ISSN 2229-5518

Seeking A Convergence Solution in Detecting Metamorphic: An Hybrid Evolutionary Stochastic Framework and Model

[1]A.O. Eboka and [2]A.A. Ojugo
[1]Department of Computer Science, Federal College of Education Technical, Asaba, Delta State, Nigeria. andre_y2k@yahoo.com
[2]Department of Mathematics/Computer Science, Federal University of Petroleum Resources Effurun, Delta State, Nigeria
ojugo.arnold@fupre.edu.ng

Abstract**—*Malware alter the behaviour of a host machine's file by self-replicating its codes unto it. On execution, some malware change its structure so that its copies have same functionality but differ in signature and syntax from its parent – making signature-based detection unreliable. Machine learning has yielded ways to evolve malware codes (even when some employ code obfuscation) to generate complex variants of base virus. This study samples metamorphic engine as hybrid with GAPSO to yield faster, highly diverse variants of base virus. It employs GA's exploratory and flexibility to learn feats within extracted data as well as PSO's speed and navigation to yield a robust optimal solution and faster, completely morphed copies of base virus. With learning rates set between [0.2, 0.35], $\phi 1$, = 1.5, $\phi 2$ = 2.5, ϖ = 0.14 and MaxGen of 500 epochs – yields better and faster convergence. Other values led to a slower convergence and/or non-convergence. Result shows the evolved variants as tested on commercial antivirus.*

Keywords—metamorphics, evolutionary, stochastic, malware,

IJSER

I. Introduction

THE computer virus is a malicious program that modifies a host machine by attaching its code and alters behaviour of other files. As it infects, it also modifies itself to include better and possibly, an evolved copy of the virus (Daodu and Jebril, 2008; Dawkins, 1989; Zakorzhevsky, 2011). Desai (2008) Brain (as the first computer virus) was a boot sector virus created in 1986 that infects the host machine resources such as files and macros, operating system, system sectors, companion files and source code. Use of Internet for data transfer has become a soft target for their widespread to help wreck havoc faster globally. Early detection of viruses is thus, imperative to minimize the damage caused.

A. Modules of a Computer Virus

Virus has 3-modules: infect, trigger and payload. Infect show its mechanism to modify its host and contain copies of it. Trigger details when and how to deliver its payload; while the payload details damage done. Trigger and payload are optional (Desai, 2008). Fig. 1a is virus pseudo-code; while Fig. 1b is an infect pseudo-code. Subroutine *Infect* selects a target from M-targets to infect when run. *Select_target* details target selection criteria as same target should not be repeatedly selected; else, reveals presence of a virus. And, *Infect_code* performs actual infection by inserting its code into the target (Ye et al, 2008).

Malware self-replicates its codes onto a machine without the user's consent, and spreads by attaching a copy of itself to some part of program file. It attacks system resources and is designed to deliver a payload that aims to corrupt program, delete files, reformat disks, crash network, destroy critical data or embark on other damage to the host machine (Szor, 2005).

```
Def Virus():
    Infect()
    If Trigger() is TRUE then
        Payload is delivered()
```

Fig 1a: Virus Pseudo-code

```
Def Infect():
    Repeat M times()
        Target = Select_target()
          If no target() THEN
              Return
   Infect code(target)
```

Fig 1b: Infect Pseudo-code

Viruses are classified into (Mishra, 2003; Orr, 2006; 2007): (a) simple virus replicates itself if launched. It gains control of the system, attaches copy of itself to another program as it spreads. After which, it transfers control back to host program. It is easily detected via search/scan for a defined sequence of bytes, known as a signature to find the virus, (b) encrypted Viruses scrambles its signature – making it unrecognizable at its execution. Its decryption routine transfers control to its decrypted virus body so that each time it infects a new program, it makes copy of both the decrypted body and its related decryption routine. It then encrypts a copy and attaches both to a target system. It uses an encryption key to encrypt its body. As the key changes, it scrambles its body so that virus appears different from one infection to another. Such virus is difficult to detect via signature. Thus, antivirus must scan for a constant decryption routine instead, (c) polymorphics consists of a scrambled body, mutation engine and decryption routine. The decryption routine gains control to decrypt both its body and mutation engine. It then transfers control to the scrambled body to locate a new file to infect. It copies its body and mutation engine into RAM, and invokes its mutation engine to randomly generate new decryption routine to decrypt its body with little or no semblance to the previous routine. It then appends this newly encrypted body, a mutation engine and decryption routine to the newly infected file. Thus, the encrypted body and the decryption routine, varies from one infection to another. With no fixed signature and decryption routine, no two infections is alike, and (d) metamorphics avoid detection by rewriting completely, its code each time it infects a new file. Its engine accomplishes this code obfuscation and metamorphism, which in most cases – is 90% of its assembly language codes.

