On Evolving Buffer Overflow Attacks Using Genetic Programming

Hilmi Güneş Kayacık
Dalhousie University
Faculty of Computer Science
6050 University Avenue
Halifax, Nova Scotia, Canada
kayacik@cs.dal.ca

Malcolm Heywood
Dalhousie University
Faculty of Computer Science
6050 University Avenue
Halifax, Nova Scotia, Canada
mheywood@cs.dal.ca

Nur Zincir-Heywood
Dalhousie University
Faculty of Computer Science
6050 University Avenue
Halifax, Nova Scotia, Canada
zincir@cs.dal.ca

ABSTRACT

In this work, we employed genetic programming to evolve a "white hat" attacker; that is to say, we evolve variants of an attack with the objective of providing better detectors. Assuming a generic buffer overflow exploit, we evolve variants of the generic attack, with the objective of evading detection by signature-based methods. To do so, we pay particular attention to the formulation of an appropriate fitness function and partnering instruction set. Moreover, by making use of the intron behavior inherent in the genetic programming paradigm, we are able to explicitly obfuscate the true intent of the code. All the resulting attacks defeat the widely used 'Snort' Intrusion Detection System.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; I.2.2 [**Automatic Programming**];

General Terms

Algorithms, Design, Security.

Keywords

Linear Genetic Programming, Mimicry Attacks, Intrusion Detection Systems.

1. INTRODUCTION

All users of virus checkers, firewalls and more generally signature-based intrusion detection systems are familiar with the need to continuously receive updates to the original base detection system. The basic nature of the intrusion detection problem is that new attacks are continuously under development. As a consequence patches to your personal firewall, virus checker or intrusion detection system are also required in order to plug the current favorite attack instance. The bottom line however, is that an omnipresent third party is required. Such a third party is responsible for recognizing unseen attacks from the log files once the system is attacked and then developing the necessary signature patch. Thus, your detector is only as good as the most recent attacks such a third party is able to correctly label. Anomaly detection (as opposed to signature-based detection) on the other hand concentrates on modeling what constitutes "normal behavior". Any deviation from the normal behavior is then flagged as an attack. This naturally results in a system able to identify novel attacks, but at the expense of false positives. That is to say, what constitutes normal behavior is not straightforward to establish, and is invariably specific to a user-application-network mix, making it impossible to carry models of normal behavior between different customers, limiting the product base for such systems.

In this work we are interested in building detection systems using a genetic programming (GP) methodology, with the aim of discovering rules suitably generic for describing a wide range of anomalous behaviors. However, there are at least two pragmatic limitations constraining the applicability of GP based detectors. Firstly, the datasets used to characterize intrusion detection problems typically consist of millions of exemplars, which implies an overhead in training time. Secondly, once trained, the model is only as good as the data available at training, a third party is again required to provide appropriate labels for new attack instances. Solutions to the first problem have been demonstrated by way of active learning algorithms [1], [2], [3].

In this work we propose to address the second problem by evolving a "white hat" attacker i.e., the purpose of this attacker is to generate the attack data for which a detector will be built. Within this context the principal objective of the attacker is to camouflage a 'core' attack in such a way that the signatures at the detector are unable to discover the true nature of the code. In doing so, we are interested in making use of the code bloat property from GP where, within this context, it provides a mechanism for hiding the real intent of the code. Finally, we focus on the case of buffer overflow attacks, where such attacks represent one of the most widely utilized models of attack. From the detection perspective, this then means that we are building a modular detection platform, with different detectors associated with specific forms of attack.

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

2. EVOLVING BUFFER OVERFLOW ATTACKS

The core behavior of an overflow attack lies in the simple observation that just because an address space of a variable declared in a program might be allocated of a specific size, this does not stop the same program from attempting to access memory outside of the allocated space. In order to make use of such a weakness, the attacker requires three components: (1) A program used by the target system that possesses an inherent overflow vulnerability; (2) Knowledge of the size of memory reference necessary to cause the overflow; and (3) The correct placement of a suitable exploit to make use of the overflow when it occurs. The skill in crafting such an attack lies in how an exploit is hidden and ensuring that the memory referenced outside of the allocated space corresponds to the code defining the desired malicious behavior.

