Test driving static analysis tools in search of C code vulnerabilities

George Chatzieleftheriou
Aristotle University of Thessaloniki
Department of Informatics
Thessaloniki, Greece
e-mail: gchatzie@csd.auth.gr

Panagiotis Katsaros
Aristotle University of Thessaloniki
Department of Informatics
Thessaloniki, Greece
e-mail: katsaros@csd.auth.gr

Abstract— **Recently, a number of full-fledged open source and commercial tools for automated code scanning came in the limelight. Due to the significant costs associated with incorporating these tools in the development process, it is important to know what defects are detected, as well as how accurate and efficient the analysis is. We focus on popular static analysis tools for C code vulnerabilities. Most of the existing benchmarks are based on code obtained from open source projects. Tools are evaluated over a set of actual vulnerabilities, but there is no systematic coverage of all the known or frequent vulnerabilities and the coding complexities in which they arise. We provide a test suite that implements the discussed requirements for the most frequent vulnerabilities from those reported in public catalogues. Four open source and two commercial tools are compared in terms of their effectiveness to correctly detect the selected vulnerabilities. A wide range of C language constructs is taken into account and a series of metrics is computed that provide insight into how the tools balance inherent analysis tradeoffs and their analysis efficiency. The results may be useful for identifying the appropriate solution for a software process, in terms of cost-effectiveness, while the methodology and the test suite may be reused in test drives with similar aims.**

Keywords-static analysis; software security; benchmark tests

I. INTRODUCTION

Static program analysis belongs to the class of problems that are *undecidable* [1]. In practice, it is implemented as an *approximation of the program's behavior* that inevitably sets limitations to the analysis capacity in correctly detecting existing code defects. Important considerations that are taken into account are: (i) the used programming language, (ii) the defects that can be detected, (iii) the analysis effectiveness, roughly given as the proportion of detected real defects and (iv) the analysis efficiency that affects the needed computing resources for code scanning.

A static analysis tool is adequate for a software project, if there is evidence that it is effective in capturing all defects considered critical for the required product quality, while it is sufficiently efficient for the size of code base to be analyzed. Tool comparisons with results from real projects are useful, but they place an evaluation bias in terms of the defects that exist in the actual benchmark, as well as the code complexity and the size of the code base.

Empirical studies with code taken from open source projects should be completed by evaluation results that systematically cover all possible or the most frequent code defects that arise in a product quality context. In this work we focus on software security, a key quality in modern software business. We evaluate four open source and two mainstream commercial static analysis tools for C, which is considered as one of the least secure programming languages. A synthetic test suite is introduced that implements common C code vulnerabilities, selected from public catalogues based on their reported frequency. Code vulnerabilities are replicated in programs with varied analysis requirements, in order to study the tools effectiveness in a wide range of coding complexities that may arise. Tool effectiveness is measured by appropriate metrics that uncover valuable information for how the tools handle the inherent analysis tradeoffs in approximating the programs' behavior. Finally, a performance benchmark provides the basis for evaluating analysis efficiency, in terms of time and memory space.

The obtained results, when combined with a cost-efficiency model [21] are used for identifying the appropriate solution for a software process. In present work, cost-efficiency is not taken into account and the results just serve in drawing valuable conclusions for the tested tools.

We are aware that the issue of reporting a flaw may be not an analysis restriction, but a matter of decision by the tool vendor, whether the flaw may cause problems. It is also true that vendors do not provide information for the possible flaws that are ignored and consequently their products do not provide transparent functionality for the end-user[1]. To this end, benchmark tests that cover a given set of flaws stimulate tool comparisons with respect to a particular code scanning purpose.

The followed methodology is general: it can be applied to quality contexts or software domains with a differentiated

[1] One of the two commercial tools was licensed to the authors for the purpose of this study. We therefore explicitly refer to the product's name, when reporting the obtained results. We also report the measurements for the second commercial tool, without explicitly referring to the product's name until the license terms are clarified.

base of critical code defects, as well as in tool comparisons for other languages. A possible scenario is for example the certification process for approving the level of trust that a mobile application needs to guarantee safe execution in a mobile platform. This certification is a prerequisite for distributing applications through internet-wide markets and the process, which is usually driven by system vendors, concerns the checking of platform specific requirements.

