An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A case of aiming in dim light

Sanjay Rawat, Laurent Mounier
Verimag Lab
Université Joseph Fourier
38610, Gières, France
{*Sanjay.Rawat, Laurent.Mounier*}*@imag.fr*

***Abstract*—We propose an approach in the form of a light weight smart fuzzer to generate string based inputs to detect buffer overflow vulnerability in C code. The approach is based on an evolutionary algorithm which is a combination of genetic algorithm and evolutionary strategies. In this preliminary work we focus on the problem that there are constraints on string inputs that must be satisfied in order to reach the vulnerable statement in the code and we have very little or no knowledge about them. Unlike other similar approaches, our approach is able to generate such inputs without knowing these constraints explicitly. It learns these constraints automatically while generating inputs dynamically by executing the vulnerable program. We provide few empirical results on a benchmarking dataset-Verisec suite of programs.**

***Keywords*-fuzzing, evolutionary algorithm, buffer overflow, vulnerability, data- and control-flow.**

I. Introduction

"*Security should be the internal property of the software*" is the phenomenon being exhibited by most of the security organizations, e.g. [1]. Under this paradigm of software development, security should be part of SDCL which implies that right from the beginning, software should be tested and verified for its security properties (for example, buffer overflow (BoF) vulnerability detection). A software can be tested *statically or dynamically* on its source (or binary) code, known as white-box approach. This approach has the advantage of having access to software's internals, for example, high-level code, control- and data-flow graphs etc. On the other extreme lies black-box approach wherein we do not have access to software internals. All we can do is to pass inputs and observe outputs. Obviously, this approach is less precise than the former one, but sometimes, it is very practical and easy to apply on a software to get quick results to a certain acceptability. In security literature, this technique is also termed as *fuzzing* [2]. It should be easy to observe the relationship between performance/result and resources required to perform analysis by an approach. White-box approaches are more complete in the sense of providing larger coverage of vulnerabilities detection; but they require more resources to conduct analysis. Black-box approach, on the other hand, is less complete as it cannot detect all the vulnerabilities whose detection is not straight forward. For example, if there are constraints on inputs to reach a vulnerable statement, fuzzing may have hard time to generate inputs that can reach that statement. But black-box based approaches require less resources to perform analysis. Figure 1 depicts this relationship by showing two clusters at both end of the curve. Cluster A represents approaches with less detection and less resource requirement. Approaches like blind fuzzing falls in this cluster. Cluster B shows approaches that provide more vulnerability detection with heavy resource requirement. White-box approaches belong to this cluster. As also shown in the figure as dotted curve, in this study we want to distort the curve a bit by proposing an approach which requires less resources to perform analysis with comparable results with approaches belonging to cluster B.

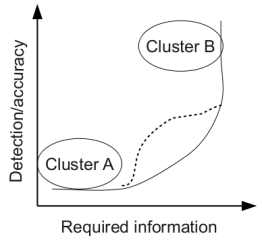

Figure 1. Relationship between performance and amount of resources/information required by an approach for security vulnerability detection.

Recently, there has been an increasing interest in dynamic analysis of vulnerabilities specially in the cases where there are constraints on string inputs and task is to generate inputs that satisfy those constraints and activate vulnerabilities [3][4]. Dynamic analysis is very accurate (and faster) in detecting certain vulnerabilities, but it faces a hard time in executing paths that activate the vulnerability. This involves the generation of inputs that execute the malicious path. As mentioned above, such techniques learn about program's internal by doing a static analysis of the application to gather malicious information flow, constraints on inputs etc. and generate inputs, which is also termed as *intelligent* fuzzing [5][6]. Looking from this perspective of dynamic input generation, problem of generating valid inputs can be mapped to *searching* for inputs that satisfy a given

path. Therefore, intelligent fuzzing can be considered as a *search problem*. As a consequence, evolutionary algorithms (EA) have been used to generate inputs for testing [7]. In this paper, we report our preliminary work that uses a hybrid EA (genetic algorithm and evolutionary strategies) for generating inputs to detect BoF vulnerabilities at runtime. We propose an approach to vulnerability analysis which generate string inputs without knowing constrains on these inputs. To the best of our knowledge, ours is the first work to explore the use of evolutionary strategies in dynamic vulnerability detection. We make use of static code analysis to generate *vulnerability execution path* by using taint data-flow technique. Dynamic analysis is used to generate inputs that execute that path. In order to learn constrains on inputs and satisfying them, we borrow tools and techniques from set-theory and *indirect constraint handling* approach of GA [8]. We device a new fitness function that computes fitness values for inputs based on the runtime dynamics of the application, which in turn incorporates constraint handling. A high level diagram of our approach is depicted in figure 2.

