Evolutionary Algorithms for Vulnerability Coverage

Shuvalaxmi Dass
Computer Science Department
Texas Tech University
shuva93.dass@ttu.edu

Akbar Siami Namin
Computer Science Department
Texas Tech University
akbar.namin@ttu.edu

***Abstract*—We present a novel idea on adequacy testing called "vulnerability coverage." The introduced coverage measure examines the underlying software for the presence of certain classes of vulnerabilities often found in the National Vulnerability Database (NVD) website. The thoroughness of the test input generation procedure is performed through the adaptation of evolutionary algorithms namely Genetic Algorithms (GA) and Particle Swarm Optimization (PSO). The methodology utilizes the Common Vulnerability Scoring System (CVSS), a free and open industry standard for assessing the severity of computer system security vulnerabilities, as a fitness measure for test inputs generation. The outcomes of these evolutionary algorithms are then evaluated in order to identify the vulnerabilities that match a class of vulnerability patterns for testing purposes.**

***Index Terms*—Software Vulnerability Testing, Vulnerability Coverage, Genetic Algorithms, Particle Swarm Optimization**

I. Introduction

The National Vulnerability Database (NVD) [3] lists over $1,644$ instances of vulnerabilities identified by their unique CVE (Common Vulnerabilities and Exposures) numbers. Some of these vulnerabilities exist partly due to improper settings of configuration parameters that govern the functionality of the given software. Testing software applications against these vulnerabilities can become tedious, and as a result, infeasible if the test has to target all the possible settings of a configuration in order to check the vulnerability of the software against some known attacks. This calls for a systematic configuration testing framework and mechanism in order to mitigate the efforts put into inspecting the given software by identifying a narrowed down set of configuration test inputs. However, from vulnerability perspective, it is not easy to check whether the tests generated for testing examines the software system against certain classes of vulnerabilities. Hence, it makes it impractical for the administrator to exercise the given software under test against any vulnerability reported in NVD.

To illustrate the overall mechanism of the proposed adequacy coverage, consider an example for vulnerabilities in MySQL. The "high" severity level of the MySQL CVE-2019-12463 vulnerability is 8.8 out of 10. There are several other vulnerabilities reported for MySQL with similar patterns as vulnerability vector and with similar severity scores. Then the major testing question is whether it is essential to examine the software under test for all the reported vulnerabilities with certain CVE numbers. It is possible to view the problem as an instance of general software testing problem and thus develop a specific adequacy criterion for covering vulnerabilities and examine the software under test (SUT) for the vulnerabilities.

This paper extends our initial idea [6] on vulnerability coverage and thus presents the novel concept of "*vulnerability coverage*," in an analogous way to conventional adequacy criterion in software testing. We apply evolutionary algorithms to generate test inputs with certain patterns and use Common Vulnerability Scoring System (CVSS) as a primary criterion to assess the vulnerability coverage (i.r., fitness) of the SUT. The paper makes the following key contributions:

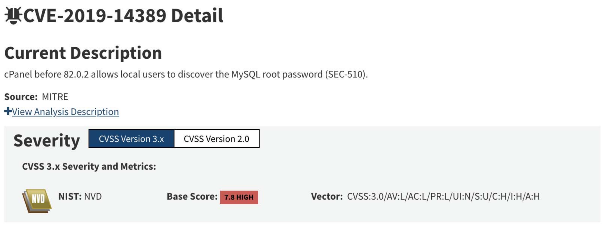
1) We present the idea of vulnerability coverage as a test adequacy criterion for inspecting the given software against certain types of vulnerabilities (Section III).
2) We perform evolutionary algorithms such as genetic algorithms (GA) and particle swarm optimisations (PSO) to generate vulnerability vector patterns (Section IV).
3) We compare the performance of both GA and PSO in generating such vulnerability vector patterns (Section V). According to our results, PSO managed to generate a more stable trend of secure vulnerability vector patterns than that of GA in a single generation.

We provide relevant background on the Common Vulnerability Scoring System (CVSS) is presented in Section II. Section III presents the idea of vulnerability coverage as an adequacy test criterion. Section IV presents the fitness function for the evolutionary algorithms. In Section V, we present experimental setup and results. Section VI reviews the related work. Section VII concludes the paper. Throughout the paper, we will be using the words "*vulnerability pattern*" and "*CVSS vector pattern*" interchangeably.

II. Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) is an open-standard industry framework, which helps cyber-security professionals to seek out information regarding ranking mechanism for the severity of vulnerabilities. CVSS captures the principle characteristics of the vulnerability by assigning a Base score rating ranging from 0 to 10 which is representative of the ease of exploitation and the damaging effect of the concerned vulnerability where 10.0 is the most easily exploitable vulnerability. The numerical scores have a qualitative assessment (low, medium, high, and critical) to provide organizations with better understanding and assessment of vulnerabilities. Some vulnerability are also given temporal and environmental scores that may modify the base score. As a proof of concept

978-1-7281-7303-0/20/$31.00 ©2020 IEEE
DOI 10.1109/COMPSAC48688.2020.00049



CVE-2019-14389 Detail

Current Description

cPanel before 82.0.2 allows local users to discover the MySQL root password (SEC-510).

Source: MITRE

Severity

CVSS 3.x Severity and Metrics:

NIST: NVD

Base Score: 7.8 HIGH

Vector: CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

Fig. 1: The description of CVE-2019-14389 for MySQL.

in generating CVSS patterns, the GA and PSO optimization algorithms are applied to the Base metrics. The Base metric consists of three sub-main metrics where each metric group comprises of a set of vector fields and the associated values it takes:

- **Exploitability Sub-Metric:** It addresses how the attack is captured. Table I lists down the vector fields.
- **Impact Sub-Metric:** It reflects the "*characteristics*" of the impacted components as shown in Table I.
- **Scope Sub-Metric:** It is a vector field acting as a separate metric which describes the change in the scope of the attack by determining whether other components are affected along with the original vulnerability. It accepts only two values: *Unchanged (U)* and *Changed (C)*.

Base score formula is calculated as follows [1]:

$$Impact\ Sub-Score(ISS) = 1 - [(1-C)*(1-I)*(1-A)] \tag{1}$$

$$Impact(IM) = \begin{cases} 6.42 * ISS & \text{if } \textit{Scope is Unchanged} \\ 7.52*(ISS-0.029) & \\ -3.25*(ISS-0.02)^{15} & \text{if } \textit{Scope is Changed} \end{cases} \tag{2}$$

$$Exploitability(EX) = 8.22 * AV * AC * PR * UI \tag{3}$$

$$Base\ Score = \begin{cases} 0 & \text{if } Impact <= 0 \\ Round(Min[(IM+EX),10]) & \text{if } \textit{Scope is Unchanged} \\ Round(Min[1.08*(IM+EX),10]) & \text{if } \textit{Scope is changed} \end{cases} \tag{4}$$

Every known vulnerability's severity can be represented as a vulnerability/CVSS vector pattern, which comprises all the aforementioned vector fields. For instance, Figure 1 shows the CVSS score and the vulnerability pattern vector for CVE-2019-14389 [3]. As shown in the figure, the score for this vulnerability is high and is quantified as 7.8 out of 10. The generated representation of the vulnerability/CVSS vector is `[AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H]`. Table II lists the vector pattern description for this vulnerability.

III. Vulnerability Coverage As Adequacy Testing

In an analogous way to the conventional definition given for "code coverage" in software testing, the vulnerability coverage is a measurement of how many known and reported vulnerabilities of the system under test (SUT) are inspected against. Similarly, the vulnerability coverage (VC) for a software system (S) can be measured as follows:

$$VC_S = \frac{(\#vulnerabilities\ inspected)_S}{(Total\ \#\ of\ vulnerabilities\ reported)_S} * 100 \tag{5}$$

Where $(\#vulnerabilities\ \ inspected)_S$ is the number of vulnerabilities inspected for the system S, and $(Total\ \#\ of\ vulnerabilities\ reported)_S$ is the total number of vulnerabilities reported for system S. Without loss of generality, this paper uses *vulnerability patterns* provided by CVSS score to measure the adequacy testing of vulnerabilities give for a software system.

It is important to note that vulnerabilities of the same pattern might be found in CVE database directly. However, given the evolutionary search algorithms presented in this paper, it is possible to identify vulnerabilities of different patterns but similar CVSS score. Hence, the use of CVSS score, as a fitness function, enables us to identify various forms and patterns of vulnerabilities within the specific level of CVSS score. Therefore, given the desired level of CVSS score, the problem of adequacy testing for vulnerability testing will be exercising the vulnerabilities with different patterns but equal CVSS scores. In following sections, we adopt two evolutionary algorithms that enable us search the input space (i.e., vulnerability pattern) that achieve a certain level of CVSS score (i.e., the fitness value).

IV. Fitness Function: CVSS Score

This section explains the genetic and optimization algorithms developed in which CVSS scores are used as fitness functions. For the ease of naming convention and understanding, we considered each CVSS pattern as a separate configuration in these algorithms.

