Large-Scale Realistic Network Data Generation on a Budget

Brian Ricks
Computer Science Department
University of Texas at Dallas
Richardson, USA
Email: bwr031000@utdallas.edu

Patrick Tague
Electrical and Computer Engineering Department
Carnegie Mellon University
Moffett Field, USA
Email: tague@cmu.edu

Bhavani Thuraisingham
Computer Science Department
University of Texas at Dallas
Richardson, USA
Email: bxt043000@utdallas.edu

***Abstract*—Many novel problems in computer networking require relevant network trace data during the research process. Unfortunately, such data can often be hard to find, which becomes a problem within itself. While generating appropriate data using in-lab network testbeds and simulators are feasible solutions, the former has limitations in terms of network scale, while the latter has limitations in the generated data. To help address these issues, we present an approach for the generation of realistic network trace data in a contained, large-scale network environment. We use network emulation to enable large-scale, in-lab networking, and a software framework we developed to support autonomous client-side protocols and services, including user-behavioral models which scale in a shared CPU environment. Our framework also enables quick experiment setup and monitoring. We show through experimentation on a low-end laptop that our approach enables network scale into the hundreds of nodes, allowing anyone with even basic hardware to generate potentially relevant, realistic network data.**

I. Introduction

Relevant network trace data is an important component for almost all facets of computer networking research. When specific data needed for a research problem is available to use, work on solving that problem can proceed. However, what if the data is inaccessible or does not exist? What if portions of the accessible data are missing? What if the data does not quite fit a problem's specification, but no alternatives exist? Indeed, this occurs more frequently than we would like, and can result in changes to a problem's approach or even to the problem specification itself.

Within the networking domain, solutions to this common issue have often relied on in-lab physical network testbeds to generate the appropriate network trace data needed to move forward on solving a problem. This may be adequate for smaller networks, such as those incorporating 5-20 nodes, but many problems rely on the need for data captured from large-scale networks. These networks may incorporate hundreds or thousands of nodes. To further complicate things, a research problem may require trace data comprising realistic, human-originated traffic. Such networks often are not feasible to build in a lab due to cost of hardware, space constraints, time required to build, and possibly the number of human volunteers needed to create the traffic that will traverse the network. This quickly becomes a challenging problem in its own right that needs to be solved before work on the primary problem can begin or continue.

Solving this problem for large-scale networks typically consists of two approaches. The first uses network simulation software to simulate a network and generate user-based traffic for capturing synthetic network data [1], [2]. However, network simulators make simplifications and assumptions in their models that may not map fully to the real-world. Furthermore, if trying to capture network trace data originating from real software, such as botnets, often times the software needs to be abstracted to a simulation model. This not only requires additional work to program, but may require protocol simplifications that produce differing behavior from the software in the real-world.

The second approach leverages collaboration among various researchers to construct large-scale, physical networking testbeds [3], [4], [5]. This approach comprises a wide-area physical network, often spanning the facilities of the participating researchers, but at the cost of network flexibility. Compromise may be necessary in terms of topology or other behavioral characteristics of the network [6]. Ideally, a researcher should be able to deploy any experiments they require within a network testbed that they can tailor for their specific needs, and that is available whenever they need to use it.

In this work, we set out to provide a solution which allows for ease of experimental setup, execution, and data collection in a large-scale network environment. Our approach utilizes network *emulation* to enable *near real-world* network traffic over an emulated network contained within a single physical computer. We do not claim completely real-world network traffic for two reasons: the physical and link layers of the network stack are simulated, which introduce assumptions typically present in network simulators, and the use of models to represent human-behavior over the emulated network.

To enable large-scale networking and experimentation in the shared resource environment provided by a single computer, we developed a framework for network traffic automation and experiment management, called eMews, which is designed with such constraints in mind. This requires a design methodology that both minimizes overall

CPU usage for the entire network, and maximizes CPU utilization at any given time during an experiment run. Our experimental results (Section V) show that even when deploying a heavily utilized wide-area emulated network on a lower-end laptop, EMEWS supports networks with hundreds of nodes.

Our main contributions are as follows:

- We provide a means to generate and collect near real-world network data on large-scale networks without requiring a large-scale, physical network.
- We incorporate autonomous models for human-based traffic generation over a large-scale network which adequately represent the desired user behavior while minimizing CPU resources.
- We create a framework for experiment management and automation that enables quick setup, execution and monitoring without researcher interaction. Our approach aims to both minimize overall CPU usage and optimize the available CPU resources among all the components managed by the framework.