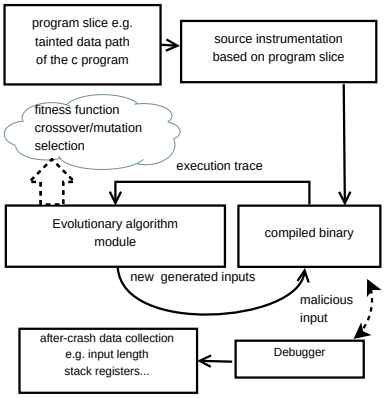

Figure 2. Flow diagram of the proposed approach wherein each major step is depicted in a box.

Program slicing is the static analysis component of the approach. It generates a tainted path from source (malicious inputs) to sink (vulnerable statement[1]) i.e. we get a *sequence* of statements that must be executed to reach the sink. Based on this information, we instrument the source code at statement level and execute it with inputs to generate runtime statistics about the program in terms of statement's execution frequencies. Proposed EA generates newer inputs from the runtime statistics. As we are targeting BoF vulnerability, EA keeps generating inputs until we get a program crash. At this point, we run the binary with a debugger to collect information about various internal structures containing information related to the crash. This action is performed to get information that is useful in order to generate real exploits. Based on this step, we can infer how easy or difficult it is to exploit this vulnerability.

[1]Currently, we look for calls to well-known vulnerable functions like strcpy, sprintf etc. In future, we plan to incorporate signatures for other vulnerable patterns so that these can be used as 'sink'.

Thus our approach shows not only the presence of the vulnerability but also its possibility of exploitation in the real world. The main contributions of the approach are:

- Automated malicious input generation.
- New fitness function based on runtime program behavior that handles constraint indirectly without knowing them explicitly.
- Hints on the level of exploitability of a particular vulnerability.

The rest of the paper is organized as follows: Section II provides necessary background to cover topics used in our proposed work, including taint data analyis and evolutionary computing. We provide a detail description of the proposed method in section III, which is followed by section IV on experimental details and discussion. To draw some comparison, section V summerizes relevant work. Finally, we conclude in section VI with a note on future work.

II. Background

In this section, we briefly discuss few topics that are building blocks of our method. Our intention is to generate inputs that can activate a vulnerability such that all the constraints along the path from source to sink are satisfied. To make discussion convenient, we take the following C code (listing 1) as an example:

```
int copyData(char *checkD, char *c2, char *buf){
  char *outD=buf;
  char *c1;
  int nchar = 0, outD = 0; // index into outfile
  for (c1=checkD;*c1 != '\0';c1++){
    if (*c1 == '='){
      if (*c1++ == '\0')
        break;
      // =\n: continuation; signal to caller it's ok to pass in more infile
      // BAD: forgot to reset out
      if (*c1++ == '\n'){
        nchar = 0;
        continue;
        }
      else {
        if (*c2 == '\0')
          break;
        nchar++;
        if (nchar > BASE_SZ)
          break;
        *outD = *c1; /* BAD */
        outD++;
        }
    }...
```

Listing 1. Excerpt of a vulnerable C code from Verisec suite of programs.

The sample code in Listing 1 is taken from Verisec suite of vulnerable programs [9]. This program is vulnerable assuming that `checkD` and `c2` are user controlled buffer strings and `buf` is fixed size buffer (of length `BASE_SZ`). If `checkD` contains more than one "=\n" (combination of = and \n to pass checks at lines 6 and 11)), `buf` can be overflowed by `c2` (line 21). So, here the problem is to generate a `checkD` string that contains "=\n" character sequence. Following two techniques - taint analysis and evolutionary computing - are used to solve the aforementioned problem.

A. Taint Analysis

As stated in the introduction we are interested in this paper in *path-oriented* fuzzing, which is based on two successive steps:

1) path selection: how to choose the paths to be exercised by the fuzzer;
2) path execution: how to find the corresponding program inputs.

We briefly discuss here how some dedicated taint analysis techniques can help to answer the first question, the rest of the paper being dedicated to the second one.

Taint analysis: Identifying the set of instructions whose execution can be influenced by user inputs can be achieved by using a (classical) technique called taint analysis. Roughly speaking, it consists in associating a (binary) *taint information* to each program variable, at each program location. Hence, a variable v is marked as "tainted" at location l if the value of v at this location depends on a user input. Let us consider, for example, the C function presented in Listing 1. We assume here that, initially, parameters `checkD` and `c2` are tainted and `buf` is untainted. Then, variable `c1` becomes tainted at location 5 (it is assigned `checkD`), and variable `nchar` becomes tainted at location 12 (it is assigned in the scope of a tainted condition, thus its value is indirectly influenced by a user input).

Taint Dependency Sequences (TDS): Once a taint analysis has been performed, a useful information is to characterize the set of execution paths (for instance on the control flow graph) that *effectively* corresponds to tainted executions leading to a vulnerable statement. In [10], a solution is proposed by means of *Taint Dependency Sequences* (TDS, for short). More precisely, each TDS $t =< l_1, l_2, \ldots, l_n >$ associated to a variable v is a sequence of program locations l_i that a program execution path should traverse in order to reach l_n with an input-dependent value assigned to v. Thus, if l_n corresponds to a vulnerable statement, this TDS set exactly characterizes the set of "dangerous" execution paths. A TDS example obtained from Listing 1 is the following: $< 5, 6, 11, 5, 6, 21 >$. It means that traversing these program locations (in this order) is a way to reach the vulnerable statement 21 with a tainted value of `c1` (which may correspond to a potential exploit).