A. Genetic Algorithm (GA)

Genetic Algorithms are based on the biological process of evolution. The idea is that over time, a pool of chromosomes will evolve to be even better (i.e., better fitness value) than the previous generation. A new generation (equal to the pool size) of chromosomes (i.e., configurations) is created with any iteration of the algorithm. This is achieved by the processes of selection, crossover, and mutation [9]. A fitness score metric is adopted as a measure to select the two fittest chromosomes from the pool that are called parent chromosomes. Then crossover takes place between the parents to produce a new child chromosome, which will have the best traits from both the parents followed by mutating of some of the characteristics of the child to introduce new traits. This process is repeated until an entirely new generation gets created.

B. GA implementation for secure configuration pool

We implemented the algorithm in Python. We first created a CVSS vector pool with the fitness score of 2.0 (i.e., the best and more secure fitness score). We refer to vector as a "string" in our implementation. We set the number of iteration/generation as 50. The entire algorithm (Algorithm 1) is divided into five parts: 1) configuration generation , 2) fitness score, 3) breeder's Selection, 4) crossover, and 5) mutation. These parts are explained below in-depth.

1) Initial Configuration Generation: As shown on lines 1 – 9 of Algorithm 1, we created a pool of 100 possible CVSS vector strings by randomly choosing corresponding permissible values from the `val` list to produce the initial pool of vector strings.

Sub-Metric	Fields	Description	Values	Score
Exploitability	Attack Vector (AV)	Reflects the proximity of the attacker to attack the vulnerable component.	Network(N)	0.85
			Adjacent(A)	0.62
			Local (L)	0.55
			Physical (P)	0.2
	Attack Complexity (AC)	Reflects the resources and conditions required to conduct the exploit on the vulnerable component.	Low(L)	0.77
			High(H)	0.44
	Privileges Required (PR)	Represents the level of privileges required by an attacker to successfully launch an exploit.	Low(L)	0.62 (or 0.68 if Scope is Changed)
			High(H)	0.27 (or 0.5 if Scope is Changed)
			None(N)	0.85
	User Interaction (UI)	Reflects whether the participation of the user is required for launching a successful attack.	None(N)	0.85
			Required (R)	0.62
Impact	Availability Impact (A)	Measures the severity of the attack on the availability of the impacted component.	Low(L)	0.00
	Integrity Impact (I)	Measures the severity of the attack on the integrity of the impacted component.	High(H)	0.22
	Confidentiality Impact (C)	Measures the severity of the attack on the confidentiality of the impacted component.	None(N)	0.56

TABLE I: Sub-Metrics.

Parameter	Description
AV: L	Denotes the vulnerability is exploited by the attacker through accessing the target system locally (L).
AC: L	Represents that the vulnerability has a Low (L) complexity of being attacked.
PR: L	Shows that a Low (L) number of Privileges are required for successfully exploiting this vulnerability.
UI: N	Denotes that no (N) User Interaction and involvement is required to launch a successful attack.
S: U	Shows the Scope (S) of the attack is Unchanged (U).
C: H	Total loss (High) of confidentiality.
I: H	Total loss (high) of integrity, or a complete loss of protection.
A: H	Total loss (High) of availability, full access denial to resources in the impacted component.

TABLE II: CVSS vector pattern description.

2) Fitness Score: As shown on lines 11 – 20 of Algorithm 1, the fitness score of the initial population of the vector strings is evaluated. We imported the `cvss` [2] python library and thus utilized the base metric score method `CVSS3`. The CVSS scores were considered as the fitness scores. The scores were valid if they were in the range of $[2.0, 5.5]$. Anything outside of that range was assigned the score as 100. We chose score 5.5 to be the upper limit since it is roughly the average score a configuration can take to be deemed reasonably secure.

3) Breeder's Selection: As shown on lines 21 – 32 of Algorithm 1, we then used Breeder's selection method. This method selects a combination of the best solutions generated by the algorithm (i.e., vectors with the low score). Furthermore, in order to avoid the problem of falling into local minima, the algorithm also picks some lucky few vectors with random vector scores.

4) Crossover: For crossover, As shown on lines 33 – 44 of Algorithm 1, we randomly swapped the values of metrics among the two parent vectors. We used a random value generator to select which parent vector to use for crossover. If $value < 0.5$, parent 1 is chosen, otherwise parent 2 would be the choice.

5) Mutation: As shown on lines 45 – 53 of Algorithm 1, the algorithm performs mutation on the CVSS vector strings by random selection of vector field whose value is also randomly selected from its permissible set of values.

We ran the GA script 100 times. Each run of the algorithm produced different pool of CVSS vector strings with different number and combinations of vector of fitness score 2.0.