The rest of this paper is structured as follows. In Section II we discuss relevant background and related work. Section III discusses the EMEWS framework and the service-driven model that it employs, including the methodology governing our design choices. Section IV discusses our behavioral models in depth. Section V presents our experiments and results, with relevant discussion. We then discuss limitations of our approach and framework, along with future work, in Section VI, and we conclude the paper in Section VII.

II. Preliminaries

In this section, we discuss relevant background and issues in the context of related work. Much prior work in dataset generation and collection have focused on network security problems, more specifically intrusion detection. Indeed, intrusion detection is a very rapidly evolving field, and the data available to researchers should evolve as well.

A. The Dataset Collection Problem

Obtaining real-world, relevant datasets for computer networking problems, such as anomaly detection, has traditionally been difficult [7], [8]. This problem is compounded when trying to apply traditional machine learning techniques to train a representative model of normal and anomalous network behavioral patterns. In fact, the lack of available training data is one reason why machine learning based detection methods have failed to gain traction in the real-world [7]. This dearth of available data can arise if a desired attack behavior does not exist in the wild (i.e., the desired behavior has not been captured), or if entities owning relevant data do not release the data to the public. In the latter case, an entity, such as a corporation, may not want to disclose the data they have collected due to privacy concerns, or due to agreements with other entities [5].

One approach to address the shortage of large-scale networking data takes many disjoint datasets and combines them using a combination of packet replay and subnet merging [9]. Replaying packets is problematic as the original traffic behavior may be inadvertently modified due to interactions of the replayed packets with other network activity, and protocols that follow a request-reply-repeat paradigm, such as SSH, may become unsynced at the replay source. Subnet merging, on the other hand, results in disjoint traffic flows, due to a lack of any interactions between the subnets belonging to different datasets.

B. Network Emulation

Network emulation in our context is a technique to enable networks contained within a single physical computer, while maintaining real-world network properties and behavior at the OSI network layer (layer 3) and above [10], [11], [12]. While network emulators have been around for a while, their use for large-scale network dataset generation, especially for intrusion detection and distributed denial-of-service (botnet generated trace data), is still relatively new.

One consequence of a multi-node network living within a single computer is shared resource usage. Unlike with a physical network where each node is itself a physical computer - utilizing its own CPU and memory, in an emulated network, every node shares the same CPU and memory - that of the physical computer in which the network resides. This includes all client and server side protocols, routing protocols, underlying network stack operations, and even the simulation of the physical and link layers. Thus, full utilization of the available shared CPU cores may result in various network timing and throughput artifacts [11]. This also implies that the emulated network's shared CPU resource usage becomes an important metric to consider [12]

In this work we utilize the CORE network emulator as a base to create a network within a single physical computer [11]. Network hosts, routers and other nodes are emulated within Linux namespace containers (LXC), with each container provided its own network stack, filesystem mountpoints, and isolated process space. The physical and link layers in CORE are simulated, which is to be expected from an emulation platform. While these layers can be simulated using more complicated simulation models [13], we opt to keep the CORE default model as it is designed for wired networks, and uses relatively low CPU resources.

C. Autonomous User Behavior

Capturing genuinely real-world user traffic, such as web browsing traffic, is a difficult problem to solve in both a simulated and physical network environment. User-generated traffic relies on real users using the network for their everyday activities, which is not feasible if the physical network consists of a testbed within a lab. One alternative, which can be applied to physical or emulated network testbeds, is having users volunteer to use a network testbed for some activity during an actively running experiment. However, human behavior tends to change when in the presence of a cue that suggests they are being watched [14], giving doubt that test subjects in a lab experiment will behave exactly as they would in the real-world. Another alternative involves using network trace data previously captured from other networks as a source to create behavioral models [15], but these previously captured datasets may incorporate characteristics that do not map past the specific network topology in which they were captured.

One approach that addresses autonomous user-behavior describes a platform for generating network data based on autonomous client services called *agents*, driven by behavior defined in *profiles* [16]. This concept of profiles provides for a higher level specification of an experiment in general. However, the specific implementation of the agents and the scalability of the system is not discussed, given that their work targeted a physical in-lab testbed, which is not as resource constrained nor as large-scale as our target environment.

