Improving Black Box Testing
By Using Neuro-Fuzzy Classifiers and Multi-Agent Systems

Marcos Álvares B. Júnior, Fernando B. de Lima Neto
Polytechnical School of Pernambuco
University of Pernambuco
Recife, Brazil
E-mail: {mabj, fbln}@dsc.upe.br

Júlio César S. Fort
Informatics Center
Federal University of Pernambuco
Recife, Brazil
E-mail: jcsf@cin.ufpe.br

Abstract—**Automated software testing has become a fundamental requirement for several software engineering methodologies. Software development companies very often outsource the test of their products. In such cases, the hired companies sometimes have to test softwares without any access to the source code. This type of service is called black box testing, which includes presentation of some *ad-hoc* input to the software followed by an assessment of the outcome. The common place for black box testing is sequential approach and slow pace of work. This ineffectiveness is due to the combinatorial explosion of software parameters and payloads. This work presents a neuro-fuzzy and multi-agent system architecture for improving black box testing tools for client-side vulnerability discovery, specifically, memory corruption flaws. Experiments show the efficiency of the proposed hybrid intelligent approach over traditional black box testing techniques.**

Keywords-**information security; software testing; black box testing; hybrid intelligent systems; neuro-fuzzy classification;**

I. INTRODUCTION

Software engineering has experienced great advancements over the past twenty years. This has directly impacted in the development of more robust and secure computational systems. Several companies have critical processes automated by using complex software, which in general is very difficult to maintain and extend. In addition, software size is growing over the time. All that has contributed to increase the automated test difficulty. Overall, the difficulty in test process is proportional to the software complexity. Test based methodologies have been proposed aiming at reducing software failures. These methodologies detect anomalous behaviors probing the software for design failures and untreated exceptions.

Software test also plays a key role in computer security. A bad designed application can lead to a risky situations regarding sensitive information. Software testing aims at discovering unplanned behavior (e.g. software bugs). Some bugs may allow attackers to violate various security levels such as integrity, availability and confidentiality in relation to the information tackled.

The discovery of failure points within large software is not a trivial task. Several automated test techniques have been developed with the intention of helping to solve this issue. One of the most popular technique is black box testing. A black box test consists on testing a given software without having any access to its source code. That is, testing by executing the software binary with specific data inputs. In general, a monitoring routine is provided for the supervision of anomalous behaviors.

A major problem with black box testing techniques is the size of the search space, which is generated by the combination among arguments and parameters. This combination makes black box testing techniques computationally expensive. Hence, the common black box testing tools have to deal with exponential complexity problems. To improve the results of black box testing, hybrid techniques such as reverse engineering and static analysis were used before [1], [2]. In several countries, reverse engineer approach has some undesirable legal implications (e.g. copyright software and digital patents).

In 2008, Hong Liu Guang *et al.* proposes an architecture to perform black box testing using genetic algorithms [3]. Guang's paper shows how computational intelligence techniques can help black box testing to find vulnerabilities. For the present paper, we present an architecture to build black box testing tools that use intelligence principles, namely, neuro-fuzzy classifiers and multi-agent systems. This tools aim at discovering memory corruption vulnerabilities under UNIX platforms.

This study investigates three hypotheses of improvement mechanisms for black box testing: (i) use of agent's cognitive module, (ii) use of agent's communication and (iii) different communication network topologies. The next three sections provide theoretical foundations for the techniques used here. Section five presents the hybrid intelligent architecture proposed; section six describes the experiments for all three hypotheses, mentioned above. The last section comments upon the results and offers directions for further investigations.

II. Neuro-Fuzzy Systems

Neuro-fuzzy(NF) is a kind of hybrid intelligent system that combines human-like reasoning style of fuzzy systems with the learning structure of artificial neural networks [4], [5].

Neuro-fuzzy hybridization is done broadly in two ways: (*i*) neural networks that are capable of handling fuzzy informations (named as fuzzy-neural networks – FFN), and (*ii*) fuzzy systems argumented by neural networks to enhance some of their characteristics such as flexibility, speed and adaptability (named as neural-fuzzy systems – NFS).