The generic buffer overflow attack consists of three components: the payload, the "NOP" (No operation) sled, and the "return address". The payload represents the shell code used to perform the malicious activity once operation of the buffer has been compromised. Although the specific content of any payload will vary (e.g. for different operating systems or exploit goals), encryption of the payload has in effect rendered detection of the payload itself impossible [4]. The buffer overflow is actually caught through the use of the NOP sled and return address components of the attack. The basic purpose of NOP sled is to maximize the likelihood of executing the payload. The NOP sled achieves this by occupying buffer space with non-operational code that correctly identifies the beginning of the payload. Thus, the likelihood of it directing program flow to the payload increases, albeit at the expense of providing a bigger signature for detection as the ratio of NOP sled to payload increases. The return address fills the buffer space following a payload with code to push program operation into the NOP sled, thus also correctly identifying the payload. Classically, a NOP sled appears before the payload and the return address follows the payload. As long as the overflowed reference falls within the NOP sled or the return address fields of the attack, the payload will be executed. In order to maximize the likelihood of this happening the payload itself is usually very short.

Needless to say, given the above basic definition of the overflow style of attack there is a wide range of approaches to achieving the exploit. Recent work has indicated that signature-based detectors might recognize a particular instance of such an attack, but are unable to generalize to the class of buffer or heap overflow attacks. Thus, given a generic overflow attack, it is comparatively easy to build different variants of the same attack, either by hand [5], [6] or automatically with suitable *a priori* information [7].

2.1 Methodology for Evolving a Buffer Overflow Attack

The principal objective of our attacker is to continuously evolve plausible attacks. As indicated above, the generic overflow attack consists of a NOP sled, a payload, and a return address. We consider two basic scenarios. Scenario-1 implies that we accept this model and assume that the payload is encrypted i.e. not detectable. The basic objective is then to find instruction sequences that are non-trivial yet do not in themselves do anything (trivial sequence might constitute a list of identical commands, where this appears to be the norm and is a characteristic widely used in signature based detection systems). Moreover, such instruction sequences should be designed to maximize the likelihood of the payload being executed, thus we attempt to find the optimal ratio of payload, NOP sled, and return address. Scenario-2 drops the preconception that an overflow attack should consist of three discrete components. Instead the operation of each element is distributed across the length of the individual. Moreover, the ideal would be for the payload to be distributed such that correct execution would always result. This implies that the return address function will always take place if the first instruction associated with the (distributed) payload is missed; whereas if execution begins before the first instruction associated with the (distributed) payload, then operation takes the form of the NOP sled.

Observations providing the basis for this approach include (1) recognizing that the GP code bloat phenomena provides the basis for the non operational command sequences, thus masking the operational or real intent of an attack command sequence. Secondly, GP solutions have been widely observed to have functionality distributed across the length of the individual. Thus, the ideal objective of an attack agent will be to build command sequences, which have the same function as the original generic attack, however camouflaged in non-operational but syntactically correct commands. Such a system is only possible if a sufficiently informative fitness function can be defined for the class of attacks as a whole. That is to say, a binary fitness function in which all unsuccessful attacks provide a fitness of zero and a successful attack a fitness of unity, is on the face of it, not much use. Section 2.2 will therefore define the objective of the attacker using multiple behavioral objectives. Finally, we recognize that we need to have multiple solutions (attacks), thus a mechanism for maintaining diversity may be necessary.

2.2 Basic Fitness Function

Categorically, the attack we are evolving is an 'execve' attack. Execve is a system call that executes a program, where the program takes the form of an argument (UNIX shell /bin/sh in our case). UNIX defines 'execve' as, `int execve(const char *path, char *const argv[], char *const envp[])`.