The authors of [10] developed a tool, freely available from [11] that computes TDS from arbitrary C programs. We used this tool for the experiments presented in section IV.

B. Evolutionary Computing

Evolutionary computing (EC) is the branch of computer science that draws inspirations from natural selection and human evolution [12]. An evolutionary algorithm (EA) is an algorithm based on the principles of EC. There are many variants of EA. Most widely known variants are: genetic algorithms (GA), evolutionary strategies (ES), evolutionary programming (EP) and genetic programming (GP). A typical EA has the following components:

- Population- a set of candidate solutions.
- Representation- definition of individuals belonging to the population.
- Evaluation Function (fitness function)- measurement of individual's ability to provide desired solution.
- Parents selection mechanism.
- Recombination and mutation- mechanism to generate new individuals (solutions) from older ones.
- Survival selection mechanism- selection of fittest individuals to generate new population.

Based on the representation of the individuals, recombination and mutation perform operations on individuals to generate new candidate solutions. While recombination takes two or more parents to generate new children, mutation operates on a single parent to generate single child. These two functions are seen as ways to bring exploration and exploitation in the solution space respectively. Regardless of variant being used, a typical EA performs the following steps to solve a problem:

Algorithm 1 Pseudo-code of a typical evolutionary algorithm

```
INITIALIZE population with random candidates
repeat
    SELECT parents
    RECOMBINE parents to generate children
    MUTATE offsprings
    EVALUATE new candidates with fitness function
    SELECT fitest candidates for the next population
until TERMINATION CONDITION is met
return BEST Solution, if found
```

We elaborate on GA and ES a bit more as our proposed method is inspired from these two algorithms.

In GA, recombination takes place in the form of crossover by combining two parents to generate two children. Mutation works by flipping bits of an individual (in case of bit-string representation) or adding a small quantity to the existing candidate solution (in case of real-values representation). The rate (and amount) of change remains constant, irrespective of search space. One aftermath of this is that GA may be trapped in local optima. ES, on the other hand, has strategy parameters, associated with each candidate solution. These strategy parameters often guide the mutation rate[2]. Basically, strategy parameter brings in self-adaptation in mutation rate. A typical method is to use Gaussian convolution for mutation [13]. Under this scheme, each individual is represented as $\langle x_1, x_2, ..., x_n, \sigma_1, \sigma_2, ..., \sigma_n \rangle$, where σ_i is standard deviation, used to draw a random number from Gaussian distribution with mean zero and standard deviation σ_i. Therefore, each individual x is mutated to x' by performing following operation:

$$x' = x + N(0, \sigma) \tag{1}$$

In practice, σ may also undergo a change by the same rule depicted in eq. 1. The purpose of using Gaussian distribution

[2]It should be noted that in ES, recombination is optional and applied on two or more parents to generate one child and therefore, strategy parameters are specific to mutation.

is that random number generated from it tends to be smaller *most of the time*, but takes a jump to produce a large number *occasionally*. The overall effect of this phenomenon is that if search is trapped in local optima, an occasional jump may bring it out of it and the search may begin towards global optima. How often this jump is required is adjusted by adapting σ, associated with each individual.

In our proposed method, we combine GA and ES to produce a simple and better malicious inputs generation algorithm. Our method is not fully ES as we do not associate a separate σ with each individual. Rather we maintain a global σ for the current generation. We use (μ, λ) ES in our implementation. We start with a population of λ solutions. Based on the fitness value, we choose μ fittest individuals and apply recombination and mutation to form λ/μ new children per individual. These newly generated λ individuals replace old generation of λ individuals. The choice of crossover over mutation is made using random number from uniform distribution i.e. we use mutation, but with a probability, we replace mutation by crossover. This gives us the chance to maintain good features of parents, while exploring more options to achieve optimal solution. The specific details of various components are provided in the next section.

III. Proposed Approach

Given a C program with string inputs, we calculate TDSs for a set of tainted input and vulnerable statement. Let $t =< l_1, l_2, \ldots, l_n >$ be the chosen TDS. Based on the each l_i, the source code is instrumented at statements corresponding to each l_i so that at runtime, execution profile of the program can be built in terms of frequencies of each of the l_is. Initial population I consists of randomly generated strings of ASCII printable characters. To calculate the fitness value of each $i \in I$, we define fitness function F_i as follows:

$$F_i = \sum_{j=1}^{k} w_j \times f_{ij} \tag{2}$$

where $w_j \in W$ corresponds to weight associated with $l_j \in t$ and f_{ij} is the execution frequency of l_j for $i \in I$. W is the set of weights for each TDS t. Selection of appropriate values for W is of paramount importance for our fitness function to yield good results. For example, if we choose to select equal weights for each l_js, and there is a nested structure in the program such that one (or more) l_j is at inner most nested statement (line 11 in Listing 1), then it will be executed very less time, as compared to l_j at outer most statements. As a result, inputs which are reaching only till outer statements will also have a high fitness values, thereby giving EA a wrong fitness impression. Therefore, for such situation, we need to assign a higher weight to inner one as compared to outer one. To tackle this issue, we propose following approach:

Dynamic Weight Calculation: Runtime analysis reveals a lot about execution trace of the program by means of frequencies corresponding to l_js. In [14][15][16] the authors discuss the dynamic analysis of program by means of *frequency spectrum analysis*. In particular, by observing the frequencies of statements, branching structure can be approximated to some extent. As discussed in above section, it may be noted that rare executed statements will have low frequencies as compared to easily executed ones. It may also be noted that statements lie deep inside a nested structure belong to rare statements. Therefore, frequency spectrum of l_js captures *IsNested* type of structure among l_js. To calculate frequency spectrum, we start with a set of inputs $I := \langle i_1, i_2, ..., i_m \rangle$ to get a frequency matrix $freq = \begin{pmatrix} \cdots \\ f_{ij} \\ \cdots \end{pmatrix}$, where f_{ij} is the frequency of l_j for the input $i \in I$. Based on the above assumption, we calculate weights dynamically by counting global frequencies of l_js and inverting them i.e. for a set of inputs I and the matrix $freq$, the weight w_j of l_j is calculated as follows:

$$w_j = \frac{1}{\sum_{i=1}^{m} f_{ij}} \tag{3}$$

Another heuristic is that labels in TDS, which are closer to vulnerable statement should have larger weights than the ones which are relatively away. This is because any input which is reaching nearer to vulnerable statement has greater chance of reaching the vulnerable statement. Given this, we multiply weights with a function whose value increases as we move from beginning to end in a TDS. One example of such function may be the indices of the elements in the TDS i.e. for $l_j \in t$, the function value will be j. Therefore, considering above formulation in mind, we update w_j of eq. 3 as $w_j \leftarrow w_j \times j$. Substituting the values obtained by updated eq. 3 into the eq. 2, we get the fitness value of each input. These values are sorted in descending order to select μ inputs to be used in recombination and mutation.

Recombination: From μ fittest inputs, we randomly choose two inputs and perform 4-point crossover, which is akin to discrete recombination of ES to form one new input. For example, let n1:= *ABCD* and n2:= *EFGH* be two chosen inputs, then new input is *AFCH*.

Mutation: Mutation is the main component of the ES based techniques. We perform mutation on individual inputs by adding or replacing characters which are generated from a different set of characters M. For example, let input be: *ABCDE*, then mutation can generate following strings with new characters $xy \in M$: *ACDE — xyABCDE — ABxyCDE — ABCDExy*. In the proposed method, this is the main step which is responsible for learning constraints automatically. As we start with no knowledge about the constraints that should be satisfied by the inputs, we desire that M should contain these characters. We resort to set-theory definitions

to present heuristics for approximating these constraints.

The *symmetric difference* between two sets A and B is defined as $A \triangle B = (A \cup B) \setminus (A \cap B)$. It contains elements which are either in A or B but not in both. If we calculate a set $diff1 = i_{1b} \triangle i_{2b}$ (i_{1b} and i_{2b} are first two best inputs), it should be containing characters that contributed in making i_{1b}, i_{2b} best[3]. There could still be few *bad* characters present and *good* characters absent in $diff1$. If we look at worst inputs (we denote them by i_{1w} and i_{2w}), we can learn something about these characters. These inputs are worst because either they have *bad* characters or they do not have *good* characters. Therefore, if we calculate $diff2 = i_1w \triangle i_2w$, it may captures few good characters as well[4]. Initialized as empty set, in each generation, the set M will be updated as $M := M \cup (diff1 \triangle diff2)$. The underlying idea of doing so is that this set should serve as the over-approximation of set of characters, constituting the constraints. But this set tend to be large and therefore, probability of choosing the *right* character is less. There should be way to reduce it. This is where we use Gaussian convolution. It should be noted that during few initials generations, the difference between the best and worst inputs is high. For first few generation (e.g. till 2nd generation), we calculate a set $notImp = (i_{1b} \cap i_{2b}) \cap (i_{1w} \cap i_{2w})$. With Gaussian normal distribution, we update M as follows: if $N(0,\sigma) > large\_number$, $M := M \setminus notImp$. As noted, set M may not contain few *good* characters and as a result, search will be trapped in local optima. In order to continue, we need a jump which is brought by adding new characters in M as follows: if $N(0,\sigma) > large\_number$, $M := M \cup new\_chars$, where new_chars is a small set of characters which are not there in M.

Remaining steps are performed according to (μ, λ)-ES algorithm as discussed earlier. Algorithm 2 shows the pseudo-code of the proposed method.

