Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software

Davide Pozza, Riccardo Sisto
Dip. di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24
I-10129 Torino, ITALY
Email: {davide.pozza, riccardo.sisto}@polito.it

Luca Durante, Adriano Valenzano
IEIIT - CNR
c/o Politecnico di Torino
c.so Duca degli Abruzzi 24
I-10129 Torino (Italy)
Email: {luca.durante, adriano.valenzano}@polito.it

Abstract— **Many of the bugs in distributed software modules are security vulnerabilities, the most common and also the most exploited of which are buffer overflows and they typically arise in programs written in the C language. This paper, focusing on static analysis tools for detecting buffer overflows in C programs, presents a methodology for experimentally evaluating and comparing the main objective features of such tools. The proposed method is based on testing all the tools on a common set of publicly available, open source software packages, and makes use of specific metrics defined to evaluate the main tool features. In particular, the evaluation aims at quantifying how close the tool is to a complete and sound tool. Our approach has been applied for an initial evaluation of the class of static analysis tools that are based on lexical analysis, using as test cases three well known network software packages. The results obtained, illustrated and commented on in this paper, offer some interesting indications.**

Keywords: static analysis, lexical analysis, buffer overflow, ITS4, Rats, Flawfinder.

I. Introduction

Nowadays distributed software is everywhere and plays an important role in our life. The number, size and complexity of software systems that interact, communicate and depend on one another is growing. Moreover, despite the evolution of programming languages, which tends to make the programming task easier, many programming errors are still quite frequent, because either the features of the new languages do not exclude all the causes of errors or some old languages, like C and C++, are still in use. A consequence of this scenario is an increase in the number of critical bugs. Programming errors are responsible for software failures, and they can have serious safety and security implications.

In this paper, attention is focused on security issues. For example, human and automated (e.g. worms) attackers can exploit software bugs in order to subvert the normal behavior of programs, in that, they are able to gain access to confidential data, to improperly take control of systems, or to cause denial of service. Security-relevant software bugs therefore can have a devastating effect.

The CERT Coordination Center (at Carnegie Mellon University) has been keeping trace of all reported security software vulnerabilities since 1995. Looking at the statistics [1] is worrying, because the number of the reported vulnerabilities is rapidly increasing. The sum of the number of vulnerabilities reported between years 2000 and 2003 has grown by about 760% compared to the sum of those published between years 1995 and 1999. The major classes of implementation security vulnerabilities, which have been identified, are buffer overflows, format strings, race conditions and weak sources of random numbers. The largest class is represented by buffer overflows, which, according to the ICAT Vulnerability Database [2], account for about 20% of all the vulnerabilities contained in the database since 1995.

Buffer overflows occur because of programming errors, such as improper buffer bound checks and/or misuse of some library functions (which work on buffers). A read buffer overflow occurs when a read operation occurs outside the intended buffer bounds. In this case, the confidential data stored in memory can be leaked. A write buffer overflow occurs when some program code tries to store more data in a buffer than it was designed to hold. Since buffers are created to contain a finite amount of data, the extra information overflows into adjacent memory areas, thus corrupting (by overwriting) the valid data held in them. Through out-of-bound writing operations attackers can modify and/or take control of the program execution, being able, in the worst case, to execute arbitrary code with root/administrator privileges. The various buffer overflow exploitation techniques are described in many technical papers. A digest of these techniques is provided in some published surveys (e.g. [3][4]).

The buffer overflow problem is typical of old programming languages, such as C and C++, whilst it cannot arise in the new generation of languages (e.g. Java and C#). However the problem is still pervasive since much software is still written in C and C++, or rely on system components that have been written in those languages. A survey of www.sourceforge.com (on September 2004) confirms this, indicating that a substantial percentage of open source projects are still using C (14,0%) and C++ (14,2%).

Detecting possible buffer overflows in a program is a difficult and time consuming task, which can be alleviated by using static and dynamic software analysis tools.