Section II describes in detail the frequent security vulnerabilities that were selected for the purpose of this study. Section III introduces the considered tools and section IV refers to the applied methodology. Results from test driving the tools with the proposed methodology are provided in section V. Measurements for the tool effectiveness are reported in section VI and for their analysis efficiency in section VII. We also consider related work in section VIII and the paper concludes with a brief discussion on the benefits of the proposed test drive and the future research prospects.

II. FREQUENT C CODE VULNERABILITIES

Public catalogues of vulnerabilities report many flaws that undermine the reliability of C programs. Vulnerabilities are reported in numerous records that sometimes overlap. Our test suite includes 30 distinct vulnerabilities selected from the Common Weakness Enumeration (CWE) catalogue [3] and the CERT C Secure Coding Standard [4] according to their frequency of appearance. They are classified into 8 broad categories as shown in Table I, in an attempt to develop a concise and comprehensive taxonomy of the most frequent C code vulnerabilities.

Category *"General"* includes three types of flaws namely, *division by zero*, *use of uninitialized variables* and *null pointer dereference*. According to the 2009 Coverity Scan Open Source Report [5], the latter stands for about 25% of all defects found in scanned open source software. In the second category of Table I we classify all flaws related to the manipulation of integers including *integer overflows*, *sign* and *truncation errors*.

Buffer overflows are the most common and dangerous vulnerabilities in C programs [2]. When exploited by a malicious user, they can cause unpleasant consequences like hacking the system running the program. *Direct overflows*, *off-by-one errors* and *unbounded copies* that appear in categories "*Arrays*" and "*Strings*", along with the *format string vulnerabilities* are accountable for the vast majority of buffer overflows.

TABLE I. FREQUENT C CODE VULNERABILITIES

Categories	Defects	Description
General	***Division by zero***	Divide a value by zero (CWE-369)
	Null pointer dereference	Dereference a pointer that is NULL (CWE-476)
	Uninitialized variables	Use a variable which has not been initialized (CWE-457)
Integers	***Overflow***	An integer is incremented in a value that is too large to store in its internal representation (CWE-190)
	Sign errors	A signed primitive is used as unsigned value (CWE-195)
	Truncation errors	A primitive is cast to a primitive of smaller size (CWE-197)
Arrays	***Direct overflow***	Out-of-bounds access of an array (CWE-119)
	Off-by-one errors	Use a min or max array index which is 1 more or less than the correct value (CWE-193)
	Unbounded copy	Copy array without checking the size (CWE-120)
Strings	***Direct overflow***	Out-of-bounds access of a string (CWE-119)
	Null termination errors	A string is incorrectly terminated (CWE-170)
	Off-by-one errors	Use a min or max string index which is 1 more or less than the correct value (CWE-193)
	Truncation errors	A string is been truncated and possible important information is lost (CWE-222)
	Unbounded copy	Copy string without checking the size (CWE-120)
Format string vulnerabilities		Invalid format-string in printf-like functions (CWE-134)
Memory	***Double free***	Call free() twice in the same memory address (CWE-415)
	Improper allocation	Misuse of functions allocating memory dynamically
	Initialization errors	Not initialize or incorrectly initialize a resource (CWE-665)
	Memory leak	Not release allocated memory (CWE-401)
	Failure check	Not check for failure of functions which are used for dynamic allocation of memory
	Access freed memory	Access memory after it has been freed (CWE-416)
File operations	***Access closed file***	Access a file which has been previousl
	Access in different mode	Access a file in a different mode than the one it has been specified when opening the file
	Double close	Close a file descriptor two times
	Resource leak	Not close a file descriptor (CWE-403)
	Access without open	Access a file without previous trying to open it
	Failure check	Not check for failure of functions which are used for opening a file
Race conditions	***Deadlock (no concurrency)***	Incorrectly manage control flow during execution (CWE-691)
	Deadlock (concurrency)	Insufficient locking and unlocking of a thread (CWE-667)
	Time Of Check, Time Of Use (TOCTOU) errors	Check the state of a resource and try to use it at a later moment based on this invalid info (CWE-367)

String truncation errors are usually introduced, when trying to prevent buffer overflows. Another frequent string manipulation problem is the *null termination errors* caused by misuse of the representation of strings in C.

"*Memory*" allocation and de-allocation flaws include *double free* attempts, *improperly allocated memory*, *initialization errors*, *memory leaks*, *absence of failure checks* and *access in previously freed memory*. "*File operation*" problems are less frequent and cannot be easily exploited in attacks. Our test suite includes instances of *redundant file closure*, *omission of file closure* (resource leak), *absence of failure check* and access in a file that either is *previously closed*, *not opened* or *opened with a different mode*.

