Computing, Information Systems, Development Informatics & Allied Research Journal
Vol 7 No 2 June 2016 - www.cisdijournal.net

An Evolvable Framework for Metamorphics

Allenotor, D.
Department of Mathematics & Computer Science
Federal University of Petroleum Resources
Effurun, Delta State, Nigeria
allenotor.david@fupre.edu.ng

ABSTRACT

Malware have the ability to alter the behavior of a host machine or file by self-replicating its codes unto it. On execution, some malware changes its structure so that its copies have same functionality. However, signatures and syntax differs from their parents. This property makes signature-based detection a hard problem. Machine learning has yielded ways to evolve malware codes (even when some employ code obfuscation) to generate complex variants of base virus. This study applies 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 Particle Swam Optimization (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. Experimented results (tested on commercial antivirus) show a trend to slower convergence and/or non-convergence.

Keywords: Stochastic, Models, Evolutionary, Optimization. PSO, and Operating Conditions.

CISDI Journal Reference Format
Allenotor, D. (2016): An Evolvable Framework for Metamorphics. Computing, Information Systems, Development Informatics & Allied Research Journal. Vol 7 No 2. Pp 33-40 Available online at www.cisdijournal.met

1. INTRODUCTION

A computer virus is a malicious program that modifies a host machine by attaching its code and alters behavior of other files. As it infects, it also modifies itself to include better and possibly, an evolved copy of the virus ([4], [5] [23]). The first computer virus, a boot sector virus was created in 1986. It infects the host machine resources such as files and macros, operating system, system sectors, companion files and source code. The use of Internet for data transfer has become a soft target for their widespread to help wreak havoc faster globally. Early detection of viruses is therefore, imperative to minimize the damage caused.

1.1 Modules of a Computer Virus

Virus has 3-modules: infect, trigger, and payload. The infect module show its mechanism to modify its host and contain copies of itself. The trigger module details when and how to deliver its payload; while the payload details damage done. Trigger and payload are optional. Figure 1 (a) is the virus pseudo-code; while Figure 1 (b) is an infect pseudo-code. In this figure, the Subroutine *infect* selects a target from M-targets to infect when executed. The *Select_target* details target selection criteria since the same target should not be repeatedly selected. If it is repeatedly selected, it will reveal the presence of a virus. 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 [22].

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

(a) Virus Pseudo-code

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

(b) Infect Pseudo-code

Figure 1: Sample Virus Pseudo-code

Figure 1 (a) shows a simple virus replicates itself if launched. It gains control of the system; attaches copy of itself to another program as it spreads. After this process, it then transfers control back to the host program. It is easily detected using search/scan for a defined sequence of bytes, known as a signature to find the virus, in Figure 1 (b), the simple virus has an encrypted scramble signature. This makes 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 if the signature system is used. It therefore means that an antivirus must scan for a constant decryption routine instead.

Polymorphics consists of a scrambled body, mutation engine, and a 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. After this process, it appends the 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 are similar. 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 accounts for more than 90% of its assembly language codes.

1.2 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 two; the signature-based scans for signature and the code emulation. In code emulation, sandbox or virtual machine is created and 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 [18] and [20]. 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 [6].

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 [9] and [8].

2. METAMORPHIC VIRUSES

Rather than use encryption, metamorphics change its code structure/appearance while keeping its functionality. It does this using code obfuscation methods as in Figure 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 [3], [17], and [21].

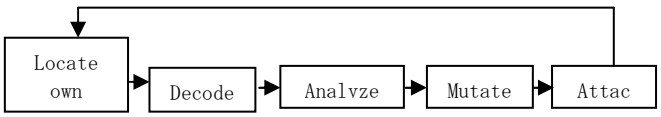

Figure 2: Distinct Signature of Metamorphic Virus

Venkatesan in [23], describe a typical metamorphic engine to consists internal disassemble to disassemble binary codes, a shrinker which has the potential to replace two or more codes with its single equivalent, an expander which replaces an instruction with many codes that performs same action, and a swapper reorders codes which swaps two/more unrelated codes. Other components of the metamorphic engine include a relocator which assigns and relocate relative references such as jumps and call, a garbager (constructor) which inserts whitespaces (do-nothing codes) to the program, and cleaner (destructor) undoes the actions of a garbager by removing whitespaces/do-nothing instructions

2.1 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 [1] and [2]:

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.

2.2 Advantage of Metamorphic Viruses

Metamorphics transform its codes as they propagate to avoid detection by using obfuscation methods to alters its behavior when it detects its execution within virtual machine (sandbox) as means to challenge a deeper analysis [13]. 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 [9]. 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 [14].

