2/33	0/33	4/33	2/33
CWE-754:Improper Check for Unusual or Exceptional Conditions	0/32	0/32	2/32	6/32	0/32	1/32	1/32
CWE-362:Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	0/31	0/31	0/31	0/31	0/31	0/31	0/31
CWE-131:Incorrect Calculation of Buffer Size	0/22	2/22	0/22	8/22	0/22	0/22	0/22
CWE-330:Use of Insufficiently Random Values	0/2	0/2	2/2	2/2	0/2	0/2	0/2
CWE-822:Untrusted Pointer Dereference	0/1	0/1	0/1	1/1	0/1	0/1	0/1

TABLE 2. DETECTION BY CATEGORIES OF WEAKNESSES INCLUDED IN THE CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS VIEW.

CERT C Secure Coding Sections	**CppCheck**	**Clang core**	**Clang alpha**	**Codesonar**	**Facebook Infer**	**Splint standard**	**Splint weak**
CWE-742: Sec. 08 - Memory Management (MEM)	98/1101	86/1101	154/1101	412/1101	138/1101	202/1101	24/1101
CWE-747: Sec. 49 - Miscellaneous (MSC)	76/937	32/937	188/937	285/937	14/937	165/937	61/937
CWE-738: Sec. 04 - Integers (INT)	87/739	44/739	146/739	219/739	14/739	157/739	59739
CWE-741: Sec. 07 - Characters and Strings (STR)	66/656	30/656	145/656	205/656	15/656	143/656	54/656
CWE-743: Sec. 09 - Input Output (FIO)	79/624	29/624	106/624	179/624	38/624	88/624	10/624
CWE-746: Sec. 12 - Error Handling (ERR)	76/564	28/564	111/564	143/564	14/564	83/564	14/564
CWE-740: Sec. 06 - Arrays (ARR)	72/564	40/564	103/564	157/564	15/564	91/564	12/564
CWE-737: Sec. 03 - Expressions (EXP)	12/553	44/553	87/553	269/553	109/553	172/553	53/553
CWE-744: Sec. 10 - Environment (ENV)	66/468	27/468	95/468	137/468	14/468	71/468	8/468
CWE-739: Sec. 05 - Floating Point (FLP)	21/262	19/262	47/262	96/262	1/262	82/262	49/262
CWE-748: Sec. 50 - POSIX (POS)	2/107	3/107	29/107	32/107	1/107	1/107	1/107
CWE-745: Sec. 11 - Signals (SIG)	0/76	0/76	29/76	16/76	0/76	0/76	0/76
CWE-736: Sec. 02 - Declarations and Initialization (DCL)	0/34	3/34	3/34	24/34	1/34	0/34	0/34

CERT C++ Secure Coding Sections	**CppCheck**	**Clang core**	**Clang alpha**	**Codesonar**	**Facebook Infer**	**Splint standard**	**Splint weak**
CWE-876:Sec. 08 - Memory Management (MEM)	107/1120	83/1120	152/1120	390/1120	146/1120	216/1120	26/1120
CWE-883:Sec. 49 - Miscellaneous (MSC)	76/937	32/937	188/937	285/937	14/937	165/937	61/937
CWE-872:Sec. 04 - Integers (INT)	87/739	44/739	146/739	219/739	14/739	157/739	59/739
CWE-875:Sec. 07 - Characters and Strings (STR)	66/656	30/656	145/656	205/656	15/656	143/656	54/656
CWE-877:Sec. 09 - Input Output (FIO)	79/631	27/631	108/631	163/631	37/631	92/631	12/631
CWE-874:Sec. 06 - Arrays and the STL (ARR)	72/564	40/564	103/564	157/564	15/564	91/564	12/564
CWE-878:Sec. 10 - Environment (ENV)	66/468	27/468	95/468	137/468	14/468	71/468	8/468
CWE-871:Sec. 03 - Expressions (EXP)	12/357	42/357	38/357	197/357	108/357	99/357	7/357
CWE-873:Sec. 05 - Floating Point Arithmetic (FLP)	21/262	19/262	47/262	96/262	1/262	82/262	49/262
CWE-882:Sec. 14 - Concurrency (CON)	13/116	0/116	1/116	20/116	23/116	17/116	2/116
CWE-879:Sec. 11 - Signals (SIG)	0/76	0/76	29/76	16/76	0/76	0/76	0/76
CWE-880:Sec. 12 - Exceptions and Error Handling (ERR)	0/32	0/32	2/32	6/32	0/32	1/32	1/32