"*Race conditions*" are related to erroneous sequence of operations in program execution paths and include the notable cases of *deadlocks* and *time-of-check-time-of-use (TOCTOU) errors*. The latter refer to any access to a program resource based on a mistimed check and can be exploited in the so-called *symlink attacks* [6].

III. TOOLS FOR THE TEST DRIVE

Many static analysis tools emerged as academic projects and some of them were later transformed into commercial products. We focus on full-fledged tools with built-in analyses, which detect most of the mentioned code vulnerabilities without having to annotate the program code. The comparison is not limited to publicly available static analysis tools, because we were also interested to explore the differences with the commercial tools, in terms of the power of the implemented analyses. The selected open source tools include Splint, UNO, Cppcheck and Frama-C. The mature commercial tool in our study was the Parasoft C++ Test. We also provide results for another mature product - without revealing the product name - in order to avoid drawing conclusions that may be biased by the detection capabilities of a single product.

A static analysis is *safe*, when it doesn't miss any flaws. It can be more or less *precise* to the extent that avoids reporting spurious errors. Precise analyses are costly, because they need to detect program paths that cannot be executed and the associated computational cost affects the analysis scalability. Whether an analysis is safe and sufficiently precise depends on three characteristics: *path sensitivity*, *context sensitivity* and *alias analysis*. A path-sensitive analysis excludes infeasible paths. Context sensitivity means that when a function call is processed, the analysis takes into account the calling context. Alias (also called *pointe*r or *points-to*) analysis computes the entities, where the variables point to. The analysis characteristics are not independent. A "sufficiently precise" alias analysis rests on a comprehensive path-sensitive and context-sensitive analysis and vice versa [14].

Splint [7] is a lightweight tool for checking large programs. The tool comes with an annotation language, where the analyst can define attributes for program objects and set allowed range of values to program variables. However, it is not possible to access control-flow and dataflow information like in UNO [8] that in this way supports the development of truly new analyses. In UNO, the provided built-in analyses search for three common C code vulnerabilities: use of uninitialized variables, null-pointer dereference and out-of-bound array indexing.

Cppcheck [9] is an easy-to-use tool that can analyze thousands of lines with precision, which can be tuned. Frama-C [10] combines static analyses embedded into an abstract interpretation framework [11] of value analysis. It implements a plug-in architecture over a kernel that controls the whole analysis and the tool can be configured to different levels of analysis sensitivity. It is also possible to extend the tool by custom plug-ins and user-defined properties written in a behavioral specification language called ACSL.

In Parasoft C++ Test [12], apart from the level of analysis sensitivity, the user can also select the flaws to be checked. The tool provides a graphical interface for expressing patterns of code defects, in order to extend the set of flaws that are already detected.

For all tools, we assess the effectiveness of their built-in analyses in the default sensitivity configuration.

IV. BENCHMARKING METHODOLOGY AND TEST SUITE

The followed benchmarking methodology is now presented, in terms of the set research goals.

- *For each tool, the exact set of vulnerabilities that are detected is found.*
  Code vulnerabilities that cannot be detected are reported, but they do not affect the measurements for the effectiveness of the implemented analyses. Multiple forms of a single vulnerability are examined. For example, a Time Of Check Time Of Use error commonly appears with different pairs of C functions (notable TOCTOU errors are the `access()-fopen()` and `lstat()-remove()` pairs). The selected vulnerabilities are therefore represented in the benchmark by more than 30 test cases. No difference was encountered in the tools response across the different forms of the tested vulnerabilities.
- *The tools analysis sensitivity is systematically assessed with a wide range of language constructs and different conditions of language semantics, under which the vulnerabilities may arise.*
  For each vulnerability 15 test cases require some sort of path sensitive analysis, 7 programs are used for testing context sensitivity and 2 programs require an alias analysis.
  All programs have a line with the tested flaw commented as `/*ERROR*/` and another line commented as `/*SAFE*/` that tests the analysis

capacity to avoid reporting spurious errors, termed as *false positives* (FP). If a truly ERROR line is ignored, a *false negative* (FN) case is encountered. FPs appeared due to lack of path-sensitivity, whereas FNs due to absence of context-sensitivity or alias analysis. Fig. 1 shows the code for a null pointer dereference test case. Our benchmarking test suite includes more than 700 programs like the one shown in the figure, where each program consists from 10 to 100 lines of code.

```
int main()
{
    int x[5]={0,0,0,0,0};

    int *y,*z;
    int m;

    y=x;
    z=0;

    m=y[2];        /*SAFE*/

    m=z[2];        /*ERROR*/

    return 1;
}
```

