Evolution of Network Enumeration Strategies in Emulated Computer Networks

Sean Harris
Department of Computer Science
Missouri University of Science and Technology
snhcn6@mst.edu

Eric Michalak
Los Alamos National Laboratory
emichalak@lanl.gov

Kevin Schoonover
Department of Computer Science
Missouri University of Science and Technology
ksyh3@mst.edu

Adam Gausmann
Department of Computer Science
Missouri University of Science and Technology
ajgq56@mst.edu

Hannah Reinbolt
Department of Computer Science
Missouri University of Science and Technology
hmrvg9@mst.edu

Joshua Herman
Department of Computer Science
Missouri University of Science and Technology
jjhf39@mst.edu

Dr. Daniel Tauritz
Department of Computer Science
Missouri University of Science and Technology
dtauritz@acm.org

Chris Rawlings
Los Alamos National Laboratory
crawlings@lanl.gov

Aaron Scott Pope
Department of Computer Science
Missouri University of Science and Technology
aaron.pope@mst.edu

ABSTRACT

Successful attacks on computer networks today do not often owe their victory to directly overcoming strong security measures set up by the defender. Rather, most attacks succeed because the number of possible vulnerabilities are too large for humans to fully protect without making a mistake. Regardless of the security elsewhere, a skilled attacker can exploit a single vulnerability in a defensive system and negate the benefits of those security measures. This paper presents an evolutionary framework for evolving attacker agents in a real, emulated network environment using genetic programming, as a foundation for coevolutionary systems which can automatically discover and mitigate network security flaws. We examine network enumeration, an initial network reconnaissance step, through our framework and present results demonstrating its success, indicating a broader applicability to further cyber-security tasks.

CCS CONCEPTS

• **Security and privacy → Network security**; • **Computing methodologies → Genetic programming**;

KEYWORDS

Network security, network emulation, genetic programming

ACM Reference format:
Sean Harris, Eric Michalak, Kevin Schoonover, Adam Gausmann, Hannah Reinbolt, Joshua Herman, Dr. Daniel Tauritz, Chris Rawlings, and Aaron Scott Pope. 2018. Evolution of Network Enumeration Strategies in Emulated Computer Networks. In *Proceedings of Genetic and Evolutionary Computation Conference Companion, Kyoto, Japan, July 15–19, 2018 (GECCO '18 Companion),* 8 pages.
https://doi.org/10.1145/3205651.3208270

1 INTRODUCTION

As computer networks have grown to encompass increasingly large quantities of sensitive applications, data, and critical infrastructure, their security has become increasingly paramount [22]. Investigations of high-profile breaches frequently report that the cause was a serious flaw or oversight in some aspect of security [21]. Consequently, offensive security in the form of penetration testing has become a six hundred million dollar industry [13], demonstrating that modern network defense strategies must be proactive. Unfortunately, the attack surface area and thus the number of attack strategies is extremely high for a large computer network, and organizations are temporally and financially resource restricted. To combat these restrictions, we propose a framework to facilitate automated, intelligent network penetration testing and vulnerability discovery to be used in addition to professional penetration testing.

One possible method for automating this is through coevolution of strategies for attacker and defender agents on a copy of the network. Our framework allows strategies for attacking a computer network to be evolved, letting attacker agents adapt to exploit weaknesses in a network or in a defensive strategy. Similarly, a population of defender agents could be evolved to develop strategies to modify the network and make it more secure, or make active responses during an attack. Work such as that of Rush et al. [20] has previously attempted this on simulated networks, but the reduction in fidelity that a simulation has compared to the complex behavior of a real network means that any results might not transfer well to the real world. Further, different simulations focus on different aspects of computer networks, and there is no standard simulator for cybersecurity scenarios, which limits the comparability of any published results. Ideally, then, such coevolution should take place on a real network, or at least an emulation of a real network, to ensure that behavior on the network provides an accurate test of agent strategies, and so that experiments are directly comparable to anything else using a real or emulated network.

The purpose of this paper is to demonstrate a proof-of-concept framework for the evolution of attacker agents in an emulated network on the task of network enumeration, a small subproblem of the larger task of attacking a network. Network enumeration is a reconnaissance task dealing with discovering the resources present on the network. We implement our framework through the use of strongly-typed genetic programming, allowing the evolution of strategies which generate sets of candidate IP addresses to scan for on the network based on addresses known through listening to network traffic or from previous scans. These scans allow the agents to efficiently discover a large proportion of the machines running on the network, information which a higher-level strategy would then use to inform its future actions. By developing infrastructure to run evolutionary algorithms (EAs) on emulated networks and demonstrating the evolution of strategies in this difficult environment, this work shows the feasibility of expanding these methods towards coevolving full attacker and defender strategies.