Our work presents user-behavioral models that are based on observations of how some users interact with a network during specific activities, and create models that reflect more general work flows, such as a crawling many web pages on a specific web site. Our models abide by a design methodology that promotes reduced CPU usage while enabling better concurrent utilization of the CPU. These user-behavioral models and our design methodology are presented in Section IV.

III. A Framework for Automation and Deployment of Network Experiments

We propose a framework, named EMEWS[1], to provide a quick and simple means of deploying large-scale network experiments which incorporate automated client-side user behavior. EMEWS aims to satisfy the following principles:

- Near real-world generation of network data, potentially including very large-scale networks and over very long periods of time.
- Quick implementation and deployment of client-side user behavior models, protocols, and other services to facilitate network data generation.
- Integrity of the experiment provided through alerts triggered when unexpected behaviors are encountered.
- Hardware implementation and physical footprint of the testbed are cost effective and appropriate for a standard academic lab.

While each principle incorporates some related work, there is yet to exist a solution that not only covers all of these principles, but which is freely available for any researcher to use.

[1]The latest version of EMEWS can be downloaded from the official project page at http://mews.sv.cmu.edu/research/emews/ or directly from GitHub at https://github.com/absolutefunk/emews.

A. Architectural Overview

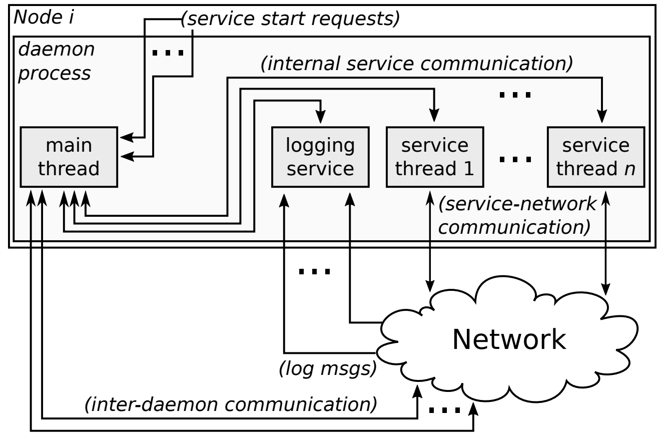

Figure 1. The EMEWS daemon architecture. Every node in the network which requires EMEWS services will contain a daemon process. If a node is the designated log server for distributed logging, then that node's daemon will also launch the logging service.

EMEWS is a distributed management and automation system that controls and coordinates various aspects of a network experiment, providing the means for quick experiment deployment and monitoring, and written for the resource constrained environment of an emulated network. As figure 1 illustrates, every node in the emulated network requiring EMEWS services (Section III-B) will launch a single daemon process. This daemon initiates and starts services as requested by external EMEWS request clients.

The framework employs a modular architecture similar to OMNeT++ [1], in which simply inheriting a base class interfaces a new component. This inheritance, combined with additional built-in helper and decorator classes, tries to minimize the amount of work a researcher needs to perform to implement a component and get the component running in the network.

The EMEWS framework is written in Python, which was selected for its speed when prototyping new logic, such as client-side automation protocols required for an experiment or other components. Python also encompasses a rich set of external libraries which can easily be incorporated when writing new components for EMEWS, further cutting down on the amount of time spent programming.

B. Services

Services provide the functional components that give EMEWS its usefulness, such as automating the client-side of a protocol. Everything that interfaces to EMEWS is a service, including the behavioral models.

One challenge with service-based architectures is scalability in a shared resource environment, such as the emulated network upon which EMEWS runs. To help overcome this challenge, we observe that many potential services, such

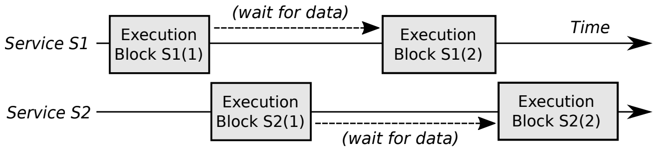

Figure 2. The service design methodology. Program logic is broken down in to small execution blocks, with periods of waiting between them. This example shows one possible way that two services could be scheduled on a single CPU core.

as common client side protocols, comprise small blocks of logic which execute, often preparing and sending data over the network, followed by a period of waiting for a response or data. For example, if sending a request to a web server for a web page, the logic to actually send the request requires little CPU resources, but the wait to receive data can often dominate in time, thus creating a task in which the web client is mostly sitting idle.

