Improving Network Applications Security: a New Heuristic to Generate Stress Testing Data

Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta
RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano
82100 Benevento, Italy
tina.delgrosso@unisannio.it, antoniol@ieee.org, dipenta@unisannio.it

Philippe Galinier, Ettore Merlo
École Polytechnique de Montréal
Computer Engineering Department
Montréal, Canada
philippe-2.galinier@polymtl.ca, merlo@info.polymtl.ca

ABSTRACT

Buffer overflows cause serious problems in different categories of software systems. For example, if present in network or security applications, they can be exploited to gain unauthorized grant or access to the system. In embedded systems, such as avionics or automotive systems, they can be the cause of serious accidents.

This paper proposes to combine static analysis and program slicing with evolutionary testing, to detect buffer overflow threats. Static analysis identifies vulnerable statements, while slicing and data dependency analysis identify the relationship between these statements and program or function inputs, thus reducing the search space.

To guide the search towards discovering buffer overflow in this work we define three multi-objective fitness functions and compare them on two open-source systems. These functions account for terms such as the statement coverage, the coverage of vulnerable statements, the distance form buffer boundaries and the coverage of unconstrained nodes of the control flow graph.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Reliability, Security

GECCO'05, June 25–29, 2005, Washington, DC, USA.

Keywords

Evolutionary testing, Security, Stress Testing, Test Data generation

1. INTRODUCTION

Testing activity consumes about 50% of software development resources thus any technique aimed at reducing software testing costs is likely to produce positive effects on cost reduction. Indeed, exhaustive and thorough testing is often too expensive and infeasible due to resource constraints.

Other techniques such as code inspection are known to be more effective but even more costly than testing. Unfortunately, defects slipped into deployed software may crash safety or mission critical applications causing threat to human beings or unacceptable economical losses. Indeed, buffer overflow is often exploited to gain unauthorized privilege or access to software systems. As reported by CERT[1], over 50% of software system vulnerabilities are caused by buffer overflows.

Recently [6] it has been proposed to use Genetic Algorithms (GA)s to generate test cases with the goal of identifying buffer overflow threats. The approach integrates knowledge of the potentially vulnerable statements and slicing to reduce the search space.

First, static analysis identifies statements that could be potentially affected by buffer overflows; these are usually statements performing operations on buffers (e.g., array access, or invocation of unsafe string copy functions). Second, slicing and data dependency are used to identify which inputs *affects* the execution of vulnerable statements. Finally, the search is guided by a fitness function balancing several terms in an attempt to smooth the landscape and efficiently guide the search.

This paper presents novel fitness functions and new results obtained by improving the approach proposed by Del Grosso et al [6]. The key idea is to drive the discovery

[1] `http://www.cert.org/summaries`

of buffer overflows by covering control flow graph unconstrained nodes, i.e., nodes that do not dominate nor post-dominate any other node [14]. Intuitively, test cases covering these nodes constitute a minimal set of test cases required to meet statement coverage criterion. Furthermore, we also considered the *distance* from the buffer boundaries as a factor guiding the search.

Finally, with respect to that proposed by Del Grosso et al, instead of generating a complete test suite, here each run of the GAs aims to detect a single buffer overflow. This decreases both required memory and average time to detect a buffer overflow.

The remainder of the paper is organized as follows. After a review of the related work, Section 3 summarizes the approach presented by Del Grosso et al [6] and describes how the approach has been improved. Section 4 describes the empirical study performed, detailing the context and presenting results. Section 5 concludes.

2. RELATED WORK

Many approaches and tools have been developed in the past to detect buffer overflow problems. Existing tools can be classified into *static* and *dynamic* tools.

Static tools detect the usage of potentially vulnerable functions, perform integer range analysis, track the string manipulation operations. Among existing tools, it is worth mentioning ITS4 [19], RATS [2] and *Splint* [12] (actually used for the work done in this paper). The main weakness of static tools is that they are rather imprecise, since the problem of statically detecting buffer overflows is, in general, undecidable.

DaCosta et al [5] proposed an approach to evaluate the security vulnerability likelihood of a function. The approach is based on the assumption that a function near a source of input may have a high likelihood of being vulnerable.