2 RELATED WORK

Evolutionary algorithms have previously been applied to the security domain in simulated environments. Mrugala et al. [16] used genetic programming to evolve attacker agents within simulated wireless sensor networks, intended to disrupt communications while remaining undetected by static defensive strategies. Attacker agents were able to improve individually upon packet suppression rates and detection rates by the defender compared to a handmade agent, but not both at once. By studying the strategies that the attackers were evolving, the authors were able to significantly improve the performance of the detection strategy.

In work by Garcia et al. [8], the authors coevolve attacker strategies and defender strategy selections on a simulated peer-to-peer network, where the attacker agents perform denial of service attacks against a set of nodes at different times as defined in the genotype. The defender selects one of three routing strategies. Several different variant forms of coevolution were evaluated. The authors found the best performance in a form of coevolution that treats each fitness the set of opposing strategies as a multiobjective problem, and selects individuals for the next generation based on the non-dominating front. The defenders were found to pick the most robust of the three strategies, which was unbeatable by the attackers on sufficiently large networks.

In research by Rush et al. [20] [19], the author coevolves attacker and defender strategies in a simulated network graph, in which the attacker agents, using strategies encoded by decision trees, are tasked with exploring and taking control of a network, and exploiting value from compromised machines. The defender has a fixed strategy during the attack, responding to alerts from intrusion detection systems by disconnecting nodes, and instead is tasked with evolving the structure of the network, along with augmentations such as intrusion detection systems. A large number of different parameters were tested for how they impacted coevolution, and evolutionary results were compared against those of a hill climber. Results found that defender strategies were difficult to evolve, and that the defenders performed best when they were tasked with making small modifications to an already functional network. Attacker strategies tended to be easier to evolve, and were found to get particular benefit from being able to make decisions based on their previous actions. Experiments with the hill climber showed that the use of coevolutionary algorithm was justified, since the search space was found to be highly multimodal.

3 BACKGROUND

3.1 Network Enumeration

Network Enumeration, often known as network reconnaissance, is the act of obtaining network information by sending IP packets and observing the responses. It is often the first step in any cyber attack as it provides critical information and the layout of the desired network [1]. Network Enumeration is often broken into the following stages: Host Discovery, or locating computer IP addresses; Port Discovery, or identifying networking port information about a specific host; and Service Detection, or determining which version of a software is running on the host. Following network enumeration, known vulnerabilities can be identified and exploited.

3.2 Strongly-Typed Genetic Programming

Genetic programming is a form of evolutionary computation which works on genotypes structured to represent a "program", a broad word which encompasses anything from algorithms to mathematical functions to circuit designs. Genetic programming represents these programs using structures of primitive nodes which receive input from and output to other such nodes. The oldest and most common way of structuring these genotypes is as a tree [11], where each vertex in the tree represents a function taking input from its child nodes, producing output either from the root node or through side effects of the functions. In this tree-based genetic programming, mutation and recombination operations occur based on subtrees: mutation replaces a subtree of the genotype with a new random subtree, and recombination replaces a subtree of one parent with another subtree transplanted from the second parent. In traditional forms of genetic programming, these primitive nodes are designed such that the output from any node serves as a valid input for any other node, such that the evolutionary operators can be applied arbitrarily. Many other forms of genetic programming exist, generally defined by using different genotype structures, such as linear genetic programming [2] which uses linear arrays of instructions interacting with registers, and Cartesian genetic programming [14] which holds a directed graph on a 2D grid of fixed size.

Strongly-typed genetic programming [15] is a variant of tree-based genetic programming which introduces a typing system into its tree structure, allowing for constraints on which primitive nodes are valid inputs for other primitive nodes, in line with the idea of type in programming languages. This removes the restriction that ordinary tree-based genetic programming has that primitive functions must be carefully designed to all work on the same data type, even when that might be unintuitive. Strongly-typed genetic programming as a result provides greater flexibility for the user in designing genotypes, and also allows the search space to be restricted to prevent function inputs which are clearly nonsensical. As a consequence, the evolutionary operators are more complicated, as they need to prevent type violations.

4 METHODOLOGY

4.1 Evolution

4.1.1 Agent behavior. Agents performing the network enumeration task have the ability to listen to traffic on their host machine, and to select specific sets of IP addresses to test using the network scanning tool *Nmap* [12]. The core idea of the agent strategy is that, short of a brute-force search, guessing IP addresses on a network involves making predictions about the structure of the network based on IP addresses which are already known. A 10.1.1.15 address with a 10.1.1.255 broadcast address suggests that there might be other addresses on the 10.1.1.0/24 subnet in use. The agents confirm the existence of a new IP address, found either by listening to network traffic or as a result of a previous scan, and use it to generate a new set of IP addresses according to a certain strategy, encoded as a genetic programming tree, to scan for. This is detailed in Algorithm 1.

