High Fidelity Adaptive Cyber Emulation

by
Samir Mammadov
Bachelor of Science
Computer Science & Software Engineering
Florida Institute of Technology
2014

A thesis submitted to the School of Computing
at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Information Assurance & Cybersecurity

Melbourne, Florida
May, 2017
We the undersigned committee hereby approve the attached thesis, “High Fidelity Adaptive Cyber Emulation” by Samir Mammadov.

Dr. Marco Carvalho  
Associate Professor, Executive Director, Dean  
School of Computing

Dr. Thomas Eskridge  
Associate Professor  
School of Computing

Dr. Adrian Peter  
Associate Professor  
College of Engineering
Abstract

Title: High Fidelity Adaptive Cyber Emulation
Author: Samir Mammadov
Advisor: Dr. Marco Carvalho, Ph. D.

While looking for a high-level adaptive traffic generation tool, we came to realize that no such tool exists that can be used for rapid development while being platform agnostic. Having reviewed a wide array of tools to either implement user models or simulate traffic, we were unable to find a tool with the right capabilities while maintaining complexity, portability and extensibility. To overcome these issues, we introduce a new adaptive user-modelling framework for the specific use case of cyber activity emulation. Our framework supports the creation of high-level user models that can react to changes in their environments and vary the way they emulate cyber activity based on those changes. We review the problems with the current tools and show how our behaviour tree based solution can be used to achieve our goals in an illustrative scenario showcasing the framework’s adaptability – a key feature most other tools are lacking. Furthermore, we show that our framework is also extensible, portable, and more conducive to rapid development than other user modelling tools currently available.
# Table of Contents

Abstract........................................................................................................................................... iii

Chapter 1 Introduction......................................................................................................................... 1

Chapter 2 Literature Review............................................................................................................... 5

2.1 Simulation vs. Emulation............................................................................................................. 6

2.2 Packet Crafting .......................................................................................................................... 7

2.3 Low-level Simulation Tools ....................................................................................................... 7

2.4 High-level Cognitive Modelling and Emulation Tools .............................................................. 9

2.5 Hybrid Tools............................................................................................................................ 10

2.6 Proprietary Tools....................................................................................................................... 11

Chapter 3 Concept Implementation..................................................................................................... 13

3.1 Capability ................................................................................................................................ 13

3.1.1 Adaptability ....................................................................................................................... 19

3.2 Portability ................................................................................................................................ 20

3.3 Extensibility and Usability.......................................................................................................... 22

3.4 Yoshka Dependencies................................................................................................................. 26

Chapter 4 Illustrative Scenario........................................................................................................... 28

4.1 Scenario Design ......................................................................................................................... 28
4.2 Scenario Tools & Implementation .................................................35
4.3 Scenario Evaluation ..................................................................37

Chapter 5 Informal Qualitative Evaluation ........................................45
5.1 Qualitative Scenario Design.......................................................45
5.2 Questionnaire Results ..............................................................48
5.3 Results Analysis .....................................................................50
5.3 Comparison of Mission Implementation .................................52
  5.3.1 Implementation in Yoshka ..................................................53
  5.3.2 Implementation in Ostinato ..................................................56
  5.3.3 Implementation in Soar .......................................................58

Chapter 6 Recommendations ..........................................................62

Chapter 7 Conclusion ....................................................................64

Bibliography ..................................................................................65

Appendix A ....................................................................................72

Appendix B ....................................................................................75

Appendix C ....................................................................................76

Appendix D ....................................................................................78
List of Figures

Figure 3.1 A block diagram showing the information flow in the framework. ................................................................. 15

Figure 3.2 An example structure of a CompositeNode using parallelism with the random order flag flipped. ................................................................. 18

Figure 3.3 A screenshot of the Yoshka front end web app............................. 23

Figure 3.4 A screenshot showing the visualization to be used in the Yoshka web GUI. ........................................................................................................ 25

Figure 4.1 A diagram representing the sample enterprise network we have designed for this scenario. ................................................................. 30

Figure 4.2 A picture of a sample behaviour YAML file.............................. 31

Figure 4.3 An example behaviour for a software developer in the software development company. ................................................................. 34

Figure 4.4 A screenshot of the scenario running being visualized in Kibana. ........................................................................................................ 37

Figure 4.5 A donut graph showing the proportion of all Tasks executed on the network................................................................. 38

Figure 4.6 A donut graph showing the proportion of all Tasks executed on the network after the attack................................................................. 39

Figure 4.7 A Kibana dashboard showing the state of the network after the mail server is disabled................................................................. 40
Figure 4.8 The behaviour we designed to showcase adaptability. ..............41
Figure 4.9 A tree diagram showing the execution path of the Adaptable node after the change threshold is triggered. ........................................42
Figure 4.10 A tree diagram showing the execution path of the Adaptable node prior to triggering the change threshold. ..................................42
Figure 4.11 Framework log files simplified and colourized to showcase adaptability. .................................................................44
Figure 5.1 A tree diagram showing the example behaviour to be implemented by the volunteer using each tool. .................................46
Figure 5.2 A snippet of code showcasing the sample Yoshka Task that was provided to the volunteer.....................................................54
Figure 5.3 Ostinato graphical user interface...........................................56
Figure 5.4 A screenshot of the Soar graphical user interface......................58
Figure 5.5 A snippet of Soar code showing an example Hello World rule...59
Figure 5.6 A screenshot of the Soar Visual Debugger user interface...........60
List of Tables

Table 2.1 A table comparing the types and supported platforms of related works. ................................................................. 12

Table 5.1 A table comparing the average times of volunteers for the mission implementation using different tools. .................................................. 48

Table 5.2 A table showing the average difficulty scores given for each tool. .................................................................. 49

Table 5.3 A table comparing the features and capabilities of Yoshka, Ostinato and Soar. ....................................................... 51
Acknowledgements

I would like to express my immense gratitude to my advisor, Dr. Marco Carvalho, for giving me a chance in times of difficulty and inspiring me to push on in the face of adversity. I would not be at this point without his support.

My colleagues Evan Stoner and Dhanish Mehta have been instrumental in the creation of this framework. I am proud to have had the chance to work with them and bring this project to fruition.

I would also like to thank many of my friends who have helped me and kept me sane during many late nights working on this thesis. Jigarkumar Rathod and Nima Aghli have been my companions on this journey. I am glad to have met and worked with them. Armando Rollins, Heather Lemieux and Tiziano Bernard have all helped me transform this thesis into a much more readable and understandable document.

Thank you.
This thesis is dedicated to my parents,
Aziz Mammadov and Svetlana Mammadova.

I would not be who and where I am today without their continual love and support.