Figure 1. An example program from our test suite

- *Provide a detailed qualitative characterization of the tools' analysis sensitivity.*
  This result improves the user's awareness, for when to expect a FP or if he can trust a tool that it will not miss an existing flaw. Having observed a consistent tool sensitivity behavior across the different vulnerabilities, we report the FPs and FNs for the different language constructs, when testing a code vulnerability that can be detected by all tools. This yields the most detailed qualitative characterization that is one possible to obtain with our test suite.
- *Provide a thorough quantitative characterization of the tools' analysis effectiveness.*
  Five metrics are used that account for the total number of FPs and FNs encountered over the code vulnerabilities that each tool can detect. These metrics highlight how the tools balance inherent tradeoffs in analysis effectiveness, like whether a tool lays emphasis on detecting as many flaws as possible or if it better suits for detecting actual flaws.
- *Provide results for the tools' analysis efficiency in terms of time and memory space.*
  Aggregate metrics that reflect all types of analysis sensitivity provide results for the computational cost when analyzing programs of different lengths. Our efficiency benchmark includes test cases that impose the same analysis demands across the different program lengths and along the code of each program. Our metrics use equal weights for the analysis demands of the three sensitivity types. They cannot provide representative measurements of the tools' efficiency in all analysis contexts, since a program may need any combination of analysis sensitivities. However, they indicate the relative differences in how the tools balance the analysis effectiveness with the price paid in efficiency.

V. TEST DRIVING ANALYSIS SENSITIVITY

Table II presents the results obtained for the tools capability to detect the code vulnerabilities of our test suite. We use the convention "Commercial Tool B" to refer to the commercial product that we do not name. Splint, Frama-C and Com. B detected all instances of uninitialized variables, while UNO and Parasoft C++ Test caught only particular instances. A frequent error, the null pointer dereference, was not detected by UNO and Cppcheck. On the other hand, Splint ignored the division-by-zero cases.

Regarding the integer flaws, Com. B detected truncation and sign errors, while the latter were also detected by Splint and Parasoft C++ Test. Integer overflows were not caught by any tool.

Overflows in arrays and strings were found by all tools except Splint. Unbounded copies could be detected only for strings and Cppcheck caught all of them. Splint, Parasoft C++ Test and Com. B caught only specific instances, but these tools were the ones that successfully identified the format string vulnerabilities.

Regarding memory errors, we observed a diverse behavior. Most tools detected the double free cases, but memory leaks were caught only by Cppcheck and partially by Splint. The latter was the only tool that identified a form of initialization error with `malloc()`. Improper memory allocation was not detected by any tool, while access to previously freed memory and absence of memory allocation check were found by most tools.

Four tools detected some file operation errors, but UNO and Frama-C ignored all of them. In the category of single-threaded race conditions, deadlocks were invisible for all tools and TOCTOU errors were caught only by the commercial products. In multi-threading programs, Parasoft C++ Test was the only tool that detected deadlocks.

A detailed characterization of the tools' analysis sensitivity is provided in Table III. Splint recognized infeasible paths only in programs with an unconditional change of the execution flow (`goto`, `return` and `exit()` function). UNO extends path-sensitivity to the simple `if-else` and `for` loop statements and when the execution flow depends on some `#define` preprocessor macro.