Our attention is focused on static analysis tools, and in par-

0-7803-9575-1/06/$20.00 ©2006 IEEE

ticular on the most lightweight ones, based on lexical analysis. Such tools look for certain patterns in the source code and, possibly using some heuristics, report the ones that might be vulnerabilities. Despite their simplicity, lexical analysis tools can help to select the parts of code that need auditing, without significantly burdening the programmer. They are particularly useful during the coding phase, where more sophisticated tools would introduce prohibitive turnaround times.

Of course, static analysis tools can miss some bugs as well as report false bugs. Thus, an important question for the user is how to evaluate and objectively compare the various tools. Up to now, not much research work has been done on experimental evaluation of static analysis tools for buffer overflow detection. The authors of the various tools have made some experiments, but using different test programs and operating conditions, so they do not provide any homogeneous basis for comparison. The first attempt to compare static analysis tools using a common base of test programs is [5], which, however, uses extremely simple artificial code samples which are not so representative of real code. Another evaluation work is [6], which reports some qualitative results of experiments where different programmers used various static analysis tools for buffer overflow detection on the same set of programs. In this case, the set of programs is significant, because it includes real world open source programs, but the aim of the work is more to evaluate human factors than to gain an objective and quantitative evaluation of the tools themselves. In short, at the moment experimental work is needed to quantitatively evaluate the tools using a meaningful set of programs and homogeneous test conditions. In particular, even rigorous methodologies for the experimental assessment of tools, capable of giving quantitative results, are missing.

This paper takes a first step in this direction. A methodology for experimental evaluation of static analysis tools for buffer overflow detection is presented along with some initial experiments performed according to the methodology on the available lexical analysis tools. The experiments have been made using real-world publicly available source codes as test cases, all of which belong to the network programming area, where buffer overflow detection is most important. In this way it is evaluated how the tools behave when analyzing real production network software, which contains commonly used programming idioms and includes complex control flows, data flows, data structures and component interactions.

The work presented here is the first part of a more extensive research project in which it is planned to investigate the whole class of static analysis tools.

The rest of the paper is organized as follows. Section II presents a brief description of the features of static analysis tools, and of lexical analysis tools in particular. Then, section III describes the evaluation methodology, the experiments performed and the results obtained. Section IV draws some conclusions.

II. Lexical Analysis

Static analysis techniques are useful to find errors that do not arise during testing or using dynamic analysis techniques, because they occur on uncovered execution paths. It is worth noting that the problem of static program analysis, in general, and the problem of finding all the buffer overflow errors, in particular, is undecidable because it is a Turing halting problem. Consequently, only approximate solutions can be obtained. An analysis tool is said to be sound if it does not report spurious warnings (i.e. warnings regarding code that does not contain errors), whilst it is said complete if it detects all the errors, possibly also reporting spurious warnings. Because of the issue of undecidability, building a tool that is both sound and complete is theoretically impossible, unless the tool is in some way restricted to operate only on a decidable subset of programs.

The existing static analysis tools that aid the programmer in finding buffer overflow vulnerabilities use different analysis techniques. In general, since each technique has its own specificity, the best results can be achieved using more than one technique. However, the cost of using sophisticated techniques can be too high and not worthy.

The simplest static analysis technique for buffer overflow detection is the one used by what are called Lexical Analysis Tools. It is a token-based analysis, searching for the presence of calls to functions included in a database. When potentially dangerous calls to such functions are detected, a warning is reported. This approach is based on the consideration that most security-related bugs are caused by misuse of library functions: some functions are always unsafe, and should always be avoided, while others can cause problems if not used properly. The database of a lexical analysis tool contains library functions that, when misused, can result in vulnerabilities. Some of such tools use heuristics to recognize some forms of security-safe usage of potentially dangerous functions, thus reducing the number of false positives that they raise.

The main lexical analysis tools have been considered. They are ITS4 [1] [7][8], Rats, [2] Flawfinder [3] and VulCAn [9].

They aim at detecting not only buffer overflow problems, but also format string bugs, race conditions, insecure temporary file and weak randomness functions usages. VulCAn can analyze programs written in C, while ITS4 and Flawfinder can analyze those written in C and C++, and Rats can also analyze those written in Perl, PHP and Python.