With this heuristic, we approximate the set of characters that should be present (or absent) in the string inputs in order to activate the vulnerability. Once a malicious input is found, we run the vulnerable program with a debugger to find out stack related information, like stack register contents. The purpose of getting this information is to further validate the exploitability of the vulnerability. Using such information, we can infer as how easy (or difficult) it is to write a real exploit for the vulnerability.

IV. Experimentation and Discussion

This section details on implementation of the framework and a set of experiments to validate the approach. We further discuss some of the findings observed during the experimentation including limitations of the approach.

Algorithm 2 Pseudo-code of the proposed EA based approach

```
INITIALIZE population I with λ random candidates
TERMINATION CONDITION: program crashed or 1000 iterations are over
M ← {}
repeat
    EXECUTE program with I
    CALCULATE fitness
    CALCULATE diff1, diff2 and notImp (till 2 generations)
    UPDATE M as M ← M ∪ (diff1 △ diff2
    if N(0,σ) > large_number then
        M ← M \ notImp
    end if
    if N(0,σ) > large_number then
        M ← M ∪ new_chars
    end if
    SELECT μ fitest candidates for the next population
    for each of μ candidates do
        if probability_mutation then
            for i = 1 to λ/μ do
                MUTATE candidate using M
            end for
        else
            for i = 1 to λ/μ do
                SELECT two parents randomly
                RECOMBINE parents to generate child
            end for
        end if
    end for
    UPDATE I with newly generated candidates
until TERMINATION CONDITION is met
```

From implementation perspective, the present work is composed of two major components- static analysis to get a program slice and EA based dynamic analysis. Static analysis is performed using STAC. Details of its implementation can be found in [10]. After the source code is analyzed by STAC, we get a set of TDSs that covers each and every path from various inputs to that statement.

Dynamic component is composed of instrumented binary of the program, EA to generate inputs and an interface to communicate with a debugger to get after-crash information. In this study, we choose to work with GDB. Based on a particular TDS for a vulnerability, we instrument lines in source code corresponding to the chosen TDS. For example, for the code given in listing 1, the TDS that we select is $\langle 5, 6, 11, 5, 6, 21 \rangle$ for the vulnerable statement at line 21. Therefore, we instrument statements after each of these labels- 5, 6, 11, 21. In this way, at runtime, we get execution trace of the program in terms of executed statement's frequency. As TDS includes control flow of the program, we get to know which branch of a conditional statement is executed. This is important to generate/select inputs which take the *right* branch.

EA is implemented as Python module. Inputs to EA module are instrumented binary and chosen TDS. EA is run until we get `SIGSEGV` signal or a predefined iteration threshold is reached. We look for `SIGSEGV` signal because BoF will result in invalid memory reference or segmentation fault. Once we get malicious inputs corresponding to the

[3]It will also contain futile characters, but we cannot eliminate them completely.

[4]It may contain few bad character also, but chances are less as being the worst inputs, bad characters should be common to them and therefore will not be included in symmetric difference. On the other hand, good characters are very less likely to be common to them and therefore, will be present in symmetric difference.

SIGSEGV signal, with the help of another Python module, we run the binary with GDB to get information after the crash which includes various stack register contents.

A. Experimentation

For empirical results, we experiment with Verisec benchmark suite [9]. The suite consists of snippets of open source programs containing BoF vulnerabilities of varied difficulties. For us the level of difficulty is directly proportional to the manipulation done on the tainted input by the program i.e. if there are checks on the input before it reaches the vulnerable statement, it is harder to generate such input. From this perspective, we find that Verisec suite has programs as simple as `bind-> CVE-2001-001-> nslookupComplain-small_bad.c`, wherein generating a larger *random* string overflows the buffer, to as hard as `edbrowse-> CVE-2006-6909-> ftpls->strchr_bad.c` wherein input should contain (or does not contain) specific characters (at specific positions) in order to reach the vulnerable statement.

Table I shows the findings of our analysis on three programs which do string (inputs) manipulations before letting it reach the vulnerable statement. We compare our results with two other approaches- random fuzzing approach and GA approach with the knowledge of characters that are being checked at various positions in TDS (i.e. constraints are known). For the second approach, we implement GA module and analyzed the code statically to get constraints. In this case, the initial population is drawn from a limited ASCII character set and the set M has much fewer *important* characters.