<3
Chapter 1
Introduction

There are a lot of difficulties when it comes to cybersecurity experimentation and the creation of useful, reproducible and globally shareable results [1]. One such difficulty stems from the pervasively expanding field of computer technology and automation into other fields such as, "energy, transportation, manufacturing, finance, healthcare, economics, human behaviour, and many others" [2]. When it comes to cybersecurity experimentation, two important factors are creating a realistic and lifelike environment and being able to easily create, test and compare user behaviours in the environment. In the past, there have been attempts (e.g. LARIAT [3], DeterLab [4], etc.) at creating a solution for these problems, yet there is still no widely available, accessible and flexible tool for all interested parties. Furthermore, there is currently no tool available to easily create adaptable user-models. By adaptability it is intended that user models can respond to certain conditions in their environment and change the course of their default behaviour. Some organizations have attempted to remedy these problems with specialized software on cyber ranges [32]. A cyber range is a virtualized environment for the training of individuals in the use of cyber tools and activities. However, these cyber ranges are often in a closed ecosystem
which makes it hard or impossible to use on other platforms [5], [4], [6]. In this paper, we will be looking at creating a tool that can help create and execute adaptive models of users' behaviours with a specific use case in non-synthetic network traffic generation.

As an example of one these cyber ranges, researchers at the Florida Institute of Technology have created a miniature version called Virtual Infrastructure for Network Emulation (VINE). VINE is a cloud based testbed that allows for the deployment of a variety of operating system images which can interact with each other over a virtual network. We use VINE to run a variety of projects and experiments where we can test the effectiveness of a certain cybersecurity tool or technique (e.g. moving target defence) [7]. However, to be able to realistically test some of these tools we need to ensure that our testing environment reflects a realistic network topology and non-synthetic background traffic. For this purpose, we have created an adaptive, behaviour tree-based framework called Yoshka. The main focus of this framework is on the capability of adaptability, while maintaining complexity, extensibility, portability and ease of use.

Since non-synthetic(realistic) background traffic is an important aspect of creating a scientifically sound experiment [2], [8], [9], [10], [33], [34] we wanted to make sure that we considered most kinds of existing tools to ensure we have one that reflects our exact requirements. Our requirements include:
• creating and controlling high level concepts to execute a cyber mission
• rapidly creating an easily manageable and reusable user model
• having the user model be able to adapt and respond to its environment
• having the tool be platform agnostic
• having the tool be easily extensible

During our research, we came to recognize that there are basically three types of tools available to researchers in our situation: the first type is a simple (usually low-level) traffic capture tool with replayability and occasional packet crafting support; the second type is a much more complex and comprehensive tool often used to model human cognition and behaviour and the third type is a hybrid of the first two types. None of these solutions matched all our requirements thus having us finally decide to create our own solution.

The reason we decided to come up with our own solution instead of using an already existing tool can be summarized into four factors:

1. *The already existing solutions were too low-level.* Half of the tools [11], [12], [13] we found and reviewed often only supported very simple features and possibilities that did not fit our expectations. They only
allowed for the control of low-level data such as packets or flows whereas we were looking for something more abstract and capable.

2. *The already existing solutions were too complex and did not relate to traffic generation.* Several tools [14], [38] we found had the exact opposite problem; they were designed to model human cognition and behaviour and inherently had more complexity. We were looking for something as powerful (or close to it), yet not as complex. These tools had nothing to do with traffic generation and did not allow for the agility and speed we were looking for.

3. *The already existing solutions were platform restricted.* Some of the other tools [4], [15], [3], [13], [16], [36] we found that were created for cybersecurity experimentation fit our needs in terms of features, but were unfortunately restricted to either a specific platform or provider.

4. *None of the traffic generation tools supported adaptability – being able to dynamically change traffic generation parameters based on changes in the environment.* All the tools [11], [12], [13] that related to traffic generation were too low level to be able to support adaptability.
Chapter 2
Literature Review

During our research, we realized that there are three main types of tools that can help create user models for the purposes of cyber activity generation. The first type is a basic tool that simply generates a specific type of traffic for a specific type of domain [9], [17], [10], [18]. Some varieties of this type of tool [11], [12] include, but are not limited to, packet capture and replay, traffic simulation, traffic emulation or some combination of these. The stages of packet crafting are further discussed in section 2.2. The second type of tool is a much more advanced framework built for cognitive modelling and refers to the theory and structure behind the human mind. The third type is simply some sort of hybrid of the first two types. We also found a number of privately owned solutions that we will mention, but, due to their closed source policy, we cannot perform a proper analysis of their inner workings [5] [6]. We compare the first two types of tools in more detail in section 2.1.

We found over 20 tools that fall into one of these two categories or somewhere in between. The first three that we review fall into the first category of simple traffic generator/simulator type tools. They are tcpreplay a packet traffic capture tool [13], Harpoon a netflow-level traffic generator [11] and Ostinato a network traffic generator and analyzer [12]. The second two
are: Soar [14] and DASH [4]. The last two can be considered a hybrid combination of the two types where a simple traffic generator tool is coupled with a user or application modelling feature. These tools are Swing [16] and RENETO [15].

2.1 Simulation vs. Emulation

When exploring these tools, it is important to distinguish between simulators and emulators. A simulator is a tool that, given an input, uses mathematical modelling or software techniques to generate an expected output based on what the model is supposed to be simulating. An emulator, on the other hand, is a tool that actually performs the actions in between the input and output stages of the process that yield in the desired output. This is important because it preserves the intricacies and details of the actual process going from input to output. In the case of simulators certain side effects of this process are often overlooked thus inherently creating non-authentic synthetic outputs [19], [20]. For example, if one wanted to generate traffic, representative of a typical office worker browsing the internet, all the packets would need to be carefully crafted to simulate the data being sent between client and server. More importantly one would have to manage all the timings between client and server connections and account for any possible mistakes made by the office worker, the connection or the server. All of this is incredibly hard to do, let alone come up with the right parameters to make it realistic. This is a concern for us since we are trying to achieve
realistic background traffic on a network. Therefore, we need Yoshka to be an emulator rather than a simulator.

2.2 Packet Crafting

There are generally four main stages in the packet crafting process: assembly, editing, play and decoding. Assembly refers to the creation of a packet, editing means changing the content of the packet, play/replay means actually sending the packets on the wire and decoding means analyzing and interpreting the packet contents after it has been sent and received [21]. There are several tools online that specialize in each of these stages such as hping3, netdude, tcpreplay, Wireshark, etc.

2.3 Low-level Simulation Tools

Tcpreplay [13] is a collection of GPLv3 licensed tools that run on UNIX and Windows (through Cygwin) and allow for use of captured traffic in the libpcap format. Users can differentiate traffic between client and server, modify network stack headers and replay the captured traffic through any number of devices. With the help of this tool network administrators can probe firewalls on the perimeter for any misconfigured rules and fix other network misconfiguration issues. This tool specifically focuses on the packet play stage of packet crafting meaning that it uses previously captured traffic that can be sent at the same rate or any other user defined rate. The main disadvantages of tcpreplay are the fact that the user is always limited to
previously captured traffic and going through the complicated process of
designing and creating one’s own packets. Re-using previously captured
traffic may not always be the best option since different scenarios and
network topologies will often yield different traffic patterns, whereas
designing and creating packet streams may be what we want, unfortunately,
there is no easy way of creating a large number and variety of these.

Harpoon [11] is a netflow-level tool that enables traffic generation. It
can use data from previously captured netflows to analyse and create
statistically similar models to replicate the original traffic in both temporal and
spatial aspects. This tool is typically used to run background traffic for
application and protocol testing. When analyzing data from previously
captured netflows, Harpoon can differentiate between inter-connection time,
source and destination IP ranges, file size and number of active sessions to
create the statistical models for TCP sessions.