ITS4, Rats and Flawfinder are context-insensitive, while VulCAn is context-sensitive, i.e. it considers the scope of variables so that it can identify the variables involved in each operation exactly.

According to its authors, VulCAn considerably reduces the number of false positives with respect to context-insensitive lexical analysis tools [9]. Unfortunately, VulCAN is not publicly available and only very little information about it has

[1] Available at http://www.cigital.com/ITS4/

[2] Available at http://www.securesw.com/

[3] Available at http://www.dwheeler.com/flawfinder/

been published. So, it has been possible to analyze in detail and compare experimentally only ITS4, RATS and Flawfinder and it is for this reason that VulCAn is no longer considered in the rest of the paper.

An important aspect that must be taken into account is how the tools provide their results. For all the tools considered, the report provided as output lists the lines of code that are suspected to contain bugs. For each different set of items in the list, a description of the potential problem along with suggestions of possible solutions are given. However, the output information should be considered just as a reminder of how the problem could arise, since no information, which would be useful to trace back the code in order to understand when a bug really exists, is provided. In short, no significant differences exist in the quality of the results given by the various tools: all of them just provide a list of lines of code which are to be analyzed further. The analysis of each warning has a considerable cost, as is also noted and experimentally verified in [6]. Thus, the amount of manual work needed to analyze the reported warnings is the predominant cost. It can be considered to be roughly proportional to the number of code lines where warnings are reported.

The sensitivity of a tool is controlled by its severity cutoff, i.e. the severity level above which the tool reports warnings. The default value of this parameter is different for the various tools. For example, when ITS4 and Rats are run with their default settings, no warning is reported for the calls to `strcpy()` functions using a fixed constant string as the second parameter, independently of the fact that the second parameter may exceed the maximum admissible length of the first parameter. Flawfinder, instead, reports such potential errors, due to its having different default settings. From a security point of view, the behavior of ITS4 and Rats is correct, since these kinds of buffer overflows are programming errors that cannot be exploited by attackers and hence they are less risky. However, users can customize the behavior of these tools, in order also to report these kinds of errors, disabling the heuristics that reduce the risk levels associated with the database functions and/or modifying the severity-cutoff level.

All the tools being considered are neither sound nor complete (because they deal only with errors caused by misuse of library functions). However, since ITS4, Rats and Flawfinder are configurable, it is possible to set them so that they are sound (they can be instructed to report only warnings for the functions that must always be avoided) or partially complete (i.e. complete with respect to the sub-problem of finding buffer overflows caused by misuse of library functions).

However, as already noted in [6], soundness and completeness are of little use when considered separately. A complete tool can be practically useless if it gives too many false positives and the same is true for a sound tool that identifies only a minimal fraction of errors. The aim of tool designers is to build a tool that is as close as possible to achieving both soundness and completeness. The extent to which this is reached can be evaluated experimentally, as explained in the next section.

III. Experimental Evaluation of Lexical Analysis Tools

The definition of the ideal tool useful to detect buffer overflows is helpful in determining how to evaluate real tools experimentally.

The ideal tool should be sound and complete, it should be able to give precise information about which lines of code contain errors and why, and it should take a negligible time to achieve these goals. With such an ideal tool, all the errors are detected and the programmer's effort needed to identify and correct them is minimum. Unfortunately, the available tools are far from this ideal case, and the analysis of a source program never leads to the certainty of having found all the errors, while a lot of manual work is involved. Typically, the programmer uses static analysis tools to get a set of warnings, and then restricts the manual search for errors to the set of source code lines indicated by the warnings. This work should lead the programmer to distinguish real errors from false positives.

It can be noted that only certain features of a tool are amenable to an objective evaluation. For example, it is not possible to establish objectively how useful warnings are in detecting buffer overflows, because this very much depends on human factors. Instead, here interest is focused on defining a methodology that enables quantitative and objective evaluations of the tools to be obtained.

