2011 35th IEEE Annual Computer Software and Applications Conference Workshops

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 tools for automated code scanning came in the limelight. Due to the significant costs associated with incorporating such a tool in the software lifecycle, it is important to know what defects are detected and how accurate and efficient the analysis is. We focus specifically on popular static analysis tools for C code defects. Existing benchmarks include the actual defects in open source programs, but they lack systematic coverage of possible code defects and the coding complexities in which they arise. We introduce a test suite implementing the discussed requirements for frequent defects selected from public catalogues. Four open source and two commercial tools are compared in terms of their effectiveness and efficiency of their detection capability. A wide range of C constructs is taken into account and appropriate metrics are computed, which show how the tools balance inherent analysis tradeoffs and efficiency. The results are useful for identifying the appropriate tool, in terms of cost-effectiveness, while the proposed methodology and test suite may be reused.**

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 restricts the analysis capability in correctly detecting actual code defects. Important considerations are: (i) the programming language, (ii) the targeted defects, (iii) the analysis effectiveness, i.e. 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 it is effective in capturing all defects that are critical for the required product quality, while it is sufficiently efficient for the size of the analyzed code base. Tool comparisons with results from real projects place an evaluation bias concerning the defects in the benchmark and the code complexity and size of the code base.

Empirical studies with open source programs should be completed by evaluation results that systematically cover the set of code defects that are frequent in a particular product quality context. We focus on software security and reliability by providing a synthetic test suite that implements C code defects, which are frequently reported in public catalogues. Code defects are replicated in benchmark programs with varied analysis requirements, in order to study the tool effectiveness in a wide range of possible coding complexities. Four open source and two mainstream commercial static analysis tools for C are evaluated. Tool effectiveness is measured by metrics that uncover valuable information for how the tools handle the inherent analysis tradeoffs in approximating the programs' behavior. A performance benchmark provides the basis for evaluating analysis efficiency, in terms of time and memory space.

The results, when combined with a cost-efficiency model [21] can be used for identifying the appropriate tool for a software process.

Reporting an actual flaw is not always connected to an analysis restriction, but it may be a decision matter of whether the flaw can cause problems and/or can be explained to the tool end-user. It is also true that vendors do not provide information for flaws that are ignored and therefore they cannot claim functionality that is transparent to the end-user. Our approach builds on a carefully selected set of flaws, in order to stimulate tool comparisons that address the aforementioned deficiency.

The followed methodology can also be applied to different quality contexts and software domains (with other code defects) and in tool comparisons for other languages. A possible scenario could be certifying an application for safe execution in a particular mobile platform. Such a certification is a prerequisite for distributing applications through internet-wide markets and the process concerns the checking of platform-specific security requirements.

Section II describes the selected code defects. Section III introduces the tools of the test drive and section IV refers to the proposed methodology. Detailed results are provided in section V and measurements for the tool effectiveness are shown in section VI. Analysis efficiency is examined in section VII. Related work is considered 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 DEFECTS

Public catalogues report many flaws that undermine the reliability and security of C programs. Flaws are reported in numerous records that sometimes overlap. Our test suite includes 30 distinct code defects selected from the Common Weakness Enumeration (CWE) catalogue [3] and the CERT

C Secure Coding Standard [4] according to their frequency. We classify them into 8 broad categories that are shown in Table I, in order to develop a concise and comprehensive taxonomy of C code defects.

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 the analyzed open source software. In the second category 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.

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 strings representation 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. We consider instances of *redundant file closure*, *omission of file closure* (resource leak), *absence of failure check* and access in a file that is either, *previously closed*, *not opened* or *opened with a different mode*.

"*Concurrency errors*" 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

We focus on full-fledged tools with built-in analyses, which detect most of the mentioned flaws. The comparison is not limited to open source 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 first mature commercial tool in our study is the Parasoft C++ Test.