Haugh and Bishop [10] proposed an approach in which the source code was instrumented to track buffer length, and function parameters were checked to eventually warn when buffer overflow problems occur. Many other tools are available, for example *Purify* [9]. Ruwase and Lam [16] proposed an approach and a tool to prevent and detect buffer overflows. We share with this work the conclusion that, despite the many attempts to tackle buffer overflow statically and automatically, in most case program execution is the only way to tackle this problem.

Despite the number of available tools, to detect likely buffer overflows, traditional testing strategies provide a marginal contribution to generate test cases capable to actually discover buffer overflow [21]. Indeed, most of the existing testing strategies focus on *regular* program operations, that is unlikely to exhibit buffer overflows. Miller et al [15] proposed a tool, *Fuzz*, used to test UNIX utility giving them inputs consisting of large streams of characters.

Dynamic approaches suffer two weaknesses: i) the need for suitable test suites, i.e., program inputs capable of detecting buffer overflows and ii) where test cases are automatically generated, this is mostly done randomly. The approach proposed in this paper aims to perform a more efficient generation of test cases using an evolutionary testing approach.

Evolutionary testing was successfully used in the past for many different purposes, describing all of them is out of scope of this paper. An exhaustive survey has been recently written by McMinn [13]. Korel and Al-Yami [11] generated test cases that violate some assertion conditions, while Tracey et al [17, 18] used GAs and simulated annealing to generate test data with the purpose of exercising the exception handling. The evaluation of these approaches, however, was limited to 200 LOC programs.

Finally, as Binkley and Harman showed [4], program slicing and data dependency analysis can be effectively used to reduce the GA search space. We use a similar approach to determine whether it exists an cause-effect relationship between program inputs and variables used in likely dangerous statements.

3. THE PROPOSED APPROACH

This section summarizes the stress testing approach presented by Del Grosso et al [6] and highlights the novelty and contribution of this paper, in particular the new fitness functions.

The approach aims to generate test cases to cause and thus detect buffer overflow. The main steps are outlined in Figure 1 and are explained in the remainder of this section. Test cases are generated using evolutionary techniques based on GAs. To reduce the search space, static analysis and slicing are used. The Software Under Test (SUT) is first analyzed using static analysis techniques, to determine the set of source code lines that could potentially be exploited to cause a buffer overflow. Second, to reduce the search space of the evolutionary testing, program slicing [23] is used to detect the cause–effect relationship between program inputs and the lines identified in the previous step.

3.1 Detecting Vulnerable Statements

This first step relies on the source code static analysis and extracts the following information:

- a list of source code statements considered vulnerable to buffer overflow;
- a list of usages of potentially unsafe C functions; and
- when possible, the estimated size of buffers.

This step can be performed with several tools such as RatScan [1] (a graphical from end to RATS) or the freely available *Splint* [12]. In general, these and other comparable tools permit the identification of program vulnerable statements and of several programming errors (such as unused variables, inconsistent types, usage before definitions, etc.).

3.2 Slicing and Data Dependency

The aforementioned static analysis identifies a number of source code statements that can potentially be affected by buffer overflow problem. However, given a set of inputs $I \equiv \{i_1, \ldots, i_j\}$, and a potentially vulnerable statement, it may happen that only a subset of I influence such a statement. In other words, sometimes there do not exist data dependencies or control dependencies between a given input and the dangerous statement. As a consequence, the search space for determining test cases can be reduced via static slicing performed with tools such as the GrammaTech *CodeSurfer*.