C. Particle Swarm Optimization (PSO) Algorithm

PSO is a widely used swarm-based optimization technique. It draws its inspiration from bee swarm, and bird flocking social behavior of particles. PSO and GA, both being different forms of evolutionary computation techniques, share some similarities. Both techniques start off with a random set of initial population/solutions and keep updating generations until it reaches an optimum solution space with respect to the fitness function. In case of PSO, it is a swarm consisting of various particles, where each particle represents a solution. Unlike GA, PSO does not make use of crossover and mutation operators to update the particles. Instead, these techniques are directed towards the global optimum by their personal best position along with the swarm's best position in the search space. PSO is also easier to implement than GA and has comparatively fewer parameters to adjust [7].

D. PSO implementation for secure pool configuration

We compared the performance of GA in generating a set of best configurations with that of PSO. We implemented the PSO algorithm in Python 3.6. To make the comparison meaningful and fair, the number of iterations and population size (swarm size) were kept similar to GA, which are 50 and 100, respectively.

The PSO algorithm is described in Algorithm 2. The algorithm takes two list as parameters: 1) `pbest_fitness` and 2) `particle_vel`. These lists maintain the initial `pbest` fitness and velocity values associated to every particle in a swarm. We defined swarm as a collection (list) of 100 initial particles whose implementation (line 2) is similar to the procedure `configuration` in GA. The algorithm returns a pool of particles with varied scores in each iteration and also stores the count of particles with score = 2.0 in every iteration in order to check how many most secure particles (configurations) are generated by PSO. We also focus only

Algorithm 1 Genetic Alg. for generation of configurations.

```
 1:              ▷ Generating initial pool of configuration vectors.
 2: procedure CONFIGURATION
 3:     $val = ['H','L','N','A','P','U','C','N','R']$
 4:     $vector\_field = ['AV','AC','PR','UI','S','C','I','A$
 5:     for each $vf$ in $vector\_field$ do
 6:         $vf = random.choice[val]$        ▷ 'val' takes
    permissible set of values based on vf chosen.
 7:     end for
 8:     return $vector$
 9: end procedure
10:              ▷ Assigning fitness score based on Best score.
11: procedure FITNESS($BestScore, vector$)
12:     $score = CVSS3(vector).score()$
13:     if ($score <= BestScore$ & $score <= 5.5$) then
14:         $fit = score$
15:     else
16:         $fit = 100$
17:     end if
18:     Return $fit$
19: end procedure
20:              ▷ Breeder's Selection: Select best vector samples.
21: procedure SELECTION($population, best\_sample, lucky\_few$
22:     $nextGen = [\ ]$
23:     $sortedPop = Sorted(population)$        ▷ descending
    order of fitness values = low cvss score to high
24:     for $i$ in range(best_sample) do
25:         $nextGen.append(sortedPop[i])$      ▷ first 'i' # of
    best configurations selected
26:     end for
27:     for $i$ in range(lucky_few) do
28:         $nextGen.append(random.choice(sortedPop))$
29:     end for
30:     Return $nextGen$
31: end procedure
32:              ▷ Creating new vector from 2 parent vectors.
33: procedure CREATECHILD($vector1, vector2$)
34:     $child\_vector = ""$
35:     for $i$ in range(len(vector1)) do
36:         if $random.random < 0.5$ then
37:             $child\_vector = child\_vector + vector1[i]$
38:         else
39:             $child\_vector = child\_vector + vector2[i]$
40:         end if
41:     end for
42:     Return $child\_vector$
43: end procedure
44:     ▷ Mutating: randomly changing a value of the vector.
45: procedure MUTATION($vector$)
46:     $vf = random.choice(vector\_field)$
47:     $modify = random.choice(val)$
48:     $index = get\_position(vf)$
49:     $vector = vector$[:index]$+modify+vector$[index+1:]
    ▷ inserting 'modify' in the vector string
50:     Return $vector$
51: end procedure
```

on the scores, which belong in the range [2.0, 5.0] in every iteration.

In a nutshell, the algorithm searches for the best fitness and velocity values for each particle until a threshold is reached (lines 7 – 34). In every iteration (lines 9 – 13), the algorithm picks the `pbest_fitness` (i.e., particle best) values as their CVSS scores `cvss_fit` with the assumption that the fitness would be better (i.e., lesser is better) than its current `pbest` fitness value. After the `For loop` ends, it then picks the global best (`gbest`) value of the swarm by the the best `pbest` value (lines 14 – 16), in this case, the least value.