TABLE I. FREQUENT C CODE DEFECTS

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
Concurrency errors	***Deadlock (no multithreading)***	Incorrectly manage control flow during execution (CWE-691)
	Deadlock (multithreading)	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)

We also provide results for another commercial product[1], in order to avoid drawing conclusions that may be biased by the detection capabilities of a single product.

A static analysis is *sound*, when it doesn't miss any flaws. It can be more or less *precise* to the extent that avoids reporting spurious errors, but by no means can be simultaneously sound and complete. In general, the more precise an analysis is the higher its computational demands are and the incurred cost affects its scalability. Common characteristics that result in more precise analyses include: *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 calling context is taken into account. Alias (also called *pointe*r or *points-to*) analysis computes the entities, where the variables point to. The aforementioned 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. It comes with an annotation language that allows to define attributes for program objects and to set range of values to program variables. However, it is not possible to access control-flow and dataflow information like in UNO [8], which in this way supports the development of truly new analyses. UNO's built-in analyses search for three common C code defects: uninitialized variables, null-pointer dereference and out-of-bound array indexing.

Cppcheck [9] can analyze thousands of lines with precision that can be tuned. Frama-C [10] provides static analyses embedded into a value analysis framework, which is based on abstract interpretation [11]. It implements a plug-in architecture over a kernel that controls the whole analysis, while the tool can be configured to different sensitivity levels. Custom plug-ins and user-defined properties written in a behavioral specification language extend the tool's functionality.

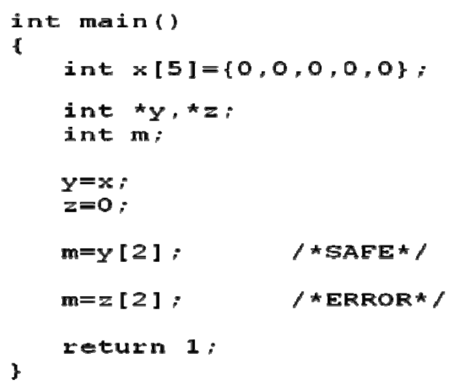
In Parasoft C++ Test [12], apart from the sensitivity level, the user can also select the flaws to be checked. The tool provides a GUI for expressing patterns of code defects, thus extending the set of flaws that are detected.

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

IV. Benchmarking Methodology and Test Suite

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

- *For each tool, we identify the defects it can detect.*
  Code defects that cannot be detected are reported, but they are not taken into account in measuring the tools' effectiveness. Different forms of a single defect are examined. For example, a TOCTOU error may occur with different pairs of C functions (notable cases are the `access()-fopen()` and `lstat()-remove()` pairs). The selected defects 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 flaws.
- *The tools sensitivity is systematically assessed with a wide range of C constructs and different conditions of language semantics, under which the defects may arise.*
  For each defect, 15 programs require some sort of path sensitive analysis, 7 programs are used for testing context sensitivity and 2 require an alias analysis.
  All programs have a line with the tested flaw commented as `/*ERROR*/` and another line commented as `/*SAFE*/` that checks the tool capability 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 a null-pointer dereference sample program. Our benchmarking test suite includes more than 700 programs like the one shown in the figure, where each of them 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 actual flaw. Having observed a consistent tool sensitivity behavior across the different defects, we report the FPs and FNs for the different C constructs, for a code defect that can be detected by all tools.
- *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 defects that each tool can detect. These metrics highlight how the tools balance inherent tradeoffs in analysis

[1] The license terms for the second commercial tool do not allow revealing the product name, as opposed to the license terms of Parasoft C++ Test.

effectiveness, like whether emphasis is given on detecting as many flaws as possible or if the tool 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. The three sensitivity types account the same in the reported metrics and therefore cannot be representative of the tools' efficiency in all possible analysis contexts (a program may need any combination of analysis sensitivities). However, these metrics indicate relative differences in how the tools balance their effectiveness with the price paid in efficiency.

V. TEST DRIVING ANALYSIS SENSITIVITY

Table II shows the results for the tools capability to detect the code defects of our test suite. We use the convention "Com. Tool B" for 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, while Splint ignored the division-by-zero cases.