In general, an input i_j is included in the search space *iff* it exists a source code line potentially vulnerable that is data dependent from the input i_j. If the input i_j is one of

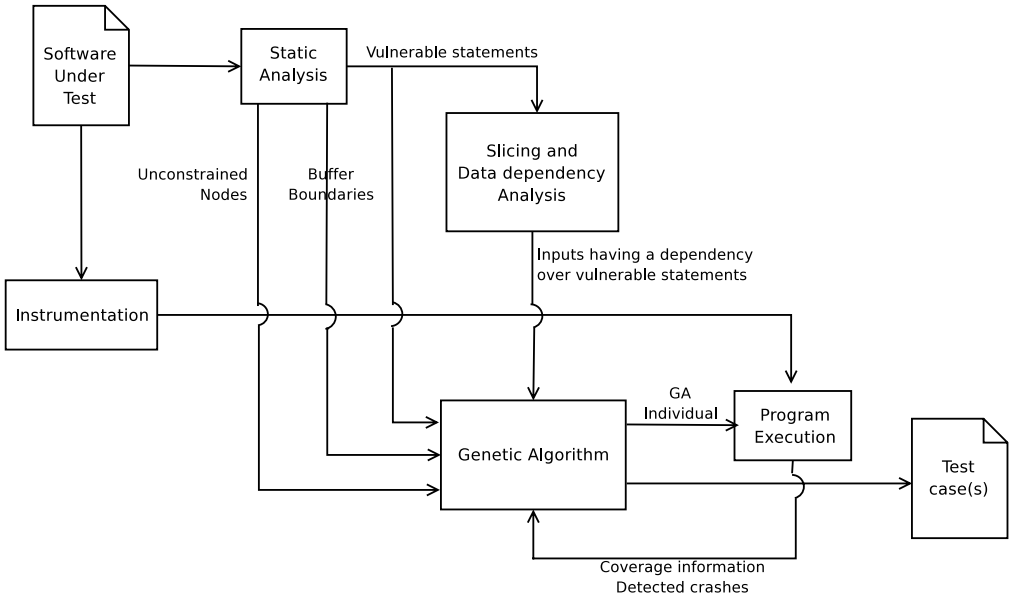

Figure 1: Test case generation process

the command–line parameters, whenever possible we capture the statement(s) that access such a parameter from the `** argv` variable. To this aim, we perform *backward slices* starting from vulnerable statements back to `argv` or to a statement using the `argv` variable.

3.3 Test Case Generation using GA

Using evolutionary testing techniques to solve our problem requires, as for any other problem tackled using GA, the definition of the chromosome, the crossover and mutation operators, the fitness function and other GA parameters (e.g., the number of generations or the population size).

The genome used by Del Grosso et al. [6] was a two-dimensional array genome where each row is a test case and each column is a program or procedure input (i.e., the k-th column is the k-th input parameter). Columns have different types matching parameter type.

In this new formulation, an individual represents a single test case, encoded as an array of test input data. Indeed, the old formulation was a compromise between the goal of detecting buffer overflows and a understated aim to generate test suites with the highest possible statement coverage.

However, the primary goal is to detect buffer overflows or other vulnerabilities. Once such a goal is achieved, the statement coverage has no practical use and no further action is required since program vulnerability is demonstrated by exhibiting a test case exposing such a vulnerability. The net effect of changing the genome representation, for a fixed population size, is a reduction in required memory and fitness evaluation time.

Choosing the right mutation and crossover operators is always a challenge, since they can greatly influence the results. We found that different crossover operators exhibit different performances for different types of input data. In particular, for numeric input the *whole crossover* was adopted (i.e., values of the offspring are linear combinations of the parents' values), while for strings, characters and, in general, for heterogeneous inputs, we used the simple one-point crossover [8].

The mutation operator was of type *creep mutation*. For numerical data, each gene is randomly selected from the genome, and then it is incremented of a particular value. For strings, mutation is performed by appending random strings to the existing gene. *Creep mutation* was adapted to our problem in such a way to promote buffer overflows. More precisely, if the input j has been selected for mutation, and if such an input has potential impact on a buffer B then the mutation is a function of the buffer B estimated size.

Optimization problems aim at searching for a solution in a search space that minimizes (or maximizes) a fitness function. In this paper three different fitness functions were compared:

1. The fitness function proposed by Del Grosso et al. [7] (we refer it as *Vulnerable coverage fitness*);
2. A modified fitness function, accounting for nesting level reached (we refer it as *Nesting fitness*); and
3. A fitness accounting both nesting and a factor accounting for the distance from boundaries in buffer accesses (referred as *Buffer boundary fitness*).