Where parameter one is the command name; parameter two contains pointers to strings that will be given to the program as arguments; parameter three contains pointers to environmental variables, which are also stored as strings. A minimalist call to the 'execve' function using the C language might have the following form,

```
int main()
{
   char *command = "/bin/sh";
   char *args[2];
   args[0] = command;
   args[1] = 0;
   execve(command, args, 0);
}
```

In order to spawn a UNIX shell prompt, 'execve' requires that the command pointer should be in the EBX register, the pointer to 'args' should be in register ECX and the pointer to the third

argument (which is in our case is the NULL pointer) should be in EDX register, Algorithm 1. Moreover, the program name '/bin/sh' should be pushed to stack. To achieve these goals, 11 assembly instructions are needed. After 10 instructions are executed (11th is the interrupt that transfers control to execve system call), registers EAX, EBX, ECX and EDX should be correctly configured and the stack should contain the program name to be executed (i.e. /bin/sh). Given the state of the stack and registers after the 10th instruction, if the values are not set correctly, greedy replacement is used to determine how many instructions are needed to correct it. For example if "/bin/sh" has not been pushed to the stack, 3 instructions are sufficient to achieve this goal. Another important point is that if the task has been half-accomplished, viable instructions should be determined. Using this principle, the fitness function summarized in Algorithm 2 returns a maximum fitness of 10 if all conditions are satisfied, otherwise it subtracts the number of instructions needed to correct the program, relative to the minimal set of sub-goals, Algorithm 1. The basic fitness function therefore takes the form of a hierarchical fitness function in which sub-goals (a) to (e) can only be completed in sequence. However, depending on the composition of the language used to evolve the attacks, there are multiple programs producing the required (buffer overflow) attack behavior.

Algorithm 1 Minimal requirements for executing a 'execve' system call for spawning a UNIX shell.

1. Register EAX contains `0x0B` i.e., the system call number of 'execve';
2. Register EBX points to '`/bin/sh0`' on the stack;
3. Register ECX points to the argument array in stack;
4. Register EDX contains `NULL`;
5. Interrupt '`0x80`' is executed;

Algorithm 2 Basic fitness function for establishing correct behavior of 'execve' exploit.

Fitness = 10;

(a) IF stack does not contain '`/bin/sh0`', THEN subtract number of instructions necessary to do so from Fitness (1 to 3);

(b) IF register EBX does not point to string from (a), THEN Fitness -= 1;

(c) IF register ECX does not point to argument array in stack, THEN subtract number of instructions necessary to do so from Fitness (1 to 3);

(d) IF register EDX != NULL, THEN Fitness -= 1;

(e) IF an INT is not executed, THEN Fitness -= 1;

2.3 Runtime Environment and Fitness Evaluation

In order to obtain the behavioral fitness requirements defined in Algorithm 2, individuals representing an exploit are executed. Although it is possible to run the attack on a 'real' environment, this approach suffers from the disadvantage that an attack can potentially crash the environment, terminating the training process as a whole. Therefore execution of the program is mimicked using a virtual runtime environment, which simulates the execution of assembly programs on 32-bit Intel Architecture. Although limited in functionality, the runtime environment is developed with sufficient functionality to execute an execve system call properly, Figure 1. Limitations take the form of explicitly prohibiting accesses to, or modification of memory or the heap. The runtime environment, Figure 1, contains simulated data structures such as:

1. General-purpose IA32 registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) that can be used in 32, 16-bit modes. 8-bit mode is available on EAX, EBX, ECX, and EDX;
2. A special purpose register for testing flag values;
3. An addressable stack;

Registers and stack are also simulated, that is to say, execution of a program does not modify the actual stack and registers of the host machine. Execution of each instruction is defined as a C function that modifies the simulated data structures of the virtual machine. Instructions are implemented using the IA32 instruction definitions from IA32 Developer's Manuals [8]. Special attention was given to flag modification in order to determine which execution branch to take e.g., the case of a control transfer instruction such as a JMP. When the interrupt instruction is called, a snapshot of the virtual machine state is taken, from which the fitness is calculated.

