Galaxy: A Network Emulation Framework for Cybersecurity

Kevin Schoonover[1†], Eric Michalak[2†], Sean Harris[1], Adam Gausmann[1], Hannah Reinbolt[1], Daniel R. Tauritz[1], Chris Rawlings[2], Aaron Scott Pope[1]

[1]Missouri University of Science and Technology:
{*ksyh3, snhcn6, ajgq56, hmrvg9, aaron.pope*}*@mst.edu, dtauritz@acm.org*
[2]Los Alamos National Laboratory: {*emichalak, crawlings*}*@lanl.gov*
[†]Equal Contributions

Abstract

The arms race of cyber warfare is growing increasingly asymmetric as defensive security practitioners struggle to successfully harden their domains without overly restricting their users, profits, and overall mission. Vulnerabilities span across technologies, business policies, and human behaviors, allowing cyber attackers to select the attack surface that best fits their strengths. This paper introduces the first version of Galaxy, a fine-control, high-fidelity computer network emulation framework designed to support rapid, parallel experimentation with the automated design of software agents in mind. Our framework provides a modular environment to experiment with arbitrary defense and attack strategies under a wide variety of business requirements and accounting for the productivity of users, allowing cybersecurity practitioners to consider the unique constraints of their real-world systems. We demonstrate the effectiveness of Galaxy for the use of an evolutionary algorithm to generate enumeration strategies for attacker agents.

1 Introduction

In this paper we introduce Galaxy, a computer network testing framework for agents to identify attacker and defender strategies on arbitrary emulated network topologies. Previously, the Coevolutionary Agent-based Network Defense Lightweight Event System (CANDLES) project [9] used simulated environments for this but did not account for variables introduced with emulation. The Coevolving Attackers and Defenders Strategies for Large Infrastructure Networks (CEADS-LIN) project, further discussed in Section 4, continued the ideas of CANDLES into emulation, but initially lacked an adequate emulation framework. Galaxy allows researchers to iteratively test security strategies on an emulated copy of their network that exists outside of their production environment. Currently, the most common network security testing available is penetration testing which is usually scope-limited due to the uptime and cost of production systems. Additionally, destructive or potentially risky attack methods such as denial of service attacks, rootkits, remote code execution, and new experimental scripts are often off-limits due to these real world constraints. Such scope limitations prevent comprehensive testing and leave unexplored attack surfaces to be speculated on in final reports where their true severity can be misrepresented.

Other common tests like code auditing and service assessments focus too narrowly, lack operational context, and assume no new vulnerabilities will be introduced in deployment or during communication. Computer service inputs and outputs (network or otherwise) can be exploited or provide useful context to attackers with access.

Any large network is inherently complex and organically changes with each system update, software deployment, employee new-hire, and change in business policy. Documentation and reality drift apart and the humans who are in charge of these systems naturally make mistakes as managing all these ingress paths becomes temporally and fiscally impossible. However, with automated testing, these attack surfaces can be more frequently explored to discover potential problems. We leverage virtualization to copy a network and perform these tests outside of production in a high-fidelity but low cost manner.

Galaxy was created with the following goals:

1. Develop a high-fidelity network clone from a configuration file, which can perform as close as possible to the actual specified network.
2. Allow a multitude of experiments to be performed on this clone in stateless succession for clean, subsequent experiments.
3. Distribute experiments across multiple physical machines to run in parallel, improving performance and runtime.

4. Provide API resources to evolve intelligent agents within the network over iterated experiments.
5. Aggregate data from the virtual network for analysis and visualization.

The remainder of this paper provides an overview of related network emulation frameworks, their limitations and how we overcame those in Galaxy, an experimental case study demonstrating the effectiveness of our approach, and the lessons we learned from it along with a blueprint for future iterations of our framework.