Algorithm 1: Algorithm for network enumeration agents

```
input  : $GPTree()$, which takes an IP address as input and
         outputs a priority queue
$Queued \leftarrow \emptyset$ (A priority queue forbidding duplicates);
$Searched \leftarrow \emptyset$ (A set);
$Found \leftarrow \emptyset$ (A set);
while evaluation time < time limit do
    if new address IP found by listening then
        add $IP$ to $Found$;
        add first 256 elements of $GPTree(IP)$ not in $Searched$
          to $Queued$ ;
    if Queued is not empty then
        $ScanIPs \leftarrow$ first 256 elements of $Queued$;
        remove the first 256 elements of $Queued$;
        $NewIPs \leftarrow Scan(ScanIPs)$;
        add all $NewIPs$ to $Found$;
        for IP in NewIPs do
            add first 256 elements of $GPTree(IP)$ not in
              $Searched$ to $Queued$;
    if Queued contains fewer than 256 elements then
        for IP in Found do
            add first 256 elements of $GPTree(IP)$ not in
              $Searched$ to $Queued$ with low priority;
output : $Found$, the set of discovered IP addresses
```

4.1.2 Genetic Programming Trees. Our genetic programming trees function by taking an input IP address and generating a set

Table 1: Genetic programming primitives used

Name	Output type	Input types
addIPs	VOID	IP_LIST, SMALL_INT, VOID
Add the IPs to the queue with given priority		
end	VOID	
Terminates execution		
buildIPs	IP_LIST	BYTE_RANGE × 4
Generates all IPs within the ranges		
constant	BYTE_RANGE	BYTE
A range containing a single constant value		
inputValue	BYTE_RANGE	
A range with only the corresponding input octet		
interval	BYTE_RANGE	SMALL_INT
An interval of given radius around the input octet		
anyValue	BYTE_RANGE	
A range containing all possible values (0-255)		
weightedByte	BYTE	
A byte literal created near 0 or 255 normally		
smallInt	SMALL_INT	
An integer value created between 0 and 10		

of new IP addresses based on the original, each given a certain priority. These new addresses are selected based on the octets present in the original, modified by the associated nodes in the tree. Four such primitive functions exist: using the corresponding value from the input, using the full range of 256 values, generating an interval of a given radius around the input value, and using a constant value initially generated by normal distribution around 0 or 255, representing the intuition that certain special addresses are often located at very low or very high addresses. On execution of the genetic programming tree during the course of the agent algorithm, this will add all IP addresses for which the selected pattern holds to the queue. For example, a generation pattern of $[inputValue, inputValue, interval(1), anyValue]$ with an input of 10.1.1.15 will add all addresses in the range 10.1.[0 − 2].[0 − 255]. The expansion strategy consists of several of these genotypes, each assigned a priority. A key principle of this design is that any possible tree should represent an at least minimally effective strategy, so that genetic operators can not render a strategy nonfunctional and the EA is able to easily find an initial gradient. The full set of genetic programming primitives are listed in Table 1. Example trees are shown and discussed in Section 6.4.

4.2 Infrastructure

4.2.1 Virtual Environment. For each evaluation, the attacker agent operates within a completely virtual network topology. This topology consists of virtual machines acting as user nodes, firewalls, routers, and various other services as well as virtual bridges that allow high fidelity network switching between the nodes. Real network fidelity ensures that the evolved agent in a virtual topology stays applicable in real networks. More importantly, a virtual topology allows us control over every aspect of an evaluation providing us the ability to limit the network flow within the topology or collect diagnostic data from the nodes while the evaluation runs.

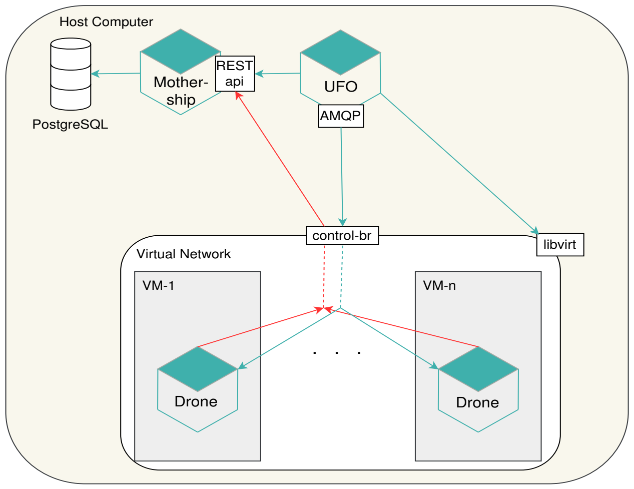

Figure 1: Design of the overarching infrastructure

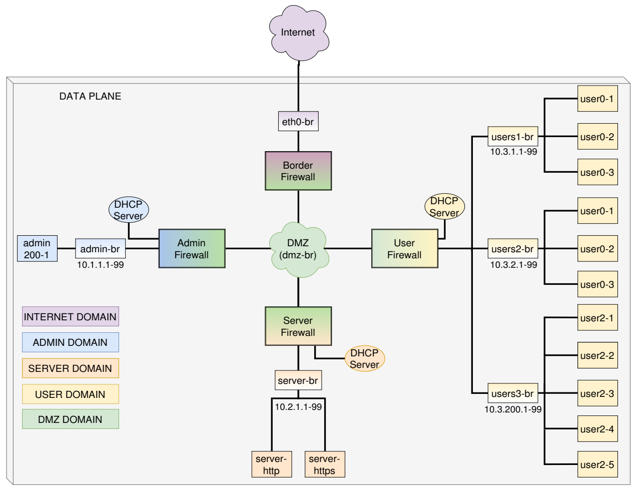

Figure 2: Network Topology used for All Experiments

4.2.2 Infrastructure Design. Figure 1 shows a high-level graphical depiction of the infrastructure's three core microservices.