For this paper we use a neuro-fuzzy classification model proposed by Ashish Ghosh [6]. The Ghosh's model works in three steps. In the first step, the system takes an input and *fuzzifies* its feature values using membership functions (MF), and provides the membership of individual features to different classes. A membership matrix thus formed contains number of rows and columns equal to the number of features and classes, respectively, present in a data set. In the second step, the membership matrix is converted into a vector by cascading all rows or columns. This vector becomes the input. The number of output nodes in the NN is the same as the number of classes present in the data set. The last step is a hard classification by performing a MAX operation to *defuzzify* the output of the NN.

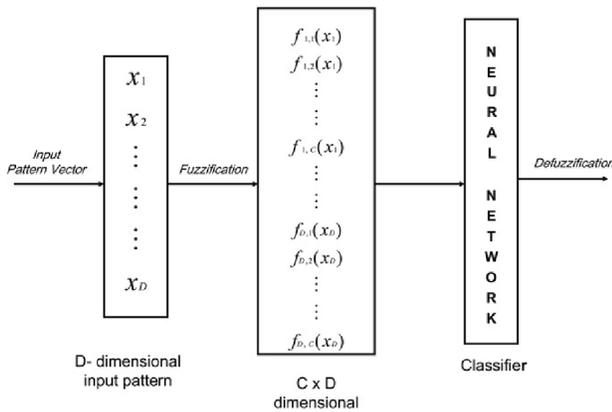An overview of Ghosh neuro-fuzy architecture is showed on Figure 1.

Figure 1: Ghosh's Neuro-Fuzzy Architecture

III. Black Box Test

According to the source code availability, Michael Sutton, in his book "*Fuzzing: Brute Force Vulnerability Discovery*" [7] classifies software tests in three main categories:

- **White Box**: test cases are generated with full access to the software source code;
- **Black Box**: test cases are generated with no prior knowledge about the software's internal structure;

- **Gray Box**: test cases are made above intermediate representations (e.g.: *Java Bytecode* or *Microsoft IL*)

For this paper, only black box tests are performed. Black box tests execute the target binary with input patterns looking for unexpected behaviors. This technique is comprised of three main steps: parameter selection, argument mutation and binary probing. Parameters selection will extract binary's parameters and make a combination of them. Parameters are data inputs for the binary. The second step fills the parameters with arguments (*payload*) which are changed for each test iteration. The last step executes the target binary using parameters and payloads and observes the occurrence of anomalous behaviors.

The biggest drawback with black box testing is the combinatorial explosion of parameters, *payloads* and binaries. The Equation 1 represents the quantity of tests cases with n parameters and only two types of *payloads* (numeric and literal).

$$f(n) = \sum_{p=2}^{n} \left[\frac{n!}{(n-p)!p!} 2n \right] \qquad (1)$$

For this Paper, we propose a generic architecture using intelligent algorithms aiming improve the efficacy to find memory corruption vulnerabilities automatically.

IV. Memory Corruption Vulnerabilities

Memory corruption is undoubtedly one of most dangerous classes of vulnerability in modern software. However, most software developers were unaware of it until the second half 1990s. The first discussion about this type of vulnerability appeared in a document known as "Computer Security Technology Planning Study" [8], which dates back from 1972. The first documented public usage of memory corruption with malicious intent came in 1988 with the Morris worm. Another groundbreaking event was the release of the article "*Smashing the stack for fun and profit*" [9], which popularized this exploitation vector.

This type of vulnerability happens when memory location essential to the execution flow gets unintentionally modified. This condition can cause program to behave unexpectedly, possibly causing denial of service or leading to arbitrary code execution, potentially allowing attackers to escalate privileges.

The root cause of this vulnerability class can roughly be traced back to bad code design (e.g.: as lack of proper input sanitization or misuse of function pointers) allowing the process to write to memory addresses it was not originally intended to write to.

The main goal of an adversary is to achieve an "*almost* 4 *bytes almost anywhere*" premise (for 32 bits architectures). This condition means to overwrite 4 bytes on any memory address.