Ostinato [12] is relatively new tool in the packet crafting trade releasing
their first stable version in 2010. Most tools focus on only one of these stages,
but Ostinato aims to be a comprehensive tool that can cover them all. The
first stage of packet assembly refers to the actual creation of the packets that
can be configured with any protocol, flags or other options. After the packets
are created one might want to edit them or edit other previously captured
traffic. Ostinato can change the value of any field of any protocol. The third
stage is actually sending the created or captured packets finally followed by
packet decoding. Although this tool has great support for a variety of protocols and platforms with powerful features at all stages of packet crafting, unfortunately it is too low-level for our use case. It is impractical to use this framework for the design and creation of adaptable user models.

2.4 High-level Cognitive Modelling and Emulation Tools

Soar [14] is a widely-used tool used to simulate human cognition and behaviour. This appeared promising, but we were also aware of the steep learning curve due to the complexity brought about with the rich feature-set. This architecture has modules to support a range of problem solving methods, memory, knowledge and learning about all aspects of tasks and their performance. Using this tool would perhaps yield the most human-like traffic down to specific details such as fatigue and human-like memory. We realized that this level of detail is only useful to us, for background traffic simulations, if it can be achieved in a reasonable timeframe.

DASH [22], [4], [23], [37] is a tool that was built by the Deter Project to help them create, "predictive modelling of human behaviour supporting definition of mental models". This tool seemed like the perfect fit, but upon further investigation we realized that there is no publicly available source code because it only runs and works on their DeterLab cyber range. DASH is based on an agent platform where computers mediate group decision-making. The agents model behaviours using a dual-process cognitive
architecture that represent rational and instinctive behaviours. Using the combination of these modules they can simulate fatigue, cognitive load and time pressure as well as human biases.

2.5 Hybrid Tools

Vishwanath & Vahdat's work on Swing [16] and Geyer et al's work on RENETO [15] can be considered as a hybrid tool of a low-level simulator and a high-level user modelling framework. Swing presents a, "closed-loop, network responsive traffic generator"[16] that builds models of application specific traffic based on captured packets and generates statistically similar live traffic. It attempts to achieve realism by taking into account packet inter-arrival rate, burstiness, size distribution, arrival rate and destination distribution. By analyzing these properties, the authors can create models of specific applications' behaviours. The latter largely presents a similar solution to the same problem, but with a different implementation and on a different platform. RENETO is based on the OMNet++ network simulation framework which is also a packet level tool like Swing. To replicate realism RENETO focuses on IP addresses, ports, timestamps and protocols. The model is then created based on the empirical cumulative distribution function of the parameters that are matched with specific applications. The parameters in these models are static and therefore "not well suited for live capture"[15].
2.6 Proprietary Tools

During our research, we also came across several privately-owned tools that seemed to fit our requirements based on the descriptions. Unfortunately, due to their closed source nature, no proper evaluation of these tools was possible. The tools are: Solarwinds' WAN Killer - a simplistic network traffic generator, Ixia's BreakingPoint [5] - an all-in-one application and security testing platform and Spirent's Avalance NEXT [6] - a tool to "generate realistic enterprise-level and carrier-grade security application traffic for load and functional testing"[6].

In table 2.1 we can quickly see all the tools that we looked at comparing their types and the platforms that they support. In the type column, US indicates user simulation, UE indicates user emulation and H indicates hybrid. In the platform column, “Any” indicates that the tool can be used on any of the three major platforms (Windows, Linux, Mac) either directly (native executable) or indirectly (running in an emulator such as Cygwin or the Linux subsystem on Windows 10). In certain cases, some of the tools are restricted to a specific closed or semi-closed platform like LARIAT or DeterLab.
Table 2.1 A table comparing the types and supported platforms of related works.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Tcpreplay</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>2  Hping</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>3  Ostinato</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>4  Seagull</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>5  Packets</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>6  Harpoon</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>7  Pktgen</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>8  Trafgen</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>9  Poisson traffic generator</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>10 Surge</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>11 Mausezahn</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>12 Soar</td>
<td>UE</td>
<td>Any</td>
</tr>
<tr>
<td>13 DASH</td>
<td>UE</td>
<td>DeterLab</td>
</tr>
<tr>
<td>14 ACT-R</td>
<td>UE</td>
<td>Any</td>
</tr>
<tr>
<td>15 Icarus</td>
<td>UE</td>
<td>Any</td>
</tr>
<tr>
<td>16 GOSMR</td>
<td>UE</td>
<td>LARIAT</td>
</tr>
<tr>
<td>17 Swing</td>
<td>H</td>
<td>ModelNet</td>
</tr>
<tr>
<td>18 RENETO</td>
<td>H</td>
<td>OMNet++</td>
</tr>
<tr>
<td>19 Netspec</td>
<td>UE</td>
<td>Any</td>
</tr>
<tr>
<td>20 AvalancheNEXT</td>
<td>UE</td>
<td>Any</td>
</tr>
<tr>
<td>21 Ixia BreakingPoint</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>22 Solarwinds</td>
<td>US</td>
<td>Any</td>
</tr>
<tr>
<td>23 Skaion</td>
<td>H</td>
<td>Any</td>
</tr>
</tbody>
</table>
Chapter 3
Concept Implementation

Based on our review of existing tools, we propose a new framework that aims to overcome the issues we have encountered while meeting the requirements we outlined. In this section, we explain our choices and design decisions as they relate to the issues of capability, adaptability, portability, extensibility and usability.

3.1 Capability

To achieve the capabilities of adaptability and responsiveness in our user models we, at first, decided to go with a behaviour tree approach. A behaviour tree is a directed acyclic graph mainly used in the video game industry to control non-playable characters (NPCs) [24], [25], [26]. This can be considered as a primitive form of AI that lays out a set of possible steps that the agent can make based on the outcomes of previous steps. We decided to approach this problem from a more abstract level than packets or flows due to our goal of balancing usability and complexity [1], [27]. To simplify this idea even further we decided to combine the concepts of behaviour trees with binary trees to restrict the outcome of each node to only two possible states: success and failure. There are two types of nodes in our framework: TaskNode and CompositeNode which both extend a general
Node class. A TaskNode represents a single atomic action that an agent can make. A CompositeNode is more complex and represents a collection of TaskNodes (represented as a string list argument) with four extra features to introduce some of the key capabilities of our framework. The four features are adaptability, distributions, a success criterion and parallelism which will be discussed later in this chapter. These two Node types can be used as building blocks to create a binary tree like graph that is easy to understand and follow, yet has the potential to model realistic human behaviour. By abstracting any task or set of tasks into a separate Node, we can rapidly create models of users' behaviours without having to worry about the individual packets. Another advantage of this architecture is that behaviour trees are highly reusable and enable composability. Given any behaviour tree, we can reuse a subtree of that original tree in any other user model for easy and rapid composition of behaviours.