TABLE 3. DETECTION BY CATEGORIES OF WEAKNESSES OF THE C/C++ CERT SECURE CODING STANDARD VIEW.

3) or they could mean that the testsuite contains itself a bug.

By manually inspecting the output of the tools, we have found an additional possibility: the tool reports an error that appears in the manifest, but the tool reports it at a different point (e.g. the tool reports a memory leak at the end of a function, while the manifest contains information about the memory leak at the allocation point).

All in all, we have manually analyzed 174 errors reported by CppCheck (out of which 132 lines correspond to a weakness in the manifest) and 250 errors reported by the Clang Static Analyzer (out of which 125 correspond to a weakness in the manifest). We report our findings below.

5.1. CppCheck

We have found that all 132 errors that CppCheck reports at line numbers included in the manifest are accurate, in the sense that the error message produced by CppCheck describes accurately the weakness specified in the manifest. In addition, the CppCheck error messages are oftentimes more precise than what is specified in the manifest.

For example, the following two examples contain an occurence of weakness CWE-562: Return of Stack Variable Address.

```
void return_local_002_func_001 (int **pp)
{
 int buf[5];
 *pp = buf;
}
```

```
int* return_local_001_func_001 ()
{
 int buf[5];
 return buf;
}
```

The weakness refers to the fact that the address of a local variable escapes the context of the corresponding function call. This is a potential vulnerability because the local variables are only valid during the function call. After the function ends, the address that escapes points to an uncontrolled place in the stack. Writing at that address would smash the stack. Note that although in both examples the address of a local variables escapes the function, it happens in a different way: in the first example, the address of the local variable is written in some output parameter, while in the second example the address of the local variable is returned directly. CppCheck reports in the first case that "Address of local auto-variable assigned to a function parameter.", while in the second case that "Pointer to local array variable returned." The same precision gain happens with other weaknesses as well. For example, in the case of weaknesses CWE-824: Access of Uninitialized Pointer or CWE-457: Use of Uninitialized Variable, CppCheck also reports which is the variable/pointer that was not initialized.

We have found that 12 out of the 42 "extra" errors (errors reported by CppCheck at line numbers that are not specified in the manifest as containing errors) are false positives. This means that CppCheck wrongly reports a weakness. We have found that most of these false positives are caused by an imprecision in the implementation of CppCheck that causes infeasible paths to be considered. For example, consider the following problematic code, extracted from the testsuite:

```
 if(staticflag == 10)
  b = ...;
 else
  a = dptr[1][1];
 printf("%d",a);
```

CppCheck reports that variable "a" is not initialized, despite the fact that `staticflag` is a global constant different from `10` and therefore the second branch of the if statement is guaranteed to be taken and initialize "a". Therefore, we consider that the precision of CppCheck would greatly increase by making it path-sensitive.

The rest of the 30 "extra" errors (errors reported by CppCheck at line numbers that are not specified in the manifest as containing errors) are true positives, meaning that they represent actual errors.