The design methodology that we propose follows this basic pattern of executing a small block of logic followed by a period of waiting. A service should be written to execute only what it needs to, including sending data, and then either wait to receive data or until some condition is met. Figure 2 gives an example consisting of two services simultaneously executing on a single CPU core. An entire block is executed, then the service waits. During this wait, another execution from another service can execute. In this way, when a service waits, another service can utilize the CPU. Notice with the example in Figure 2, after execution block $S2(1)$ finishes, service $S1$ still hasn't received data, so the CPU is idle until data is received by service $S1$ and execution block $S1(2)$ can start. Once execution block $S1(2)$ finishes, notice that execution block $S2(2)$ from service $S2$ starts immediately. This could be due to coincidental timing of service $S2$ receiving data right as execution block $S1(2)$ finished, or having received the data earlier but continuing to wait for execution block $S1(2)$ to finish.

To enable this type of CPU scheduling, each service runs in its own thread under a single EMEWS daemon process on each node (Figure 1). To prevent CPU context switching while an execution block is actively running, we take advantage of a feature in the CPython interpreter called the *global interpreter lock (GIL)*, which only allows context switching during an I/O call, such as waiting for data over a network, or even when a services sleeps. This guarantees that our execution blocks will run in their entirety without fear of context switching.

One question to ask is why minimizing CPU context switching is important. As CPU context switching itself requires additional overhead, this overhead starts to add up in an emulated network with possibly thousands or tens of thousands of concurrently running services. Therefore, by using a design methodology of atomic blocks of logic, we can context switch only when it would happen regardless (during network I/O, for example), and prevent any other occurrences.

C. Service Monitoring

An important aspect of any network experiment is the ability to monitor the state of all active components during an experimental run. With respect to EMEWS, all running daemons and the services they each contain enable log messages to be emitted, notifying the researcher of any events, including anomalies.

EMEWS incorporates per-node or distributed logging. Per-node logging logs to a file on each node, whereas distributed logging uses one designated node as a centralized log server (Figure 1), and all other nodes send log entries to this designated node. The designated log server node maintains a single log file for the entire network.

In using distributed logging, all the researcher needs to do is monitor a single log file. While this method is pretty simple, it requires log traffic to traverse the network, which may not be desired. One workaround is only emitting log messages for anomalous events, so non-anomalous experimental runs will not generate log traffic over the network.

Monitoring a running network experiment using per-node logging requires a method to monitor each node's log file. One approach, which takes advantage of CORE's underlying directory structure on the physical host computer, is to traverse every emulated node's root directory as it is represented on the physical host, opening each log file found. This way any event from any node can be logged without network traffic being generated.

IV. Client Behavioral Models

In this section we discuss the client-side automation protocols we implemented as services in EMEWS, and their underlying behavioral models. These models serve more as a proof of concept that expressive-yet-simple models can be run within EMEWS that preserve its network scalability. Additional behavioral models and their services can be written and added to EMEWS, which we actively encourage.

A. Methodology

Ideally, any EMEWS behavioral model should minimize CPU usage. Therefore, simpler models should be preferred providing they represent the client-side behavior adequately. While the term "adequately" is rather subjective, for very large-scale networks, the computational overhead of these models can greatly affect the upper bound on network size if they are employed on many of the network nodes.

In keeping with the design methodology of our services, we split the user behavior into execution blocks of small behavioral patterns governed by single distributions, and sequence the blocks to form the completed pattern.

The following behavioral models represent generalized scenarios based on observations of real user behavior. However, they clearly do not cover every possible user scenario, nor do they cover specific nuances that would require a much more complicated model, or a model which could not be split nicely into small execution blocks.

B. SSH

The behavioral model we use for SSH client traffic follows a pattern of initiating an SSH connection with some server from a list, executing some commands, and exiting the session. This behavioral pattern follows from a common pattern observed in academia in which a user needs to connect to an SSH server to perform a series of tasks that are often repeated, for example logging in to collect data from a previously run experiment and launch a new variation of the experiment.