The main logic controller of our framework is called the Engine. It is responsible for handling control flow, timing, dynamic variable initialization, thread initialization and more. By default, the Engine uses seconds to manage all timings, but this can be adjusted to minutes or hours through the Engine configuration file. Yoshka supports the use of dynamic variables which means that the output of one Task can be used as the input of another. Since these variables are determined and allocated at runtime, the Engine creates trees with placeholder values that are replaced during behaviour
initialization and before the execution of the individual TaskNode. The five most important classes in our framework can be seen in the block diagram of figure 3.1. This diagram represents the information flow of a behaviour file and its parameters.

![Figure 3.1 A block diagram showing the information flow in the framework.](image)

Both Task and Composite nodes can be executed one or more times and keep track of their status. Based on this status (success or failure) the Engine will direct the execution down the respective behaviour path. A behaviour is what we call the general set of actions that an agent can make. This behaviour is a directed, acyclic graph written as a YAML file that is used as one of the required inputs for the framework. YAML [28] is a type of human readable file format used for data serialization. The main reason we decided to use YAML is its highly readable hierarchical design and clean key to value mapping implementation. Given such a behaviour file and a set of arguments, the file is verified and processed, the Engine is configured and the main execution loop begins. The arguments are discussed in further detail in section 3.2. For example, given a YAML behaviour file as an input, the main entry class Yoshka handles the arguments and then calls the InputHandler class passing on the reference to the behaviour file. This InputHandler class determines what type of file format is used and then calls the appropriate verification class. What this means is that the framework is designed in a way
to be extensible and support other file formats and inputs which is further discussed in section 3.3.

To ensure ease of use of our framework we decided to include a YamlVerifier class to check for syntactic and logical errors when it comes to constructing behaviours. For example, some of the internal constraints on our behaviours specify that every Node should have a unique name, zero or two children, the correct number and type of arguments of Task specified, etc. In case there is a mistake in the behaviour, a descriptive error message is logged letting the user know what part is incorrect. Given the behaviour is correct, it then gets passed on to the YamlReader class which builds an internal model of this behaviour. Internally this model is represented as a hashmap using the TaskNode and CompositeNode classes. The reason we use a hashmap and not a tree is because our behaviours support having multiple parent nodes for reusability. Having finished building this data structure, it finally gets passed on to the Engine that begins the execution. The Engine also makes sure to keep track of Task data such as duration, status and the use of any dynamic variables. After Task execution, the time of completion is measured and compared to the intended duration of that Task. If it completed earlier than it should have, it will wait for the difference. It is also possible to specify a range of values for the duration having the Engine randomly pick a value in that range.
When the Engine is executing these nodes, it checks whether the current Task happens to be an instance of CompositeNode. In this case the control flow is slightly altered to ensure that all the Tasks that are part of the collection are run according to the specified number of threads, success criterion and distribution as described further on. The Engine also keeps a watch, with the help of the WatchService class, on the behaviour file and Engine configuration file as either can be edited live during execution. In case a file is changed, the Engine discards the old information and updates to the new changes.

When using the CompositeNode in behaviours, the user has two options: using the distribution and adaptability features together or the success criterion and parallelism features together. These features are specified by the arguments of the CompositeNode and cannot, currently, all be used at the same time.

The success criterion feature is a pair of integers that are used to determine how the CompositeNode is evaluated as one whole Node. The first integer (min) represents the minimum number of Tasks that can pass for the CompositeNode to succeed and the second integer (max) represents the maximum number of Tasks that can pass for it to succeed. For example, if the user wants to create a CompositeNode with an AND condition for the individual sub-Tasks then s/he would specify both integers to be the same number as there are Tasks contained within the CompositeNode. In general,
both integers must be positive and no larger than the total number of Tasks in the CompositeNode. For another example, if the user wanted the CompositeNode to be evaluated as an XOR (returns success if and only if there is only one successful Task in the collection) then the user specifies the maximum value as 1 and the min value as 0. The Engine automatically figures out this success criterion to return the correct result.

Parallelism is determined by the number of threads argument of the CompositeNode. This argument must be a positive integer that controls the number of threads the thread pool is allowed to instantiate when executing

![Diagram of task structure](image)

*Figure 3.2 An example structure of a CompositeNode using parallelism with the random order flag flipped.*
the Tasks of the CompositeNode. This thread pool uses the ThreadMonitor class to keep track of all the threads and log any key events.

3.1.1 Adaptability

The key differentiating part of our cyber emulation framework is its ability to adapt to the environment as the environment evolves and changes. We realize this feature by implementing a simplistic memory model along with our distribution feature. The distribution feature enables a non-deterministic approach to the execution of Tasks. For example, the user can specify a type of distribution (e.g. uniform, Gaussian, Poisson) in the arguments of a CompositeNode that enforces the selection policy for that Node. The selection policy determines which Task is to be run during the next execution. Given a certain distribution, the selection policy is determined by a random sampling of the selected distribution mapped to all possible Tasks. For example, if a Poisson distribution is selected and the CompositeNode contains three possible Tasks, then depending on the sample value, the corresponding Task will be selected for execution. The Engine keeps track of how many times a particular Task of the CompositeNode fails (what we call the change threshold) and how many subsequent actions have passed since then (what we call forgetfulness). For example, if a CompositeNode contains three TaskNodes that an agent can take and one of them fails more times than the change threshold value, then the distribution based selection
policy is overridden and a subsequent TaskNode is selected for execution. To prevent that TaskNode from never running again we also decided to implement forgetfulness, which is simply modelled as a counter of actions since the last change threshold trigger. We deem this as an important part of our framework primarily because we can study the effectiveness of a certain attack on a network with responsive user-agent behaviours.

The user can also reuse variables between Tasks or use the output of one Task as the input of another. This feature works in tangent with the behaviour tree data structure as the Tasks can become dynamic and respond to changes in the environment they are running in. Our framework makes sure that all nodes in the behaviour tree are context aware in a semi-automatic manner. This means that any Task can utilize a global register to share information with other Tasks in the same behaviour. This works automatically based on manually predetermined keys thus making the process semi-automated.

3.2 Portability

The Yoshka framework is packaged as a Java Archive (.jar), so that it is portable, and requires four mandatory arguments and has the option for five additional arguments. The four mandatory arguments are: frequency (-f), absolute path to the logger configuration file (-l), absolute path to the Engine configuration file (-c) and the absolute path to the behaviour file(s) (-
b). Frequency must be a positive integer that determines how many times the Engine should run the specified behaviour file(s). In case this integer is zero, the behaviour(s) will run forever. The logger configuration file is a requirement imposed by the log4j library used inside the framework. The Engine configuration file lists all the possible configurable parameters used in the Engine, such as: maximum and minimum numbers of threads, time scale, change threshold and forgetfulness of the memory model, etc. Finally, the behaviour file path specifies either the individual behaviour file or a directory of multiple behaviour files to be executed. These multiple behaviour files can be linked and reference one another thus allowing for modular and extensible behaviour design. In the future, we hope to create the capability for Yoshka to internally agglomerate these behaviours files into one large model.