3. MACHINE LEARNING AND SOFT INTELLIGENT COMPUTING

Soft intelligent computing aims to merge Artificial Intelligence (AI) with other fields, so as to create a synergetic field, dedicated to solving problems using optimization techniques. It simultaneously, exploits numerical data as well as explores human knowledge through the use of statistical pattern analysis tools, mathematical models and symbolic reasoning. 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 uses Evolutionary Algorithms which can perform quantitative data processing to ensure qualitative statements of knowledge and experience as natural languages. Inspired by behavioral patterns and evolution laws in biological population, its tuning explores 3-basic achievements; (i) 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, (ii) robustness estimates a model's effectiveness, and (iii) 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. 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.

3.1 Models Limitations and Fitness Function

Stochastic models 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.

3.2 Genetic Algorithm

Genetic Algorithm (GA) as inspired by Darwinian evolution (survival of fittest), consists of a dataset chosen for natural selection with potential solutions. The GA has 4-operators namely:

a. Initialize – Individual data are encoded into format suitable for selection. Each encoding 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 encodes 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: (i) simple crossover for binary encoded pool via particular- or multi- point; and all genes are from one parent, and (ii) 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 (CGA) is one of the many variants of GA with 3-belief spaces defined as follows: (i) Normative – notes the specific range of values to which an individual is bound, (ii) Domain belief – has information about the task domain), (iii) Temporal belief – has information about the search space that is available), and (iv) 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 [9] and [19].

3.3 Particle Swarm Optimization

Particle Swam Optimization (PSO) is 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 [10]. Encoded, PSO handles discrete value as [15]:

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 Equation (1) and Equation (2) distributed as thus:

$$X_0^1 = X_{min} + rand(X_{max} - X_{min}) \quad (1)$$

$$V_0^1 = \frac{X_{min} + rand(X_{max} - X_{min})}{\nabla t} = \frac{Position}{Time} \quad (2)$$

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_{t+1}^i. To avoid trapped at local optima, it uses Equation (3) with V_{t+1}^i as particle's velocity at t+1, $\varpi+V_t^i$ is current motion, X_t^i is particle position, $\phi 1$*rand()[$(P_i-X_t^i)/\nabla t$] is particle's influence factor and $\phi 2$*rand() [$(P_t^g-X_t^i)/\nabla t$] is swarm's influence factor:

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

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 Equation (4):

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

A particle may hold values beyond X_{max} and X_{min}, due to its current position and updated velocities (that grows rapidly) [3] and [11]. As a result, particles may diverge instead of converge. They are dragged back to the nearest side constraint using Equation (5) (that handles particle velocity explosion constraints via linear exterior penalty, if such particles violate bound value). In [16], the PSO algorithm was given as:

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

1. Input: Generations size, Output: set of permuted solutions.
2. Randomly Initial created solution population.
3. Set $\phi 1 = 1.5$, $\phi 2 = 2.5$, 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 excite out of bound, then compute f(x) //bring them back
11. End if: End For Each Solution

4. EXPERIMENTAL DESIGN FRAMEWORK

4.1 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 [12] and [22].

4.2 The Framework (Hybrid Model)

The presented framework is an adaptation of [38] 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/attributes are: 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 (for example, see Figure 3).

The 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 Equation (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, these conclusions as adopted in [38], therefore, (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. in [25], facts 2 and 3 are established once executed within a sandbox in an operating system at real-time.

Therefore, I theorized unlike in [38] that if I 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, I 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 using Equation (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 that for every F, give that:

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

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, since the 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. That means all conflicts are resolved (for example, see Equation (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.

Selection and mutation in GA ensures the first 3-beliefs are met; velocity/position updates in PSO ensures that the fourth belief space is satisfied. Also, influence function determines number of mutations takes place; A knowledge of how close task is to solution, has direct impact on how model is processed. The algorithm stops when best individual has a fitness of 0.

4.3 Trade-offs 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 in signature search. This method is proved inherently impractical, time-consuming and costly because each metamorphic requires its own detection program. Similarly, 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 using crossover/mutation. This feat can be adapted in areas of software evolution. The 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. The obtained resulting morphed copies are tested against normal files and commercial virus scanners.

5. RESULTS AND FINDINGS

Using fitness function and selection criteria that is common to both GA and PSO, it was discovered that learning rates set between 0.2 and 0.35, and PSO parameters set thus: $\emptyset_1 = 1.5, \emptyset_2 = 2.5,$ MaxGen $= 500,$ epochs and $\varpi = 0.14$ 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 Figure 4a and Figure 4b.