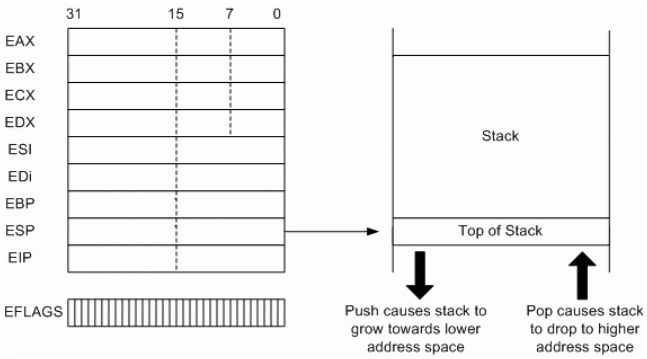

Figure 1. Virtual runtime environment for fitness evaluation.

2.4 Linear GP

Individuals are represented using linear GP in which instructions are composed from a 2-byte opcode and two operands (each 1-byte) i.e. all instructions have the same number of bytes. Table 1 defines the instruction set architecture where IMM32 denotes the 32 bit immediate values (immediate is the term used for constants), REG32 denotes the 32 bit general purpose registers, EREG32 denotes the extended 32 bit registers and LREG8 denotes the low 8 bit registers, Table 2.

Individuals are defined using a fixed length format, thus initialization is defined over the total range of permitted program lengths, Table 3. Selection takes the form of a steady state tournament over 4 individuals. The children from the best performing half of the tournament overwrite the individuals corresponding to the worst half of the tournament, taking their place in the population. Search operators take three forms: two point crossover, instruction mutation, and instruction swap, Table 3. Crossover is therefore constrained to exchanging an equal number of instructions (a page) between two individuals. The number of instructions per page is allowed to vary from 1 instruction to max instructions per page as the fitness function

reaches a new plateau, as in the page-based Linear GP framework [9]. Mutation selects a single instruction with uniform probability and replaces with a different instruction from the instruction set, Table 1. The swap operator selects two instructions from the same individual with equal probability and interchanges their respective positions.

Table 1. Linear GP instruction set

Instruction	Instruction Type	Parameter 1	Parameter 2
INT	Control Transfer	`0x80`	N/A
CDQ	Data Transfer	N/A	N/A
PUSH		IMM32	N/A
PUSH		REG32	N/A
MOV		EREG32	EREG32
MOV		LREG8	`0x0B`
ADD	Binary Arithmetic	REG32	REG32
SUB		REG32	REG32
INC		REG32	N/A
DEC		REG32	N/A
MUL		REG32	N/A
DIV		REG32	N/A
AND	Logic	REG32	REG32
OR		REG32	REG32
XOR		REG32	REG32
NOT		REG32	N/A

Table 2. Parameter types

Parameter Type	Options
IMM32	`0x68732f2f` (“hs//”), `0x6e69622f` (“nib/”)
REG32	EAX, EBX, ECX, EDX
EREG32	REG32 + ESP
LREG8	AL, BL, CL, DL

Table 3. GP parameters

Parameter	Setting
Crossover	Page based crossover with 0.9 probability
Mutation	Uniform instruction-wide mutation. with 0.5 probability
Swap	Instruction swap within an individual with 0.5 probability
Selection	Tournament of 4 individuals
Stop Criteria	At the end of 50,000 tournaments.
Population	500 individuals with 10 pages and 3 instructions per page.
Training Time	Approximately 6 hours.

The details of the linear Genetic Programming methodology itself is not particularly important, however, previous work utilizing Grammatical Evolution (GE) indicated that the linear representation provides a more direct method for successfully evolving buffer overflow attacks (the search operators in GE were not particularly efficient at manipulating register references) [10].