The five optional arguments are seed (-s), verify (-vf), generate (-g), input (-i) and package (-p). “Seed” allows the user to input a value to be used as the seed for all the random number generation in the framework to ensure repeatable results. The “verify” option lets the user verify the correctness of their behaviours in terms of syntax and logic without having to also run them. The last three options provided to the user are tools that must be used in conjunction and allow the user to create custom Tasks more easily and quickly. The “generate” option enables the generation of source files to be finalized by the user, the "input” option describes the necessary and optional
components of the Tasks to be generated and the “package” option specifies the root package name for these Tasks.

### 3.3 Extensibility and Usability

The Task is an abstract class in our Java-based framework that can be used to implement custom actions. We designed the Engine in such a way that the user can use the framework with a set of their own custom Tasks to achieve any sort of functionality supported by Java programming. For example, we created a set of FTP, Web, SQL and SMTP APIs to help us represent users generating traffic on a network. We used these APIs to create a set of specific traffic classes extending the Task class. We have also packaged more generic Tasks into Yoshka to allow the user to emulate offline behaviour as well. These Tasks include file manipulation and executing arbitrary commands in the Linux shell. Thus, any action that can be done with a shell command can be emulated by our framework. Since any class can extend this Task object, our framework can represent any type of behaviour programmable in Java. The only method the user has to implement in this class, when creating custom Tasks, is "run()". This is the main method used to determine what the Task is supposed to do when called by the Engine for execution. As mentioned previously, we chose to use Java to overcome the main limitation of a lot of other frameworks – portability.
Yoshka also supports the automatic generation of these Task source files through the -g, -i and -p options to maintain ease of use. For example, if the user runs the framework with the options “-gp myTasks -i /home/tasks.yml”, then Yoshka will create however many Tasks were specified in the “tasks.yml” file in a new package called “myTasks”. All of these Tasks would already have the correct location and general skeleton generated for the user, leaving them only to implement the aforementioned “run()” method.

As mentioned earlier, we have an InputHandler class to determine the file format of behaviour file input to the framework. This is done in order to support extensibility for other file formats in the future. The class is designed in a way that given the format of the behaviour file is not YAML, the user can either write code to convert the new format into YAML and reuse the Verifier.
and Reader classes or simply write new Verifier and Reader classes for the new format.

The process of user model creation starts with a web based user interface. The UI is composed of two panes that are located side by side as seen in figure 3.3. The first pane contains a list of possible Tasks (sorted by type), templates and previously created behaviours. The second pane contains a simple list based visualization of the behaviour tree. We have also created a new tree based visualization, as seen in figure 3.4, yet to be integrated into the web GUI. The user can create and manage the structure of the tree by simply dragging and dropping the desired Tasks. For example, the user can drag any Task from the first side pane onto the tree canvas to create that node. If that is the first node on the canvas it is automatically assumed to be the root node. Dragging another Task onto an already existing node will create two more nodes that will be assumed as the children of the first. The Task that is dragged on is selected as the success child by default, with a blank Task template for the failure child. During this process one can
click on any of the nodes to edit them and specify any special properties like the ones previously mentioned for CompositeNode as well as some other ones like run-once and duration. Run once is a directive for the Engine to make sure that the Task gets executed only once, even if the whole behaviour is set to run multiple times. This can be useful in situations where a first-time set-up step might be necessary. Duration is another important part of the user-model design process as it directly affects the realism of the whole model. The user can choose any positive integer or range of integers for the duration of any node. If a range of integers are specified (“5-15”), the Engine randomly and inclusively selects a value within the specified bounds. This value must not be less than 0.
At the end of the process, the user can choose to save the current behaviour as a template for future reuse and/or export it as a YAML file for Yoshka. It is also possible to create these user model behaviour files directly through a text editor, if one so chooses.

3.4 Yoshka Dependencies

The Yoshka framework has several Java library dependencies that it requires to function properly. These dependencies are:

- TrafficGen APIs
- Log4j2
- SnakeYAML
- TestNG
- Apache Commons CLI
- Apache Commons Math
- Apache Commons Lang
- JSON
- Freemarker

The traffic generation APIs let us easily generate HTTP, FTP, Git, SMTP and SQL traffic, the Log4j2 logging library, the SnakeYAML library for parsing YAML files into Java code, the TestNG framework to aid in testing of Java code, the Apache commons CLI framework to aid in the creation and management of command line options, the Apache commons Math
framework to help create and manage distributions, the Apache Commons Lang package for auxiliary data structures, the JSON library to help parse JSON data into Java code and the Freemarker library to help generate source files.

To manage these dependencies, we use Maven. Maven is a dependency and package management tool used with Java code. While it is not strictly necessary to use with Yoshka, it is highly recommended due to the tedious nature of dependency and package management of Java code using standard Java commands.
Chapter 4
Illustrative Scenario

We designed this framework to be portable and extensible enough to be used in a variety of use-cases, but our primary purpose for it is the generation of non-synthetic background network traffic. We used some Apache libraries to help us write the APIs we could use in the framework to create a realistic traffic generator. These APIs make use of the libraries to make direct calls to the services we want to replicate (SMTP, FTP, SSH, etc.) thus ensuring the authenticity of the traffic produced through emulation rather than simulation. By making calls to these services that establish real connections between servers and clients using real protocols, we can guarantee that any intricacies of the traffic produced is natural and not an artifact of the generation process.

4.1 Scenario Design

To illustrate the proposed framework, we introduce an example of how an experiment may be constructed. We use the framework to design the scenario and then provide ways to evaluate it.

Consider a scenario where there is a testbed, created on a virtualized cloud provider, and provisioned with the services and routes to represent a
software development company. There are also adversaries present in this scenario causing service interruption to the company.

This testbed has multiple domains representing various departments in the company and an external domain which represents the Internet, where the customers of the company are located.

The services that are required for the scenario are described below and depicted in Figure 4.1:

1. DNS Server: A server to resolve names to IP addresses. There can be one or there can be many interconnected.
3. FTP Server: A proftpd server to allow the storage of files on the server that others can access using the file transfer protocol.
4. Database Server: A MySQL server to allow the maintenance of various records.
5. Web Server: An Apache web server to emulate the company’s external and internal web services.
6. Git Server: A version control server for the developers to maintain their code.
Figure 4.1 A diagram representing the sample enterprise network we have designed for this scenario.

It is at the user’s discretion to decide on the number of each type of server or in which domain they are to be located. The assumptions that we make are that the DNS will be able to resolve the requests and the resolved addresses will be reachable.

We then begin defining our user models for various domains which would emulate employees of the company.

We have developed two ways to create these behaviors:

1. Web GUI. Using a drag & drop web GUI where the user can drag various nodes and provide configurations for the nodes and build a complex behavior tree from it.
2. YAML file. The user can also choose to write their behaviours in YAML (an example will be shown below) which allows for more control and gives access to more features of Yoshka.