A. Virus Detection Mechanisms

Antivirus software detects, prevent and remove all malware, including but not limited to viruses, worms, Trojans, spyware and adware. Antivirus use strategies namely: heuristic search, cyclic redundancy check, logic search and spy on processes to scan for viruses. Detection mechanism is broadly grouped into: (a) signature-based scans for signature, and to evade it – virus makers create new virus strings that can alter their

structure while keeping its functionality via code obfuscation method, and (b) code emulation creates sandbox or virtual machine, so that files are executed within it and scanned for virus. Once the virus is detected, it is no longer a threat – since it is running in controlled environment that limit damage to host machine (Singhal and Raul, 2012; Rabek et al, 2003).

Antivirus often impairs system performance, and incorrect decision may lead to security breach as it runs at the kernel of the operating system. If an antivirus uses heuristics, its success depends on the right balance between positives and negatives. Today, malware may no longer be executables. Macros can present security risk and antivirus heavily relies on signature-detection. Metamorphic and polymorphic viruses, evades and makes signature detection, quite ineffective (Filiolel, 2005).

Studies have shown that AVs effectiveness has decreased against unknown or zero-day attacks. This problem has been magnified by the changing intent of virus makers. Independent testing on all the major virus scanners consistently shows none to yield 100% detection. The best ones yield 99.6% detection, while lowest is 81.8%. Consequent of the fact that all scanners can yield a false positive result as well, identifying benign files as malware (Hashemi et al, 2008; Grimes, 2001).

II. METAMORPHIC VIRUSES

Rather than use encryption, metamorphics change its code structure/appearance while keeping its functionality. It does this via code obfuscation methods as in fig 2. Its engine reads in a virus executable, locates code to be transformed using its locate_own_code module. Each engine has its transformation rule that defines how a particular opcode or a sequence of opcodes is to be transformed. Decode module extracts these rules by disassembling. Analyze module analyzes current copy of virus and determines what transforms must be applied to generate the next morphed copy. Mutate module performs the actual transformations by replacing an instruction (set) with the other its equivalent code; While, Attach module attaches the mutated or transformed copy to a host (Cohen, 1987; Desai, 2008; Orr, 2007 and Sung et al, 2004).

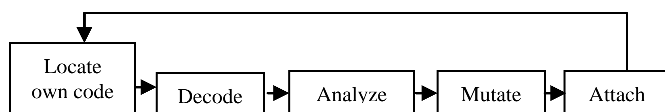

Fig. 2: Distinct Signature of Metamorphic

Venkatesan (2008) note that a typical metamorphic engine may consist of: (a) internal disassemble to disassemble binary codes, (b) a shrinker replaces two or more codes with its single equivalent, (c) an expander replaces an instruction with many codes that performs same action, (d) a swapper reorders codes by swapping two/more unrelated codes, (e) a relocator assigns and relocate relative references such as jumps and call, (f) a garbager (constructor) inserts whitespaces (do-nothing codes) to the program, and (g) cleaner (destructor) undoes the actions of a garbager by removing whitespaces/do-nothing instructions (Desai, 2008; Konstatinou, 2008).

Feats of an effective metamorphic engine includes: (i) must be able to handle any assembly language opcode, (ii) shrinker and swapper must be able to process more than one instruction concurrently, (iii) garbager is used moderately, not to affect actual instructions, and (iv) swapper analyzes each instruction so as not to affect next instructions' execution (Orr, 2007; Sung et al, 2004; Walenstein et al, 2007).