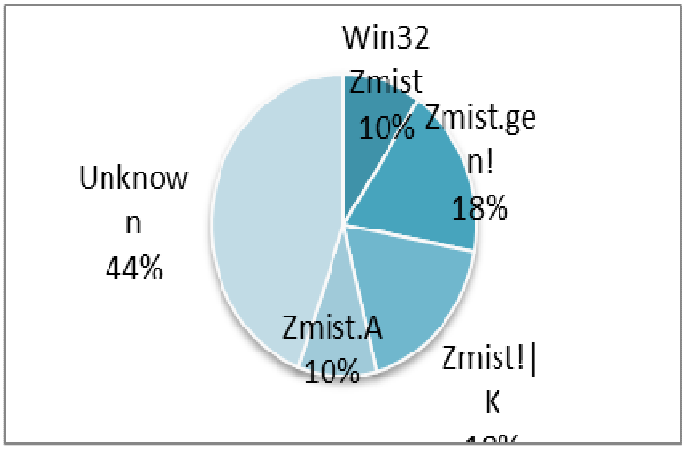

Fig. 4a: Evolved Variants Scanned with Eset

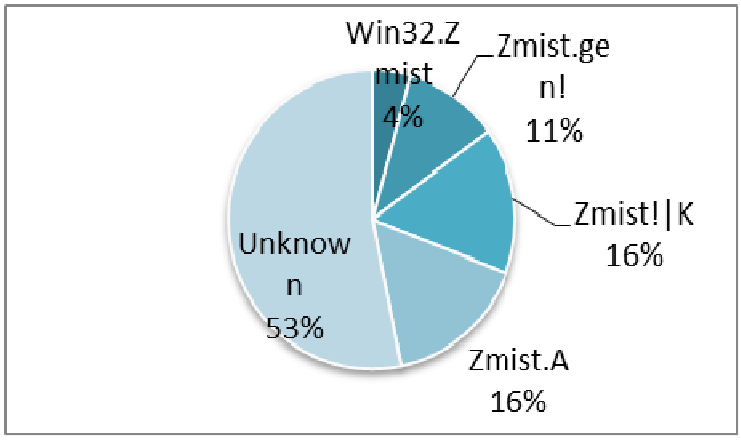

Fig. 4b: Evolved Variants scanned with Norton Symantec

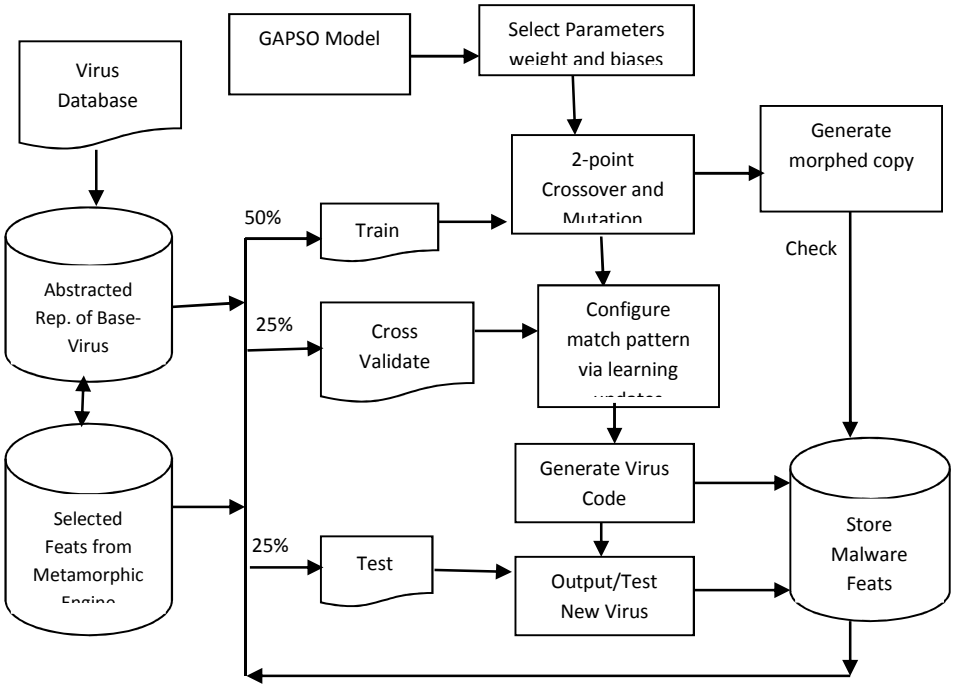

Fig. 5: Experimental Model for Virus Generation

6. CONCLUSION

With [38], 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 [8]. A wide variety of replicative and non-replicative malware can also be evolved using proposed framework to increase network security research and study.

REFERENCES

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

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

[3] Clerc, M. *The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization*, In Proceedings of Evolutionary Computation (IEEE), 1999. p123-132.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] Singhal, P and Raul, N. *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, 2012, p61

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

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

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

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

[24] Noreen, S. Murtaza, S, Shafiq M. Z, and Farooq, M. Evolvable malware. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation, GECCO '09, pages 1569–1576, New York, NY, USA, 2009. ACM.