2 Related Work

There exist several generic frameworks for network emulation such as Emulab [6] and DeterLab [8]. However, while both work well for building and allocating test networks, we encountered limitations making them unsuitable for projects such as CEADS-LIN. In Emulab, we had to filter out all of our evaluation-facilitating control protocols because they tainted the experiment's overall traffic that agents scan and utilize to make strategic decisions. Galaxy has a similar problem with the control bridge discussed in Section 3, but we plan to remove it as a dependency in future versions. Deterlab, a cyber security focused framework, isolated their emulated networks, but did not give us a clean solution to implement or filter out our control protocols. Additionally, Deterlab's dynamic routing, which creates subnets, uses virtual local area network (VLAN) tagging instead of physically separating nodes into LANs. Through dynamic routing, we had less control over the allocated router nodes than other nodes in the network. Deterlab did provide direct link networking which emulates a physical connection; however, direct link networking was physically restricted to six connections per node – two of which are employed internally by Deterlab's infrastructure – which limited the topology and size of networks we could emulate to small and unrealistic enterprise networks. Finally, we were unable to establish our own VLANs inside our allocated networks, a feature we plan on using in future work.

During network creation, Emulab and Deterlab need to image all nodes with their specified OS, link them using internal routers, and expose the network to the researcher. In our experiments, the build time was roughly 30 minutes for a 20 node network. Software agents such as evolutionary algorithms require statelessness between the potential thousands of evaluations; however, the setup cost made these types of experiments infeasible and continually provisioning networks in the background would greatly limit the node resources available to other users on the platform. Galaxy aims to provide improved methods to gather results, send commands into and out of the network, and provide stateless networks.

3 Design

Figure 1 shows a self-contained, high-level graphical depiction of Galaxy and its components; a distributed example is discussed in Section 3.3. Galaxy attempts to completely isolate the virtual network from the host system by operating in two logical layers: the host layer and the virtual layer. The host layer provisions the VMs and orchestrates an evaluation. The virtual layer contains an isolated, fully virtualized network where experiments are run through leveraging the KVM hypervisor. Results are then aggregated back to the host layer through Galaxy to be analyzed.

3.1 Building Topologies

Arbitrary network topologies are specified in a hierarchy of configuration files, where each configuration file defines a VM, or a node. Galaxy parses this hierarchy to build the network on the host computer. The network configuration contains a subdirectory for each node to be built in the network. The name of a subdirectory defines the name of the VM. Each subdirectory contains the following: a yaml file specifies the host's virtual bridge configurations; a text file enumerates the packages, the OS for the node[1], and a post-build bash script that further customizes the node during build; and an overlay directory which optionally contains a partial Linux file system to be copied onto the built node.

Prior to Galaxy automatically building the network, the initial setup of the virtual interfaces and virtual bridges needs to be manually configured on the host computer. We address this issue through *bridge-utils* in the next iteration of Galaxy.

Galaxy builds the experimental network in the following fashion:

1. The network builder parses the configuration directory, ingesting all node configurations.
2. The appropriate config file is passed to *vmbetter*, designed by Sandia National Laboratories for their minimega project [1], which builds a VM disk image with the specified packages and copies the overlay's contents onto the root of the disk image allowing the researcher to overwrite Linux system files. It finalizes the build by running the post-build bash script in a chroot environment within the VM disk image, allowing arbitrary researcher configuration.
3. Each VM is defined in *libvirt* as an XML file to leverage *libvirt's* API; Galaxy can then perform operations on the VM such as start, stop, snapshot, and restart. The XML specifies all aspects of the

[1]The host's OS is determined by *vmbetter*; only Debian-based systems are supported.

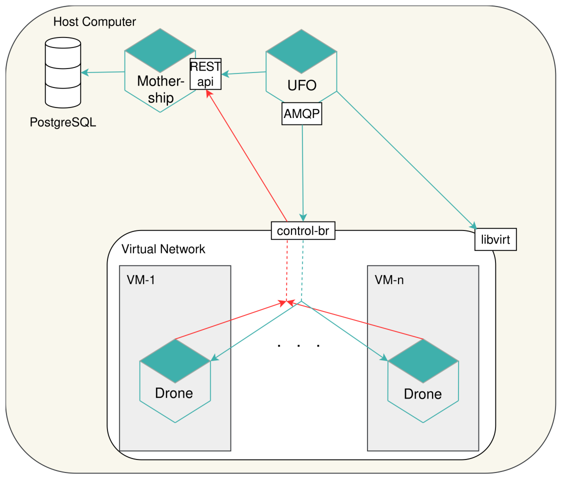

Figure 1: Galaxy Design

VM such as the filesystem, network interfaces, and hypervisor features to apply. Due to the complexity of *libvirt's* XML domain specification, we built template XML files for each unique interface instead of in-code XML generation, which would require intimate knowledge of *libvirt's* XML format. The generated XML templates contain an optional *libvirt*-defined machine type specific to each operating system that helps KVM take advantage of the operating system's full-virtualization features.