3. RESULTS

As indicated in the introduction, two basic approaches are considered in the design of buffer overflow attacks. In the first case we retain the concept of three readily identifiable components to the attack (NOP, payload, and return address). This was previously demonstrated using Grammatical Evolution (GE) [10]. However, in the second scenario in which the objective is to develop the attack itself whilst maximizing the probability of executing the malicious code, GE proved very inefficient at manipulating register references (using the standard GE search operators). By using linearly structured GP, we expect to avoid this problem. In the following we describe a series of three experiments in which the instruction set is incrementally expanded, thus increasing the search space, but providing for greater freedom in the resulting program content (thus a wider range of behavioral properties). This case results in code that has the capacity to intermix attack and obfuscation.

In all cases the fitness function takes the form of Algorithm 2, augmented with an additional term to measure the likelihood of an attack being executed. Specifically, since all individuals have a fixed length of 30 instructions and it takes 11 instructions to describe the attack, there are up to 19 instructions denoting introns with respect to the malicious code (effective NOPs). If the approximated return address was not accurate enough to jump to the first instruction, jumping to an effective NOP region would allow an attack to deploy successfully. If execution of a successful attack fails (i.e. an inaccurate return address) by jumping past a relevant instruction, the location of the instruction is called the failure point. The probability of execution is defined as (failure point ÷ number of all possible points); or a denominator of 19 in this case.

3.1 Minimal Instruction Set

In the first experiment, we consider an instruction set composed from the minimum subset of instructions necessary to build the malicious exploit alone (first 5 instructions of Table 1 plus the XOR instruction). This represents a minimal search space in which the relevant instruction sequence, instruction arguments, and intron behavior are all expressed in terms of the 6 instructions known to describe the minimalist exploit.

Figures 2 and 3 summarize the probability of executing a working exploit and number of unique individuals under the basic fitness function (baseline) and basic fitness function with the additional objective of maximizing the probability of executing a valid exploit. From Figure 2, it is apparent that including the additional objective doubles the likelihood of executing the exploit. A unique individual differs from all others in the population by at least one or more instruction. Figure 3 indicates the population diversity is maintained throughout the generations, albeit with evolution incorporating the additional objective enforcing an additional constraint on diversity.

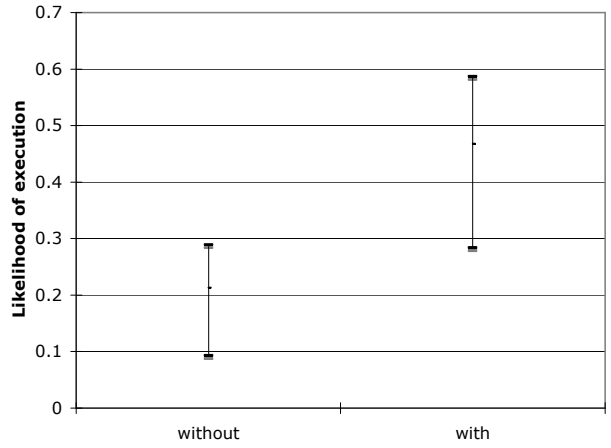

Figure 2. Likelihood of executing an attack: With and without additional fitness objective (1st , 2nd and 3rd quartile).

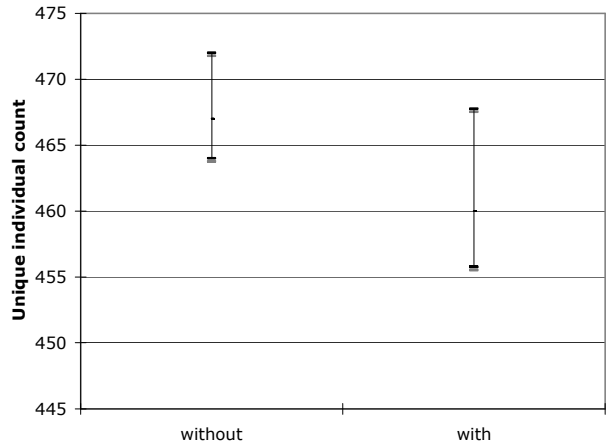

Figure 3. Population diversity: With and without additional fitness objective (1st, 2nd and 3rd quartile).