The main indexes that can be evaluated objectively are how close a real tool is to a complete and sound tool and how long it takes to perform the analysis. Another feature that may be evaluated is the tool scalability, i.e. if and how such indexes vary when the programs being analyzed grow in size.

The definition of metrics suited to completeness and soundness is not obvious and calls for a preliminary discussion. Completeness is related to tool efficacy (it describes the ability of the tool to report as many real errors as possible). Soundness is related to tool efficiency (it describes the ability of the tool to report errors without reporting false positives): few false positives means low manual auditor effort will be needed to perform the task. A question that arises when trying to define quantitative completeness and soundness performance indexes concerns what units are to be used for errors, false positives and warnings. In this paper it is assumed that each line of code may contain at most one error and the unit used is the line of source code: each tool gives warnings that refer to lines of code and for each line of code indicated by a warning there may be either exactly one real error or a false positive.

The extent to which completeness is achieved by a tool when analyzing a given program can be measured by the fraction of real error lines reported by the tool. Similarly, soundness can be evaluated by the fraction of false positives in the reported warnings. If TP denotes the number of true positives (i.e. real error lines) given by the tool and FP the number of false positives (i.e. false error lines) given by the tool, it follows that the total number of warnings (in code lines) given by the tool is $FP+TP$. If LOE is the total number of lines of errors in the program analyzed, tool completeness for

a given program can be evaluated by the fraction TP/LOE and tool soundness for a given program can be evaluated by the fraction $TP/(TP + FP)$. Of course, the completeness measure makes sense if $LOE > 0$, i.e. if the test program contains at least one error, and the soundness measure makes sense if $TP + FP > 0$, i.e. if the tool gives at least one warning.

Another important aspect of an experimental evaluation is how to select the set of test programs to be used as samples. In this paper this set is called the *program basis*.

Of course, in order to provide uniform and comparable results for all the tools, they must all be tested using the same program basis and under the same conditions. Furthermore, to get results of practical interest, the experiments should use a program basis made up of real code, containing real world errors, and not artificial code samples.

Additional criteria to be followed when selecting the program basis were identified as follows:

- The programs in the program basis must be publicly available, so that the program basis is itself publicly available and anyone can use it to repeat the experiments or to evaluate other tools.
- To be representative of a large class of real programs, the program basis should contain programs of different sizes (and hence different complexities), possibly written by several hands (so as to contain a wide range of programming idioms) and having different security reputations (i.e. programs known to contain several vulnerabilities and programs considered to have a secure implementation).
- The program basis should be made up of programs that are potentially prone to buffer overflow problems, i.e. programs that belong to the class of network software and intensively use buffers, performing several operations on them.

The methodology just outlined has been applied to experimentally evaluate the available lexical analysis tools for buffer overflow detection. Specifically, ITS4 (version 1.1.1) [7][8], Flawfinder (version 1.21), and Rats (version 2.1) have been evaluated.

The experiments presented here are only a first step towards a complete experimental evaluation of these static analysis tools: the specific aim of the experiments has been to evaluate the tools when run with their default settings, with a minimal, but sufficiently significant, program basis.

The program basis that has been used for these experiments, although composed of only 3 software packages, satisfies all the above mentioned criteria. It includes: net-tools 1.46, WU-FTPD 2.5.0 and Pure-FTPd 1.0.17a.

Net-tools is a package that implements several system commands related to networking, such as netstat, ifconfig, route, and so on.

WU-FTPD was the most popular and widely used FTP daemon for Unix systems, at least until 2001. However, now it is no longer being developed, supported or widely used, probably because it has sadly become famous for the high number of discovered security problems. Version 2.5.0 is an old version with several known vulnerabilities[4].

Pure-FTPd is defined as a free (BSD), secure, production-quality, standard-conformant, efficient and easy to use FTP server. The authors of the program audited the code several times before releasing it, and no buffer overflow vulnerabilities have been reported yet. By accurately analyzing the source of the program a single and very simple programming error was found.