4. A snapshot is taken to store the VM's initial state. This preserves a clean state of the node to ensure consistency and fidelity across evaluations. Because we were unable to programmatically snapshot a perfectly initialized node, there is an extra overhead cost for starting from a shutdown state. We were concerned that if a snapshot was not taken *perfectly* before all processes had started, potential memory states could be inconsistent between evaluations[2].

We decided to integrate *Ansible*[3] into Galaxy to leverage playbooks to distributedly build networks across multiple host computers in parallel. The current implementation restricts each host computer to one network and thus one evaluation at a time. This is a result of virtual bridges and VMs being assigned statically. In order to arbitrarily modify a network during an experiment, Galaxy would need to dynamically build and link bridges to virtual subnets of lightweight custom-function VMs.

Dynamically building bridges requires a dictionary of all active bridges, their associated virtual network interfaces, and which virtual subnet they control. There are many edge cases to account for in reconstructing the network. For example, adding a new physically separate subnet off of a router which has no available physical interfaces requires rebuilding the router VMs and subsequently relinking all associated subnets and nodes. In the worst case, this feature could potentially be abused by agents to "run out the clock" of evaluations, reducing the fidelity of the experiment and increasing average evaluation time. To facilitate efficient tracking of topological changes during an evaluation, we considered the solution of preallocating extra virtual bridges on the host and extra physical interfaces on routers, alongside a sophisticated graph-based representation of the network topology. However, this was postponed to a future iteration due to its inherent complexity.

Lightweight custom-function VMs that only have the desired packages and features and not a superset of all common features are important to our experiment fidelity, as superfluous packages could be leveraged by malicious agents. Unfortunately, custom building each unique VM incurs a significant build cost of 9 minutes for each VM, which quadruples evaluation times from 2 minutes when we create a new node. A caching solution such as *OverlayFS* for VM images would provide necessary modularity to perform these changes quickly.

3.2 Components

This section discusses the three core microservices that operate between the host computer and the VMs: Mothership, UFO, and Drone.

3.2.1 Mothership

Mothership provides access to a *PostgreSQL* database that stores experiment metadata such as results, evaluations, and nodes through a *Flask*[4] REST API interface. The REST API enforces a single access point for data storage that prevents concurrent resource updates from multiple distributed networks. Through Mothership, a researcher or agent can interact with an endpoint, schedule an evaluation to be run, or query the scheduler for the next evaluation to run.

Mothership encapsulates objects, called models or resources, and the data associated with those objects into tables. Mothership's current resources can be interacted with through the evaluation endpoint, the queue endpoint, and the results endpoint. An evaluation resource contains all evaluation metadata including its current status, any arbitrary researcher-created data file that will be put onto specified nodes, and evaluation start and end times. The evaluation endpoint allows researchers to register evaluations to be run on the framework. Evaluations are added to a queue in the Mothership database; a UFO

[2]This assumption is based on preliminary experiments and requires further validation.

[3]`https://www.ansible.com/`

[4]`http://flask.pocoo.org/`

process then polls the queue endpoint to receive the next evaluation. If a queued evaluation exists, Mothership will provide the agent with the required metadata for running the evaluation. The nodes endpoint stores a list of all registered nodes in an experiment. Mothership internally utilizes this resource to determine the origin of results. The results endpoint allows agents to report results during and after an evaluation. Different types of results can be easily implemented depending on the evaluation needs of the researcher. REST endpoints make these resources available with full create, read, update, and delete (CRUD) functionality.