A. *Metamorphic Code Obfuscation Methods*

Metamorphic engine uses code obfuscation to yield morphed copies of original program. Obfuscated code is more difficult to understand and can generate different looking copies of a parent file as it operates on both control flow and data section of a program (Wong, 2006). Code obfuscation is achieved via (Borello and Me, 2008; Desai, 2008 and Avcock, 2006):

a. Register Usage Exchange/Renaming – modifies the register data of an instruction without changing the codes itself, which remain constant across all morphed copies. Thus, only the operands changes.
b. Dead Code inserts do-nothing (whitespace) codes that do not affect execution via a block or single instruction so as to change codes' appearance while retaining functionality.
c. Subroutine Permutation aims to reorder subroutines so that a program of many subroutines can generate (n-1)! varied routine permutations, whose addition will not affect its functionality as this is not important for its execution.
d. Equivalent Code Substitution replaces instruction with its equivalent instruction (or blocks). A general task can be achieved in different ways. Same feat is used in equivalent code substitution.
e. Transposition/Permutation – modifies program execution order only if there is no dependency amongst instructions.
f. Code Reorder inserts unconditional and conditional branch after each instruction (or block), and defines branching instructions to be permuted so as to change the programs' control-flow. Conditional branch is always preceded by a test instruction which always forces the execution of the branching instruction.
g. Subroutine Inline/Outline is similar to dead code insertion in that subroutine call are replaced with its equivalent code as Inline inserts arbitrary dead code in a program; while outline converts block of code into subroutine and replace the block with a call to the subroutine. It essentially does not preserve any logical code grouping.

B. *Advantage of Metamorphic Viruses*

Metamorphics transform its codes as they propagate to avoid detection by using obfuscation methods to alters its behaviour when it detects its execution within virtual machine (sandbox) as means to challenge a deeper analysis (Lakhotia et al, 2004). Virus writer use weaknesses of AVs, as limited to static and dynamic analysis, and attacks these: (a) data flow, (b) control flow graph generations, (c) procedure abstract, (d) property verification, and (e) disassembly – all means to counter scans, to identify such metamorphic viruses (Konstantinou, 2008). To mutate its code generation, metamorphics analyze their own codes and must re-evaluate the mutated codes generated (since complexity of transformation in the previous generation has a direct impact on its current state, how a virus analyses and transforms code in its current generation). Thus, they employ code conversion algorithm that helps them detect their own obfuscation and reordering (Ojugo, 2010).

III. MACHINE LEARNING AND SOFT INTELLIGENT COMPUTING

Aims to merge Artificial Intelligence with other fields, so as to create a synergetic field, dedicated to solving problems via optimization. It simultaneously, exploits numerical data as well as explores human knowledge via statistical pattern analysis tools, mathematical models and symbolic reasoning (Ojugo et al, 2012a). The models must be robust, so that with partial truth, imprecision, uncertainty and noise applied to its input, it yields an output guaranteed of high quality. Such model use Evolutionary Algorithms – capable of performing quantitative data processing to ensure qualitative statements of knowledge and experience as natural languages. Inspired by behavioural patterns and evolution laws in biological population, its tuning explores 3-basic feats: (a) adaptation to yield agents, void of local minima with high-diverse random immigrants introduced to slow convergence and a balance between exploitation and exploration so that learning feats of change, biases its solution accordingly, (b) robustness estimates a model's effectiveness, and (c) decision is flexible as uncertainty feats can impacts a model's future state in forecasts while focusing on its goal state and ease with blackbox integration (Ojugo, 2013).

Statistical pattern analysis has proven the most successful technique to detect metamorphic viruses via machine learning heuristics. It involves evolutionary optimization frameworks and models such as neural networks, Bayesian model, hidden Markov model, gravitational search, genetic algorithm etc – all of which are well known modeling tools and recently, used in detection of polymorphics cum metamorphics.

A. *Models Limitations and Fitness Function*

Stochastic model are often time consuming and their speed often shrink as they approach optima with their use of hill-climbing technique that often gets them stuck local minima. They also require extra computational power to search, and are computationally intensive and expensive to implement.

A fitness function evaluates if an optimal is found, as model learns data feat/relationship, compare forecast versus observed values. Its performance measures fitness value (Ojugo, 2012).

B. *Genetic Algorithm*

GA as inspired by Darwinian evolution (survival of fittest), consists of a dataset chosen for natural selection with potential solutions. Individuals with genes close to its optimal solution, is fit as determined by the fitness function determines (Perez and Marwala, 2011). GA has 4-operators namely:

a. Initialize – Individual data are encoded into format suitable for selection. Each encodings has its merit/demerit. Binary encoding is computationally more expensive to achieve. Decimal encoding has greater diversity in chromosome and greater variance of pools generated; float-point encoding or its combination is more efficient than binary. Thus, it encode as fixed length vectors for one or more pools of different types. The *fitness* function evaluates how close a solution is to its optimal – after which they are chosen for reproduction. If solution is found, function is *good*; else, is *bad* and not selected for crossover. The fitness function is the only part with knowledge of task. If more solutions are found, the higher its fitness value.
b. Selection – Good fit individuals close to optimal are chosen to mate. The larger the number of selected, the better the chances of yielding fitter individuals. This continues until one is chosen, from the last two/three remaining solutions, to become selected parents to new offspring. Selection ensures the fittest individuals are chosen for mating but also allows for less fit individuals from the pool and the fittest to be selected. A selection that only mates the fittest is *elitist* and often leads to converging at local optima.
c. Crossover ensures that individual of fitter gene is exchanged to yield a new, fitter pool. There are 2-types of crossover namely: (a) simple crossover for binary encoded pool via particular- or multi- point; and all genes are from one parent, and (b) arithmetic crossover allows new pool to be created by adding an individual's percentage to another. The one to be used depends on encoding type used.
d. Mutation alters chromosomes by changing its genes or its sequence, to ensure new pool converges to global minima. Algorithm is stopped either when an optimal is found, or after a number of runs or once no better solution is found. Genes change based on probability of mutation rate, and mutation improves the needed diversity in reproduction.

Cultural GA is one of the many variants of GA with 3-belief spaces defined as follows: (a) Normative – notes the specific range of values to which an individual is bound, (b) Domain belief – has information about the task domain), (c) Temporal belief – has information about the search space that is available), and (d) spatial belief has topographical data about the task with time as a specific feat. In addition, CGA has an influence function that mediates between its belief spaces and the pool – to ensure that individuals (that are altered or not), all in the pool conforms to the belief space. CGA is chosen so as to yield a pool that does not violate its belief space – since it will also help reduces the number of possible individuals GA generates till an optimum is found (Reynolds, 1994; Hassan and Crossley, 2004).

C. *Particle Swarm Optimization*

PSO as a population based optimization method that predicts motion in swarm collective intelligence and specify a model of randomly initialized candidates distributed in space to find its optima. Its search for optimal solution on large amount of data is assimilated and/or shared by the entire swarm – so that particles are generated who have adapted to their environment (via computed fitness function) with all constraints satisfied in a number of moves. Also, desirable traits evolve within the swarm though swarm's composition remains as particles with of better traits replace those with weaker ones (Homaifar et al, 1992).

Encoded, PSO handles discrete value as (Ojugo et al, 2012):

a. **Position/Velocity Update:** Particle is solution in space that changes its position during a move, based on updates. Swarm is initialized with random-generated positions X_i and velocities V_i as in Eq. 1 and Eq. 2 distributed as thus:

$$X_o^1 = X_{min} + rand(X_{max} - X_{min}) \quad (1)$$

ISSN 2229-5518

$$V_o^1 = \frac{X_{min} + rand(X_{max} - X_{min})}{\nabla t} = \frac{Position}{Time} \quad (2)$$

Hu et al (2005a,b) velocity update is achieved via fitness function to yield particle's best (a function of its current position, current swarm's global best (P_g^i) and each particle best position P_i (current and previous). It uses the effect of current motion (V_i^t) to give direction for the next move V^i_{t+1}. To avoid trapped at local optima, it uses Eq. 3 with V^i_{t+1} as particle's velocity at t+1, $\varpi+V^i_t$ is current motion, X^i_t is particle position, $\phi1$*rand()[(P_i-X^i_t)/∇t] is particle's influence factor and $\phi2$*rand() [(P^g_t-X^i_t)/∇t] is swarm's influence factor:

$$V_{t+1}^1 = \omega + V_{(t)}^1 + \varphi_1 rand() \frac{(P^1 - X_t^1)}{\nabla t} + \varphi_2 rand() \frac{(P_t^2 - X_t^2)}{\nabla t} \quad (3)$$

b. **Position Update** is achieved in 2-steps namely: via fitness function calculation as repeated until convergence criteria is reached and via velocity/position update. Stop criterion is set at, when maximum change in best fitness is smaller than needed in the number of moves, as in Eq. 4:

$$X_{t+1}^1 = X_t^1 + V_{t+1}^1 * \nabla t \left| f(P_t^g) - f(P_{t-g}^g) \right| \in \varepsilon \quad (4)$$

Kennedy and Mendes (2002) and Clerc (1999) a particle may hold values beyond X_{max} and X_{min}, due to its current position and updated velocities (that grows rapidly). As such, particles may diverge instead of converge. They are dragged back to the nearest side constraint via Eq. 5 (that handles particle velocity explosion constraints via linear exterior penalty, if such particles violate bound value).