The next step in the algorithm is to calculate the velocity (lines 17 - 31) where $particle\_vel(particle)$ fetches the velocity of the given particle. The lines 18 - 28 describe how the velocity is evaluated for every particle. The velocity metric measures the distance between the fitness score (pbest) and the best score. The particle is updated whenever its current velocity value is greater than its previous one.

The particles are updated using `update_particle` in a similar manner to the configuration mutation in GA. More specifically, any one out of the eight vector fields (i.e., AV, AC, etc.) is constructed whose value is chosen randomly from its corresponding set of permissible values. For example, if 'AV' is selected, then any value in the list of{H, L, N, A} can be randomly selected.

The target global best value was set to 10.0, particle_velocity in the range $[0, 8]$ where 0 and 8 are the minimum and maximum number of differences between two particles, respectively. Since each particle (CVSS vector) constitutes of only 8 vector fields (AV, AC, etc). The fitness_range is set between the range $[2, 10]$ where 2.0 is deemed as the best fitness score and 10.0 is the maximum CVSS score any particle can get which means highly unfit.

V. Experimentation and Results

We ran our Python scripts, developed for implementing the GA and PSO algorithms, 100 times on the CVSS population in order to evaluate the performance of the evolutionary algorithms in generating the most secure patterns. The performance was measured on the basis of three evaluation metrics:

1) Number of instances of vulnerability patterns with the target score (e.g., score = $(2.0, 3.0)$) in each run.
2) Mean hamming distance (diversity) of the CVSS vectors.
3) Standard deviation of the scores calculated for the set of population produced.

1) Diversity of Vulnerability Patterns: It is important to produce a diverse set of instances of vulnerability vector patterns to ensure the thoroughness of test inputs (i.e., vulnerability pattern) generation and thus avoid generating redundant test inputs where test input refers to an instance of vulnerability vector pattern produced by the algorithms. We collected the data for the three evaluation metrics for various range of target scores including $S \in$ 2.0, $S \in$ (2.0, 3.0], $S \in$ (2.0, 4.0] and $S \in$ (2.0, 5.0]. As a representative example, Figure 2 shows the plots for all the aforementioned three metrics for CVSS vector strings falling into $S \in$ (2.0, 3.0] for both GA and PSO.

Algorithm 2 PSO for generation of configuration.

```
procedure PSO(pbest_fitness, particle_vel)
    swarm = [particle() for $i$ in range(swarm_size)]    ▷ Initialize 100 particles
    iteration = 0
    Threshold = 50
    total_count = [ ]    ▷ To store count of particles with score = 2.0 in every iteration
    best_score = 2.0
    while iteration $<$ Threshold do
        count = 0
        for each particle do    ▷ Calc Fitness
            if $cvss\_fit(particle) < pbest\_fitness(particle)$ then
                pbest_fitness(particle) = cvss_fit(particle)
            end if
        end for
        if $pbest\_fitness(particle) < gbest(swarm)$ then
            gbest(swarm) = pbest(particle)
        end if
        for each particle do    ▷ Calc Velocity
            if $pbest\_fitness(particle) < best\_score$ then
                continue
            else
                velocity=pbest(particle)-best_score    ▷ Current Velocity
                if $velocity == 0.0$ then
                    count+=1    ▷ count of particles of score = 2.0
                end if
                if $velocity < particle\_vel(particle)$ then
                    $particle\_vel(particle) = velocity$
                else
                    particle = update_particle(particle)
                end if
            end if
        end for
        $iteration+ = 1$
        $total\_count.append(count)$
    end while
    return  total_count
end procedure
```

As expected, the number of instances of the generated vulnerability patterns for each run is smaller for the target score of 2.0 (i.e., most secure) and it is higher when the target fitness score is in range $(2.0, 5.0]$ (i.e., least secure). It implies that when a lower level of vulnerability is targeted (i.e., more secure with CVSS score = 2.0), there are "not" too many alternatives for patterns. On the other hand, if the vulnerability levels and security is relaxed (i.e., CVSS score $<= 5.0$ then over 60 alternatives could be produced for pattern matching. A combination of such varying level of CVSS scores might be beneficial to increase the search space when implementing a moving target defense platform.

The bar plots depicted in Figure 2.(a) and 2.(d) demonstrate the number of occurrences of CVSS patterns (i.e., y-axis) against the number of runs (i.e., x-axis) when the target target CVSS scores is $(2.0, 3.0)$ for GA and PSO, respectively. A glance at the charts indicates that GA is able to generate more instances of the CVSS patterns targeting the desired level of security (i.e., $(2.0, 3.0)$).