TABLE I
SOME FEATURES OF THE PROGRAMS SELECTED FOR EXPERIMENTS.

Program	Ver.	# file src.	# file lib.	LOP	LOP src.	LOC src.	LOE	LOE / LOC src. %
Net-tools	1.46	14	49	10878	4850	4146	50	1.206
WU-FTPD	2.5.0	32	19	20772	17738	13582	65	0.479
Pure-FTPd	1.0.17a	111	8	29275	28201	25230	1	0.004

Tab. I provides some information about sizes and defect rates of the selected programs, so that the reader can see that our selection meets the stated criteria. For each program, the table reports the number of source (`# file src.`) and support-library (`# file lib.`) files. Then it reports the total number of lines (`LOP`), the number of lines of the source files only (`LOP src.`) and, since programs normally contain comments and empty lines, the net number of non-comment, non-empty lines of code (`LOC src.`). Finally, it reports the number of lines where buffer overflow errors were found (`LOE`) and the defect ratio in percentage (`(LOE/LOC src.)%`). LOE takes into account all the errors that were known in advance and the additional ones that were found by performing an accurate auditing guided by the warnings given by the available static analysis tools. Of course, considering the size of the programs, it is possible that some errors are not included in this figure, because they were not detected. So, the real total number of lines of code containing errors is unknown, and, for the purpose of these evaluations, only the errors that were discovered are considered.

All the experiments were performed using an Athlon 1800 XP+ machine equipped with 1GB of RAM and running Linux Red Hat 9.0.

The following procedure was used to obtain the experimental results.

Each tool was run once, with its default settings. Before proceeding with the manual investigation of the warnings raised by the tools, the output of each tool was filtered preliminarily, in order to eliminate the parts which where of no interest to this study.

Of course, all warnings not concerning buffer overflows but other vulnerabilities, such as format string bugs and race conditions were all eliminated beforehand.

In addition, it was decided to exclude from our evaluation the potential buffer overflow vulnerabilities related to the functions `getopt`, `getopt_long` and `syslog`, because

[4]For the known buffer overflow vulnerabilities, see CERT advisories CA-1999-13, CA-2001-07 and CA-2001-33

only some old library implementations of such functions suffer from buffer overflow problems. Consequently, the warnings related to such functions were also eliminated from the outputs.

Finally, all the warnings that give generic hints to the programmer without indicating precisely the possible location of errors were eliminated from the outputs. These include the warnings (emitted by Flawfinder) regarding the `strlen` functions, because, if `strlen` works outside the buffer bounds, the programming error is not due to the `strlen` itself, but to some other portion of the code. Similarly, it was decided to delete the warnings issued by Flawfinder and Rats for all the statically sized buffers, recommending that bounds checking be performed, functions that limit length be used, and that the size is larger than the maximum possible length be ensured.

The next step of the procedure was to audit code by hand, since it is necessary to find out which warnings correspond to either false positives or real errors respectively. The code auditing was performed independently twice, in order to reduce the risk of human errors in the categorization of warnings, since all the lines of code subject to buffer overflow problems were not known beforehand.

Auditing was performed with the aid of the powerful source code navigation features of CodeSurfer [10]. This tool significantly helped to alleviate the work needed to understand and carefully examine the code in order to assert presence or absence of errors for each line of code reported by the tools. It is interesting to note that, on average, understanding when a warning is an error is less expensive than proving that a warning is a false positive. In fact, in order to determine the presence of an error it is sufficient to find a path where a buffer can be used outside its allocation size, while asserting absence of errors requires that correctness be proved for all the possible execution paths. Therefore, while judgments have been made by humans, CodeSurfer has been invaluable to understand interaction between functions, pointer aliasing and to avoid the examination of execution paths being overlooked.

The results obtained were entered in a database maintaining information about which tools issued a warning (for a file-line pair) and what kind of warning it was. Each warning was classified as either *FP* (False Positives) or *TP* (True Positives). *TP* were further classified as *BO* (Buffer Overflows), for all the buffer overflow problems, or *PE* (programming errors), for generic programming errors that are not buffer overflow problems.