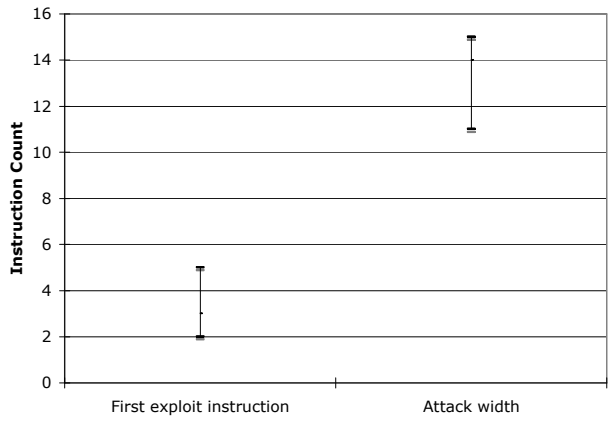

Figure 4. Intron / exon characteristics of successful attacks (1st, 2nd and 3rd quartile).

Figure 4 details the distribution of introns within the attack. If the removal of an instruction does not affect the outcome, it is considered as intron. Figure 4 shows that the first instruction of the exploit code starts in the first third of the program. The attack width, which is defined as the number of instructions between the first and the last exploit instruction, shows that introns are mixed with the exploit code, Figure 4.

Table 4. Evolved attack compared with the core attack from which the fitness function is developed.

Evolved Program	Core Attack	Sub-goals
PUSH 0x68732f2f		
MUL EAX		
PUSH EBX		
MUL EDX		
CDQ		
CDQ		
SUB EAX, EAX	XOR EAX, EAX	(d)
MUL EDX	CDQ	(d)
PUSH EDX		
MOV CL, 0x0b		
PUSH EDX		
DEC ECX		
DEC ECX		
MOV EBX, ESP		
PUSH 0x6e69622f		
PUSH EDX	PUSH EAX	(a)
PUSH 0x68732f2f	Same	(a)
PUSH 0x6e69622f	Same	(a)
MOV EBX, ESP	Same	(b)
MOV ECX, EDX	PUSH EAX (step 1)	(c)
CDQ		
MUL EDX		
PUSH ECX	PUSH EAX (step 2)	(c)
PUSH EBX	Same	(c)
MOV ECX, ESP	Same	(c)
MOV AL, 0x0b	Same	(e)
INT 0x80	Same	(e)
PUSH EDX		
PUSH 0x6e69622f		
MOV DL, 0x0b		

Table 4 provides a comparison between an evolved attack and the core attack from which the fitness function was developed. Exploit code is shown in bold whereas the remaining instructions of the program act like introns. It is apparent that the evolved attack discovered different ways to attain the sub-goals (a), (c) and (d) in Algorithm 2. The evolved attack executes successfully and spawns a UNIX shell.

3.2 Extended Instruction Sets

Two additional experiments are conducted, each augmenting the base instruction set as follows:

- Basic 6 instructions plus 6 arithmetic instructions, Table 1;

- Basic 6 instructions, plus 6 arithmetic, plus three logical, Table 1.

The results are detailed in terms of mean fitness, number of hits (i.e. number of programs, which has fitness above or equal to 10), and the mean probability of execution for the basic instruction set and the two increments, Figures 5 6, and 7 respectively. All three figures indicate an improved characteristic when arithmetic instructions are included in the instruction set. Thus, arithmetic instructions were either helpful in attaining objectives in different ways or they were good introns. Although the introduction of logical instructions worsen the results compared with the inclusion arithmetic instructions, the results are still better than the basic instruction set. Hence, extending the search space by adding new instructions does not have substantial negative impact on deploying successful attacks.