We will now demonstrate the creation of a simple behavior which would check the web service and if it can't access it, send an email to tech support for help. The first action is designed to take 10 seconds. The parameters of some of the tasks are specified in angle brackets to indicate that they can be dynamically changed before or after deploy time.

```
- name: Visit web service
  task: web.GetRequestTask
  args:
    name: Visit web
    url: <web_server>
    success: returnSuccess
    failure: send mail to IT
    root: true
    duration: 10

- name: send mail to IT
  task: mail.SendMailTask
  args:
    name: send mail
    mailServer:<mail_server>
    from: <user>@<mail>
    subject: Can't access web service
    message: Web service is giving bad requests
    recipients: <user>@<mail_server>

- name: returnSuccess
  task: generics.ReturnSuccessTask
  args:
    name: success
```

*Figure 4.2 A picture of a sample behaviour YAML file.*
Above is a simple example demonstrating how one would define a behavior of some user or agent. It is also possible to create a group of behaviors which can be linked to each other just like individual Tasks for simpler and easier behavior creation, management, maintenance and reuse. A group of behaviors can constitute some sort of broader mission or goal when executed together.

For instance, in the case of the aforementioned software development company, we model customers who collaborate with the company designers to create a new product. Once the software development company’s product is finalized, the developers upload all their documents to the FTP server and let the engineering team know about the new product and where the documents are uploaded in the FTP server using an email message. One example of such a behaviour we designed can be seen in figure 4.3. In this behaviour the user model performs three main actions in sequence which are sending an email, committing some code and creating a bug ticket. The engineering team then starts developing the new product and keeps the design team in the loop through email. The development of features is emulated by randomly uploading a file to the Git server. Once the product is developed, the engineering team lets the customer support team know about the new product, via email, and lets the technical operations team deploy the new code. The customers can then visit the web server and file any bug reports or issues with technical support. Along with this, there would be an
accounting department issuing paychecks to all employees periodically. If an employee is not paid, then that employee would stop working jeopardizing the workflow of the software development company.

We assume that the users of Yoshka will have some type of virtualization provider setup along with all the instances having SSH installed on them for easier management and automation purposes.
Figure 4.3 An example behaviour for a software developer in the software development company.
4.2 Scenario Tools & Implementation

For the backend of the scenario we are using OpenStack and our cyber range interface VINE. As mentioned previously, the Virtual Infrastructure for Network Emulation is an interface to the OpenStack backend that lets us manage, deploy and configure various virtual machines and the connections between them. In our case, all the machines use the Ubuntu 14.04 image.

In this case, we import our testbed which contains the various domains, address spaces and instances within each domain. Once the testbed is up and running, we need to be able to provision the network and tweak any additional network configurations. To achieve this goal, we use a tool called Ansible.

Ansible [29] is an IT automation tool used to easily control a large number of machines - either virtual or physical. Ansible uses a script called a playbook to leverage control over machines. Given a large testbed, we use Ansible playbooks to provision the entire network and setup networking and the various services required for our scenario. The Ansible playbooks also setup an ELK stack instance to aid in emulation evaluation. We have created playbooks to deploy Yoshka along with the user models to the client machines. The playbooks install Yoshka as an Upstart service on the instances. Upstart is a tool that comes packaged with Ubuntu 14.04. It is, “an
event-based daemon which handles starting of tasks and services during boot, stopping them during shutdown and supervising them while the system is running” [30]. If not for Ansible and Upstart, we would have to manually perform these steps. We made sure to make Yoshka independent of these tools for the sake of portability. Neither Ansible or Upstart are required to use Yoshka.

The previously mentioned ELK stack refers to the Elasticsearch, Logstash and Kibana suite of software [31]. Elasticsearch is a, “distributed, RESTful search and analytics engine … that stores your data” [31]. Using this tool one can easily search and query the necessary data. Logstash is an, “open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to your favorite ‘stash’” [31]. Kibana lets the user visualize the Elasticsearch data and navigate the Elastic Stack. Once again, we use this suite to help us visualize the events generated by Yoshka. While the ELK stack is not necessary for Yoshka to run, we use it to help us better organize and interpret the data produced.
4.3 Scenario Evaluation

For logging purposes, we use the Log4j2 library which comes with a feature to forward log events to Logstash through the logging configuration file. We set up an ELK stack on one of the machines of the software enterprise network to be able to visualize the data. The IP address of that machine is what gets defined in the logging configuration file for the socket connection. The framework keeps track of all Tasks and their metadata such as status, duration, etc. After the log data is sent to the ELK machine, we

Figure 4.4 A screenshot of the scenario running being visualized in Kibana.
need to set up a few filters to be able to parse the logs and retrieve the data we need. Logstash then automatically uses these filters to parse the data and save it to be queried by Elasticsearch. These events can be then visualized using Kibana dashboards as depicted below.

As we can see from figure 4.4, there is a dashboard that takes various information from the framework and depicts it in appropriate graphs. On the left side is several line graphs that each show the status of a host running their respective behaviour. The green line indicates a successful Task status and a red line indicates a failed Task status over time. On the right side, we have a large donut graph representing the total number of Tasks and their proportionate frequency on the network. The bigger the slice, the more times that Task has been run. This graph is depicted in figure 4.5. As we can see, the two Tasks that are run most often are MailLoginTask and

![Figure 4.5 A donut graph showing the proportion of all Tasks executed on the network.](image)
MailLogoutTask. Right under this graph is a number of line graphs representing the health of the services on the network (mail, FTP, etc.) At the very bottom are several smaller donut graphs each belonging to a separate user model. The slices in the donut graph represent the proportions of different Tasks that are being executed in those behaviours. Under these graphs is a table listing the same information in text format.

During the scenario, we simulate an attack on the network by disabling the Mail server, thus causing all the dependent user models to start failing. This can be seen in figure 4.6. As seen from the graphs, all the user models dependent on the Mail server start indicating failure on their line graphs on the left. The services health graphs are mostly the same, except for the Mail server line graph which shows a fail state. Due to this change, the proportion of Tasks executed on the network also changes because of the way the user

![Figure 4.6 A donut graph showing the proportion of all Tasks executed on the network after the attack.](image-url)
models are designed. No longer are ReadMailTask, MailLoginTask and MailLogoutTask dominant on the donut graph as seen in figure 4.7.

Thanks to the log files produced by Yoshka and the visualizations produced by Kibana, we can verify the intentions of our user models and evaluate the scenario as a whole. Using this framework, one can design any sort of mission that involves cyber activity and then evaluate its success or failure independently, or in conjunction with other tools.

To demonstrate the adaptability feature we have set up a simple user model consisting of four actions in a different scenario from the software development company. We construct this behaviour as is seen in figures 4.8, 4.9 and 4.10. Figure 4.8 is the actual behaviour file that we use as the input
for Yoshka. Figure 4.10 is graphical representation of the same behaviour prior to adaptability being triggered and figure 4.9 shows the execution path after adaptability is triggered.

```
- name: AdaptableNode
task: core.CompositeNode
args:
  - name: adaptableNode
    nodes: executeCommand, readFile, check
distribution: poisson
root: true

- name: CheckProcesses
task: generics.ExecuteCommandTask
args:
  - name: command
    command: ps aux | grep yoshka
duration: 5-15

- name: ReadFile
task: generics.ReadFileTask
register: true
args:
  - name: read
    file: ./src/main/testResources/distribution.txt
duration: 5-15

- name: Echo
task: generics.ExecuteCommandTask
args:
  - name: check
    command: echo "$read.file.contents"
duration: 5-15
```

*Figure 4.8 The behaviour we designed to showcase adaptability.*

The root node of the behaviour is the adaptable node which uses the Poisson distribution for its selection policy. This node contains three other Tasks: executing a Linux command to check running processes, reading a nonexistent file and executing another Linux command to echo a message. We set the change threshold to 3 failures and the forgetfulness threshold to 5 iterations in our Engine properties. This means that given a Task in an
Adaptable node, if one of them fails three times in a row, then that Task is guaranteed not to run for at least 5 following iterations, even if it selected for execution by the distribution. We can see this behaviour in the logs produced as seen in Appendix A and simplified in figure 4.11.