An interesting true positive, which reflects an important feature of CppCheck, occurs in the `st_cross_thread_access.c` testcase. The relevant lines are the following:

```
#if !defined(CHECKER_POLYSPACE)
 if (st_cross_thread_access_005_thread_set != NO_THREAD)
 {
  ;
 }
 else
 {
  ...;
#endif /* defined(CHECKER_POLYSPACE) */
 }
```

Note that the closing brace of the else branch is outside of the `ifdef`. The macro `CHECKER_POLYSPACE` is not defined in the testcase and therefore the code above compiles succesfully. However, CppCheck analyzes the code for all possible definitions of macros and it therefore reports the error "Invalid number of character '{' when these macros are defined: 'CHECKER_POLYSPACE'." We believe that these closing braces should be fixed in the testsuite.

Another class of errors (10 occurences) that are detected by CppCheck but are not specified in the manifest is represented by uninitialized variables. The relevant lines from one such example are the following:

```
 st_overflow_005_s_001 s;
 st_overflow_005_func_001(s, 10);
```

The variable `s` is a `struct` that is not initialized and which is passed as a parameter to a function. The function does not read `s` at all. CppCheck reports at the function call the fact that `s` is not initialized. The manifest follows a different philosophy: because `s` is not read by the function that was called, it does not matter that it was not initialized.

Several other true positives occur because of a difference in philosophy between the testsuite and CppCheck. Consider the following (minified) example:

```
void uninit_pointer_011 ()
{
 unsigned int * ptr,a=0;
 ptr = (unsigned int *)malloc(10*sizeof(unsigned int *));
 int i;
 if (ptr!=NULL)
 {
  for(i=0; i<10/2; i++)
   ptr[i] = i;
 }
 for(i=0; i<10; i++)
 {
  a += ptr[i]; /*ERROR:Uninitialized pointer*/
 }
}
```

The CppCheck reports that there is a memory leak, since `ptr` is not deallocated at the end of the function. The testsuite does not specify the memory leak, since there is another weakness (uninitialized pointer) which occurs on the code path.

Finally, CppCheck reports an error in the following code (shown minified here):

```
void f(wrong_arguments_func_pointer_012_s_001 st,
      wrong_arguments_func_pointer_012_s_001* st1)
{
    int temp;
    int i=0;
    memset(st1, 0, sizeof(*st1));
    for (i = 0; i < MAX; i++)
    {
     st.arr[i] = i;
     st1->arr[i] = st.arr[i]+i;
     temp += st.arr[i];
    }
}
```

This testcase illustrates another vulnerability, but the line `temp += st.arr[i];` contains a read of the variable `temp`, which was not initialized. This weakness is not specified in the manifest of the testsuite, but is reported by CppCheck.

We have also found that the "extra" errors that CppCheck reports contain three instances of weaknesses which are acknowledged in the testsuite as comments in the C++ code, but these weaknesses do not appear in the manifest file. This is obviously a deficiency in the testsuite.

5.2. Clang Static Analyzer (core checkers)

Of the 125 weaknesses in the testsuite that the Clang Static Analyzer detects (based on line numbers), we found that 6 weakness are wrongly categorized, because the error reported by Clang at the corresponding line number does not match the weakness in the testsuite manifest. The rest of the 119 weaknesses are correctly counted.

Of the six problematic cases, we have found that three are false positives (clang reports an error that does not exist at that line), which means these are instances of a deficiency in the Clang Static Analyzer. However, in the other three cases, it reports true bugs, different from the bugs specified in the testsuite manifest. Therefore the manifest file should be enriched with these potential vulnerabilities.

Here are the relevant lines of such an example:

```
int flag=10;
(flag == 10) ?
    (ptr = (int*) malloc(10*sizeof(int))) :
    (a = 5);
```

The testsuite manifest specifies that a tool should report a CWE-561: Dead Code error on the second line (because the assignment `a = 5` is never executed). However, the Clang Static Analyzer reports a type mismatch between the second (of type `int *`) and third (of type `int`) arguments of the ternary operator.