$$f(x) = \varphi(X) + \sum_{i=1}^{N_{max}} X_i * \max[0, g_i(x)] \quad (5)$$

Ojugo et al (2012a) PSO algorithm is as thus:

1. Input: Generations size, Output: set of permuted solutions.
2. Randomly Initial created solution population.
3. Set ϕ1 = 1.5, ϕ2 = 2.5, MaxGen = 500 epoch and T = 0
4. Set N = total solution and set generationCounter = 0
5. For each solution in population
6. Position = min(X) + rand{(max(X) – min(X)}
7. Update Velocity = (Potion / Time)
8. If best individual fitness is close to solution
9. Then compute new position as V^i_{t+1} : End if
10. If particles excites out of bound then compute f(x) //bring them back
11. End if: End For Each Solution

IV. Experimental Design Framework

A. Virus Abstract Representation

The study uses Real Permutation metamorphic engine (as adopted for generation of Zmist., Zperm and Zmorph viruses). It uses substitution, transposition and trash (all permutation) methods to build viruses of the same functionality. The engine changes its opcode, generating new variants from old versions (authored by Zombie and extracted from VX Heaven). Zmist at its release was one of the most complex binary viruses ever written. It uses Entry-Point Obscuring that supports a unique method called code integration and occasionally inserts jumps after every instruction in a code section, pointing to the next instruction. It extremely modifies applications and files from one generation to next. Thus, allowing it extreme camouflage and making Zmist, more of a perfect (and sometimes anti-heuristic) virus (Konstantinou, 2008 and Szor, 2005).

B. The Framework (Hybrid Model)

Our framework is an adaptation of (Noreen et al, 2008) that aims to evolve new malware from a known virus database. The first step is high-level of abstract representation (or genotype) of given virus that requires great understanding of the virus functionality and structure. It determines quality of evolution achieved by proposed framework, while including functional details of the virus characteristics and that of the metamorphic engine in use. Some known feats and attributes includes: date, domain, application-to-infect, port number, email attachment, registry variable, mail-body, file extension, process terminated, peer-to-peer propagation etc, which forms its abstract representation of the base virus to be taken as input into system (see fig 3).

Second step is the application of the evolutionary algorithm to the high-level representation. Thus, dataset is divided into: train (50%), cross validation (25%) and test (25%). The fitness of offspring as evaluated in Eq. 6 is a function of the similarity measure of the genes (chromosomes) with that of all stages of the framework. Individuals that evolved but do not match the training samples, their feats are stored and forms input to the next iteration. Thus, we have these conclusions as adopted by Noreen et al (2008) thus: (1) new individuals are malware to be used during testing, whose abstract represented feats are fed-back into model, (2) new individual is an unknown Zmist virus, and (3) new individual is (not) Zmist virus. Like Noreen et al (2008), facts 2 and 3 are established once executed within a sandbox in an operating system at real-time.

We theorized unlike Noreen et al (2008) that if we generated Zmist virus for test (as against having testing data from base virus abstract representation) – it invalidates the experiment to some degree as mutation yields a better and fitter generation. Instead, we argued that a combination of the old feats and new feats as extracted from both dataset and metamorphic engine at each stage of the process as well as feedback into the system to generate newer variants to yield greater evolution (backward compatibility in terms of functionality to the base virus).

GA initializes the hybrid with an entire population of 500-input (suitable abstract representation of base virus), computes individual fitness of each individual via Eq. 6 as well as selects 30-individual via tournament method to yield the new sub-pool (and determine individuals to proceed for mating). Selected data are moved for crossover and mutation so that model or network learns static/dynamic feats in the obtained data. With 30-individuals selected via tournament and 2-point crossover used, other parents contribute to yield new pool whose genetic makeup is a combination of both parents. Mutation will yield 3-random genes that are allocated new random value that still conforms to belief space. The number of mutation applied, depends on how far CGA is progressed (and how fit is the best fit individual in the pool). Thus, number of mutations equals fitness of the fittest individual divided by 2. New individuals replace old ones with low fitness values (Ojugo et al, 2013a,b).

$$F = \sum_{i=1}^{k} \frac{f_i}{k} \quad (6)$$

Each particle in PSO (30-individuals from CGA) are moved over and encoded as a string of positions in multidimensional space. Position/Velocity updates are performed independently in each dimension (a merit of PSO). Though, not for such an evolution/permutation task, as candidate solutions depends on each other. Thus, two/more particles have can have same value for both velocity and position. Particles can also have values outside the boundary after an update, which breaks the rule of permutation. Thus, all conflicts are resolved (see Eq. 5). With larger velocity, particles explore more space and will likely change (though all update formulas remain the same). Velocity is limited to absolute values, which represents the difference between particles). This continues till an individual in the pool with a fitness of 0 is found. Thus, the solution has been reached (Ojugo et al, 2013).