The first layer is comprised of the host computer. Every service in this layer runs directly on the host computer's hardware. This layer primarily manages the running and coordination of evaluations by hosting the main administrative components of the infrastructure. *Mothership* hosts all network enumeration results and metadata the infrastructure and the evolutionary algorithm require to orchestrate each evaluation and generate new genetic programming trees, respectively. Moreover, *Mothership* is responsible for queuing evaluations to be run on the virtual network. The *UFO* runs the evaluations by interacting with the virtual machines using a virtualization API, *libvirt* [18], and communicates with the attacker agent on the virtual machines. For each evaluation, a *UFO* loads the genetic programming tree onto the node the attacker is on, starts the attacker agent, and resets the virtual machine after 60 seconds.

The second layer contains the virtual network. This layer encapsulates the network topology and everything which runs within a virtual machine. The control bridge (control-br) is an important virtual bridge on the host machine which allows point-to-point (or exclusive) communication between the host and each virtual machine in the virtual network. Results and metadata communication must use this communication channel to ensure agents stay isolated in the emulated environment.

Each virtual machine runs the *Drone* service. *Drone* provides metrics about the virtual machine's network connectivity which will be used in future experiments.

4.3 Network Layout

4.3.1 Building the Network. The underlying idea behind the network building process is to ensure statelessness between evaluations and experiments. Any action the attacker agent performs on the network alters the state of the network that the action affects with regards to system logs or system memory. In future work, a defender could utilize this change in state if not reset to more quickly identify an attacker than would otherwise possible, skewing subsequent evaluations. To mitigate these problems, the network is build using the following procedure:

(1) The network building tool reads from a configuration directory which describes the network topology.
(2) The builder feeds these configurations into *vmbetter*, a tool designed to work in Sandia National Laboratories' *minimega* [5] project, which builds a custom virtual machine images.
(3) These virtual machine images are registered with the virtualization library *libvirt* [18] and connected using virtual bridges specified in the configuration.
(4) A snapshot is created which can be later reverted to for a fresh machine state.

4.3.2 Network Design. The network design simulates a basic small scale enterprise network with three main enclaves.

The administrative enclave contains the machine a network administrator would use to access the network. Currently, there is no difference between an administrative machine and a regular user machine, but future experiments will require this administrative machine to access certain services. A website administrator agent runs on this machine and generates SSH traffic to the two servers located in the servers enclave.

The servers' enclave hosts internal services an enterprise needs to run. In this specific instance, two servers are running HTTP and HTTPS servers, respectively. This domain provides a target for generated traffic which increases the fidelity of the network by providing the "white noise" of normal network users.

Lastly, the users enclave encapsulates all of the users subnets located in the enterprise. These user nodes would include things like guests accessing internal services, employees working on the network, and other normal network usages. On each of these nodes, a web crawler agent generates traffic to the HTTP and HTTPS servers to further increase the "white noise" on the network.

The core router infrastructure in our network is static, mimicking an enterprise environment. All non-routing nodes are dynamically served network information through internal DHCP servers sitting on each domain's firewall node. At the beginning of an evaluation each node is given a new IP address to prevent agents from learning static addresses.

We have intentionally started our attacker agent in the administrative domain allowing the agent to see the traffic from the website administrator but not the user agents. Our attacker must then evolve to detect these unknown nodes. For the purposes of this experiment, being located on the admin node provides no special advantage to the attacker beyond seeing its traffic.

4.4 API

Our Application Programming Interface (API) abstracts low-level implementation into high-level easy-to-use actions for attacker, defender, user agents, and their programmers. Agents require the ability to manipulate network structures, gather network and host forensics, review logs, among other sensory actions in order to execute high-level strategies and objectives over the course of both development and evolution. The main features currently implemented leverage Libnmap [17] and Pyshark [9] for interacting with raw data from hosts.

5 EXPERIMENTS

5.1 Experimental Procedure

To begin an experiment, we build the virtual network on specified host computers using *Ansible* [6]. Upon completion, the evolutionary algorithm can then submit sets of evaluations to the infrastructure to be run. The infrastructure takes each genetic programming tree and runs the attacker agent with that tree on an available virtual network, executing in parallel with evaluations on other virtual networks. The evaluation runs for sixty seconds, reports the results to the infrastructure, and then the network is reset for the next evaluation.