TABLE II. TOOLS CAPABILITY TO DETECT THE C CODE CULNERABILITIES OF THE TEST SUITE

Categories	Problems		Splint	UNO	Cppcheck	Frama-C	C++ Test	Com. B
General	*Division by zero*		✗	✓	✓	✓	✓	✓
	Null pointer dereference		✓	✗	✗	✓	✓	✓
	Uninitialized variables	*Integers*	✓	✓	✗	✓	✓	✓
		Strings	✓	✗	✗	✓	✗	✓
		Arrays	✓	✗	✗	✓	✓	✓
		Pointers	✓	✗	✗	✓	✓	✓
Integers	*Overflow*		✗	✗	✗	✗	✗	✗
	Sign errors		✓	✗	✗	✗	✓	✓
	Truncation errors		✗	✗	✗	✗	✗	✓
Arrays	*Direct overflow*		✗	✓	✓	✓	✓	✓
	Off-by-one errors		✗	✓	✓	✓	✓	✓
	Unbounded copy		✗	✗	✗	✗	✗	✗
Strings	*Direct overflow*		✗	✓	✓	✓	✓	✓
	Null termination errors		✗	✗	✗	✗	✗	✓
	Off-by-one errors		✗	✓	✓	✓	✓	✓
	Truncation errors		✗	✗	✗	✗	✗	✓
	Unbounded copy	*strcpy*	✗	✗	✓	✗	✗	✓
		strcat	✗	✗	✓	✗	✗	✓
		gets	✓	✗	✓	✗	✗	✓
		sprintf	✓	✗	✓	✗	✗	✗
		strncpy	✗	✗	✓	✗	✓	✓
		strncat	✗	✗	✓	✗	✓	✗
		fgets	✗	✗	✓	✗	✓	✓
		snprintf	✗	✗	✓	✗	✓	✓
Format string vulnerabilities			✓	✗	✗	✗	✓	✓
Memory	*Double free*		✓	✗	✓	✗	✓	✓
	Improper allocation		✗	✗	✗	✗	✗	✗
	Initialization errors	*malloc*	✓	✗	✗	✗	✗	✗
		realloc	✗	✗	✗	✗	✗	✗
	Memory leak	*malloc*	✓	✗	✓	✗	✗	✗
		calloc	✓	✗	✓	✗	✗	✗
		realloc	✗	✗	✓	✗	✗	✗
	Failure check		✓	✗	✗	✗	✓	✓
	Access freed memory		✓	✗	✓	✗	✓	✓
File operations	*Access closed file*		✗	✗	✗	✗	✗	✓
	Access in different mode		✗	✗	✗	✗	✗	✗
	Double close		✗	✗	✓	✗	✗	✓
	Resource leak		✗	✗	✓	✗	✓	✗
	Access without open		✗	✗	✗	✗	✗	✗
	Failure check		✓	✗	✗	✗	✓	✓
Race conditions	*Deadlock (no concurrency)*		✗	✗	✗	✗	✗	✗
	Deadlock (concurrency)		✗	✗	✗	✗	✓	✗
	Time Of Check, Time Of Use (TOCTOU) errors		✗	✗	✗	✗	✓	✓

TABLE III. ANALYSIS SENSITIVITY OF THE TESTED TOOLS IN THEIR DEFAULT CONFIGURATION

Analysis	Language constructs	Splint	UNO	Cppcheck	Frama-C	C++ Test	Com. B
Path Sensitivity	Simple if-else statement	FP	✓	✓	✓	✓	✓
	Complex if-else statement	FP	FP	✓	✓	FP	✓
	Typical for loop	FP	✓	FP	✓	✓	✓
	Complex for loop with break command	FP	FP	FP	✓	✓	✓
	While loop with continue command	FP	FP	FP	✓	✓	FP
	Do-while loop with continue command	FP	FP	FP	✓	✓	FP
	Switch statement	FP	FP	FP	FP	✓	✓
	Goto statement	✓	✓	FP	✓	✓	✓
	For loop with arrays	FP	FP	FP	✓	✓	FP
	For loop with pointer arithmetic	FP	FP	FP	✓	✓	FP
	Conditional operator	FP	FP	FP	✓	✓	✓
	Return statement	✓	✓	FP	✓	✓	✓
	Exit function	✓	✓	FP	✓	✓	✓
	Define constant	FP	✓	✓	✓	✓	✓
	Enumeration	FP	FP	FP	✓	✓	✓
Context sensitivity	Simple function calls	FN	FN	FN	FN	✓	✓
	Static variables	FN	FN	FN	FN	✓	✓
	Global variables	FN	FN	FN	FN	✓	✓
	Function pointers	FN	FN	FN	FN	✓	✓
	Structs	FN	FN	FN	FN	✓	✓
	Unions	FN	FN	FN	FN	✓	✓
	Typedef	FN	FN	FN	FN	✓	✓
Alias analysis	Direct assignments	✓	✓	FN	✓	✓	✓
	Casting assignments	✓	FN	FN	✓	✓	✓

Cppcheck avoided FPs only in simple or complex `if-else` blocks and in programs with `#define` macro. Frama-C and Parasoft C++ Test yielded only one FP, which in the first case concerns the analysis of a `switch` statement and in the second case a complex `if-else` statement. Com. B detected all infeasible paths, apart from paths in programs with `while` or `for` loops.