The model consists of the following parameters:

- $sc(t)$: The amount of time to wait before starting the next SSH session during timeslice t.
- $cn(t)$: The number of commands to execute during the SSH session at timeslice t.
- $cc(t,i)$: Command i to send to the SSH server during the SSH session at timeslice t.
- $cd(t,i)$: The delay between command i and command $i+1$ during the SSH session at timeslice t.

Each parameter is sampled with respect to a specific timeslice of execution, t. For each timeslice t, one SSH session is initiated, commands executed, and session terminated. A single timeslice will incorporate many execution blocks.

We assume that $sc(t)$ and $cd(t,i)$ are uniformly distributed, due to such behavior consisting of a naturally high degree of randomness and the relative computational simplicity of sampling. We use a discrete uniform distribution parameterized by lower bound a and upper bound b to define the range of sampled values. As such, each sample x is bounded as $a \leq x \leq b$.

We assume that $cn(t)$, and $cc(t,i)$ are normally distributed, which enables us to use distribution parameters to model implicit temporal dependencies which would be difficult to represent without a more complicated model or breaking our design goals. This allows for a general assumption that for any timeslice t, $P[y(t+1)|y(t)] = P[y(t+1)]$, where $y(t)$ is a model parameter such as $cn(t)$. We imply a dependency between $y(t+1)$ and $y(t)$ by inducing a high probability of sampling similar values for timeslice t, $t+1$, etc. This is accomplished by using a very low standard deviation. The same follows for sequences of commands. We have implicitly that command $i+1$ is dependent on command i due to the sequence of commands appearing to following from one another, even though explicitly $P[cc(t,i+1)|cc(t,i)] = P[cc(t,i+1)]$.

We use a variation of the normal distribution called the *truncated normal*, which is similarly bounded above by a and below by b, and further defined by μ, the expectation, and σ, the standard deviation. We set $a = 0$, and $b = |list|$, where $|list|$ is the number of items in a list that corresponds to one of our model parameters, such as the total number of commands in the list for $cn(t)$ or $cc(t,i)$. These constrain the sample to an index value (once rounded), suitable for item retrievals from lists. We further define $\mu = b/2$, which provides a simplification if we assume that the specific value we are expected to sample does not matter, as long as it is *consistent* across multiple samples for timeslices t, $t+1$, $t+2$, ... , $t+m$, and for commands i, $i+1$, $i+2$, ... , $i+n$. In other words, we care about the overall behavior, not specific instances of the behavior. For example, assuming some value of μ based on a fixed value for b during timeslice t, suppose we sample $cn(t) = 5$. This value fits the behavioral pattern provided that $P[cn(t+1) = 5]$ is high. We induce this dependency, as discussed in the previous paragraph, by setting σ to a small value (0.5 - 1.0 worked well in trial runs).

This gives the following PDF f_{tn} for our truncated normal distribution:

$$f_{tn}(x;\mu,\sigma) = \frac{\phi(\frac{x-\mu}{\sigma})}{\sigma(\Phi(\frac{\mu}{\sigma}) - \Phi(\frac{-\mu}{\sigma}))}, \tag{1}$$

in which ϕ is the normal PDF, Φ is the normal CDF, and x is a generated sample within the range $[0,b]$.

One final note regards the dependence relationship in $cc(t,i)$ between command i and $i+1$. Commands are sampled *without* replacement, and a dependency is introduced due to b changing between samples. However, this dependency is both desired and comes without any additional computational overhead other than a list copy (to preserve the original list of commands for future sessions). In terms of desirability, similar sequences of commands per SSH session are produced with high probability given an appropriate σ. Further, when a selected command deviates from the expectation, the rest of the sequence now has a high probability of being shifted to some other sequence of commands (based on our μ being fixed to the middle command in the list), mimicking potential user behavior during similar real-world deviations.

To complete the definition of the model, we define the following parameters which need to be set manually (all of them distribution parameters): a_1 and b_1, which define the range of time for $sc(t)$, a_2 and b_2, which define the range of time for $cd(t,i)$, σ_1, which defines the standard deviation for $cn(t)$, and σ_2, which defines the standard deviation for $cc(t,i)$.

C. HTTP

The behavioral model we use for HTTP (and by extension HTTPS) is based on an observation that for some web sites, such as online banking, a pattern is often followed in which given the first hyperlink (link) clicked, the subsequent links tend to follow a consistent pattern across multiple accesses of the site. For example, if the first link clicked is 'login to online banking', then the next link clicked has a high probability of being the 'login' button after entering credentials.