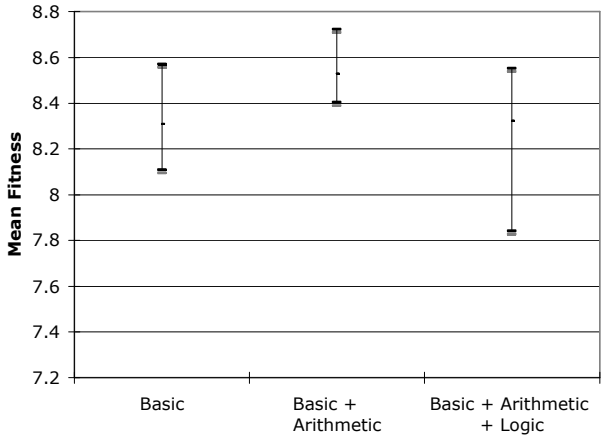

Figure 5. Mean fitness averaged over 20 runs (1st, 2nd and 3rd quartile).

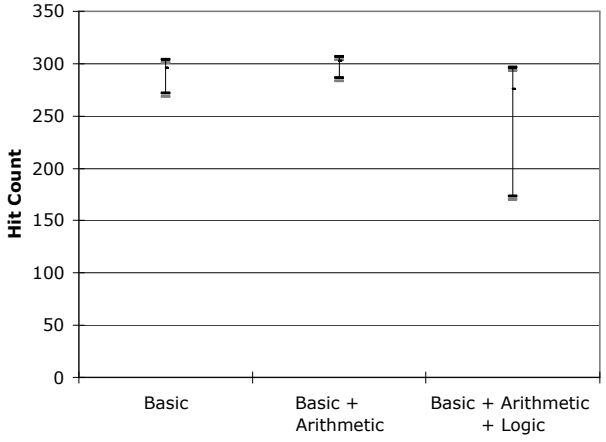

Figure 6. Hit Count averaged over 20 runs (1st, 2nd and 3rd quartile).

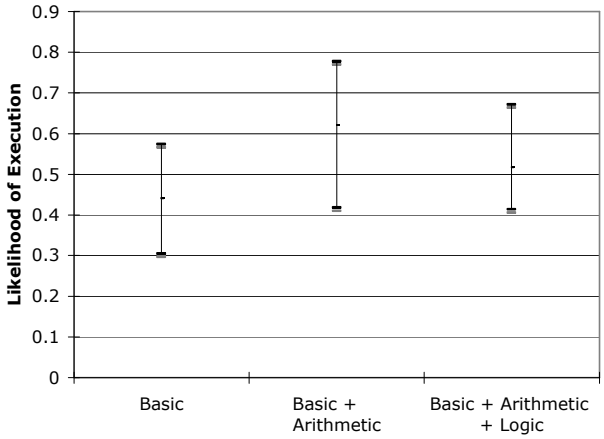

Figure 7. Mean likelihood of exploit execution averaged over 20 runs (1st, 2nd and 3rd quartile).

4. RELATED WORK

Previous works have demonstrated the use of random modifications to a 'core' attack with the objective of highlighting weaknesses in standard signature based detectors [7]. Updates to a detector based on an immune system using a Genetic Algorithm (GA) as the 'attacker' have also been proposed [11]. In this case, however, no attack as such is built (the problem simplifies to function optimization). A GA has also been proposed for acting as an attack agent in an artificial 'server-user-hacker' environment. In this case the GA is used to construct a hacker behavior from a predefined set of attack scripts [12]. As a consequence there is no attempt to obfuscate the true intent of the intended behavior, or discover alternative methodologies for achieving the same objective. Finally, in a previous work, we demonstrated that Evolutionary Computation (GE in particular) may be used to establish the composition of buffer overflow attacks comprising of a predefined exploit, NOP sled, and return address [10]. The result being a set of attacks capable of defeating Snort, a widely used signature based intrusion detection system.