3.3.1 Vulnerable coverage fitness

First and foremost, the fitness should account for the ability of a test suite to discover crashes. However, a fitness function solely related to detected or undetected vulnerability would lead to a flat landscape (representing program executions that do not generate exceptions) with few spikes (exceptions), and therefore to a random search.

To obtain a more favorable landscape driving the search towards the spikes, dynamic information needs to be used. It was experienced that some possible elements to use in a fitness function are the following ones:

- *statement coverage*: a thorough code coverage contributes to increase the likelihood that an exception is raised or that a buffer overflow is caused.
- *vulnerable statement coverage*: much in the same way as above, covering vulnerable statements increases the likelihood of generating exceptions and overflows. Clearly, covering such statements must have a higher priority than others, therefore this factor needs to be weighted more than the *statement coverage* factor.
- *number of executions of vulnerable statements*: we found that, together with the *vulnerable statement coverage* the number of executions of vulnerable statements needs to be considered. In fact, the higher this number is, the higher the likelihood of generating exceptions and buffer overflows will be.

Overall, the fitness function, to be maximized, was defined by Del Grosso et al [6] as:

$$F(g) = w_1 \cdot scov \ + \ w_2 \cdot log(k) \cdot vcov \ + \ w_3 \cdot crash \quad (1)$$

where:

- *scov* is the statement coverage ratio;
- *vcov* is the vulnerable statement coverage ratio;
- *crash* represents the number of buffer overflows detected by the test suite;
- k is the number of executions of vulnerable statements. To avoid that it excessively dominates other factors, it is normalized using a logarithm; and
- w_1, w_2 and w_3 are real, positive weights, indicating the contribution of each factor to the overall fitness function.

3.3.2 Nesting fitness

Despite the encouraging results, the above fitness function reflects the double contrasting goals of producing a test suite with high statement coverage and to whilst detecting buffer overflows. A heuristic for improving the fitness function comes from the definition of *unconstrained nodes* [3] in a control flow graph. Unconstrained nodes are nodes that do not dominate nor postdominate any other node. In other words, to cover an *unconstrained node*, it is necessary to properly design a test case that meets the path condition to such a node.

We also observed that *unconstrained nodes* often correspond to statements at the maximum nesting level in the program or procedure under test. Obviously, each test case covering an *unconstrained node* will also cover many other statements and, in particular, vulnerable statements. Finally, if a vulnerability is detected, then the test data generation process achieves his goal.

According to the above observations, the fitness function (1) was modified as follows:

$$F(g) = w_1 \cdot scov \ + \ w_2 \cdot log(k) \cdot vcov \ + \ w_3 \cdot nesting \quad (2)$$

where *nesting* is the observed maximum nesting level corresponding to the current test case.

3.3.3 Buffer boundary fitness

More sophisticated and complex fitness functions can be conceived, for example, fitness functions that explicitly use buffer boundaries in the fitness computation. However, since multiple buffers with, possibly, dynamically allocated sizes may be involved in each SUT, the complexity of fitness evaluation and the information collection process will be increased. At the current stage of the research, we are more focused on demonstrating the feasibility of the approach and on evaluating simple fitness functions that can be easily computed with state–of–the–art tools and technology. Some problems, such as pointer handling and dynamically defined buffer sizes will be part of future work.

It can be argued that the above fitness functions (i.e., equations (1) and (2)) do not directly account for terms related to buffer overflow. To further improve the fitness function, at least for programs where the buffer sizes can be estimated at compile time, we added to equation (2) a new term accounting for the *distance* from the buffer *boundaries*. More precisely, this new fitness is defined as:

$$\begin{aligned} F(g) \quad &= \quad w_1 \cdot scov \ + \ w_2 \cdot log(k) \cdot vcov \ + \\ &w_3 \quad \cdot nesting + w_4 \cdot max_i\{min_j(L_{i,j} - SB_i)\} \end{aligned} \quad (3)$$

where SB_i is the i-th buffer estimated size and $L_{i,j}$ is the i-th buffer upper limit that was read or written in the $j-th$ buffer access. Clearly, the term favors test cases having the smallest distance from the buffer *borders* and promotes test cases accessing memory locations above the buffer sizes.