Selection and mutation in GA ensures the first 3-beliefs are met; while velocity/position updates in PSO ensures that the fourth belief space is met, as time is of paramount interest. Also, influence function determines number of mutations takes place; And knowledge of how close task is to solution, has direct impact on how model is processed. Algorithm stops when best individual has a fitness of 0 (Ursem et al, 2002).

C. *Tradeoffs and Issues in Metamorphic Malware*

Researchers designed routines to detect metamorphics (one-by-one) and detect varied sequences of code known to be used by given mutation engine via signature search. This method is proved inherently impractical, time-consuming and costly as each metamorphic requires its own detection program. Also, the mutation engine can seemingly randomly, generate billions variation of virus and different engines used by metamorphics make any identification somewhat unreliable. This has led to, mistakenly identifying one virus for another. Thus, researchers have modeled statistical method to associate signature to such metamorphics based on probability.

Hybrids are difficult to implement though its encoding via structured learning addresses existing statistical dependencies amongst variables to yield better pool via crossover/mutation. This feat can be adapted in areas of software evolution. Use of GAPSO hybrid with the metamorphic engine's obfuscation will yield Zmist virus variants in the shortest time of high and discrete, morphed copies. Our resulting morphed copies are tested against normal files and commercial virus scanners.

V. Results and Findings

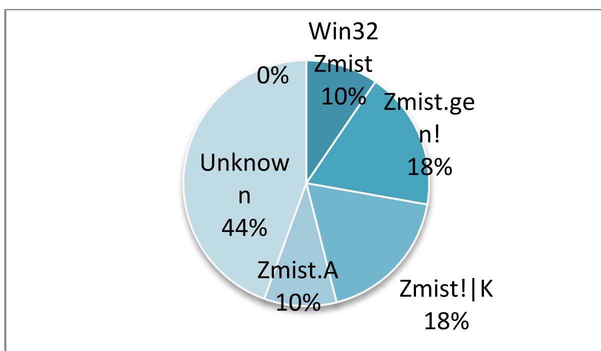

Fig. 4a: Evolved Variants Scanned with Eset

In summary, with fitness function and selection criteria that is common to both GA and PSO, it is discovered that learning rates set between 0.2 and 0.35, and PSO parameters set thus: $\phi1$, = 1.5, $\phi2$ = 2.5, MaxGen = 500 epochs and ϖ = 0.14 yields a better and faster convergence. Other parameter values led to a slower convergence and sometimes, non-convergence. When tested against commercial antivirus, the evolved virus as scanned with ESET detects 56% of generated variants; while Norton Symantec detects 47% as in fig. 4a and 4b.

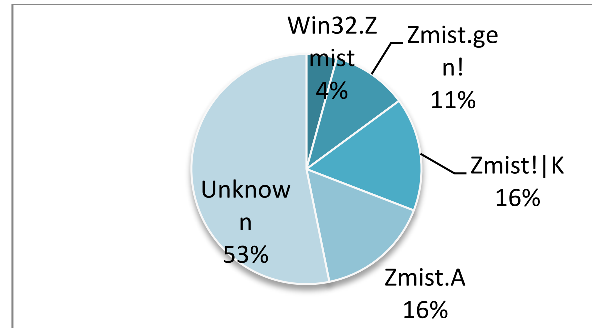

Fig. 4b: Evolved Variants scanned with Norton Symantec

VI. Conclusion

With Noreen et al (2008), we note that though the proposed framework is posed as an evolvable malware system. It can be adapted to software evolution – which is a process associated with modifying existing software for both backward and forward compatibility as well as emphasizes component reuse (Gray and Klefstad, 2005). Also, a wide variety of replicative and non-replicative malware can also be evolved via proposed framework to increase network security research and study.

References

[1] Aycock, J., (2006). *Computer Viruses and malware*, Springer Science and Business Media.

[2] Borello, J and Me, L., (2008). *Code obfuscation techniques for Metamorphics*, www.springerlink.com/content/233883w3r2652537

[3] Cohen, F., (1987). *Computer viruses: theory and experiments*, Computer Security, 6(1), p22-35.