5. CONCLUSION

In this work, GP is used as a "while hat" attacker with the objective of altering the core attack to make it undetectable by signature based intrusion detection systems. Results show that code bloat property of the GP provides suitable means to hide the actual attack by mixing exploit instructions with introns that have no effect toward the success of the attack. Furthermore, evolved attacks discover different ways of attaining sub-goals associated with building buffer overflow attacks, hence mimicking the core attack with different instructions. Consequently, it becomes harder for a signature-based detector to detect the resulting attack variant. Our experiments focused on formulating a suitable fitness function and defining instruction sets. Results showed that employing an additional 'likelihood' objective increased the chances of deploying successful attacks. Expanding the instruction set provided additional intron behavior and supported different avenues for mimicking sub-goals associated the core attack. In order to observe the detection of the evolved attacks, successful attacks are tested against Snort, which is a widely used network based intrusion detection system. Successful attacks are

transmitted over a network, where Snort is deployed, monitoring the network traffic. All of the 2110 attacks avoided detection by Snort.

Future work will expand our experiment scope to heap based overflows and worms. Furthermore, we are interested in developing a gaming environment in which attackers and defenders will coevolve. Such an "arms race" will avoid the requirement for a third party to explicitly label exploits after the fact, but provides the opportunity to actually preempt new attack behaviors before they are encountered. In such a scheme, one class classification will be employed to build a detector from training data consisting of attack exemplars alone, thus avoiding the potentially open ended problem of characterizing 'normal' behavior.

6. ACKNOWLEDGMENTS

This work was supported in part by Killam pre-doctoral scholarship of the first author and NSERC, MITACS and CFI grants of the second and third authors. All research was conducted at the NIMS Laboratory, http://www.cs.dal.ca/projectx/.

7. REFERENCES

[1] D. Song, M.I. Heywood, A.N. Zincir-Heywood. A Linear Genetic Programming Approach to Intrusion Detection. In *Proceedings of Genetic and Evolutionary Computation Conference*, GECCO, Springer-Verlag, Lecture Notes in Computer Science, 2724, pages 2325-2336, 2003.

[2] R. Curry, M.I. Heywood. Towards Efficient Training on Large Datasets for Genetic Programming. In *Canadian Conference on Artificial Intelligence*, pages 161-174, Springer-Verlag, Lecture Notes in Artificial Intelligence, 3060, May 2004.

[3] D. Song, M.I. Heywood, A.N. Zincir-Heywood. Training Genetic Programming On Half a Million Exemplars: An Example from Anomaly Detection, *IEEE Transactions on Evolutionary Computation*, 9(3): 225-239, June 2005.

[4] ADMmutate. http://www.ktwu.ca/security.html

[5] D. Wagner, P. Soto, Mimicry Attacks on Host-based Intrusion Detection Systems, *ACM Conference on Computer Security*, pages 255-264. 2002.

[6] K.M.C. Tan, K.S. Killourhy, R.A. Maxion, Undermining an Anomaly-based Intrusion Detection System using Common Exploits, *In 5th International Symposium on Recent Advances in Intrusion Detection*, pages 54-73. Lecture Notes in Computer Science, LNCS 2516, 2002.

[7] G. Vigna, W. Robertson, D. Balzarotti, Testing Network Based Intrusion Detection Signatures Using Mutant Exploits, In *ACM Conference on Computer Security*, 2004.

[8] IA-32 Intel, Architecture Software Developer's Manual Volumes 2A, 2B: Instruction Set Reference, A-M, M-Z, 2005

[9] M.I. Heywood, A.N. Zincir-Heywood. Dynamic Page Based Crossover in Linear Genetic Programming*, IEEE Transactions on Systems, Man and Cybernetics - Part B*, 32(3), pp 360-388, June 2002.

[10] H.G. Kayacik, A.N. Zincir-Heywood, M.I. Heywood, Evolving Successful Stack Overflow Attacks for Vulnerability Testing, In *21st Annual Computer Security Applications Conference*, Dec 5-9 2005.

[11] G. Dozier, D. Brown, K. Cain, J. Hurley, Vulnerability analysis of immunity-based intrusion detection systems using evolutionary hackers, In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 263-274. Lecture Notes in Computer Science, LNCS 3102, 2004.

[12] J. Budynek, E. Bonabeau, B. Shargel, Evolving Computer Intrusion Scripts for Vulnerability Assessment and Log Analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1905-1912. ACM SIGEVO, Volume 2, June 25-29 2005.