Weights $w_{1,\ldots,4}$ are selected using a trial-and-error, iterative procedure, guided by plotting the values of the three factors and of the fitness function over the GA evolution.

3.3.4 GA parameters

We set-up, according to our experience and to what is proposed in literature, the other GA parameters. In particular, we used an elitist GA, with the two best individuals kept alive across generations. The population is composed of 70 individuals, and we have analyzed the GA behavior over 500 generations (while this parameter can vary across case studies). Finally, the we set the crossover probability $p_{cross} = 0.7$ and the mutation probability $p_{mut} = 0.01$.

3.4 Tool support

As described above, static analysis devoted to identify potentially vulnerable statements was performed using *Splint*, while data dependencies and slices were computed using *CodeSurfer*.

The GA was implemented using *GaLib*[22]. The code coverage for `C` programs was measured using the freely available coverage tool *gcov*, distributed with the GNU C compiler[2].

To compute the fitness function, our tool executes the instrumented source code. The rows of the individual genome (for the vulnerable coverage fitness) or the individual itself (for the other two), are the input test cases. Then, the program output is used to detect exceptions and segmentation faults (i.e., the *crash* factor), while the *gcov* coverage output used to compute the *scov*, *vcov* and k factors.

[2] `http://www.gnu.org`

4. EMPIRICAL RESULTS

To gain preliminary evidence of the relative performance of the newly proposed fitness functions, we experimented with two different case studies: a white noise generator[3] and a function contained in the FTP client *cmdftp*[4].

Table 1 reports, for each test case, LOC, cyclomatic complexity and vulnerable LOC (as detected by *Splint*.

Case study	LOC	Vulnerable Stmts	Cycl compl.	# of Inputs
Whitenoise	331	20	80	7
readline_tab (FTP client)	45	5	18	3

Table 1: Case study metrics

The white noise generator can be considered as somewhat representative of scientific programs, while the FTP client as representative of network applications (for which preventing buffer overflows is very relevant).

4.1 Random search

A preliminary step was conducted to assess whether or not the fitness function has an influence, if any, on the search process. To this aim we generated test data starting from a fixed initial population applying the genetic operators but disregarding the fitness value in the selection process. The procedure was repeated several times for both programs, each time 500 generations were created but no crash was observed.

A further preliminary experiment was conducted to assess the effectiveness of *pure* random test generation. The experiment described above was conceived so that the random test data generation was initialized with the same values used for the *real* GA search, in other words, the same mutation and crossover operators were applied but fitness values were disregarded. However, it can be argued that creep mutation can influence results, though no crash was observed. To avoid possible bias, we did a further preliminary test; we simply randomly generated numbers and string and generated values were used as parameter inputs. Again, no buffer overflow was observed and the computation was stopped after about 300000 random evaluation for FTP client and 150000 for the white noise generator.

4.2 Searching from a given initial point

This approach aimed at mimicking a situation in which the knowledge of an expert is used to define the initial search point. In other words, we assume that a domain expert defines initial input values; the search algorithm starting from that point in the search space moves toward search space regions were buffer overflow is observed. Data were collected for the fitness functions described in Section 3, i.e., equations (1), (2) and (3). Experiments were replicated 10 times and collected data plotted to gain a first insight subsequently we applied hypothesis testing to the three fitness functions.

Figure 2 shows the cumulative plot of crashes for the Whitenoise program; a similar plot was obtained for the FTP client. Fig. 2 data show a ranking of the three fitness functions and that the best performance is obtained by the fitness (3). Summary statistics are reported in Table 2. No substantial difference between means end medians were observed and thus median is not reported.