This type of path-based pattern in web browsing has also been observed in the domain of link prediction [17], [18], though the models used are much more complex than what we believe a large-scale emulated network can handle in bulk. Thus, we developed a model that can represent this type of path-based work flow, using the same principles as for SSH (Section IV-B).

The model consists of the following parameters:

- $hc(t)$: The amount of time to wait before starting the next HTTP session during timeslice t.
- $hv(t)$: The HTTP server to connect to at timeslice t.
- $pn(t)$: The number of web pages to request during the HTTP session at timeslice t.
- $pp(t, i)$: Web page i to request during an HTTP session at timeslice t.
- $pd(t, i)$: The delay between web page request i and web page request $i+1$ during the HTTP session at timeslice t.

We use the same distributions and definitions of a, b, and μ as for the SSH behavioral model (Section IV-B). Parameters $hc(t)$, $hv(t)$, and $pd(t, i)$ are uniformly distributed, whereas $pn(t)$ and $pp(t, i)$ are normally distributed (using Equation 1).

To keep the number of web pages crawled to a realistic value, we use a heuristic in which once the first web page is returned to us ($i = 0$), and the index of the next link to crawl selected, we use this index as b when sampling $pn(t)$. Thus, $E[pn(t)] = E[pp(t, 0)]/2$. Note that crawling will also stop if a web page is reached that consists of no links.

We define $P[pp(t, i : i > 0)|pp(t, 0)] = P[pp(t, i : i > 0)]$ and represent the dependency implicitly by sampling $pp(t, 0) \sim TN(\mu, \sigma_3)$ and $pp(t, i : i > 0) \sim TN(\mu, \sigma_4)$, where $TN(\mu, \sigma)$ represents the truncated normal distribution we are sampling from (Equation 1), σ_3 is the standard deviation used when sampling the first link to click ($i = 0$), and σ_4 is the standard deviation used when sampling all subsequent links ($i = 1...b - 1$). We set $\sigma_3 > \sigma_4$ to signify that after the first link is clicked, subsequent links clicked have a higher probability of following a specific pattern (i.e., a higher probability of the middle-index link being clicked).

When a web page is requested through a link, that web page is returned, containing a new set of links. Thus, b often changes between page requests, inducing a dependency between $pp(t, i)$ and $pp(t, i + 1)$. Even if b doesn't change, most likely the list of specific links will. This follows from the real-world in which a link that a user clicks is directly dependent on the web page that contains the link, which itself is dependent on the previous link that loaded the page.

To complete the definition of the model, we define the following parameters which need to be set manually (all of them distribution parameters): a_3 and b_3, which define the range of time for $hc(t)$, a_4 and b_4, which define the range of time for $pd(t, i)$, σ_3 and σ_4, which defines the standard deviation for $pp(t, i)$, for the first link clicked and subsequent links clicked, respectively, and σ_5, which defines the standard deviation for $pn(t)$.

V. Experiments

We aim to see how well an emulated network running eMews scales up. We start with a small emulated network consisting of a set of 100Mb and 1Gb LANs connected to a few routers, and deploy eMews SSH and HTTP services on a majority of the nodes, each connecting to various servers on other nodes and emulating sessions. This emulated network also consists of an iPerf 1Gb client-server session running to serve as a canary for throughput anomalies. The iPerf client and server are disconnected from the rest of the network, to minimize throughput fluctuations due to other traffic. Starting from a modestly sized network topology, we continue to scale up the emulated network by adding more client and server nodes until either the throughput from the iPerf session starts to drop, or we run out of computing resources.

The eMews HTTP and SSH services were configured to be more aggressive to increase overall network utilization. For HTTP, model parameters $hc(t)$ and $pd(t, i)$ were configured to decrease the average time spent waiting to connect to a web server, and clicking on links, respectively. For SSH, model parameters $sc(t)$ and $cd(t, i)$ were configured to decrease the average time spent waiting to connect to an SSH server, and sending commands, respectively. We also purposely configured the distributed logger to log all events, which provided another source of traffic to increase our network utilization.

All experiments were run for 1 hour to help ensure stability in CPU usage (ie, the average CPU usage should not increase over time). We also monitored average memory usage, which also should not increase during our experiments.

While data collection was not a primary focus of our experiments, we did collect PCAP data from a single node on our emulated network. This node ran both an SSH and HTTP server, which captured data originating from our SSH and HTTP eMews services, respectively.