[4] Clerc, M., (1999). *The swarm and the queen: towards a deterministic and adaptive particle swarm optimization*, In Proceedings of Evolutionary Computation (IEEE), 5, p123-132.

[5] Daoud, E and Jebril, I., (2008). *Computer Virus Strategies and Detection Methods*, Int J. Open Problems Computational Mathematics, 1(2), www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf

[6] Dawkins, R., (1989). *The selfish gene (2nd edition)*, Oxford Univ. Press

[7] Filiol, E., (2005). *Computer Viruses: from Theory to Applications*, New York, Springer, ISBN 10: 2287-23939-1.

[8] Gray, J and Klefstad, R., (2005). *Adaptive and evolvable software systems: techniques, tools and applications*, 38th Annual Hawaii Int. Conf. on System Sciences, p274, IEEE Press.

[9] Grimes, R., (2001). *Malicious Mobile Code: Virus Protection for Windows*, O'Reilly and Associates, Inc., Sebastopol, CA, USA.

[10] Hashemi,S., Yang, Y., Zabihzadeh, D and Kangavari, M., (2008). *Detecting intrusion transactions in databases using data item dependencies and anomaly analysis*, Expert Systems, 25(5), p460, doi:10.1111/j.1468-0394.2008.00467.x

[11] Hassan, R and Crosswley, W., (2004). *Variable population-based sampling for probabilistic design optimization and with a genetic algorithm*, Proceedings of 42nd Aerospace Science, p32, Reno: NV.

[12] Hassan, R., Cohanin, B., De Wec and Venter, G., (2004) *Comparison of PSO and GA,* Proceeding of 44th Aerospace Sci.,Washington, p56.

[13] Homaifar, A.A., Turner, J and Ali, S., (1992). *N-queens problem and genetic algorithms"*, Proceedings of IEEE Southeast conference, p262.

[14] Hu, X., Eberhart, R.C and Kennedy, J., (2005a). *Solving constrained nonlinear optimization problems with PSO,* Proceeding Multi-conference on Systems, Cybernetics and Informatics, p234.

[15] Hu, X., Eberhart, R.C and Shi, Y., (2005b). *Swarm intelligence for permutation optimization: case study of n-queens*, Proceedings of Genetic Evolutionary Computing on Memetic Algorithms (IEEE), p243

[16] Kennedy, J and Mendes, R., (2002). *Population structure and particle swarm performance*, Proceedings of Congress on Evolutionary Computation (IEEE), p1671, Honolulu: Piscataway.

[17] Konstantinou, E., (2008). *Metamorphic virus: Analysis and Detection,* Technical report (RHUL-MA-2008-02), Dept. of Mathematics, Royal Holloway, University of London.

[18] Lakhotia, A., Kapoor, A and Kumar, E.U., (2004). *Are metamorphic computer viruses really invisible*? Part 1, Virus bulletin, p5-7.

[19] Mishra, P., (2003). *Taxonomy of software unique transformations*, www.cs.sjsu.edu/faculty/stamp/students/FinalReport.doc

[20] Ojugo, A.A, (2010). *The computer virus evolution: polymorphics analysis and detection*, J. of Academic Research, 15(8), p34 – 46.

[21] Ojugo, A., Eboka, A., Okonta, E., Yoro, R and Aghware, F., (2012). *GA rule-based intrusion detection system*, J. of Computing and Information Systems, 3(8), p1182.

[22] Ojugo, A.A., and Yoro, R., (2013a). *Computational intelligence in stochastic solution for Toroidal Queen task,* Progress in Intelligence Computing Applications, 2(1), doi: 10.4156/pica.vol2.issue1.4, p46

[23] Ojugo, A.A., Emudianughe, J., Yoro, R.E., Okonta, E.O and Eboka, A.O., (2013b). *Hybrid artificial neural network gravitational search algorithm for rainfall runoff,* Progress in Intelligence Computing and Applications, 2(1), doi: 10.4156/pica.vol2.issue1.2, p22.

[24] Orr, (2006). *The viral Darwinism of W32.Evol: An in-depth analysis of a metamorphic engine*, http://www.antilife.org/files/Evol.pdf

[25] Orr, (2007). *The molecular virology of Lexotan32: Metamorphism illustrated*, http://www.antilife.org/files/Lexo32.pdf

[26] Rabek, J., Khazan, R., Lewandowski, S., Cunningham, R., (2003). *Detection of injected, dynamic generated and obfuscated malicious code*, Proceeding ACM Workshop on Rapid Malcode, p76.