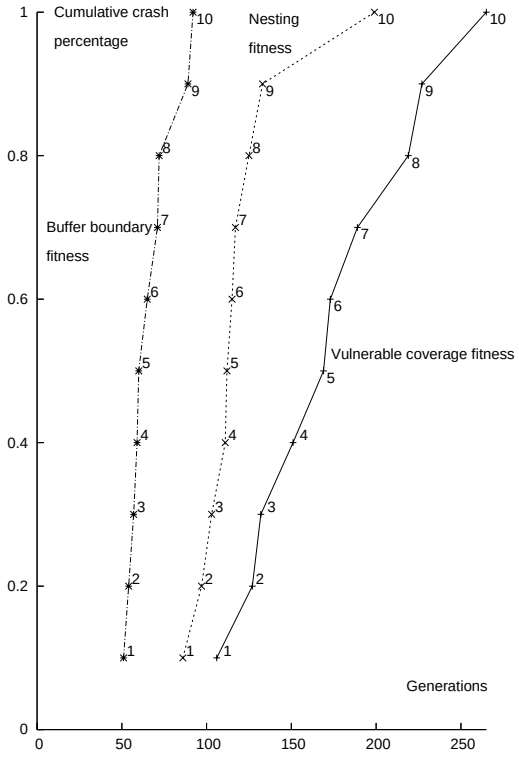

Figure 2: Whitenoise cumulative percentage of crashes for fixed initial population.

Program	Vulnerable Coverage Fitness (1)	Nesting Fitness (2)	Buffer Boundary Fitness (3)
	Mean (Std.Dev.)	Mean (Std.Dev.)	Mean (Std.Dev.)
Whitenoise	125 (36)	120 (31)	67 (14.1)
readline_tab (FTP client)	155 (53)	123 (40)	76.7 (35.47)

Table 2: Search from an initial point defined by an expert.

Hypothesis testing was performed by means of a modified t-test comparing the means of two independent samples, with different variances [20], assuming as the null hypothesis that the populations have the same mean i.e., no difference between means exists. Table 2 data supports with strong evidence (5 % confidence level) that *buffer boundary* fitness (3) outperforms both the *vulnerable coverage* (1) and *nesting* (2) fitness. Null hypothesis cannot be rejected between fitness functions (1) and (2). This clearly indicates the benefit of using buffer size and boundary distance to drive the GA evolution.

[3] `http://www.eecs.umich.edu/pelzlpj/whitenoise`
[4] `http://freshmeat.net/projects/cmdftp/`

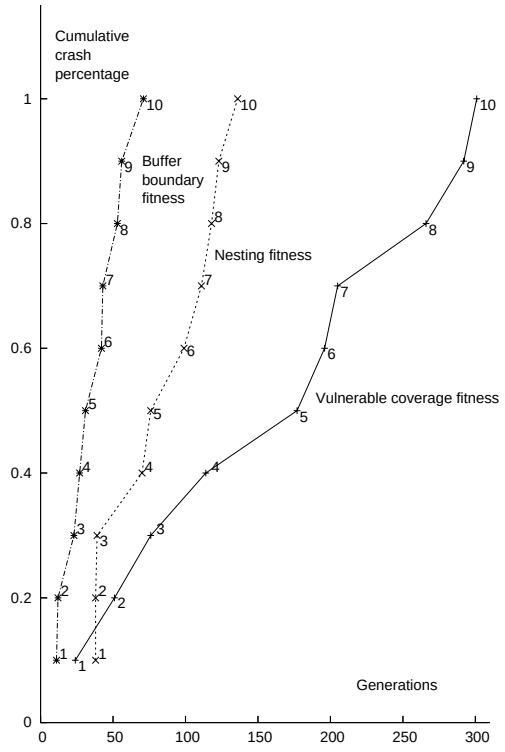

Figure 3: FTP client cumulative percentage of crashes for random initial populations.

4.3 Searching from a random initial population

This situation corresponds to a scenario where no *a-priori* knowledge is available. The initial population is randomly generated from uniform distributions over the legal value ranges and string length for numerical and string arguments respectively.

As in the previous scenario each experiment was replicated 10 times and data were collected. Figure 3 reports the cumulative percentage of crashes for the FTP client; as for data plotted in Figure 2, a ranking of fitness functions is evident. Obtained summary statistics are reported in Table 3.

Program	Vulnerable Coverage Fitness (1)	Nesting Fitness (2)	Buffer Boundary Fitness (3)
	Mean (Std.Dev.)	Mean (Std.Dev.)	Mean (Std.Dev.)
Whitenoise	144.7 (53.0)	89.9 (15.7)	62 (10.7)
readline_tab (FTP client)	170.2 (100.4)	84.3 (37.8)	36.9 (19.6)

Table 3: Search from a random initial population.