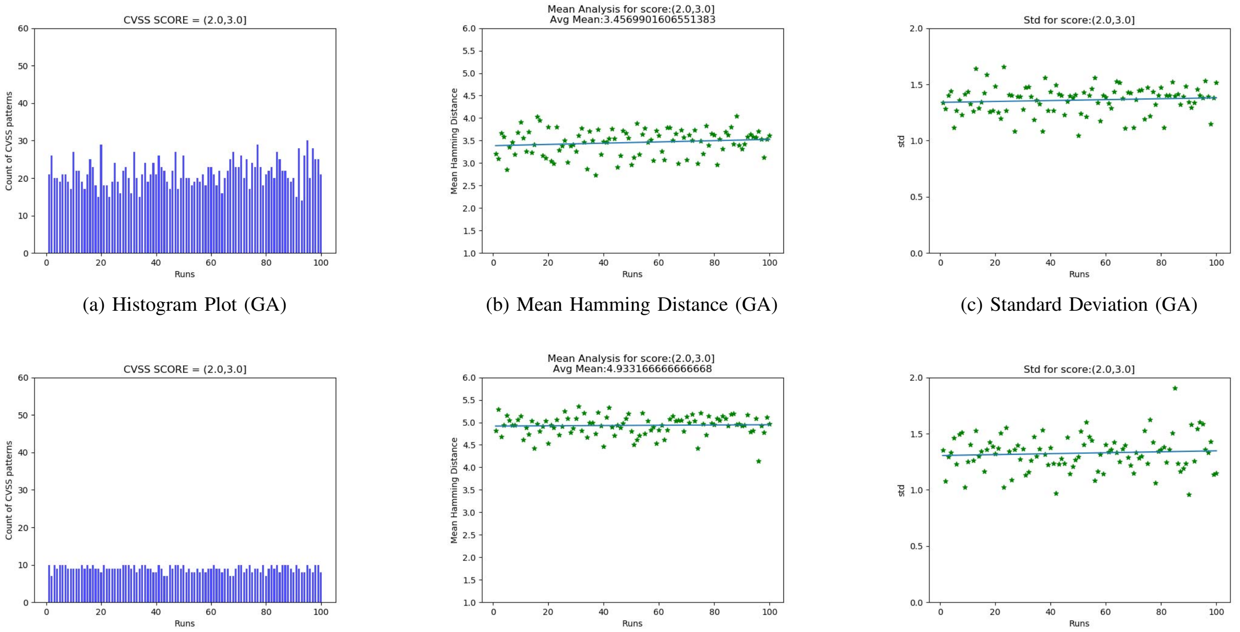
The scatter plots given in Figure 2.(b)-(e) and Figure 2.(c)-(f) denote distribution of the mean hamming distance and standard deviation against the runs, respectively. To ease comprehending the trend of the mean values, a regression line is fitted into the scatter plots to capture the overall trend. The Hamming distance addresses the "*diversity*" of the vulnerability vector patterns generated by the algorithms based on the count of corresponding unequal values of each vector fields among strings. The smoothing lines for mean values of Hamming distance demonstrate similar trends for each target value for the fitness score.

The mean values of the hamming distance (y-axis) in all the cases remain unchanged over the runs and are mostly scattered between 3.0 and 3.7 for GA (i.e., a diversity of the vulnerability vector pattern generated) and between 4.5 and 5.5 for PSO. As demonstrated in scatter plots shown in Figure 2.(b) and 2.(c), the instances generated by the GA algorithm is less diverse compare to the instances generated by PSO. The mean of the hamming distances between the instances generated by GA and PSO are 3.45 and 4.93, respectively. This indicates that even though the PSO algorithm generates far fewer instances of CVSS patterns for the given fitness, it produces more diverse instances of patterns.

To illustrate the variations of such vulnerability patterns generated, plots 2.(c) and 2.(f) illustrate the trend of the values of the standard deviations for Hamming distance over the runs. There is a light reduction in standard deviations while running GA for all cases. The observed standard deviations for all cases is somewhere between 0.5 and 1.5. When combined together, the mean and standard deviations of the hamming distance can serve as an indication of the diversity of the vulnerability vector patterns produced by the algorithms and thus helps in measuring the thoroughness of test case generation and thus vulnerability selections in which the generation of redundant patterns (i.e., test inputs) is avoided.