All experiments were run on a Dell Inspiron laptop, consisting of an Intel Core i7-7500U CPU running at 2.70GHz (dual core with hyperthreading), 8GB of RAM, and running Ubuntu 17.10 (kernel 4.13.0-39). Our experiments used CORE 5.0 and eMews 0.32. In addition to CORE and eMews, our emulated network also ran instances of Apache2 2.4.27, OpenSSH 7.5p1, and iPerf 3.1.3.

A. Scalability Results

Looking at Table I, the overall scalability of the emulated network was surprisingly good considering the hardware in which we performed the experiments. Most surprising was why we stopped at 263 nodes. With an average CPU usage

Node Count	Host Count			Client Count			CPU Usage (Avg)
	Total	HTTP	SSH	Total	HTTP	SSH	
75	11	8	4	40	27	17	21%
157	19	16	4	86	58	37	34%
215	26	23	4	118	82	50	47%
263	30	27	4	146	103	62	65%

Table I
OUR NETWORK EMULATION SCALABILITY RESULTS INCLUDE MULTIPLE TRIALS WITH VARIABLE NODE COUNT IN THE EMULATED NETWORK. HOST COUNT (TOTAL) AND CLIENT COUNT (TOTAL) REPRESENT THE TOTAL NUMBER OF NODES THAT ARE CONFIGURED AS HOSTS AND CLIENTS, RESPECTIVELY. SOME HOSTS AND CLIENTS ARE CONFIGURED TO HANDLE BOTH HTTP AND SSH SESSIONS, WHICH IS WHY THE SUM OF HTTP AND SSH CLIENTS IS GREATER THAN THE TOTAL COUNT.

of 65%, it would seem we had some room left to grow, but unfortunately we did not have enough memory to proceed. In fact, during the 215 node trial, the network was consuming 7.1GB of RAM and 5.4GB of swap (out of 7.9GB). When we scaled up to 263 nodes, we hit 7.5GB of RAM consumed and 7.2GB of swap. At this point, 6 EMEWS services could not start due to insufficient memory.

The iPerf results were very stable at 958Mbps average throughput per trial, rounded to the nearest megabit per second. For each trial, iPerf comprised 7% of the average CPU usage. Because we hit our memory limit before we could exhaust CPU resources, this result is to be expected.

CPU usage seemed to scale linearly as the node count increased. This would suggest that, given more RAM, we could have reached a node count in the upper 300s. Considering that the experiments comprise of hundreds of concurrent threads running on a dual core processor, an emulated network consisting of 300+ nodes is quite acceptable. With even a modest desktop computer, emulated networks with thousands of nodes should be possible.

One last point to touch upon regards CPU and memory usage over time. Average CPU usage remained very consistent over our experimental runs, as expected. Memory usage spiked during EMEWS service initialization at the beginning of each experimental run, which was also expected. EMEWS services initialize using external client processes that request them to the daemon (Figure 1), and these processes terminate after a request. What was unexpected is that memory usage slowly started increasing again as our experimental runs proceeded. To measure exactly how long an experiment can run before exhausting memory, we re-ran our last experiment (263 node count), which comes close to exhausting memory during EMEWS service initialization. The experiment ran for 10 hours before memory exhaustion. The underlying reason could be EMEWS, a network protocol running outside of EMEWS (an SSH or HTTP server, for example), or the CORE network emulator. Further investigation is needed to discover the culprit.

VI. LIMITATIONS AND FUTURE WORK

While our solution is a step closer to reaching the ideal goal of cost-efficient, in-house, real-world network experimentation, some limitations inherent to our approach need to be addressed.

Experimentation utilizing emulated wireless networks pose many of the same challenges as using a network simulator. While the network stacks are real-world, the physical medium in an emulated wireless network is still simulated, with many of the same limitations and simplifying assumptions made in network simulators. Thus, some wireless experiments, especially those which require measurement of physical signal properties such as received signal strength (RSSI) [19], may not be suitable for emulation.

In a wired setting, as discussed in Section II-B, network emulation provides for real-world network behavior from the network layer (layer 3) upward. Experiments which focus on the physical or link layers may not be suitable for emulation.