Another example (minified) is also quite interesting:

```
typedef struct {
 int a;
 int b;
 int c;
} ll;
ll *llg;
void littlemem_st_004 ()
{
 char buf[10];
 llg = (ll *)buf;
 lgg->c = 1;
}
```

A cast is used to convert a local 10-byte buffer to a (presumably) 12-byte structure. Writing to the third element of the structure will therefore have the effect of writing past `buf`, thereby smashing the stack. Unfortunately, the Clang Static Analyzer cannot detect this error, but it detects another: a pointer to the local variable `buf` escapes the function call through the global variable `llg`, which can potentially lead to a vulnerability. We believe this weakness should be added to the manifest of the testsuite.

Of the 125 "extra" errors reported by the Clang Static Analyzer, we have found that 113 errors are irrelevant to our study, in the sense that they do not represent weaknesses. For example, most of these irrelevant errors are of the form `pthread.h:743: warning: declaration of built-in function '__sigsetjmp' requires inclusion of the header <setjmp.h>`, which is simply a consequence of our system configuration. The other type of irrelevant errors are of the form `underrun_st.c:30: note: array 'buf' declared here`, meaning that they are not true errors, but clarifications of previous error messages output by the tool.

Of the other 12 "extra" errors, 8 are true positives, 2 resulted in compilation errors on our system and 2 are false positives. Here is an interesting (minified) example of a true positive, which illustrates an imprecision in the testsuite manifest:

```
int i = 0;
char** doubleptr=(char**) malloc(10* sizeof(char*));

for (i=0;i<2;i++)
{
 doubleptr[i]=(char*) malloc(10*sizeof(char));
 if(doubleptr[i]!=NULL)
  doubleptr[0][0]='T';
}
```

The Clang Static Analyzer reports that the line `doubleptr[0][0] = 'T';` contains potentially a NULL pointer dereferencing. This can occur when the malloc fails at the first iteration but succeeds at the second iteration. However, this weakness does not appear in the manifest (the testcase illustrates other weaknesses).

6. Interpretation and Future work

As is apparent in Table 1, CodeSonar finds the greatest number of weakness in the testsuite, but it also finds the most "extra" errors and its running time is the greatest in the testsuite. The "extra" errors need to be manually inspected to decide whether they are relevant or not.

CppCheck and Splint are the fastest tools, but at the cost of some imprecisions, some of which have been discussed in the manual analysis of the CppCheck tools in Section 5.1. The Clang Static Analyzer (core checkers) offers a good tradeoff. The manual inspection of the "extra" errors revealed that most of them are warnings related to our system configuration (e.g. double declarations due to the standard library inclusion order) or simply extra information on other errors (e.g. lines where uninitialized variables were declared). We still need to manually inspect the output of the remaining tools. At first sight, Facebook's Infer seems to be very good at catching memory issues such as NULL pointer dereferencing or double frees.

In the future, we will expand our comparison with other static analysis tools for which at least an evaluation version is available. The architecture of our framework allows to easily add other static analysis tools to the comparison, as all that is needed is a function to parse the output of the static analyzer.

Currently, we have manually inspected only the results reported by CppCheck and Clang. We plan to extend our manual inspection to the results reported by other tools as well, to see if there are bugs relevant for BitDefender and not found by the others. We will also improve our analysis by performing additional statistics, e.g., the number of vulnerabilities that can only be found by a given tool, and by formalising the subtle difference between false-positives and irrelevant "extra" errors.

In addition to expanding the set of tools that we compare, we plan to support test suites other than Toyota ITC as well.

We will present these results to the partner company and based on the results, we will choose an appropriate open-source static analysis tool to customize for the needs of Bitdefender.

Acknowledgment

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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] NIST-SARD, "Software Assurance Reference Dataset (SARD) Testsuites," https://samate.nist.gov/SARD/testsuite.php, 2017.

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

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

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

[12] 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.

[13] 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.

[14] 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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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

[22] 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.

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

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

[25] 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.

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

[27] 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.