From figure 4.11 we can see that ReadFile task is selected for execution in the first step and fails since the required file is not found. We remove the file on purpose to induce failure and showcase this feature. After a few more iterations, the ReadFile task gets selected for execution for the fourth time and fails again, but this time triggering the change threshold value as seen indicated in blue right before step #12. When the distribution
selection policy selects the ReadFile task for execution again, it gets overridden by the adaptability feature and the next Task is selected for execution instead – Echo as seen in step #12. After five more iterations, the forgetfulness feature gets triggered and the ReadFile task is available for execution once again. This can be seen in the second blue line right after which the ReadFile task is indeed selected for execution and is actually executed.
Step #1 Running node - name: readFile, task: generics.ReadFileTask, args: {name=readFile, file=./src/main/testResources/distribution.txt}, duration: 515,
Step #6 Running node - name: readFile, task: generics.ReadFileTask, args: {name=readFile, file=./src/main/testResources/distribution.txt}, duration: 515,
Step #12 Running node - name: echo, task: generics.ExecuteCommandTask, args: {name=echo, command=echo message}, duration: 5-15,
Step #18 Running node - name: readFile, task: generics.ReadFileTask, args: {name=readFile, file=./src/main/testResources/distribution.txt}, duration: 515,

Figure 4.11 Framework log files simplified and colourized to showcase adaptability.
Chapter 5
Informal Qualitative Evaluation

To help us evaluate our framework in a qualitative way we decided to conduct an informal study to assess the framework’s ease of use in terms of speed, capability, extensibility and feature set. Testing this framework in terms of the realism of the network traffic would be invalid since at that point we would be evaluating the ability of the user to create user models rather than the capabilities of the framework by itself. It is also difficult to compare performance figures with other similar tools since there is no other tool that has the same capabilities as Yoshka out of the box. Therefore, we created a table to qualitatively compare Yoshka with one low level traffic generation tool and one high level cognitive modelling tool. The two pieces of software that we will be comparing Yoshka against are Ostinato and Soar respectively.

5.1 Qualitative Scenario Design

To help us make these comparisons and evaluation we designed a simple scenario for a group of volunteers to complete using each of the three tools. The scenario involves the volunteer to do the following stages:

1. Create and implement a predetermined set of Actions that the user model should be able to make.
2. Create and implement the logical structure required to represent the mission.

3. Execute, if necessary, debug the mission and collect results.

4. Answer a questionnaire regarding the experience.

The predetermined set of Actions necessary to be created are creating a text file, sending an HTTP GET request, writing to a text file and indicating mission success or failure. The mission to be executed is a simple conditional sequence of actions that starts with creating a file, on success, sending that file or writing to that file and on failure failing the whole mission. This mission is depicted in figure 5.1.

![Figure 5.1 A tree diagram showing the example behaviour to be implemented by the volunteer using each tool.](image)

The Adaptable Task node in this diagram is supposed to indicate that the user model should also be able to adapt to failing conditions in the environment. In this case on success of creating a file, the model should
attempt to do the default task of sending a “GET” request. If this action fails, then, instead of failing the mission straight away, the model should adapt to this failure and attempt to write to the file instead. If and only if both the default and alternative tasks fail, then the model can fail the mission; otherwise the model is successful.

The implementation and execution part of each stage of the scenario is up to the volunteer using one of the three tools and its documentation. The questionnaire is designed to evaluate the following aspects of each of the three tools compared with each other: portability, extensibility, capability, ease of use, learning curve and completeness. The full questionnaire can be seen in Appendix B.

During the scenario, we keep track of how long it takes the volunteer to complete each stage and the whole scenario itself, as well as the difficulty experienced by the volunteer during each step. To measure these two aspects, we used a stopwatch and a subjective scale from difficult (1) to easy (5) to rate the volunteer’s experience respectively.

The volunteer is given up to fifteen minutes, prior to using each tool, to familiarize oneself with the tool's environment and documentation. Following this period, the volunteer has a total of thirty minutes to complete the entire mission. To help the volunteer figure out what s/he needs to do, a small instruction sheet labelled “goals” is provided (Appendix C). This instruction sheet lists all the aforementioned stages to the volunteer with brief
explanations regarding some of the required capabilities, such as adaptability and a diagram showing the logical structure of the mission. The volunteers are also permitted to use any other documentation or help they can find online. This whole process and study is explained to the volunteer using a standard script (Appendix D).

We chose the 15 and 30 minute timings as a reasonable amount of time, tending to the lower bound, to complete a scenario of this level of complexity. Most users who use these tools would like to be able to complete their goals in as little time as possible. We assume that most users would not want to spend more than an hour on any one task and thus decided that 45 total minutes (maximum) would be an acceptable amount of time to complete a task with this level of complexity.

5.2 Questionnaire Results

Having interviewed five colleagues with varying levels of experience in general software development skills, we got the following results.

Table 5.1 A table comparing the average times of volunteers for the mission implementation using different tools.

<table>
<thead>
<tr>
<th></th>
<th>Yoshka</th>
<th>Ostinato</th>
<th>Soar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing Tasks</td>
<td>16m37s</td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>Implementing Mission Structure</td>
<td>6m02s</td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>Execution and Debugging</td>
<td>6m11s</td>
<td>DNF</td>
<td>DNF</td>
</tr>
</tbody>
</table>
Table 5.2 A table showing the average difficulty scores given for each tool.