[27] Reynolds, R., (1994). *An introduction to cultural algorithms*, IEEE Transaction on Evolutionary Programming, p131.

[28] Singhal, P and Raul, N., (2012). *Malware detection module using machine learning algorithm to assist centralized security in Enterprise networks*, Int. J. Network Security and Applications, 4(1), doi: 10.5121/ijnsa.2012.4106, p61

[29] Sung, A., Xu, J., Chavez, P., Mukkamala, S., (2004). *Static analyzer of vicious executables,* Proceedings of 20th Annual Computer Security Applications Conf., IEEE Computer Society, p326-334.

[30] Szor, P., (2005). *The Art of Computer Virus Research and Defense*, Addison Wesley Symantec Press. ISBN-10: 0321304543, New Jersey.

[31] Ursem, R., Krink, T., Jensen, M.and Michalewicz, Z., (2002). *Analysis and modeling of controls in dynamic systems*. IEEE Transaction on Evolutionary Computing, 6(4), p378-389.

[32] Walenstein, R., Mathur, M., Chouchane R., and Lakhotia, A., (2007). *The design space of metamorphic malware*, Proceedings of 2nd Int. Conference on Information Warfare, p243.

[33] Wong, W., (2006). *Analysis and Detection of Metamorphic Computer Viruses*, Master's thesis, San Jose State University, http://www.cs.sjsu.edu/faculty/students/Report.pdf

[34] Venkatesan, A., (2008). *Code Obfuscation and Metamorphic Virus Detection*, Master thesis, San Jose State University, www.cs.sjsu.edu/faculty/students/ashwini_venkatesan_cs298report.doc

[35] VX Heavens Virus Collection, Available at: http://vx.netlux.org/

[36] Ye, Y., Wang, D., Li, T and Ye, D., (2008). *Intelligent malware detection based on association mining*, J. Computer Virology, 4(4), p323–334, doi: 10.1007/s11416-008-0082-4.

[37] Zakorzhevsky, E.R., (2011). *Monthly Malware Statistics*, www.securelist.com/en/analysis/204792182/Monthly_Malware_Statistics_June_2011.

IJSER

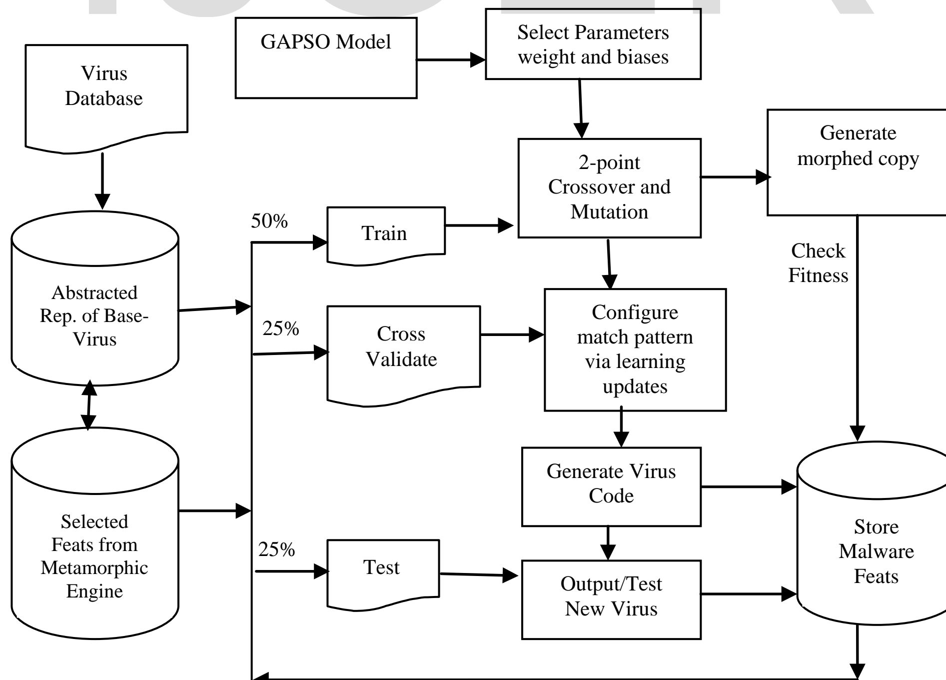

Fig. 3: Experimental Model for Virus Generation