The experiments performed in this paper focuses mainly on *stack* and *heap* overflows [10]. However, other classes of vulnerability, such format string bugs, integer overflows or command injection, might be uncovered while performing black box testing the built prototype.

V. CONTRIBUTION

Modern black box testing techniques analyses the binary systematically and sequentially. These techniques perform brute force searches to find parameters and payloads which break the software binary at hand. This kind of search is not computationally efficient as it tests all existing possibilities. To help on that, we propose an architecture for performing black box tests based on collaborative work and learn ability of (multi)agents. The proposed architecture also aims at accuracy improvement on the vulnerability discovery process because of cumulative experiences though neuro-fuzzy classifiers.

The proposed model has two main components: (i) Environment Profiler and (ii) Agents. The Environment Profiler searches for all candidate binaries[1] on the *file system* to be tested; these candidates are placed in a tree structure. This tree acts as search spaces for the agents. The tree leafs represent the binaries and the path from the root to the leaf represents the binary path.

The Environment Profiler stores control structures like the agent position and its status. Control structures are used as communication channels among agents. These channels are also used for alerts propagation of anomalous events. Control structures contain the state (tuple with *position* and *action*) of all agents. Another responsibility of the Environment Profiler is to collect machine specifications (e.g. CPU capacity and memory amount) and to determine the agent population size. The execution flow of the Environment Profiler is:

1) Enumerate the machine resources;
2) Determine the agent population size;
3) Find binaries inside the *file system*;
4) Build a tree structure;
5) Initialize control structures;
6) Initialize the population.

The second main component of the proposed model is the Agent. Agents is responsible for execute Black Box tests against the target binary. For this paper, a cognitive agent model is used; this means the agent can perceive the environment, perform cognitions and decide suitable actions. The Agent architecture is composed by four modules:

- **Controller module**: defines what action the agent will perform;
- **Cognitive module**: learn about the current experience;
- **Fuzzy module**: handle the binary parameters and arguments performing tests cases;

---

[1]Set-User ID binaries changes the user ID on execution

- **Communication module**: inform the population about some occurrence.

The controller module supervises the execution cycle of an agent. This icicle is showed in Figure 2. The agent starts choosing an random binary to test. Each binary is tested exclusively by one agent. The agent can communicate anomalous events and recruit another agent to work on a specific region of the search space. If the agent receives privileged information of its neighborhood, the controller can broke the agent execution cycle and redirect it for another binary. Before abandoning the current analysis, the fuzzy process status is saved inside the tree leaf created by the Environment Profiler module.
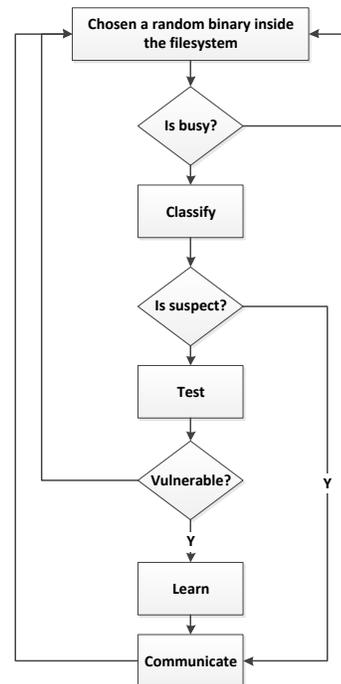

Figure 2: Agent execution fluxogram

There are several possibilities of agent neighborhood topologies, reefer to Figure 3. For this paper we use the *regular lattice* topology, where each agent is linked with four neighborhood agents. This means that, if an agent discovers a vulnerable binary, four other agents will receive this information. This architecture enables the exploration of suspected regions without stopping the exploration of the search space. The recruited agents perform local searches on leafs (binaries) from the same branch of the suspected binary.