Future work will support specifying topologies through the node endpoint instead of through the currently implemented configuration directories.

3.2.2 UFO

A UFO programmatically runs evaluations based on a provided evaluation time, e.g., how long the UFO will allow the network to run per evaluation. The default evaluation time is sixty seconds, to allow for thousands of evaluations to occur in a reasonable amount of time. The UFO can start, stop, and revert each VM; however, each action adds around 5 seconds – the time it takes for the node to power cycle back to fetching networked status – to the setup time for each node on the network.

The UFO communicates with agents in the virtual network from the host computer through the control-br (control bridge) as shown in Figure 2 using asynchronous message queueing protocol (AMQP)[5]. AMQP was chosen to reduce the development time of developing a custom, reliable network messaging system for agents. The control bridge is a specialized bridge connected to all VMs that allows for administrative commands, traffic, and other network management overhead to pass transparently from each node to the host computer directly. A point-to-point control bridge isolates administrative traffic from virtual experiment traffic, helping prevent network pollution and increasing experiment fidelity. Drones, software agents, and the UFO subscribe to a *RabbitMQ*[6] message broker where the UFO can publish messages to exchanges. Drones and software agents consume messages containing commands in real time from the UFO.

In order to determine which evaluation to run, the UFO polls the Mothership queue endpoint every five seconds to collect the newest evaluation metadata before running the following procedure:

1. Query the Mothership node endpoint for the list of all nodes in the current evaluation topology.
2. Revert all collected VMs to their initial snapshot.
3. Start all reverted VMs.
4. Send the START command signaling that all machines have successfully booted and that the experiment can formally begin. The START command synchronizes agent start times, preventing false results due to an uninitialized network.
5. Send the UPDATE command every five seconds, informing any agent or drone within the network to send its latest update to Mothership.
6. Send the STOP command after the experiment window has ended. Any agent or drone must report its final results and gracefully shutdown in preparation for the next evaluation.

[5]`http://www.amqp.org/`

[6]`https://www.rabbitmq.com/`

3.2.3 Drone

Each node contains a Drone daemon which collects data and metrics to report to Mothership. During the building topology phase, the Drone is automatically included in each VM's configuration file along with a postbuild command that automatically starts the Drone on launch. Currently, the Drone only collects connectivity results but in future iterations will collect richer network and node states. After receiving an update command from the UFO, the Drone pings a specified node, usually a core routing node, on the network as a basic connectivity test. If the ping is successful, the Drone reports a success to Mothership. Results signal if the node properly interacted with the virtual network during an evaluation. Missing one or more connectivity results informs the researcher that a fatal (intentional or otherwise) network error occurred during the evaluation. These signals can then be used to troubleshoot whether an evaluation was invalid and should be repeated. The Drone inherently reduces the network fidelity by inflating network traffic, but in turn provides the researcher with fault tolerance to prevent faulty evaluations.

3.3 Distribution

Figure 2 shows an example of how Galaxy can be distributed across multiple host computers increasing the number of concurrent evaluations. The design employs a master/worker architecture where the host computer is the *master* and all other machines are *workers*. Mothership only needs to be located on the master and will automatically handle the scheduling and registration of evaluations for all connected workers. Mothership's scheduler ensures only unprocessed evaluations are assigned to UFOs when they poll. Each worker requires a proxy to forward traffic such as UFO pulling requests and results to Mothership. The proxy preserves the internal virtualized network IP address that allows Mothership to resolve which node sent the results across machines.