2) The Contributions of Each Permissible Value in each Vector Field: It is also important to investigate whether certain settings of each vector field contributes to security configuration differently than its counterpart. Table III shows the frequency (i.e., in terms of percentage) of each value permissible for each vector field, as listed in the base metrics. As reported in Table III:

– **AV**: The most contributing value is P (i.e., Physical) for GA (ranging from 45.43 to 59.22%). The PSO algorithm highlights two values of P and L as the most contributing to the security level of the patterns. This observation indicates that if the severity of the vulnerability needs



(a) Histogram Plot (GA)

(b) Mean Hamming Distance (GA)

(c) Standard Deviation (GA)

(d) Histogram Plot (PSO)

(e) Mean Hamming Distance (PSO)

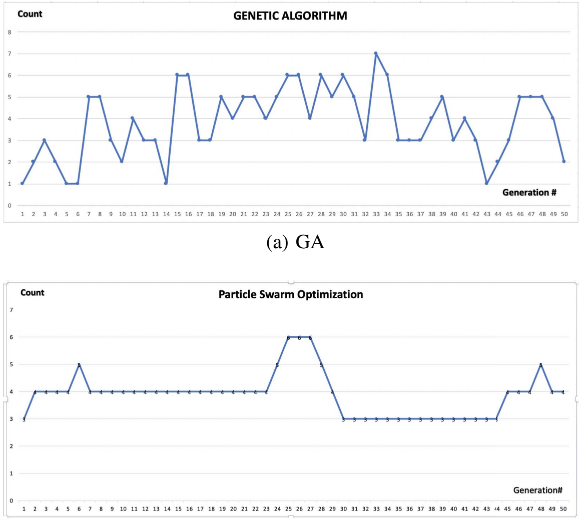
(f) Standard Deviation (PSO)

Fig. 2: Histograms, Mean Hamming Distances, Standard Deviations of CVSS vectors for $S \in (2.0, 3.0]$.

to be reduced, no other values or means of attacks (i.e., Network (N), Adjacent (A), and somewhat Local (L) is allowed for exploiting the vulnerability.

- **AC**: There is a mixed situation for attack complexity and there is no clear winner between Low (L) and High (H) complexity level to launch the exploitation.
- **S**: The dominant setting for this variable is C, except the case for GA when the target score is 2.0.
- **UI**: There is a mixed situation for the level of user involvement for exposing the vulnerability.
- **C**: There is also a mixed situation for confidentiality settings among GA and PSO algorithms.
- **I**: A similar mixed situation for this case. However, it is also observed that in most cases a None (N) risk to integrity is needed to reduce the impact of exploiting the vulnerability.
- **A**: Furthermore, there is a a mixed situation for availability where there is no clear dominant setting value.
- **PR**: The two dominant setting values for the level privileges are L and H.

We also executed the GA and PSO scripts with single run to sense their performances. Figure 3 illustrates the results of one run with 50 generations/iterations where y-axis is the number of vulnerability pattern produced whose score = 2.0; whereas, the x-axis is the generation index (i.e., 50 generations). As it is observed, GA could manage to generate two vulnerability vector patterns with score 2.0 on its 50-th iteration along with highly unstable trend; whereas, the PSO algorithm demonstrated a more stable trend with four pattern generated with score 2.0.



(a) GA

(b) PSO

Fig. 3: # CVSS patterns with score 2.0 over 50 generations.

VI. Related Work

Crouse and Fulp [4] used genetic algorithms to deploy a Moving Target Defense (MTD) platform and make computer systems more secure through temporal/spatial diversity in configuration parameters that govern how a system operates. Later on, they developed an MTD by simulating 256 virtual machines of similarly purposed computers where each computer was initially configured with an extremely vulnerable configuration making them prone to all sorts of attacks.