The open source tools showed a lack of context sensitivity yielding FNs in all tested types of context for the function calls. With the commercial tools, all flaws were detected, irrespective of the used type of context. For Cppcheck, FNs also appeared in the programs that required an alias analysis. UNO failed when the alias analysis concerned casting assignments and the other tools reported all existing vulnerabilities.

VI. TOOL EFFECTIVENESS

Tool effectiveness for the considered vulnerabilities was quantified by classifying every commented line either as true positive, true negative, false positive or false negative. The number of cases for every possible result determined the values of five metrics: *accuracy*, *precision*, *recall*, *specificity* and *F-measure*.

Accuracy is the ratio of correct classifications over the number of observations. Fig. 2 shows that Frama-C, Parasoft C++ Test and Com. B achieved the best scores

around 0.85. For UNO, the accuracy score was 0.72, while for Cppcheck and Splint was respectively 0.64 and 0.56.

Precision is the ratio of the number of true positives over the number of reported errors. An ideal tool (score 1.0) would report only true code defects. The best tools in the test drive were Frama-C with 0.93, Com. B with 0.88 and Parasoft C++ Test with 0.81. All other tools exhibited lower precision, with Splint having the lowest score (0.5).

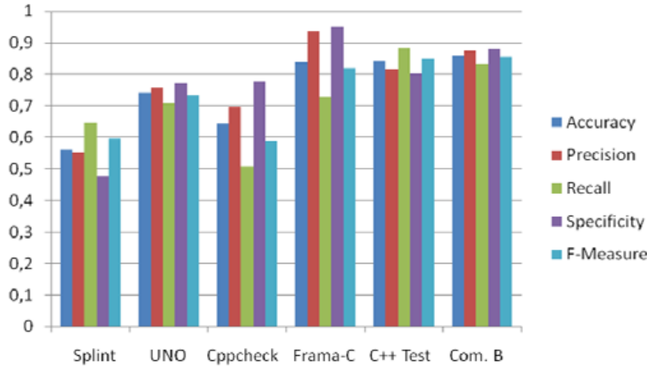

Figure 2. Tool effectiveness on the considered C code vulnerabilities.

Recall, also known as *true positive rate*, *is the ratio of the number of true positives over the number of existing errors*. An ideal tool (score 1.0) would have detected all existing errors. The best tools in the test drive were Parasoft C++ Test with 0.9 and Com. B with 0.82. Apart from Cppcheck that has fallen short with 0.5, all other tools achieved a score around 0.7.

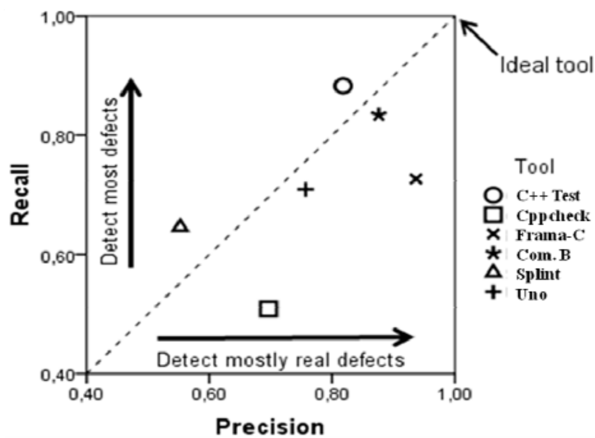

Figure 3. How the tools balance the tradeoff between precision and recall

Specificity is also called *true negative rate* and *is the ratio of the number of true negatives over the sum of true negatives and false positives*. The top score 1.0 corresponds to the absence of FPs. In the test drive, the highest score was 0.95 by Frama-C. Com. B was ranked second with 0.9. Parasoft C++ Test, UNO and Cppcheck scored in the level of 0.8 and Splint remained behind with 0.5.

The F-measure provides an aggregate measure (harmonic mean) for precision and recall, two metrics that possess an intrinsic tradeoff. The two commercial tools were ranked first with a score around 0.85 that places them closer to the ideal tool. Fig. 3 shows how the tools balance the tradeoff between precision and recall. Frama-C achieved a slightly lower F-measure, but with more emphasis in detecting the true defects. The F-measure score for UNO was 0.7 and for all other tools less than 0.6.