If an agent finds a broken binary, it improves its internal knowledge database using its cognitive module. Agents can suspect that a binary is vulnerable based only on its previ-
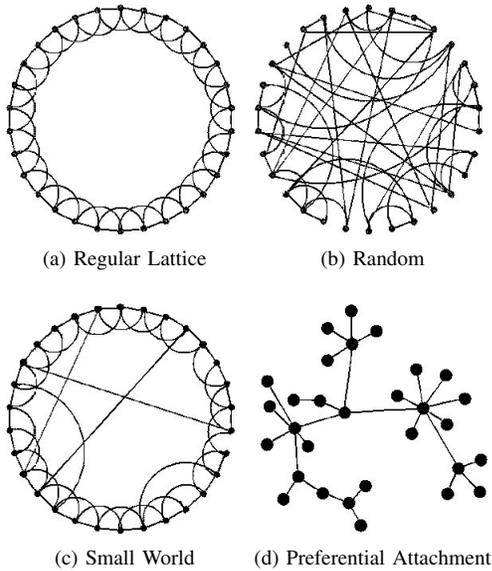
Figure 3: Four popular topologies used for multi-agent systems communication

ous experience. The cognitive module contains an artificial neural network which learns about the found vulnerable software. The neural network topology was found experimentally. This topology was composed of four artificial neurons on the hidden layer and two artificial neurons on the output layer. This means that the produced tool must be calibrated. The agent learns about vulnerable application patterns and evolves its cognitive module. The input data for the cognitive module is a vector of the binaries characteristics like:

- **Is active**: if the binary is active in memory;
- **Parameters cardinality**: the quantity of binary parameters;
- **Position**: the location on the tree.
- **Last use**: the lapse between current time and binary last execution time;
- **Last update**: the lapse between current time and binary last update time;
- **Size**: the binary size in Kilobytes.

Notice that the ANN tries to solve a classification problem of two classes of binaries, namely, "vulnerable" and "safe". The agent retrains the cognitive module for every new vulnerable binary found. All vulnerable binaries and a limited pool of safe binaries characteristics are saved inside an agent local database.

The fuzzy module performs black box tests selecting parameters and creating patterns of data input. The tested binary is executed using debug mode, this allow the operator to receive signals from the operating system. Signals can indicate, for example, memory corruption flaw. For UNIX based operating systems the memory corruption signal is

the *SIGSEGV* (i.e. *Segmentation Fault*). The fuzzy module extracts the binary parameters using regular expressions from *help command* or *manpage*[2]. From the possibilities of payloads and selection of parameters, the search space is built and one test case is created. *Fuzzification* process stops when an operating system *SIGSEGV* signal is produced or a maximum value of payload mutation iterations is reached. The fuzzy module operation can be represented by the algorithm 1.

Algorithm 1: Fuzzy Module Operation Algorithm

Extracts binary parameters from *help command* or *Manpage*;
foreach i *between* 1 *and the quantity of parameters* **do**
    Select subset of parameters with i size;
    Calculate combinations of this subset;
    foreach *combination* C **do**
        foreach *payload type* pt **do**
            Generate a payload vector pv with the same size of C;
            while *mutate* pv **and** *stop criteria* **do**
                Test the binary using the combination of parameters with the payload vector;
            end
            if *stop criteria* **then**
                return C and pv
            end
        end
    end
end

The communication module is responsible to exchange information between an agent and its neighborhood. The agent can send and understand basically two types of messages: "*I found a suspect binary*" and "*I finished my job*". If a suspect binary is found the agent alert its neighborhood agents only. This mechanism prevents premature stops on the search space exploration.

VI. EXPERIMENTS

Experiments were carried out using an Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz with 4.0 GB of RAM. We choose to use three identical virtual machines with Linux Ubuntu Server 9.10 (kernel 2.6.31-20). Using the proposed approach, we built a neuro-fuzzy system for discovering client-side memory corruption vulnerabilities.

As a proof of concept purpose, we use intentionally broken binaries obtainable at uCon Security Conference capture-the-flag[3] website [11]. Then, we select three pairs

---

[2]In some UNIX based systems each application has a digital user manual called *manpage*.
[3]Security Conference live challenges of crack-me and memory corruption vulnerabilities

of broken binaries and put each pair on a different virtual machine (randomly placed inside the file system). Ubuntu server default installation contains 34 *SUID* binaries. Each virtual machine contains 36 *SUID* binaries (34 default *SUID* binaries and 2 previously known broken). All experiments were average over 30 executions.