A summary of the results of the categorization is presented in tables II and III. Tab. II shows the number of warnings issued by each tool for each program in the program basis and, cumulatively, for the whole program basis (last line). The warnings, as stated above, were classified according to their type (i.e. TP, and hence either BO or PE, and FP). Tab. III shows the same data but as percentages of the number of lines of the analyzed code.

At first glance, it can be noted that ITS4 and Rats behave more or less in the same way, moreover Flawfinder is the tool that raises the highest number of warnings (having the highest number of both FP and TP).

TABLE II

CATEGORIZATION OF THE REPORTED WARNINGS.

Program	Warnings		ITS4	Rats	Flawfinder
Net-tools	FP + TP		64	64	148
	TP	BO	36	37	44
		PE	0	0	0
	FP		28	27	104
WU-FTPD	FP + TP		128	114	255
	TP	BO	38	35	57
		PE	1	1	1
	FP		89	78	197
Pure-FTPd	FP + TP		45	32	156
	TP	BO	0	0	0
		PE	0	0	0
	FP		45	32	156
Program Basis	FP + TP		237	210	559
	TP	BO	74	72	101
		PE	1	1	1
	FP		162	137	457

TABLE III

REPORTED WARNINGS AS PERCENTAGES OF THE LINES OF CODE.

Program	Warning Ratio %	ITS4	Rats	Flawfinder
Net-tools	(FP+TP)/LOC	1.544	1.544	3.570
	TP/LOC	0.868	0.892	1.061
	FP/LOC	0.675	0.651	2.508
WU-FTPD	(FP+TP)/LOC	0.942	0.839	1.877
	TP/LOC	0.287	0.265	0.427
	FP/LOC	0.655	0.574	1.450
Pure-FTPd	(FP+TP)/LOC	0.178	0.127	0.618
	TP/LOC	0.000	0.000	0.000
	FP/LOC	0.178	0.127	0.618
Program Basis	(FP+TP)/LOC	0.552	0.489	1.301
	TP/LOC	0.175	0.170	0.237
	FP/LOC	0.377	0.319	1.064

Looking at Tab. III it can be noted that all the tools, on average, have an acceptable usability for low to medium sized software packages, in the sense that they issue a reasonable number of warnings. In fact, checking 1-2 lines of code every 100 lines, having, in the worst case, about 25 FP every 1000 lines of code is affordable. Of course, for large programs, this warning rate can be prohibitive. It is also worth noting that the ratios of warnings seem to decrease as the size of the programs increases, but this is probably due to the simultaneous increase of secure (or at least believed secure) code implementation of programs.

TABLE IV

THE COMPLETENESS FIGURE TP/LOE (PERCENTAGES).

Program	ITS4	Rats	Flawfinder
Net-tools	72.00	74.00	88.00
WU-FTPD	56.52	52.17	84.06
Pure-FTPd	0.00	0.00	0.00
Program Basis	62.50	60.83	85.00

Going on to consider the main performance figures, Tab. IV reports the completeness figure (TP/LOE) as a percentage for each tool-program pair and the cumulative figure for the whole program basis (last line). FlawFinder is the tool that, according to these experiments, gets closer to full complete-

ness. Its 85% of buffer overflow errors caught confirms the assumption that most buffer overflow errors are related to improper usage of library functions: the known errors which do not fall into this class are less than 15% of the total in our program basis. It can also be noted that the completeness of all the tools decreases when the defect rate and size of the programs increase.

TABLE V
THE SOUNDNESS FIGURE $TP/(TP+FP)$ (PERCENTAGES).