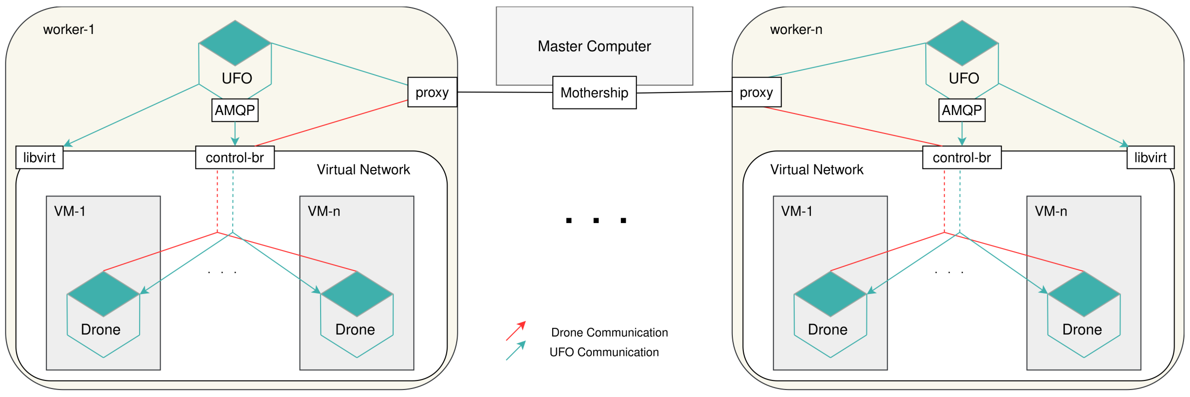

Figure 2: Distributed Galaxy Design

4 Application

CEADS-LIN is a research project that studies the use of evolutionary algorithms (EAs) to develop strategies for cyber attacker and defender agents in emulated computer networks. An initial step in those attacker strategies is network enumeration, the reconnaissance action of scanning a network to discover resources that are available, but are not being advertised, such as the IP addresses of machines on the network. In recent work, Galaxy was used to study methods of evolving strategies for attackers that can perform network enumeration more efficiently than using a brute force approach [4]. Attacker agents relied on the results of packet sniffing and network scanning software being executed on an emulated model of an enterprise network in order to learn the network's layout, and evolve specialized strategies for determining which address ranges to scan. We summarize this recent work to illustrate the effectiveness of Galaxy and the lessons we learned.

4.1 EA Background

Evolutionary algorithms are a type of stochastic optimization algorithm, inspired by the biological principle of natural selection. EAs have long been studied for their applicability as black-box optimization algorithms for a broad range of problems [2]. An EA stores a population of candidate solutions, and iteratively replaces members with new solutions based on previous high-performing solutions, said to have high "fitness". Such a fitness score might represent measures such as error, incurred cost, or objectives satisfied. Solutions are stored in data structures called genotypes, which are designed to enable modification by evolutionary operations. Mutation allows solutions to search and optimize locally, while recombination allows the components of multiple solutions to be merged into a new solution, allowing for exploration of the search space.

4.2 EA Constraints

The runtime of an EA is generally dominated by the cost of its fitness evaluation; this is particularly the case in emulations which need to run in real-time. Each solution must be tested against the problem it is meant to solve, and the number of solutions in need of evaluation typically ranges in the thousands.

The EA's ability to find high-performing solutions is dependent on the accuracy of the fitness landscape, which for this work is in turn dependent on the accuracy of the underlying network emulation. The application of EAs to network security problems, then, has strong bearing on the design of any network emulation framework intended to accommodate them.

Most obviously, the time that it takes to perform an evaluation for an EA must be as low as possible. As the process of starting and resetting an emulated network runs the risk of incurring a heavy overhead in time, the emulation framework should be optimized for doing this in rapid succession. Fortunately, EAs inherently allow parallel computation, which mitigates the expensive evaluations, at the cost of requiring parallel hardware. Thus, in order for a network emulation framework to best support the running of EAs, it must additionally be capable of distributing evaluations across many machines.

EAs are also infamously capable of exploiting unintentional irregularities in evaluation functions [7]. While some applications might only be weakly affected by a slightly inaccurate behavior on an emulated network, if the EA finds that it can gain fitness by exploiting this behavior, it will rapidly do so and greatly amplify the effect of that irregularity on the experiment. Therefore fidelity is of high importance.

4.3 Network Topology

Figure 3 shows the network topology the CEADS-LIN project utilizes for all evaluations that is built and maintained by Galaxy. This network consists of 18 VMs running various services and 6 virtual bridges that provide layer-2 switching between these nodes. On this network there are three main node domains, which helps our network model a business network with high fidelity.