The attacker strategy is assigned a fitness equal to the number of discovered IP addresses during evaluation. After all evaluations have been run for the current generation, the EA collects the results from the infrastructure, generates a new generation of individuals, and again submits them to be run. This procedure repeats until termination of the EA.

5.2 Parameters

One of the key difficulties with evolution in an emulated environment is combating the time constraints inherent to agents executing commands in real-time. Additionally, the overhead of restoring the network state between each evaluation further lengthens experiment times. While work on simulated computer networks, such as that by Rush et al. [20], can run nearly 100,000 evaluations per experiment, a single evaluation on our emulated network takes multiple minutes, meaning that the run-time of only 1,000 evaluations can exceed 50 hours, excluding parallelization. Future reductions in infrastructure overhead can improve this somewhat, but there remains the necessary requirement of running agents in real-time in order to maintain fidelity of the experiments. This time limitation is fundamental to the use of a real network and limits the performance of the EA: it would be desirable to use larger populations or to run multiple evaluations per attacker to reduce noise, but these things would come at an extreme time cost, so evolution can only be given fairly limited resources. Table 2 lists the parameters chosen for these experiments.

Table 2: Evolutionary parameters

Generation limit	50
Selection population (μ)	18
Child population (λ)	18
Total evaluations	900
Mutation fraction	25%
Recombination fraction	75%
Initial tree height	3-7
Parallel evaluations	6

Fitness proportionate selection is used to select individuals for parent selection, primarily due to certain properties of the fitness function used. Since evaluations take place in an emulated environment, a small amount of noise can result in individual evaluations having unusually high or low fitness scores; fitness proportionate selection reduces the bias of this. For survivor selection, truncation selection was chosen, since a level of elitism is needed to maintain the relatively small integral gains late in evolution of occasional new strategies which find only one or two more IP addresses than their rivals.

Genetic programming trees are generated using a variant of the grow method from strongly-typed genetic programming [15]. This variant precomputes the possible heights that can be reached for a subtree with a given data type as the root, and then makes sure to select primitives such that at least one subtree reaches the desired height. The other subtrees are generated as normal through the grow method. No parsimony pressure was needed to control the growth of the trees during evolution, because larger trees tend to add more addresses to the queue for each input, and as such an excessively large tree is usually less efficient in search than a moderately-sized one.

5.3 Alternative Optimization Strategies

In order to gauge the effectiveness of the EA as a method of generating attacker agent strategies and justify its use for this problem, two simpler optimization strategies were implemented: a random generator, and a first-choice hill climber. These were each run for the same number of evaluations that the EA was given. The random agent generator simply generates agents at random in the same way that the EA generates its initial population, and returns the best generated strategy. The first-choice hill climber stores its best strategy, and successively generates children by mutation of the best strategy, keeping the one with the highest fitness between the stored strategy and the children. Due to the availability of parallel evaluation, several children are generated in parallel groups, rather than each in sequence. This includes the initial genotype, which selects the best of several randomly generated trees.

6 RESULTS

6.1 Evolutionary Results

Five runs of the EA were recorded. Figure 3 shows a graph of best and average fitness over time for these runs. Evolution was usually

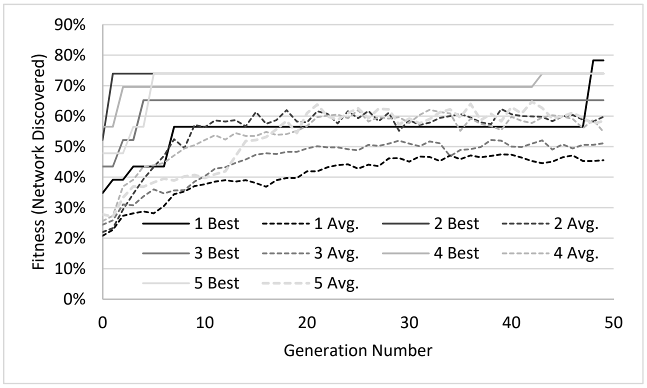

Figure 3: Average and best fitness over time for the EA, across each of five runs

rapidly able to find IP expansion strategies which could discover 74% of the network, which corresponds to the entire network excepting the 10.3.200.0/24 subnet, which it tended to have difficulty finding due to the 200 octet's distance from any others. Only one of the five experimental runs developed a strategy able to discover that subnet. Of particular note is that all experiments appeared to suffer from premature convergence, often even before reaching that 74%, indicating that they were frequently getting trapped in local optima. This is explored further in Section 6.4.

6.2 Random Generation

900 randomly-generated strategies were analyzed to determine the effectiveness of evolution compared to a trivial algorithm. The average fitness of a random strategy was 20%, far less than the average fitness of strategies generated during evolution, which reached 55% in later generations. This demonstrates that evolutionary operators were much more effective at generating new strategies than random chance. A histogram of the generated strategies' fitnesses is provided in Figure 4. Only 8 of the 900 random strategies were able to find more than 50% of the network, and just 1 out of 900 reached a fitness of 74%, even though such strategies were often discovered very early on in evolution.