Program	ITS4	Rats	Flawfinder
Net-tools	56.25	57.81	29.73
WU-FTPD	30.47	31.58	22.75
Pure-FTPd	0.00	0.00	0.00
Program Basis	31.65	34.76	18.25

Tab. V shows the percentage of true positives over the total number of raised warnings for each analyzed program, and for the whole program basis, thus measuring the soundness of the tools. Unfortunately, as soundness is related to efficiency, the tools do not seem to be very efficient. In particular, ITS4 and Rats behave better than Flawfinder. Considering how the lexical tools work, it is obvious that their efficiency is better when they analyze programs containing lots of misused functions. Accordingly, it can be noted that the soundness of all the tools decreases when the defect rate and size of the programs increase. Specifically, it can also be noted that the soundness loss of ITS4 and Rats is higher than that of Flawfinder, when they analyze programs with lower defect rates and of larger sizes.

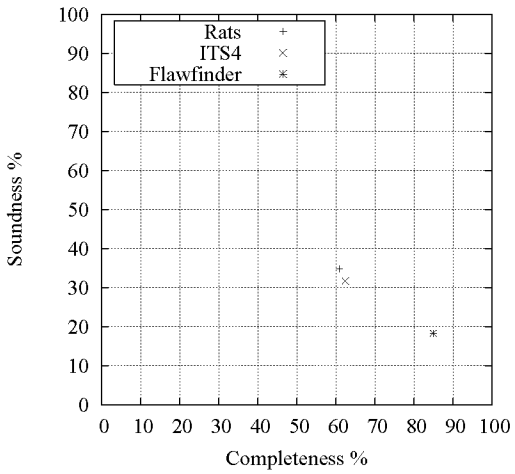

Fig. 1. Graphical representation of completeness and soundness (percentages).

Fig. 1 shows graphically how far the available tools are from the ideal one. The graph was obtained using the cumulative percentage values of soundness and completeness of the tools as Cartesian coordinates. Therefore, the ideal tool would be located in the top-right corner of the graph, since it is 100% sound and complete.

Looking at the figure it can be noted that all the tools are closer to being a complete tool than to being a sound tool. Moreover, the similarity of Rats and ITS4 is quite evident. The tool that actually most resembles the ideal one is Rats, followed by ITS4, while, as has already been noted, the results show that the most complete tool is Flawfinder.

The execution times of all the tools are in the range of few seconds, so execution time is not a critical parameter for our comparison.

A more interesting question to consider is the advantage that can be obtained by using more than one tool. In other words, it is interesting to study the set-relationships between the warnings raised by the various tools, and in particular those regarding true positives.

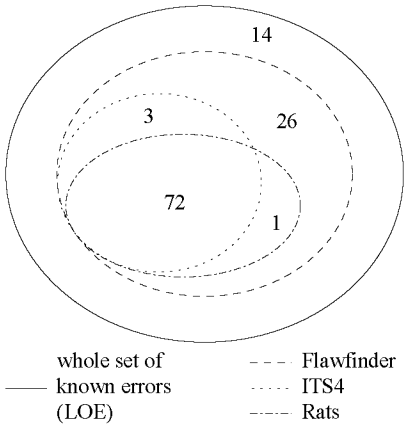

Fig. 2. Set relationships between sets of TP.

Fig. 2 shows such results. It is worth noting that the outermost ellipse contains all the TP which it was possible to discover also by using other tools and that 14 TP were not detected by any of the lexical analysis tools considered. Looking at the numbers of common TP, it can be noted that they are high, confirming the similarities of the tools considered.

An interesting result is that the TP reported by Flawfinder are a super-set of those reported by ITS4 and Rats. So, the errors that can be found by Flawfinder in our program basis are a strict super-set of those that can be found by each one of the other tools. A large intersection instead is observed between the TP reported by ITS4 and those reported by Rats, even though each tool reports at least one error not reported by the other.

In our experiments, Flawfinder detects more errors (i.e. it issues more TP) than the other tools, but this advantage has to be traded against a high number of additional FP (which amounts to about 20% of the total, as can be deduced from Fig. 1).

From this point of view, and on the basis of the preliminary results collected, it is possible to say that the use of ITS4 or Rats during development can be an efficient choice, since they have the best soundness index and show the best compromise between soundness and completeness. Flawfinder produces more warnings than the other tools and, hence, its use requires more code auditing work and its efficiency is low. However,

it has been helpful in finding more errors than the other tools. Thus, it should be used when a thorough analysis is needed and typically only for the final code auditing, so that as many residual errors as possible will be found.