The admin domain contains all of the machines that are used to perform administrative tasks on the network. This includes actions such as system administrators accessing machines to update, upgrade, or repair services and other administrator privileged tasks. Currently, the single admin node has no special privileges or access to the network and acts equivalently to a user node in a different subnet. However, an admin agent runs on this node and generates SSH traffic to the two web servers on the server domain.

Secondly, the server domain represents all internal services a network would need to support its employees. In this specific topology, the two services are internal HTTP and HTTPS servers. These services mainly act as a destination for traffic generation for this specific experiment; however, future networks could include different types of critical services such as central authentication or high value servers.

Lastly, the user domain encapsulates all of the subnets containing users who would be operating in a real network. Within a standard network, these user nodes would include employees accessing internal services on the network, guest users temporarily allowed on the network, or other normal network users. On each node in the subnet, a user agent generates traffic to the internal HTTP and HTTPS servers located in the server domain.

While the breadth of services and the privileges of the admin node will be expanded in future work, this topology emulates a small business. This network showcases many of the main features of the Galaxy emulation framework. All the nodes are built using *Ansible*, making developing arbitrary network configurations as simple as modifying configuration files and directories. Moreover, during this building process, the HTTP server, HTTPS server, and the various agents automatically start on the VMs because of the build processes. Within the CEADS-LIN project, both the attacker and the EA utilize Mothership for reporting and retrieving results for evaluations, respectively. Lastly, all the evaluations are controlled by the UFO using the *libvirt* API.

4.4 Agents

CEADS-LIN employs three types of agents: attacker agents, defender agents, and user agents. Attacker agents

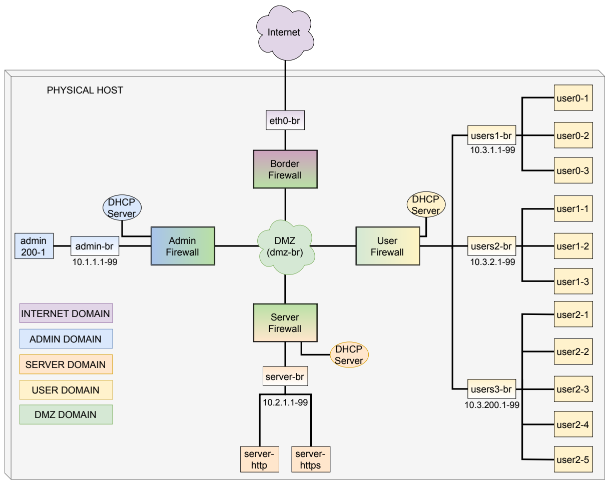

Figure 3: Network Topology for CEADS-LIN

are initialized on user nodes with the task of enumerating the network through various evolved enumeration strategies. Defender agents, while not implemented in the described experiment, will for example control traffic flow among nodes aiming to disconnect any users with malicious traffic. User agents exist to add realistic network traffic; they uniquely add traffic for the attacker to learn from, and network connections that the defender must keep intact. Users will additionally prevent the defender from taking extreme measures, as the defender's fitness will be equally based on maintaining user connection uptime and impeding attackers.

4.5 Experiments

Experiments consist of an EA submitting a series of evaluation requests to Mothership. These requests contain a file describing the genotype of the attacker strategy. When performing an evaluation, Galaxy follows these steps:

1. Galaxy receives the data for an evaluation to be run in a virtual environment sent by the EA.
2. A UFO is scheduled to run this evaluation and prepares the virtual environment by placing the evolved agent parameters onto the virtual network.
3. The UFO starts the virtual network, which in turn automatically starts the attacker agent and all services on the various VMs.
4. The attacker agent begins listening for packets at its location and executing network scans of IP ranges according to the strategy described by the genotype, reporting results of discovered machines to Mothership.
5. After sixty seconds, the evaluation concludes and

the network is shut down. Mothership is informed that the evaluation has completed.

6. The EA uses the information reported by the agent and stored by Mothership to calculate the fitness of the attacker strategy.