Table I
LIST OF THE PROGRAMS USED IN EXPERIMENTATION AND CORRESPONDING RESULTS

S.No.	Application Name		# LoC	Constraints	EA	GA	Random
1	sendmail	mime _fromqp	65	'=n'	154	20	370[5]
2	sendmail	buildfname	52	'&' and not(,;%)	16	6	96
3	edbrowse	ftpls	49	'-- ' (in the beginning)	*	35	*

In the table, columns 2–3 denote the vulnerable program in the Verisec suite. Column 5 shows constraints that an input must satisfy in order to reach vulnerable statement. Column 6 shows the number of iterations (generations) taken by our approach to generate inputs that crash the application. These numbers are average taken over 20 different runs. In the case of *edbrowse* program, out of 20 times it could generate malicious inputs only 4 times at the iteration number 800, 85, 224 and 985. Column 7 shows the same for pure GA based method with constraint knowledge. The last column shows the number of iteration taken by random fuzzing approach. In the case of *edbrowse* program, it could not generate malicious inputs. This comparison shows the effectiveness of EA enabled inputs generation *viz-a-viz* randomly generated inputs, specially in the case, when we have knowledge about the precise path to reach the vulnerable statement by means of TDS. Following we show a typical output of our tool for the program at S. No. 3 in the table. In that program, there exists a BoF when `strcpy()` is called with the fixed length buffer `user[USERSZ]` and user controlled input. Before calling `strcpy()`, the input string must have '`--` ' as first three characters. For convenience, we denote `<space>` by `S` in the output shown in figure 3. Line 2-4 shows the malicious

```
1 Generation# 85
2 Malicious inputs: --S--%-aSa*-n**naSa1b=4*7e56ekg*2fie*
3 1S**n11%a1*an1as%%n-*-1sa*11%%%1s1Sn%%*-%%a-1%1%-S-a*-*
4 *%***na%1n**111%-ansn-%aaa**s-**aS*n1- .....
5 Calling GDB...
6 EIP is overwritten by:  1s1S  at index:  68
7 EBP is overwritten by:  1%%%  at index:  64
8 ESP is pointing to:  n%%*-%%a-1%1%-S-a*-**%***na%1n**111
9 %-ansn-%aaa**s-**aS*n1- ....  at index:  72
```

Figure 3. Output of the tool.

input that caused the program to crash, followed by its length 107. Line # 6 shows the status of `eip` which is overwritten by `1s1S` at an offset 68 in the string and line 9 shows the contents pointed by `esp` at offset 72. With this information, one can construct a real exploit with the following skeleton:

```
<--S...Ax67...><4 bytes address to 'jump esp' instruction>
<...shellcode..(starting at 72th byte>
```

Based on this information, we can infer that it is easy to exploit `edbrowse` program. On the other hand, in the case of program shown in listing 1, we find it difficult to exploit as values of `eip, esp` were not always affected meaningfully by user controlled input which makes it hard to construct a meaningful exploit.

B. Discussion

Table I shows a clear distinction among the three approaches *viz.* proposed EA based approach, GA assisted approach with partial knowledge of constraints and the random fuzzing. As expected, EA based approach outperforms random fuzzing by a huge difference, given the fact that both have no knowledge of constraints. GA based approach performs better than the proposed one mainly because in the former case, initial population is seeded with *special characters* (constraints on inputs) known a priori and the set M contains limited characters including special characters. As a consequence, probability of selecting the right character during mutation is higher, making it converging to the desired input faster. On the other hand, EA based approach starts with all printable ASCII characters. The set M is

[5]It could not generate malicious inputs 5 times out of total 20 times. So the average is taken over 15 iterations

updated at runtime by learning from best and worst inputs. As a result, normally set M has large set of characters which makes it probabilistically difficult to select right characters during mutation. This explains why does it performs better in the case of program 2 than in programs 1 and 3. Program 3 is even more complex as it has constraints on the position as well as on the sequence of characters. The desired input must be of the form "$--$ *any_character_string*". Therefore, in the case of GA based approach with smaller M, probability of generating such a string is still higher when compared to our approach. Nevertheless, one interesting thing to note about later case is that though it could not generate the correct string frequently, we observe that set M contains the *special characters* for the corresponding program. Therefore, one limitation of the proposed approach that we can note immediately is that constraints on position and sequence of characters (more than 2) together are hard to solve. We need to find some heuristic to reduce the size of M and to incorporate the notion of position in mutation. Another rather obvious limitation is that it cannot generate string which contains longer substring as a constraint, specially when it is a valid world of a grammar e.g. English words.

V. Related Work

A. String constraint solving

Popularity of web-based applications and ever increasing attacks on such services has given enough motivation to develop techniques for vulnerability detection for such applications where tainted malicious inputs are mainly *strings*. Therefore, apart from executing path from tainted source to sink, another challenge is to generate strings that satisfy all the constraints along the path. Recently, in [3] a solver, called HAMPI, is proposed for strings constraints. HAMPI normalizes the inputs into the *core* forms to express memberships to many regular expressions. These regular expressions also serves as patterns for a given vulnerability (e.g. SQL injection). After converting them to bit-vectors, a solver is applied to solve the constraints and if satisfiable, a string is generated that satisfy all the constraints along the path. However, in order to use HAMPI, all the constraints should be known a priori which may be a difficult task for certain complex applications. Another approach by Yu *et al.* makes use of DFA based forward and backward analysis of the intermediate nodes on the path from source to sink [4]. The idea is to generate a regular expression that represents *vulnerability signature* i.e. a pattern for malicious inputs. During forward and backword analysis, a DFA is constructed to denote a regular expression which is based on a vulnerability pattern. This regular expression represent the input that can exploit the vulnerability at the sink. Again, in this study also, user should be knowing constraints in terms of regular expression.