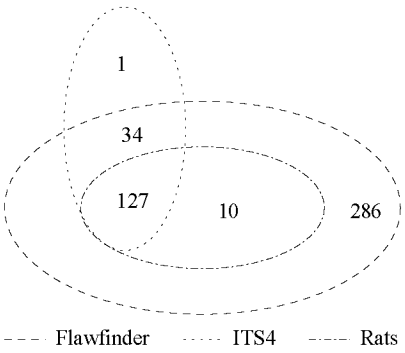

Fig. 3. Set relationships between sets of FP.

One possible strategy to minimize the impact of false positives is to use either ITS4 or Rats during development, marking the lines of code investigated and eventually patched, and then, at the end of development, use Flawfinder to find additional TP. In this last analysis, only the warnings related to lines not previously examined would be considered. It is worth noting that, even in this way, the human work needed is considerable, since Flawfinder emits many additional warnings. Moreover, the additional auditing activity turns out to be rather inefficient, as can be deduced by looking at Fig. 3, which shows the set relationships between FP. In fact, the number of additional FP when Flawfinder is used after ITS4 is 296 and it is 320 when Flawfinder is used after Rats, which corresponds to a decrease of soundness indexes of 13.43% and 16.52% respectively.

In general, it can be observed that all the available lexical tools appear to be rather far from being ideal in their behavior. In particular, while they seem to be able to reach considerable completeness (Flawfinder), their soundness still appears poor.

IV. Conclusions

Evaluation and comparison of static analysis tools for buffer overflow detection on an objective basis calls for experiments to be made in which the various tools are all tested under the same conditions and using real-world programs as test cases. Since no much previous work is available on this topic, this paper has proposed a possible basis for objective quantitative evaluation of static analysis tools for buffer overflow detection. The proposed experimental evaluation methodology has been used to make an initial evaluation and comparison of some lexical analysis tools when applied to network software packages. The experiments and their results are limited to the default settings of the tools. They show quantitatively the main features of the three available tools that have been tested: ITS4, Rats and Flawfinder. From these data it is possible to understand the relative strengths, weaknesses and peculiarities of the tools with their default settings. These results could subsequently be refined using a larger program basis and can be used as a reference for objective comparisons, possibly including new tools that will become available in the future.

The work presented in this paper naturally complements the work presented in [6], which addresses the problem of evaluating human factors, i.e. how useful, on average, the warnings given by the various tools are for the programmer.

Acknowledgment

This work was partially supported by the Italian National Council of Research in the framework of the project ICT-P06-IEIIT-M5: "Metodi e Strumenti per la Progettazione di Sistemi Software-Intensive ad Elevata Complessità".

References

[1] "Cert coordination center." [Online]. Available: http://www.cert.org/stats/cert_stats.html#vulnerabilities

[2] "Icat metabase." [Online]. Available: http://icat.nist.gov/icat.cfm

[3] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[4] "Buffer overflow and format string overflow vulnerabilitites," *Software Practice and Experience*, vol. 33, no. 5, 2002.

[5] J. Wilander. and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," in *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, Nov. 2002, pp. 68–84.

[6] J. Heffley and P. Meunier, "Can source code auditing software identify common vulnerabilities and be used to evaluate software security?" in *Proceedings of the 37th Hawaii International Conference on System Sciences*, Big Island, Hawaii, Jan. 2004.

[7] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, Dec. 2000, pp. 257–271.

[8] J. Viega, J. T. Bloch, T. Kohno, and G. McGraw, "Token-based scanning of source code for security problems," *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 3, pp. 238–261, 2002.

[9] M. Weber, V. Shah, and C. Ren, "A case study in detecting software security vulnerabilities using constraint optimization," in *SCAM*, Venice, Italy, 2001, pp. 3–13.

[10] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins, "Tool support for fine-grained software inspection," *IEEE Software*, vol. 20, no. 4, pp. 42–50, 2003.