This process repeats for each evaluation sent to Galaxy by the EA. Using Galaxy's distributed parallel computation, these evaluations are run six at a time, proportionally decreasing the runtime of an experiment by a factor of six. Evaluation continues over many generations and thousands of evaluations of the EA, as its population gradually improves its strategies for searching the network.

The runtime of an experiment is the product of the length of a single evaluation (rebuild time plus running time), the number of children per generation, and the number of generations that the EA runs for, divided by the number of parallel evaluations that can be run. For the specific experiments discussed here, this is five hours[7]. In future experiments, as the complexity of the attacker's task increases, the length of an evaluation will likely need to increase. The use of coevolution to evolve defender agents alongside attackers will further multiplicatively increase the length of an experiment, as each attacker must be evaluated against several defenders, and vice versa. Galaxy's abilities to quickly rebuild a network and distribute evaluations in parallel are therefore critical to the use of an EA, because without them the time to run a single experiment would rapidly grow infeasible.

4.6 Results

Figure 4 displays the fitness scores of the EA's best individual and average value over time, both averaged across five experimental runs. Testing with randomly generated attacker agents found that they were able to discover an average of 20% of the network. Meanwhile, while the EA runs started similarly, after 400 evaluations (generation 10) the average attacker could find 55% of the network and the best could find nearly 70% of it. By the end of most experiments, top-performing strategies could find 74% of the network within one minute, for the five-subnet network shown in Figure 3.

Typical high-performing network enumeration strategies involved first prioritizing the discovery of new subnets with nearby addresses, by searching for analogues of known addresses such as using 10.1.1.254 to infer 10.2.1.254. Once those were checked, agents would exhaustively search the /24 subnets around known addresses. When those were exhausted, many agents would also try to guess low-numbered addresses in the hopes of finding any undiscovered user subnets.

[7] $(60 + 60 \text{ seconds}) \times 18 \times 50 \div 6 = 5$ hours

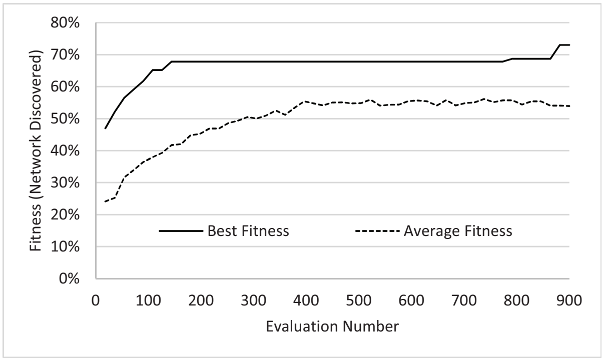

Figure 4: Fitness of the EA populations over time

These results indicate that the evolved agents were successfully able to evolve effective strategies that were far better than random, and within a reasonable timeframe. While the actual network enumeration strategies developed are not particularly interesting from a security perspective, the success of evolution in discovering them indicates that Galaxy provides an environment supportive to the evolution of agents.

4.7 Lessons Learned

Shifting from simulated to emulated experiments removed the luxury of real-world abstractions. For example, VMs must be power cycled, or reverted, to reset state as opposed to quick internal variable modifications in simulations. Subsequently, each action in the emulation environment now incurs real-world time costs. These consequences come from the inherent nature of cybersecurity where attackers notoriously have near infinite time, money, and patience, whereas our experiments are heavily time and resource restricted. CEADS-LIN required a tradeoff between the evaluation time and total experiment time. At 900 evaluations per experiment, every second of an evaluation counts for 15 minutes of experiment time. Increasing the evaluation time allowed the agent to perform more realistic actions but inhibited quality evolution as fewer evaluations could be performed in a reasonable amount of time.

Originally, several attack features were developed for the evolutionary agents that required long periods of time to complete; dictionary attacks, hash cracking, and brute force remote ssh authentication attacks all may run for hours on a designated host computer and the VMs have less time and compute power. Therefore we pivoted from authentication attacks into network scanning, which requires significantly less runtime and provides a foundation for more elaborate attacks going forward.