6.3 Hill Climber

Three runs of a hill climber were recorded, in order to better analyze the fitness landscape. The results of the EA suggested that there were problems with premature convergence to local optima. As hill climbers do nothing but converge to a local optimum, they are well-suited to examining these issues further. The best and average fitness values over time for the hill climber are shown in Figure 5 and a comparison with the EA and with random search is shown in Figure 6. While the EA produced higher-fitness individuals on average than the hill climber as a result of recombination, both were nearly identical on average in terms of best fitness over time. In two of the three runs, the hill climber was in fact able to to reach similar maximum performance to the EA, reaching local optima with 74% fitness early on. This provides further evidence that future research should include a focus on escaping local optima, which is discussed in Section 6.4. However, the hill climber used was still

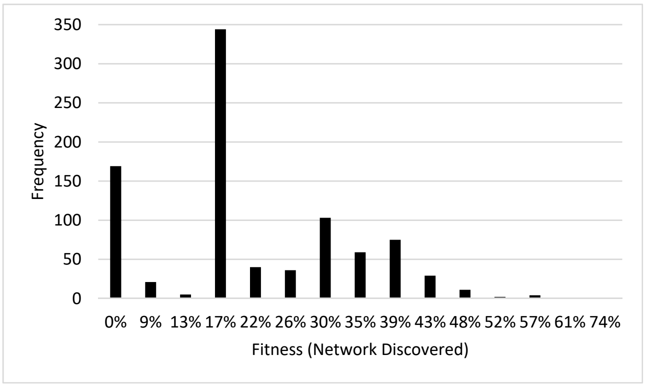

Figure 4: Fitness distribution of 900 strategies produced through random generation

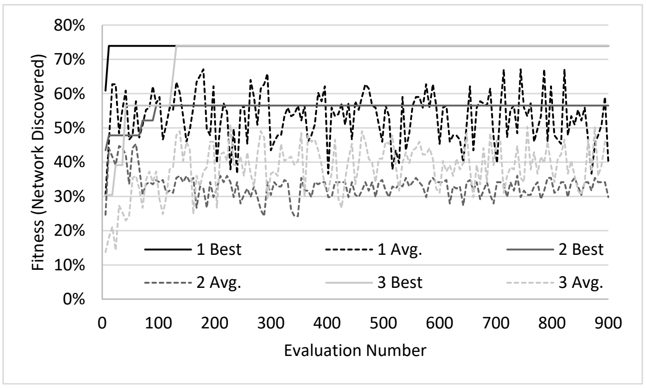

Figure 5: Average and best fitness over time for the hill climber, across each of three runs

working with the same genetic programming trees, under the mutation operator, so its unexpectedly high performance still supports the effectiveness of this work's selection of genetic programming primitives and agent design. The hill climber relies on the smooth gradient produced deliberately by the agent structure in order to function as well as it does.

6.4 Individual Strategies

The most successful individuals tended to develop similar strategies. Given an input IP address consisting of four octets, high-performing strategies mostly responded by searching intervals on the second and third octets in order to locate new subnets near already discovered ones, as well as searching the full set of possible values on the fourth octet in order to find the addresses randomly distributed by DHCP in that subnet. An example of such a strategy is shown in Figure 7.

This strategy first searches for subnets near known ones, with the same fourth octet value as the known IP. When those have all been searched, it then searches the entire [0–255] range for the fourth octet on each subnet it is located. Finally, if nothing else is found, it looks for any possible IP addresses with a 3 in the second octet, which allows it to locate addresses on the user

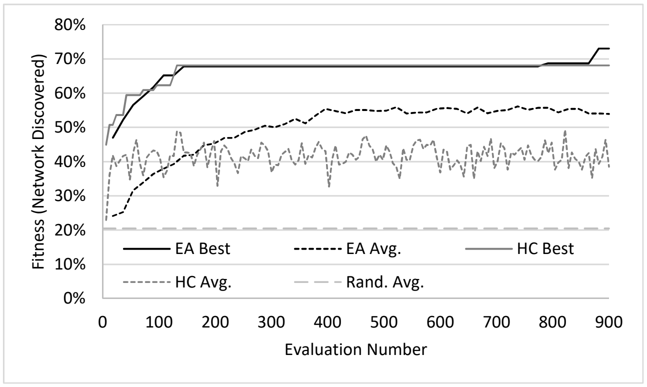

Figure 6: Comparison of average and best fitnesses over time between the EA, hill climber, and random generation, averaged over all runs.

For input IP a.b.c.d:

- Queue a.[b±8].[c±1].d (Priority 2)
- Queue a.b.c.[0–255] (Priority 3)
- Queue a.3.[0–255].[0–255] (Priority 5)

Figure 7: Example strategy with fitness of 74%

subnets through brute force. Upon finding one, it will have new higher-priority values to search again.