Regarding integer flaws, Com. B detected truncation and sign errors, while the latter were also detected by Splint and 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 those that successfully identified the format string vulnerabilities.

Regarding memory errors, 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 concurrency errors with no multithreading, 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 in the control flow (`goto`, `return` and `exit()` function). UNO extends path-sensitivity to the simple `if-else` and `for` loop statements and when the control flow depends on some `#define` preprocessor macro.

Cppcheck avoided FPs only in simple or complex `if-else` blocks and in programs with the `#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 function call contexts. With the commercial tools, all flaws were detected, irrespective of the calling context. For Cppcheck, FNs also appeared in programs that required an alias analysis. UNO failed when the alias analysis concerned casting assignments, but the other tools detected all flaws.

VI. TOOL EFFECTIVENESS

Tool effectiveness for the considered defects was quantified by classifying every commented line either as true positive, true negative, FP or FN. The number of cases for every possible result determined 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).

Recall or *true positive rate*, *is the ratio of the number of true positives over the number of actual 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.

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

TABLE II. TOOLS CAPABILITY TO DETECT THE C CODE DEFECTS 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		✓	✗	✗	✗	✓	✓
Concurrency errors	***Deadlock (no multithreading)***		✗	✗	✗	✗	✗	✗
	Deadlock (multithreading)		✗	✗	✗	✗	✓	✗
	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	✓	✓	✓

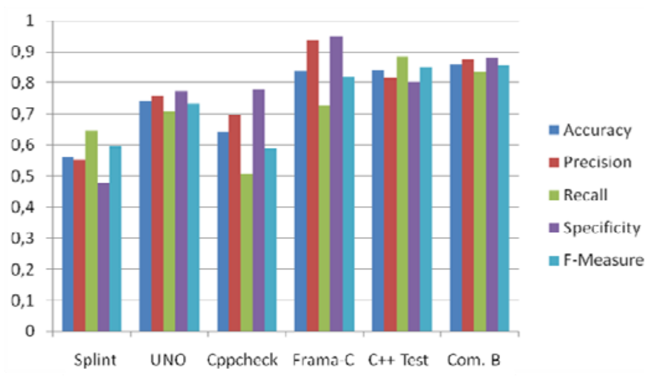


Figure 2. Tool effectiveness on the considered C code defects

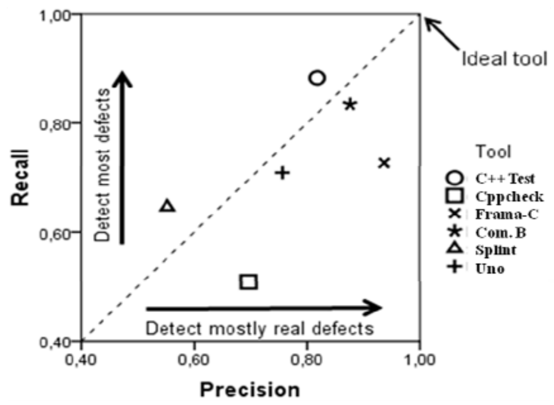


Figure 3. How the tools balance the tradeoff between precision and recall

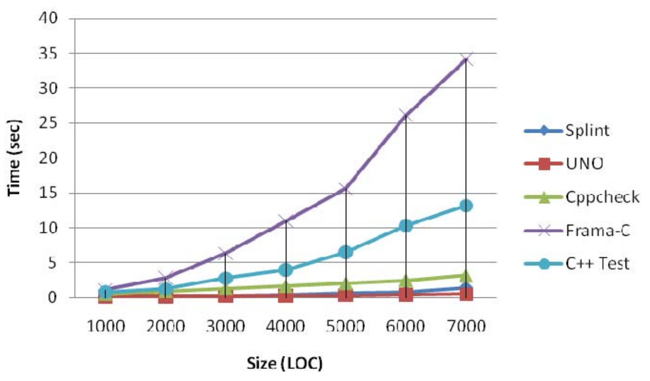


Figure 4. Average time for three analysis cases: path-sensitive, context-sensitive and alias analysis