We realize three experiments: (i) use of agent's cognitive module, (ii) use of agent's communication module and (iii) different communication network topologies.

A. Use of Agent's Cognitive Module

This experiment investigates the influence of the cognitive module inside the agents. We tried two configurations of the tool: with and without the cognitive module enabled. Each configuration is trained using the first virtual machine. After finished the training stage, we test the efficacy of these two instances, to find other machine's vulnerabilities (environment #2 and #3). The Table I shows the experiment results.

Table I: Agent's Cognitive Module Influence

Environment	With cognition	without cognition
1	3h 47m	4h 10m
2	1h 51m	3h 57m
3	1h 05m	3h 53m

For the training stage, both configurations find vulnerabilities using the same amount of time. For the test stage, we can observe the expressive gain of the agents with cognition enabled. The proposed approach finds vulnerabilities quicker because it possess 10 experienced agents inside the population (i.e. the discoverer agent and its neighborhood). The neighborhood information is rebuilt for each fuzzy analysis avoiding group overspecialization and elitist behavior.

B. Use of Agent's Communication

This experiment investigates the influence of the communication module inside the agents. We tried two configurations of the tool: with and without the communication module enabled. Each configuration is trained using the first virtual machine. After finishing the training stage, we tested the efficacy of these two instances, to find others machine's vulnerabilities (environment #2 and #3). The table II shows the experiment results.

Table II: Agent's Communication Module Influence

Environment	With communication	without communication
1	4h 21m	4h 01m
2	1h 25m	3h 44m
3	1h 13m	4h 05m

For the training stage, both configurations find vulnerabilities using the same amount of time. For the test stage,

we observed the expressive gain of the agents with communication enabled. This improvement is due to the agent capability of informing its neighborhood about a suspicious region.

We also can visualize the communication module's effect by observing the average distance among the agents over the time Figure 4. The distance between two agents is measured by the quantity of tree nodes between their positions. The agents movement, inside the tree, is through the leafs (each leaf represents a binary). For example, two agents, inside the same directory and testing different binaries, have the distance equal to 1. This index acts as a clustering indicator. The proximity of population agents leads to formation of small search groups. This behavior can mean the formation of local search processes.
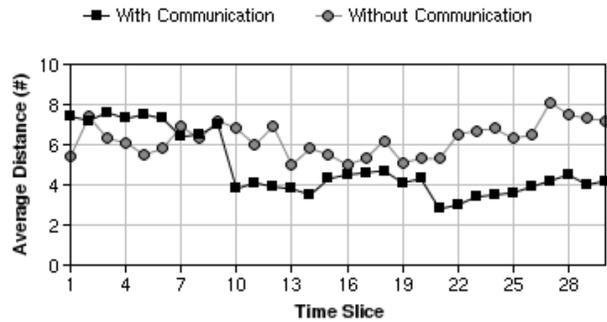

Figure 4: Evolution of the population's average distance over the time

Communication effect can be noted through the chart's inflections. An agent communicates its neighborhood about the discovery of vulnerable binary. Neighbors agents get closer, to explore the nearby regions. Due to this approximation, the average distance among the agents tends to reduce. The used communication topology (*regular lattice*) limits the insider information propagation and prevents overspecialization. This communication mechanism allows continuous exploration of the search space.

C. Different Communication Network Topologies

This experiment compares different configurations of communication network topologies. We compare the system's accuracy using four topologies: full connected, regular lattice, random and small world. Different instances of the tool were configured for test each topology. These instances were trained and executed using two different virtual machines. The table III shows the experiment results.