Post and Sinz [8] bridged the gap between configuration in-

Vector Field	**Values**	**[2.0]**		**(2.0, 3.0]**		**(2.0, 4.0]**		**(2.0, 5.0]**	
		GA	**PSO**	**GA**	**PSO**	**GA**	**PSO**	**GA**	**PSO**
AV	P	**59.22**	**29.29**	**49.31**	23.74	**47.85**	23.19	**45.43**	22.25
	L	14.02	22.22	21.40	**26.75**	21.90	**26.98**	22.32	**26.95**
	A	13.76	26.26	18.67	26.15	18.41	25.25	19.11	25.33
	N	12.98	22.22	10.60	23.34	11.83	24.56	13.13	25.45
AC	L	32.98	44.44	48.63	**50.90**	**50.83**	**50.17**	**51.85**	**51.56**
	H	**67.01**	**55.55**	**51.36**	49.09	49.16	49.82	48.14	48.4
S	U	**60.25**	48.48	47.87	48.39	47.15	48.96	46.50	48.56
	C	39.74	**51.51**	**52.12**	**51.60**	**52.84**	**51.03**	**53.49**	**51.43**
UI	N	33.24	43.43	48.51	**50.0**	49.30	**50.40**	49.79	**50.70**
	R	**66.75**	**56.56**	**51.48**	**50.0**	**50.69**	49.59	**50.20**	49.29
C	N	**62.85**	25.25	**61.84**	31.46	**54.77**	29.16	**53.36**	28.88
	L	37.14	**37.37**	38.15	**35.67**	45.19	**36.73**	45.82	34.91
	H	0.0	**37.37**	0.0	32.86	0.02	34.09	0.80	**36.20**
I	N	**67.27**	**39.39**	**64.09**	30.46	**55.98**	27.21	**54.54**	25.84
	L	32.72	34.34	35.90	**38.47**	43.82	**37.88**	44.37	35.72
	H	0.0	26.26	0.0	31.06	0.18	34.90	1.07	**38.42**
A	N	**69.87**	26.26	**66.70**	28.05	**58.96**	27.95	**57.95**	27.17
	L	30.12	**47.47**	33.29	**37.17**	40.98	34.95	41.55	34.01
	H	0.0	26.26	0.0	34.76	0.05	**37.08**	0.49	**38.81**
PR	N	40.25	26.26	29.75	28.05	30.81	27.95	30.58	27.17
	L	0.0	**47.47**	29.83	**37.17**	31.13	34.95	31.70	34.01
	H	**59.74**	26.26	**40.40**	34.76	**38.05**	**37.08**	**37.71**	**38.81**

TABLE III: % of contribution of each permissible value in all the score ranges across 100 runs of GA and PSO.

formation and verification process by introducing a new technique named *Configuration Lifting*. The technique converts all the variants over which a software is verified into a meta-program thereby making the application of configuration-aware verification techniques like static analysis, and model checking more efficient.Dai et al. [5] introduced the concept of *configuration fuzzing* in order to check the vulnerabilities that appear only at certain conditions by randomly modifying the configuration of the running application at specific execution points. During the deployment phase, this technique ceaselessly fuzzes the configuration and looks for a vulnerability that rises due to the violation of of security invariants.

VII. Conclusion

We introduced the novel idea of "vulnerability coverage," a methodology to examine software under test against certain classes of vulnerabilities as reported by National Vulnerability Database (NVD) adequately. The introduced idea makes use of an open industry standard tool called Common Vulnerability Scoring System (CVSS) as a metric to measure fitness in order to generate a pool of vulnerability vector patterns that attains a secure level of CVSS score. For adequacy testing of the underlying software, the software under test is then inspected against all those filtered representative sets of vulnerabilities with similar vulnerability vector pattern that were selected from the generated pool. The paper compared two evolutionary-based algorithms namely Genetic and Participle Swarm Optimization algorithms on the basis of their performance in generating a pool of vulnerability patterns and the results indicated a similar performance achieved by both algorithms.

The concept of adequacy criterion is a new approach and hence has a larger scope of improvement. An adequacy criterion based on vulnerability coverage is a novel technique in the best of our knowledge. This approach can be further improved by taking into consideration several other metrics including temporal and environmental ones present in CVSS and National Vulnerability Database (NVD). We also built our experiments based on the range of 2.0 and 5.5. Additional experimentation would be needed to further study the effect of such range. Moreover, the concept needs tool support and further empirical studies which can aid in thorough and systematic searching for vulnerabilities reported in the NVD database based on the matching property for the goal of security testing and then investigate the effectiveness of such adequacy criterion.

Acknowledgment

This work is supported in part by funding from National Science Foundation under grants no: 1516636 and 1821560.

References

[1] Common vulnerability scoring system v3.0: Specification document. https://www.first.org/cvss/v3.0/specification-document, Access 2019.

[2] CVSS 3.0. https://pypi.org/project/cvss/, Accessed 2019.

[3] National vulnerability database. https://nvd.nist.gov/, Accessed 2019.

[4] M. Crouse and E. W. Fulp. A moving target environment for computer configurations using genetic algorithms. In *Symposium on Configuration Analytic and Automation (SAFECONFIG)*, 2011.

[5] H. Dai, C. Murphy, and G. Kaiser. Configuration fuzzing for software vulnerability detection. In *2010 International Conference on Availability, Reliability and Security*, pages 525–530, 2010.

[6] Shuvalaxmi Dass and Akbar Siami Namin. Vulnerability coverage for adequacy security testing. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing*, pages 540–543, 2020.

[7] Mei-Ping Song and Guo-Chang Gu. Research on particle swarm optimization: a review. In *International Conference on Machine Learning and Cybernetics*, 2004.

[8] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 347–350, 2008.

[9] Kumara Sastry, David Goldberg, and Graham Kendall. *Genetic Algorithms*. Springer, Boston, MA, 2005.