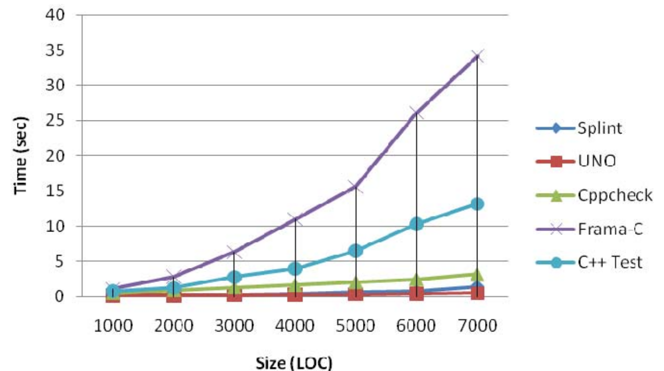

Figure 4. Average time for three analysis cases: path-sensitive, context-sensitive and alias analysis

VII. ANALYSIS EFFICIENCY

Tool effectiveness is just the one side of the coin, since a relatively high F-measure comes with a price in analysis efficiency.

We measured the demands in time and memory space for a series of program analyses that could be accomplished on our experimental platform in reasonable time. The experiments took place on a 1.7GHz machine with 2GB of RAM and the length of the benchmark programs varied between 1000 and 7000 lines of code. The used programs have been generated according to the methodological considerations of section IV, meaning that for each program size three test cases with different requirements of analysis sensitivity are considered, namely path sensitivity, context-sensitivity and alias analysis.

Fig. 4 shows the average analysis time for all tools except Com. B that was not possible to run on the same operating system. Parasoft C++ Test and Frama-C that exhibited high precision are on average more than three or respectively seven times slower than UNO and other tools in programs with 7000 lines. It is also noteworthy that this gap increases rapidly for programs with more than 5000 lines.

Fig. 5 shows the average peak memory usage for the same static analysis tasks. UNO and Cppcheck scale smoothly for programs with up to 7000 lines, in contrast to Splint that exhibited a greedy demand for memory in

programs with more than 3000 lines. Frama-C and Parasoft C++ Test, which implement comparatively more effective analyses, incur constantly increasing memory costs with the program size, but Frama-C scales slightly better.

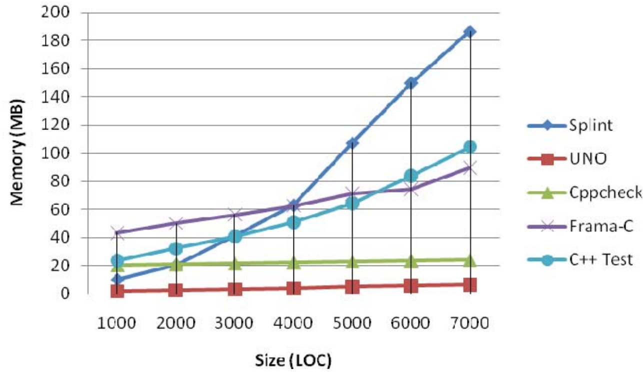

Figure 5. Average of peak memory usage in three cases: path-sensitive, context-sensitive and alias analysis

VIII. RELATED WORK

In [15], Wilander and Kamkar examined publicly available code scanners for their ability to detect buffer overflows and format string vulnerabilities. The scanning capability of some of the tested tools is restricted to a form of primitive lexical analysis and the study was limited to a relatively small number of programs written for the mentioned code vulnerabilities.

Zitser et al. [16] proposed 14 model programs that simulate an equivalent number of reported real-world vulnerabilities found in open-source software. Our approach differs in terms of the underlying methodological considerations of section IV and in the criteria used for the code vulnerabilities covered by the test suite.

Kratkiewicz and Lippmann [17] developed 291 small C programs to test the error detection capabilities of five static analysis tools. In that study only buffer overflows are considered. It is also worth to note that the results in [16] and [17] include only one commercial tool, thus failing to provide a spherical view of the analysis characteristics encountered in mature commercial products.

Newsham and Chess [18] propose a prototype benchmark for source code analyzers of C and Java programs. Their approach seems to be promising, because they try to combine artificially created test cases and test cases produced from real-world applications. There is no comparison based on metrics that as in our case yield a quantitative characterization of the tools' effectiveness and efficiency.

One of the most interesting related works is the so-called BegBunch by Cifuentes et al. [19]. BegBunch is a static analysis test suite divided in two sub-sections, where accuracy or scalability of static analysis tools can be studied. The two sub-suites are independent from each other, as opposed to our benchmark, where the analysis efficiency benchmark was derived from the test cases used for studying the tools' effectiveness. BugBench uses synthetic test cases taken from other projects, like SAMATE [13] and does not utilize information and data reported in public catalogues.