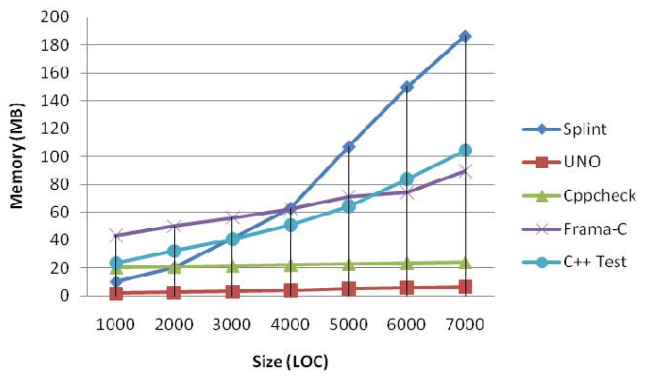


Figure 5. Average of peak memory usage in three cases: path-sensitive, context-sensitive and alias analysis

was 0.7 and for all other tools less than 0.6.

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 were generated according to the methodological considerations of section IV, meaning that for each program size three test cases with different analysis sensitivity requirements 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 increasing program sizes, but Frama-C scales slightly better.

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 tools is restricted to a form of 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 defects 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 code analyzers of C and Java programs. Their approach seems to be promising, because they try to combine artificially created test cases with test cases from real-world applications. There is no comparison based on metrics like in our case, which would provide 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 defects that in public catalogues are reported with comparatively high frequency. The test suite is available online at `http://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.

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.

Acknowledgments
This work was performed in the framework of the TRACER project, which is partly funded by the Hellenic General Secretariat of Research and Technology (09ΣΥΝ-72-942).

REFERENCES

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

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

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

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

[5] Coverity Scan Open Source Report, Available: 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". In *Proc. of the 21st Annu. Computer Security Applications Conf.*, Washington, DC, 2005, pp. 13-22.

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

[8] G. Holzmann. "UNO: Static source code checking for user-defined properties". In *6th World Conf. on Integrated Design and Process Technology (IDPT '02)*, Pasadena, CA, USA, 2002.

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

[10] Cppcheck – A Tool for static C/C++ static code analysis. Available: http://sourceforge.net/apps/mediawiki/cppcheck

[11] P. Cousot, P. and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In *Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages* (POPL '77). ACM, New York, NY, 1977, pp. 238-252.

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

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

[14] M. Bravenboer and Y. Smaragdakis. "Strictly declarative specification of sophisticated points-to analyses". In *Proc. Of 24th ACM SIGPLAN Conf. on Object oriented programming systems languages and applications* (OOPSLA '09). ACM, New York, NY, USA, 2009, pp. 243-262.

[15] J. Wilander and M. Kamkar. "A comparison of publicly available tools for static intrusion prevention". In *Proc. Of 7th Nordic Workshop on Secure IT Systems*, November 2002, pp. 68-64.

[16] M. Zitser, R. Lippmann, and T. Leek. "Testing static analysis tools using exploitable buffer overflows from open source code". In *Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of software engineering* (SIGSOFT '04/FSE-12). ACM, New York, NY, USA, 2004, pp. 97-106.

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

[18] T. Newsham and B. Chess. "ABM: A Prototype for Benchmarking Source Code Analyzers". In *Proc. of the Workshop on Software Security Assurance Tools, Techniques, and Metrics* (SSATTM '05). Long Beach, California, 2005.

[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". In *Proc. of the 2nd Int. Workshop on Defects in Large Software Systems* (DEFECTS '09), New York, NY, USA, 2009, pp. 16-20.

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

[21] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. "An Evaluation of Two Bug Pattern Tools for Java". In *Proc. of Int. Conf. on Software Testing, Verification, and Validation* (ICST '08), 2008, pp. 248-257.