The *regular lattice* communication topology leads to better results. This topology finds the vulnerable binaries at least 15% faster than the other tested approaches. The *Small World* topology achieves satisfactory results, but the random

Table III: Communication Topology Network Influence

Topology	Time to find the vulnerabilities
Regular Lattice	1h 51m
Full connect	5h 15m
Random	3h 32m
Small World	2h 10m

criterion can lead to premature convergence. *Full Connected* topology is the worst tested method. The entire population is informed about each vulnerable binary discovery. The agents follow the discoverer; stopping their current work (the exploration process is interrupted). The random topology approach inhibits the formation of local groups process harming the search process. For better visualization this results, we can observe Figure 5. This figure represents the population clustering index using each communication topology over the time. The "time slice" is the total time to the algorithm reach the stop criteria (showed on the table III) divided by 30. For each topology the "time slice" means different amount of time.
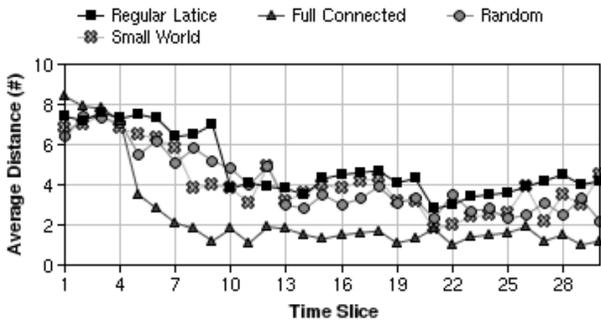

Figure 5: The evolution of the population average distance over the time using different communication topologies

Observing Figure 5, we can note that periodic movement is an important aspect for the algorithm efficacy. This periodic movement indicates the efficiency in exploration and exploitation of the search space. The seasonal behavior is better obtained using the *Regular Lattice* communication topology.

VII. CONCLUSION

In this paper, we put forward a new approach to perform black box testing by using neuro-fuzzy classifiers and multi-agent systems. The general idea is that the test process is distributed among autonomous agents. These agents ought to work cooperatively, distributing the effort of searching vulnerabilities. Several network topologies were tested aiming at the creation of adequate mechanisms for agent communication.

The experiments carried out show that the proposed model can generate adaptive tools for discovery of security vulnerabilities. In our approach the gathered experience in one fuzzy section is stored for later use. This means that the more the tool is used, the more it improves its accuracy. The proposed model may be generalized for several other types of software for vulnerability discovery.

A *proof of concept* was built to find client-side memory corruption vulnerabilities. We tested the prototype with security challenges such as broken binaries. These challenges were placed into three Linux virtual machines. The prototype was able to find all vulnerable binaries inside the virtual machines. For each tested environment the vulnerable binaries were found quicker than the previous one. This behavior shows the learn ability of the agents.

All these observable results motivate us for pursue on future development of intelligent mechanisms which improve Black Box testing. Future work may include the incorporation of cooperative work on the same binary capability and distribution over replicated environments. These approaches balance the load on CPU and memory.

REFERENCES

[1] Z. Wu, J. W. Atwood, and X. Zhu, "A new fuzzing technique for software vulnerability mining," *International Conference on Software Engineering*, 2009.

[2] H. Kim, Y. Choi, D. Lee, and D. Lee, "Practical security testing using file fuzzing," *10th International Conference on Advanced Communication Technology*, 2008.

[3] H. L. Guang, W. Gang, Z. Tao, M. S. Jian, and C. T. Zhuo, "Vulnerability analysis for x86 executables using genetic algorithm and fuzzing," *Third International Conference on Convergence and Hybrid Information Technology*, 2008.

[4] J. Jang, C. Sun, and E. Mizutani, *Neuro-Fuzzy and Soft Computing*. Prentice Hall, 1997.

[5] D. Nauck, F. Klawonn, and R. Kruse, *Foudations of neuro-fuzzy systems*. Wiley, 1997.

[6] A. Ghosh, B. Shankar, and S. K. Meher, "A novel approach to neuro-fuzzy classification," in *Neural Networks*. Elsevier Science Ltd, 2009, vol. 2, no. 1.

[7] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, 1st ed., A.-W. Professional, Ed. Addison-Wesley Professional, 2007, no. 321446119.

[8] P. S. Browne and M. Rockville, "Computer security technology planning study," *AFIPS Joint Computer Conferences*, 1976.

[9] A. One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 49, 1996.

[10] Huku, "Exploiting dlmalloc frees in 2009," *Phrack Magazine*, vol. 66, 2009.

[11] M. Alvares and G. Bittencourt, "Capture the flag challenges," 2009, uCon Security Conference.