If a researcher wishes to conduct experiments to learn about how humans interact within a network environment, or research exploring various human quirks that may arise while using a network, they clearly would need an actual human in the loop. This type of data collection most likely would require either a large-scale public network (that the researcher could collect network trace data from), or a smaller network with many users.

The SSH and HTTP behavioral models may not represent some scenarios very well, and certainly, there are many in which they will not represent well at all, even with model parameter tuning. However, one strength of EMEWS is the ability to quickly implement custom services, including new behavioral models. Indeed, one motivation for the creation of EMEWS was to enable this kind of expansion of individual services under a common framework.

Auto-labeling of network trace data is a feature we would like to implement in EMEWS. While the foundation for this already exists, an EMEWS service would need to be written to correlate and label the traces.

Another future work direction is implementing a botnet EMEWS service. Flooding-based botnets do not require human-based behavioral models for traffic generation, as these botnets generate traffic autonomously. With the command and control overhead already implemented, a researcher would only need to define the botnet traffic generation model, or simply set the parameters for the default model. We think this would open the door for large-scale botnet network trace data generation.

VII. Conclusion

In this paper, we showed that large-scale, realistic network data generation using network emulation and our eMEWS framework is indeed possible on cheaper hardware. While we do not expect most academic labs to be confined to laptops for performing experiments, our results give good insight into how well even a modest desktop could perform. And by implementing additional eMEWS services, many different types of networking experiments and data generation can be performed, all within the comfort of a lab.

References

[1] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ser. Simutools '08. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 60:1–60:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1416222.1416290

[2] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34. [Online]. Available: https://doi.org/10.1007/978-3-642-12331-3_2

[3] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir, "Experiences building planetlab," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 351–366. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298489

[4] E. Jaffe, D. Bickson, and S. Kirkpatrick, "Everlab: A production platform for research in network experimentation and computation," in *LISA*, 2007.

[5] A. J. Aviv and A. Haeberlen, "Challenges in experimenting with botnet detection systems," in *Proceedings of the 4th Conference on Cyber Security Experimentation and Test*, ser. CSET'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2027999.2028005

[6] H. Pucha, Y. C. Hu, and Z. M. Mao, "On the impact of research network based testbeds on wide-area experiments," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '06. New York, NY, USA: ACM, 2006, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1177080.1177097

[7] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 305–316. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.25

[8] W. Haider, J. Hu, J. Slay, B. Turnbull, and Y. Xie, "Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling," *Journal of Network and Computer Applications*, vol. 87, pp. 185 – 192, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804517301273

[9] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet detection based on traffic behavior analysis and flow intervals," *Computers & Security*, vol. 39, pp. 2 – 16, 2013, 27th IFIP International Information Security Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404813000837

[10] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 271–284, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844154

[11] J. Ahrenholz, "Comparison of core network emulation platforms," in *2010 - MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*, Oct 2010, pp. 166–171.

[12] G. Calarco and M. Casoni, "On the effectiveness of linux containers for network virtualization," *Simulation Modelling Practice and Theory*, vol. 31, pp. 169 – 185, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X1200161X

[13] J. Ahrenholz, T. Goff, and B. Adamson, "Integration of the core and emane network emulators," *2011 - MILCOM 2011 Military Communications Conference*, pp. 1870–1875, 2011.

[14] M. Bateson, D. Nettle, and G. Roberts, "Cues of being watched enhance cooperation in a real-world setting," *Biology Letters*, vol. 2, no. 3, pp. 412–414, 2006. [Online]. Available: http://rsbl.royalsocietypublishing.org/content/2/3/412

[15] K. V. Vishwanath and A. Vahdat, "Realistic and responsive network traffic generation," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '06. New York, NY, USA: ACM, 2006, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/1159913.1159928

[16] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Computers & Security*, vol. 31, no. 3, pp. 357 – 374, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404811001672

[17] J. Zhu, J. Hong, and J. G. Hughes, "Using markov chains for link prediction in adaptive web sites," in *Soft-Ware 2002: Computing in an Imperfect World*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 60–73.

[18] A. L. Montgomery, S. Li, K. Srinivasan, and J. C. Liechty, "Modeling online browsing and path analysis using clickstream data," *Marketing Science*, vol. 23, no. 4, pp. 579–595, 2004.

[19] B. Ricks and P. Tague, "Isolation of multiple anonymous attackers in mobile networks," in *Network and System Security*. Springer International Publishing, 2015, pp. 32–45.