B. Forward symbolic execution

Symbolic execution is a general technique which allows to reason about a whole set of program executions at one time. Program states are represented by logical formulas, that are updated when assignments are executed. When a conditional statement is encountered, one of the branch is chosen, and the corresponding predicate is added to a so-called *path-condition* (the conjunction of all the predicates to be satisfied in order to reach the current state). Using this technique, deciding if a vulnerability (e.g., a buffer overflow) can occur at a given location l can be reduced to a satisfiability problem: is there a valuation of inputs that satisfy the path condition for reaching l in a program state which "activates" the vulnerability. Depending on the logic used to express such a constraint (and hence on the data types and predicates encountered in the original program) it can be solved by an automated tool (like SMT solvers).

Several vulnerability (or error) detection tools have been developed using this technique; one can cite for instance PATHCRAWLER [17], SAGE [18], DART [19], CUTE [20], EXE [21], or PEX[22]. One of the main issues in forward symbolic analysis is the so-called *path selection problem* [23], which consists in deciding which branch to follow first in case of conditional statement (with the risk of getting stuck inside loops those termination cannot be detected). In most cases the main objective is to increase some *path coverage* criterion. Classical strategies are then depth-first search, breadth-first, or even random searches. A more recent solution, called "Dynamic Symbolic Execution[5]" (DSE) consists in executing the program first using concrete input values, collecting the set of predicates traversed during this execution, and looking for a new input valuation which satisfies all these predicates but one (hence leading to a new execution path).

Recently, an extension of the PEX tool has been proposed which includes a fitness-guided search strategy to "drive" the execution towards a particular test target (e.g, a not-yet-covered branch) [24]. The fitness function they use to score a given execution is based on a distance computation between the valuation of a target predicate obtained for this execution, and the valuation required to satisfy the predicate. The next input is then chosen by flipping the most promising branching node of the best path executed so far (according to the fitness function). Compared with our proposal, this approach still requires the use of a constraint solver. Moreover, it does not allow to target an explicit (tainted) path.

The work by Brumley et al. [25] proposes a technique-APEG to generate inputs that exploit a given vulnerability. From smart fuzzing standpoint, their work adds a way to learn (new) constraints on inputs and source of vulnerability. This is accomplished by comparing patched and un-patched

[5]or *concolic execution*

version of the same application. After learning constraints, it follows the standard way of using a constraint solver to solve the constraints and based on this solution, generate inputs. Our approach tries to learn constraints (i.e. checks) without the patched version.

C. Evolutionary computing

Genetic algorithms have been used to generate test cases for program testing [26][7]. We find only one reference on the application of ES in software testing [27]. In that study, $(\mu + \lambda) - ES$ is used to test C programs that takes real inputs. Empirical results show a clear advantage of ES over GA. In our study, however, we tweak a bit the traditional usage of ES and apply it on string inputs. Grosso *et al.* have used GA to generate inputs automatically that trigger BoF vulnerability in the application [28][29]. The fitness score of input is based on its ability to covering maximum code and reaching vulnerable statements and weights associated with each of these factors. They have used linear programming to adjust weights automatically which is an additional step in the whole process of input generation. In our approach, we have used readily available statements frequencies to approximate nested statements and their weights. Sidewinder is a tool for analyzing binaries to detect vulnerabilities using GA assisted fuzzer [30]. In this approach, control flow is modeled as Markov Process and fitness function is defined over Markov probabilities associated with state transition on control flow graph. Inputs are generated using *grammatical evolution*. However, there is no mention of tackling constraints while generating inputs. Our approach is close to this approach with added feature of simplifying the weight calculation and path to traverse to reach vulnerable statements, even in the presence of simpler constraints. On the similar lines, Liu *et al.* construct *control dependence predicate path* (CDPPath) from the binary of the application and apply GA to construct inputs to reach vulnerable statements [31]. Their fitness function depends on the number of predicates in CDPPath covered by inputs. However, this study treats each predicate equal which may result in stagnation during later stage of searching and in contrast to our approach, does not try to satisfy the string constraints as such.

VI. Conclusions and Future Work

We present preliminary results of our work on smart fuzzing using EA. We note that majority of the programs that are vulnerable to BoF attack accepts strings as inputs from users and manipulate them in dangerous way. However, in many cases, there are certain checks performed on strings to validate it as per structural requirements. This poses certain constraints on input strings. In order to activate the vulnerability, input strings must satisfy these constraints. Our intention is to generate strings that have potential to exploit a given BoF vulnerability without knowing these constraints a priori. These constraints are learnt as inputs are evolved by using EA. We make use of set-theoretical definitions to approximated such constraints. We conduct a set of experiments to test our approach and we find promising results. However, as another outcome of experimentation, we note down certain shortcomings and hurdles in our approach which constitute our future work.