Finally, Schmeelk [20] has recently introduced the design of a repository, in order to integrate benchmarks with publicly available fault taxonomies like the CWE. He also pinpoints the need for a unified benchmarking framework.

IX. CONCLUSIONS

Static analysis can improve the reliability of C programs, only if it is really effective for the code defects that usually arise in a project at an affordable cost. The effectiveness of a tool encompasses quantitative evidence for the tradeoff between precision and efficiency and qualitative evidence for the analysis sensitivity with respect to the language constructs. We introduced a methodology for test driving static analysis tools and a test suite implementing code vulnerabilities that in public catalogues are reported with comparatively high frequency. The test suite is available online at `mathind.csd.auth.gr/static_analysis_test_suite`, together with the benchmark for evaluating analysis efficiency. The results from test driving four open-source and two commercial tools showed that only one open-source tool competes the commercial products, in terms of precision, but at a very high cost in efficiency. We provided a detailed report on the tools analysis sensitivity with respect to most C language constructs. This contribution, when completed with statistics from real software projects (number of FPs and FNs) set a complete view for the tools' effectiveness.

Further development of test driving static analysis tools can be directed towards studying their effectiveness with "weighted" metrics, where the weights will depend on statistics for the distribution of code defects in software projects (or catalogues).

A tool's adequacy for a software process is also affected by its extensibility perspectives and the ease of use, but eventually the best solution in terms of cost-effectiveness depends on the monetary cost.

REFERENCES

[1] W. Landi, "Undecidability of static analysis", ACM Letters on Programming Languages and Systems, 1(4):323-337, 1992

[2] C. Cowan, P. Wagle, C. Pu, S. Beattie and J. Walpole, "Buffer overflows: Attacks and defenses for vulnerability of the decade", Proc. Of DARPA Information Survivability Conference and Exposition, 119-129, 2000

[3] CWE list (1.10). URL http://cwe.mitre.org/data/published/cwe_v1.10.pdf

[4] R.C. Seacord, "The CERT C Security Coding Standard", Addison-Wesley Professional, 1st Edition, 2009

[5] Coverity Scan Open Source Report, URL: http://scan.coverity.com/report/Coverity_White_PaperScan_Open_Source_Report_2009.pdf

[6] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison and J. West, "Model Checking An Entire Linux Distribution for Security Violations". Proc. of the 21st Annual Computer Security Applications Conference, 13-22, 2005.

[7] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis", IEEE Software, 19(1):42-51, 2002

[8] G. Holzmann, "Static source code checking for user-defined properties", Pasadena, CA, USA, 2002

[9] Frama-C. URL http://frama-c.com/

[10] Cppcheck – A Tool for static C/C++ static code analysis, *http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page*, Accessed: 31 October 2010

[11] P. Cousot, P. and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages ACM, New York, NY, 238-252, 1977

[12] Parasoft C++ Test, URL http://www.parasoft.com/

[13] NIST. Samate - software assurance metrics and tool evaluation. URL: http://samate.nist.gov.

[14] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses", Proc. Of OOPSLA '09: 24th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2009

[15] J. Wilander and M. Kamkar. "A comparison of publicly available tools for static intrusion prevention". Proc. Of 7th Nordic Workshop on Secure IT Systems, pages 68–84, 2002

[16] M. Zitser, R. Lippmann, and T. Leek. "Testing static analysis tools using exploitable buffer overflows from open source code". SIGSOFT Softw. Eng. Notes, 29(6):97–106, 2004

[17] K. Kratkiewicz and R. Lippmann. "Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools". Proc. Of Workshop on the Evaluation of Software Defect Detection Tools (BUGS'05), 2005.

[18] T. Newsham, B. Chess, "ABM: A Prototype for Benchmarking Source Code Analyzers". Workshop on Software Security Assurance Tools, Techniques, and Metrics. U.S. National Institute of Standards and Technology (NIST) Special Publication(SP) 500-265, 2006

[19] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. "Begbunch: benchmarking for C bug detection tools". Proc. of the 2nd International Workshop on Defects in Large Software Systems, New York, NY, USA, 16-20, 2009.

[20] S. S. Schmeelk. 2010. "Towards a unified fault-detection benchmark". Proc. of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '10). ACM, New York, NY, USA, 61-64, 2010

[21] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An Evaluation of Two Bug Pattern Tools for Java", 248-257, 2008