For every node in the network, VM overhead actions such as power cycling and snapshot reverting adds approximately 4 seconds to every evaluation's setup cost. The 18 nodes in the CEADS-LIN experiments added in total 60 seconds to each evaluation, in conjunction with other overhead mentioned in Section 3, doubling the overall experiment time. As a result, we ran fewer evaluations, consequently reducing the evolution quality. In future work, we will reduce operational overhead, allowing experiments to contain more complex emulated attacks and exploits.

5 Future Work

Even though initial applications of Galaxy have been successful with the CEADS-LIN project, there are a number of current limitations to the framework. Each physical host computer can only support a single virtualized network. When it comes to small networks like the one described in Figure 3, many instances could be run on a single machine increasing the number of networks scaling better across distributed hosts. Additionally, working with Galaxy requires advanced knowledge of the underlying infrastructure. To address these problems, future versions of Galaxy will attempt to containerize the infrastructure and create a graphical user interface for interacting with it.

5.1 Containerization

VM emulation is bulky and computationally expensive. The setup and management costs burdens evaluation performance through the additive costs of building and running many evaluations. Containers use fewer system resources, have faster power cycle actions, and unlike VMs, which emulate the entire architecture stack, containers share the host kernel at the cost of fidelity. Ultimately, we weighed the overhead reductions and performance increases against future kernel attack restrictions as kernel panics will be shared across all nodes on host. Handigol [3] and Heller [5] utilize container-based virtualization techniques for high-fidelity emulation, which will be explored in a future version of Galaxy.

5.2 Graphical User Interface

A user-friendly web-GUI microservice will be added to Galaxy to help interface with the infrastructure. The goal is to remove the requirement of understanding Galaxy's underlying complexities to run experiments and visualize evaluation progress, results, and other metadata.

6 Conclusion

Modern cybersecurity testing and research in enterprise-grade networks is limited by uptime-dependent systems which constrain the breadth and depth of permitted testing. Moreover, general cybersecurity testing of network topologies requires a large investment in physical hardware or low-level virtualization knowledge. Galaxy attempts to solve this problem by providing an emulation framework that enables cybersecurity researchers and professionals to emulate network topologies with high-fidelity. By leveraging virtualization, Galaxy creates a high-fidelity clone of a network and takes snapshots to always have a copy of this known base state. With this cloned network, researchers can perform experiments to see how the network performs with certain kinds of attacks such as in the CEADS-LIN project. The successful use of Galaxy in conjunction with CEADS-LIN demonstrates that the techniques presented here are effective for facilitating the use of EAs on emulated computer networks.

Acknowledgements

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

References

[1] Crussell, J., Erickson, J., Fritz, D., and Floren, J. minimega v. 3.0, version 00. `https:/www.osti.gov/servlets/purl/1312788`, 2015.

[2] Eiben, A., and Smith, J. *Introduction to Evolutionary Computing*, second ed. Springer, 2015. ISBN 978-3-662-44873-1.

[3] Handigol, N., Heller, B., Jeyakumar, V., et al. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 253–264.

[4] Harris, S., Michalak, E., Schoonover, K., et al. "Evolution of Network Enumeration Strategies in Emulated Computer Networks". *Genetic and Evolutionary Computation Conference* (Jul 2018). Forthcoming.

[5] Heller, B. *Reproducible Network Research with High-fidelity Emulation.* PhD thesis, Stanford University, 2013. pp. 33-42.

[6] Hibler, M., Ricci, R., Stoller, L., et al. Large-scale Virtualization in the Emulab Network Testbed. In *USENIX 2008 Annual Technical Conference* (Berkeley, CA, USA, 2008), ATC'08, USENIX Association, pp. 113–128.

[7] Lehman, J., Clune, J., Misevic, D., et al. The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities. *ArXiv e-prints* (Mar. 2018).

[8] Mirkovic, J., Benzel, T. V., Faber, T., et al. The DETER Project: Advancing the Science of Cyber Security Experimentation and Test. In *2010 IEEE International Conference on Technologies for Homeland Security (HST)* (Nov 2010), pp. 1–7.

[9] Rush, G., Tauritz, D. R., and Kent, A. D. 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* (2015), ACM, pp. 859–866.