As mentioned earlier, we plan to investigate the feasibility to reduce set M further by applying some heuristics, for example, learning operands of comparison operator at runtime by instrumenting the code. Another improvement could be to keep more information on string structure, for example, ordering and relative position of characters etc. Currently, our method requires source code of the program to be analyzed to get program slice (tainted data path). Therefore, finally we intend to perform such analysis on the binary of the program which extends its usage to more applications where source code is not available (e.g. COTS).

Acknowledgement: We are thankful to Vulcain project for financial support and annonymous reviewers for their useful comments.

References

[1] Nist, "Information security in the systems development life cycle." [Online]. Available: http://csrc.nist.gov/SDLCinfosec

[2] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[3] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for string constraints," in *Proc. of the 18th ISSTA'09*. New York, NY, USA: ACM, 2009, pp. 105–116.

[4] F. Yu, M. Alkhalaf, and T. Bultan, "Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses," in *Proc. of the IEEE/ACM Int. Conf. ASE'09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 605–609.

[5] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, 1st ed. Addison-Wesley Professional, July 2007.

[6] J. DeMott, "The evolving art of fuzzing," in *Proc. of DEFCON'06*. DEFCON CD, 2006. [Online]. Available: http://www.vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf

[7] T. Mantere and J. T. Alander, "Evolutionary software engineering, a review," *Applied Soft Computing*, vol. 5, no. 3, pp. 315–331, 2005, application Reviews.

[8] A. E. Eiben, "Evolutionary algorithms and constraint satisfaction: Definitions, survey, methodology, and research directions," in *Theoretical Aspects of Evolutionary Computing*, ser. Natural Computing Series, L. Kallel, B. Naudts, and A. Rogers, Eds. Springer-Verlag, 2001, pp. 13–58.

[9] Verisec, "Verisec benchmark dataset," October 2008. [Online]. Available: http://se.cs.toronto.edu/index.php/Verisec_Suite

[10] D. Ceara, L. Mounier, and M.-L. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," in *Proc. of the IEEE Int. workshop MDV'10*. IEEE Computer Society, 2010, pp. 371–380.

[11] "STAC - Static Taint Analysis." [Online]. Available: http://code.google.com/p/tanalysis/

[12] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer Verlag, 2003.

[13] S. Luke, *Essentials of Metaheuristics*, 2009, available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[14] T. Ball, "The concept of dynamic analysis," in *Proc. of 7th European Software Engineering Conference*, ser. LNCS, vol. 1687. Springer, 1999, pp. 216–234.

[15] ——, "What's in a region? or computing control dependence regions in near-linear time for reducible control flow," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, pp. 1–16, 1993.

[16] T. W. Reps, "The use of program profiling for software testing," in *Informatik 97, GI Jahrestagung*, 1997, pp. 4–16.

[17] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: automatic generation of path tests by combining static and dynamic analysis," in *Proc. European Dependable Computing Conference*, ser. LNCS, vol. 3463. Springer, 2005, pp. 281–292.

[18] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Proc. of the 2nd int. workshop on Random testing*. New York, NY, USA: ACM, 2007, pp. 1–1.

[19] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.

[20] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proc. of the 10th European SE conf. with 13th ACM SIGSOFT int. sym. FSE*. New York, NY, USA: ACM, 2005, pp. 263–272.

[21] C. Cadar, P. Twohey, V. Ganesh, and D. Engler, "Exe: A system for automatically generating inputs of death using symbolic execution," in *Proc. of the 13th ACM Conf. CCS 06*, 2006.

[22] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *Proc. of the 2nd int. conf. on Tests and proofs*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153.

[23] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. of the IEEE Security & Privacy Symposium*. IEEE, 2010.

[24] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. of the 2009 IEEE/IFIP Int. Conf. DSN 09*. IEEE, 2009, pp. 359–368.

[25] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. of the IEEE Symposium on Security & Privacy*. IEEE Computer Society, 2008, pp. 143–157.

[26] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[27] E. Alba and J. F. Chicano, "Software testing with evolutionary strategies," in *Proc. of the 2nd Int. Workshop RISE'05*, ser. LNCS, vol. 3943. Berlin Heidelberg: Springer, 2005, pp. 50–65.

[28] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers & Operations Research*, vol. 35, no. 10, pp. 3125–3143, 2008.

[29] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo, "Improving network applications security: A new heuristic to generate stress testing data," in *Proc. of the conference on Genetic and evolutionary computation*. ACM, 2005, pp. 1037–1043.

[30] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," in *Proc. of the 23 Annual Computer Security Applications Conference*. ACSAC, 2007.

[31] G.-H. Liu, G. Wu, Z. Tao, J.-M. Shuai, and Z.-C. Tang, "Vulnerability analysis for x86 executables using genetic algorithm and fuzzing," in *Proc. of the 3rd Int. Conf. on Convergence and Hybrid Information Technology*. IEEE Computer Society, 2008, pp. 491–497.