Table 3 data supports with strong evidence (5% confidence level) that fitness function (3) outperforms the fitness functions (1) and (2). Furthermore, at the same confidence level, the fitness function exploiting the unconstrained node idea i.e., nesting fitness performs better than the fitness proposed in [6] corresponding to equation (1). This seems to suggest that the expert wrongly selected the initial point for the populations of Table 2 or at least that the initial point challenges fitness (1) and (2). Interestingly, the fitness (3), on our data set, seems to be able to "escape the trap", supporting the idea that it moves the search in a smoother landscape.

Data are still preliminary and it pays to be cautious in the interpretation. At the time of writing we are working to increase the number of trials, to evaluate new string crossover operators and to enlarge our dataset of programs.

5. CONCLUSIONS AND WORK IN PROGRESS

The proposed approach used a consolidated technique, i.e., evolutionary testing, with the purpose of detecting program crashes or exceptions caused by buffer overflows. The evolutionary approach has been complemented with existing static analysis techniques, slicing and data dependencies. Such techniques contribute to reduce the search space and to allow the GA converging faster. The proposed fitness functions account for statement coverage, vulnerable statement coverage, number of executions of vulnerable statements, maximum reached nesting and distance from the limit of buffers to drive the evolution toward fitness convergence, avoiding a flat landscape and thus a random search.

Our preliminary results support the hypothesis that maximum reached nesting and distance from the limit of buffers help to guide the search. In particular, a fitness function accounting for the distance from buffer boundaries, on our data set, outperforms fitness functions not using the same factor.

Work-in-progress is devoted to improve genetic operators for strings, to augment the number of trials and, in general, to apply the approach to large-industrial software systems.

6. REFERENCES

[1] Beetlesoft RatScan. *http://www.beetlesoft.com*.

[2] Secure software solutions, rats, the rough auditing tool for security. *http://www.securesw.com/rats/*.

[3] G. Antoniol and E. Merlo. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *CASCON*, October 1999.

[4] D. Binkley and M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering*, 30(11):715–735, Nov 2004.

[5] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 266–276, Amsterdam, The Netherlands, Oct 2003.

[6] C. Del Grosso, G. Antoniol, and M. Di Penta. An evolutionary testing approach to detect buffer overflow. In *Student Paper Proceedings of the International Symposium of Software Reliability Engineering (ISSRE)*, St. Malo, France, Nov 2004.

[7] C. Del Grosso, M. Di Penta, and G. Antoniol. An evolutionary testing approach to detect buffer overflows. In *International Symposium on Software*

Reliability Engineering (student paper), pages 77–78, St Malo, Bretagne, France, November, 2-5 2004.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

[9] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proceedings of the Winter USENIX Conference*, Washington, DC, USA, Aug 1992.

[10] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities.

[11] B. Korel and A. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the International Conference on Software Engineering*, Berlin, Germany, 1996.

[12] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *In Proceedings of the USENIX Security Symposium*, Washington, DC, USA, Aug 2001.

[13] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14:105–156, June 2004.

[14] E. Merlo and G. Antoniol. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *Proceedings of CASCON-99 - ponsored by IBM Canada and the National Reasearch Council of Canada*, pages 173–186, Mississauga (Ontario), November 8-11 1999.

[15] B. Miller, L. Fredricksen, and B. So. Empirical study of the reliability of unix utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, Dec 1990.

[16] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, Feb 2004.

[17] N. Tracey. *A search-based automated test-data generation framework for safety critical software*. PhD thesis, University of York, 2000.

[18] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software - Practice and Experience*, 30(1), 2000.

[19] J. Viega, J. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, pages 3–17, Dec 2000.

[20] S. G. W. and C. W. G. *Statistical Methods*. Iowa State University Press, 1989.

[21] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, USA, Feb 2000.

[22] M. Wall. GAlib - a C++ library of genetic algorithm components. *http://lancet.mit.edu/ga/*.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.