The hill climber was able to find similar solutions fairly quickly, indicating that such a solution can be found by evolution using only small, incremental improvements. However, this strategy leaves no clear incremental improvement that would allow it to discover the 10.3.200.0/24 subnet, which is located much further away from the others than the use of interval expansions would be able to cover. A strategy that can discover the 10.3.200.0/24 subnet would instead need to be searching for shared octet 4 addresses across different subnets on octet 3, and might particularly exploit the fact that the user subnets all share a 10.3.*.254 address. The immediately appealing success of the former strategy however makes it difficult for the EA to explore different approaches due to a lack of factors promoting population diversity.

Low population diversity can be caused by excessive selection pressure or the properties of the fitness landscape, but also by issues such as the low population being used in this work due to the time constraints of evaluation. This indicates that a first step in improving the performance of the EA should be analyzing metrics of diversity in the population over time, such as those presented in Burke et. al [3]. Insufficient diversity in the population need not only be improved through adjusting parameters to encourage it, however. It is not uncommon to directly promote diversity through various methods, such as including it as a second objective function in a multiobjective framework. A particular mechanism of measuring diversity for genetic programming trees is given by Burks and Punch [4]. By increasing diversity in the population, the EA can be made to hold on to more novel genotypes even if they are not as immediately lucrative to exploit, which would help with the problems seen here.

7 CONCLUSION

Despite the remaining difficulties in need of further study, this work serves as a proof of concept of the idea that effective strategies for a cyber-attacker agent can be evolved on an emulated computer network running in real-time. Unlike in a simulation, these emulated agents are interacting with all the complexities of a real network, and have to deal with problems such as unreliable network scanning functions and benefits such as overhearing network control packets, all of which might not have been implemented in a simulation, and all of which allow evolved agents to be that much better adapted to the minutia of functioning in the real world. While the task being attempted by the agents is simple on its own, it is one of many parts of a much larger problem, all of which can be explored in an emulated network using similar principles in the future.

8 FUTURE WORK

8.1 Coevolution

A key component in replicating a realistic security scenario is the presence of a defender who controls the computer network and is attempting to keep the network secure against the attackers. The defender agent should be able to take actions in the interests of security both in preparation for future attacks, such as configuring intrusion detection software and modifying the network topology, and during the attack, such as shutting down computers or network connections. All of these sorts of actions can have a significant cost to the defender, in terms of cost and effort to implement, but also in terms of impact to the user. A network can be made perfectly secure by blocking all traffic, but this also makes it useless. The defender, then, must learn to make a reasonable tradeoff between security and usability.

Allowing competitive coevolution between the attacker and defender agents, such that the two populations are evolved together with combined fitness evaluations, provides for an interesting set of possibilities. Such an arrangement would ideally result in an “arms race” effect in which the attackers would evolve to better exploit new vulnerabilities in the network, while the defenders would need to evolve to defend against those new strategies. Effective coevolution without cycling ensures that population members maintain effectiveness against opponents from past generations. As a result, attackers and defenders evolved in this way should be effective against most attack or defense strategies that developed during evolution. One could imagine using a replica of an existing computer network as a baseline, and evolving a set of modifications and security strategies tailored directly to that network able to stand up reasonably against everything the attacker was able to develop.

8.2 Benign-Human Mimicking Agents

An additional part of the network ecosystem is important for replicating real network behavior is the users, who provide a variety of benefits to the attacker. Attackers can gain information about the structure of the network by listening to user communications with network services. User activity also makes the state of the network more dynamic: for example, users logging into machines might deposit additional credentials into memory that an attacker can use. Finally, users provide cover for the attacker: attackers often rely on the complexity of real users to blend in, making detection more complicated. Therefore, the inclusion of realistic user agents will force both attackers and defenders to learn these real-world skills.

Previous research has been mostly limited to replays of user network traffic and creating agents by hand. Both methods are insufficient, since agents created from replays are unresponsive to changes in the network, and creating agents by hand can be time-consuming and lead to overspecialization. Research is underway to create a custom hyper-heuristic employing genetic programming that evolves benign-human mimicking software agents for cyber-security emulations. To analyze network traffic, netflows are captured and then the entropies of the bidirectional netflows are calculated over a sliding window of time. The resulting time series of entropies are then used in supervised learning, in the form of clustering algorithms and learning classifiers, to distinguish between realistic and artificially generated traffic.

8.3 Expanded Tasks

With network enumeration serving as the first step in a much larger problem space, our future agents will need to combine future problems with our current work to represent broader and more complex tasks. Initially, network enumeration should be expanded to allow agents to pivot onto nodes with escalated permissions and different network traffic, giving them more tools to reach the entirety of a network.

Beyond network enumeration, the next step is to introduce exploitation tasks, including stealing credentials and locating high-value targets to exfiltrate data from. Exploitation also introduces further opportunities for a defender agent to detect attacker actions. With these features implemented, the attacker agents will require strategies spanning the full breadth of an attack scenario.

Time is a significant obstacle in the emulation of these tasks. In addition to the necessity of executing these actions in real-time, a common tactic for real-world attackers is positioning one's self on a network, and then waiting for a long time for the right conditions. It is obviously impractical to spend weeks on a single evaluation, so it will become necessary to find a method of skipping the networks forward in time while minimizing loss of fidelity.

8.4 Evaluation Time

The overhead of building and resetting the test network along with the requirement to run agents in real-time result in an extremely long evaluation time, causing many present and future difficulties for this work. Future expansions to the network will increase the former, and increased complexity of the agents' goals will increase the latter. Further, the introduction of coevolution will multiply the number of fitness evaluations needed. Therefore, reducing the amount of time spent per evaluation is critical to expanding this work beyond a proof of concept. One way of achieving this is to simply skip a significant portion of the evaluations altogether. This can be done through methods like fitness approximation, which can estimate fitness based on that of other individuals. Work such as that by Gustafson and Vanneschi [10], and by Burke et. al [3] suggest a genotype similarity measure, while Esparcia-Alcázar and Moravec [7] suggest measuring phenotype similarity. Further, while the aim of this work focuses on emulated networks, simulation of agent strategies beforehand has the potential to detect obviously nonfunctional individuals and terminate their evaluation early.

ACKNOWLEDGEMENTS

This work was supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 259565.

REFERENCES

[1] Elias Bou-Harb, Mourad Debbabi, and Chadi Assi. Cyber Scanning: A Comprehensive Survey. *IEEE Communications Surveys Tutorials*, 16(3):1496–1519, Third 2014.

[2] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, Feb 2001.

[3] Edmund K. Burke, Steven Gustafson, and Graham Kendall. Diversity in Genetic Programming: An Analysis of Measures and Correlation with Fitness. *Trans. Evol. Comp*, 8(1):47–62, February 2004.

[4] Armand R. Burks and William F. Punch. An Efficient Structural Diversity Technique for Genetic Programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 991–998, New York, NY, USA, 2015. ACM.

[5] Jonathan Crussell, Jeremy Erickson, David Fritz, and John Floren. minimega v. 3.0, version 00. https://www.osti.gov//servlets/purl/1312788/, 2015.

[6] Michael DeHaan. Ansible. https://www.ansible.com/, 2012.

[7] Anna I. Esparcia-Alcázar and Jaroslav Moravec. Fitness approximation for bot evolution in genetic programming. *Soft Computing*, 17(8):1479–1487, Aug 2013.

[8] Dennis Garcia, Anthony Erb Lugo, Erik Hemberg, and Una-May O'Reilly. Investigating Coevolutionary Archive Based Genetic Algorithms on Cyber Defense Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1455–1462, New York, NY, USA, 2017. ACM.

[9] Dor Green. Pyshark. http://kiminewt.github.io/pyshark/, 12 2013.

[10] Steven Gustafson and Leonardo Vanneschi. Crossover-based tree distance in genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):506–524, Aug 2008.

[11] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, Detroit, MI, USA, 20-25 August 1989. Morgan Kaufmann.

[12] Gordon Lyon. Nmap. https://nmap.org/, 1997.

[13] MarketsandMarkets. Penetration testing market worth 1,724.3 million usd by 2021. https://www.marketsandmarkets.com/PressReleases/penetration-testing.asp, 2016.

[14] Julian F. Miller and Peter Thomson. Cartesian Genetic Programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming*, pages 121–132, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[15] David J. Montana. Strongly Typed Genetic Programming. *Evol. Comput.*, 3(2):199–230, June 1995.

[16] Kinga Mrugala, Nilufer Tuptuk, and Stephen Hailes. Evolving attackers against wireless sensor networks using genetic programming. *IET Wireless Sensor Systems*, 7(4):113–122, 2017.

[17] Savon Noir. Libnmap. https://libnmap.readthedocs.io/, 5 2013.

[18] Red Hat, Inc. Libvirt virtualization api. https://libvirt.org/, 2005.

[19] George Rush. Cyber security research frameworks for coevolutionary network defense. Master's thesis, Missouri University of Science and Technology, Missouri S&T, Rolla, MO 65409, Fall 2015.

[20] George Rush, Daniel R. Tauritz, and Alexander D. Kent. Coevolutionary Agent-based Network Defense Lightweight Event System (CANDLES). In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 859–866, New York, NY, USA, 2015. ACM.

[21] Xiaokui Shu, Ke Tian, Andrew Ciambrone, and Danfeng Yao. Breaking the Target: An Analysis of Target Data Breach and Lessons Learned. *CoRR*, abs/1701.04940, 2017.

[22] United States Computer Emergency Readiness Team. Russian Government Cyber Activity Targeting Energy and Other Critical Infrastructure Sectors. https://www.us-cert.gov/ncas/alerts/TA18-074A/, March 2018.