<table>
<thead>
<tr>
<th></th>
<th>Yoshka</th>
<th>Ostinato</th>
<th>Soar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing Tasks</td>
<td>3</td>
<td>1</td>
<td>1.6</td>
</tr>
<tr>
<td>Implementing Mission Structure</td>
<td>4.6</td>
<td>x</td>
<td>1.2</td>
</tr>
<tr>
<td>Execution and Debugging</td>
<td>4.8</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

As we can see in table 5.1, none of the participants were able to finish the task in time when using either Ostinato or Soar and only one participant was not able to finish the task with Yoshka. The two former tools also received the lowest scores on the 1-5 difficulty scale where applicable (1/5, 1.6/5, 1.2/5). Yoshka, on the other hand, has an average time of 16 minutes and 37 seconds for implementing Tasks, and around 6 minutes for implementing the mission structure and the execution and debugging stages of use each. None of the volunteers chose to use the code generation method for implementing the first stage. It was later discovered through additional comments, that they felt uncomfortable risking to use that feature due to the lack of documentation. Compared to the other tools Yoshka also received higher scores in terms of ease of use and understanding; a medium score of difficulty (3) for implementing tasks and an almost maximal score (4.6 and 4.8) for the other two stages of use.
5.3 Results Analysis

As we can see in Table 5.1 Yoshka is the only tool that the volunteers could successfully complete the mission with in the allotted timeframe. Despite Soar’s large documentation pool and resources, the complexity of the architecture made it very difficult and seemingly impossible to complete the task in time. The comments provided by the volunteers indicated that there was not enough time to learn how to use Soar and apply that knowledge to the problem. In Ostinato’s case, there was a lot less documentation, but also a lot less complexity and capability. The volunteers were unable to finish the task with this tool because it lacked the capability of supporting higher level abstract concepts such as tasks, and missions. The general feedback from volunteers favoured Yoshka compared to the other tools as expressed in the difficulty scores and some of the additional comments sections provided on the questionnaire. Looking at the difficulty scores, we see that the volunteers found implementing tasks to be the most difficult part of the process. The other two stages of the process (especially execution and debugging) were deemed trivial as indicated by the almost perfect difficulty score of 4.8/5 and the additional comments.

Based on these results and the comments from the volunteers we constructed the following table to help us compare the capabilities, ease of use, extensibility, portability and completeness of each tool.
As we can see from table 5.3, we were able to achieve all of the design goals we envisioned for Yoshka so far. There is much room for improvement, as we will discuss in the next chapter, but compared to the two closest tools we can find in the field, Yoshka is either more capable and adaptable or easier to use in terms of speed. Some other improvements to the framework were identified thanks to the additional comments from the volunteers. These comments mostly revolved around the lack of a graphical user interface for the entire process, especially evaluation during execution, and some issues with documentation.

* It is important to note that even though Ostinato is available on all three major platforms (Windows, Mac, Linux) it is free only on Linux, whereas Yoshka and Soar are free on all three platforms.

<table>
<thead>
<tr>
<th></th>
<th>Yoshka</th>
<th>Ostinato</th>
<th>Soar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptability</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Extensibility</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Portability</td>
<td>✔️</td>
<td>✔️*</td>
<td>✔️</td>
</tr>
<tr>
<td>Completeness</td>
<td>User modelling and traffic emulation</td>
<td>Traffic simulation only</td>
<td>User modelling only</td>
</tr>
</tbody>
</table>
By completeness we mean to evaluate whether each tool has the capability for cyber emulation “out of the box”. Yoshka is the only tool out of these three to support this feature. Ostinato only supports traffic generation though simulation and Soar only support general cognitive modelling with no built-in functionality for cyber emulation capability “out of the box”.

Yoshka bridges the gap between simple traffic simulators and complex cognitive modelling frameworks and thus requires a comparison of different capabilities and features using different tools. Compared to Ostinato, and other low-level simulation type tools, Yoshka can support the same general functionality of network traffic generation via emulation, but with extra capability such as adaptability and the expressivity leveraged by higher level concepts and constructs. Compared to Soar, and other high-level cognitive modelling type tools, Yoshka supports capability such as adaptability, but also makes it much easier to be used, especially in rapid prototyping type tasks.

5.3 Comparison of Mission Implementation

To help further compare the three tools, we have implemented some of the key features of the mission (as much as possible) using each tool. We will begin with the implementation of the mission in Yoshka, followed by Ostinato and ending with Soar.
5.3.1 Implementation in Yoshka

Implementing all three stages of the goals document is designed to be simple and streamlined in Yoshka. The user is provided with a few pages of documentation and examples for various use cases. For the first stage of implementing a Task in Yoshka the user has two options: use the documentation and examples to write the necessary code by hand, or use the -gip options to generate most of the necessary code and only implement the run() method by hand. The sample code provided to the user can be seen in figure 5.2.

For this scenario, the user has several choices about how to complete the first stage of the scenario. Yoshka comes prepackaged with Tasks to indicate mission success and failure, as well as Tasks for HTTP traffic generation located in the web package. Given the fact that it also supports arbitrary Linux commands, the volunteers can use the generic ExecuteCommandTask to complete the remaining file manipulation actions required in the first stage. Optionally the user can also choose to implement the two file manipulation Tasks as separate classes to make them more readable and reusable.
public class SampleTask extends Task {
    private final static Set<String> requiredKeys = Stream.of("name", "filename").collect(Collectors.toSet());
    private final static Set<String> optionalKeys = Collections.emptySet();
    private String filename;
    private String message;

    /**
     * Default constructor.
     */
    public SampleTask() {}

    /**
     * Real constructor.
     * @param parameters to set.
     */
    public SampleTask(final Map<String, Object> parameters){
        this.name = (String) parameters.get("name");
        this.filename = (String) parameters.get("filename");
        this.message = (String) parameters.get("message");
    }

    /**
     * Execute task.
     * @return status of task.
     */
    @Override
    public boolean run() {
        File file = new File(filename);
        try {
            if (file.createNewFile()){
                return true;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    /**
     * Getter for required keyset.
     * @return Set of strings.
     */
    @Override
    public Set<String> getRequiredKeys() {
        return requiredKeys;
    }

    /**
     * Getter for optional keyset.
     * @return Set of strings.
     */
    @Override
    public Set<String> getOptionalKeys() {
        return optionalKeys;
    }
}

Figure 5.2 A snippet of code showcasing the sample Yoshka Task that was provided to the volunteer.
For the second stage of the scenario the user needs to implement the logical structure of the mission. This is easily achieved using the behaviour files in Yoshka. Once again the user has the option between using the web GUI or simply using a text editor. However, since the web GUI does not yet support advanced CompositeNode features like adaptability, the user must resort to using the text editor to be able to complete this stage. It is still possible to use the GUI to create an initial general, logical structure as specified in the goals diagram (figure 5.1) and then export the YAML file to edit by hand to reflect the adaptability requirements. This behaviour YAML file looks like the previously shown example in Chapter 4, figure 4.1.

For the third and final stage the user needs to execute and, if necessary, debug the Task and behaviour files. Before the user can run the code, it is necessary to repackage Yoshka into a Java archive file (.jar). This can be easily done with the help of Maven. To do this we run the following command: “maven package -P uber” in the same directory as the project. The -P option indicates to use the uber profile already provided in the pom.xml file of the project. After the packaging is complete the user can finally execute the user model using the following command: “java -jar <path to Yoshka.jar> -c <path to Yoshka Engine configuration file> -b <path to behaviour file> -l <path to logging configuration file> -f <times to run>. All the default configuration files are provided with the Yoshka framework.
5.3.2 Implementation in Ostinato

Implementing any of the stages of the mission in Ostinato proved to be practically impossible. This is primarily because Ostinato does not support any higher-level concepts such as abstract tasks, missions, adaptability or logical control flow structure. This is simply a traffic generation tool that one can use to simulate packets.

To generate traffic, Ostinato provides the user with a graphical user interface for packet crafting which can be seen in figure 5.3. To start generating traffic the user needs to select a port in the left pane of the window.
and then create a stream for that port in the right pane of the window. After
the stream is created it can be edited by double clicking it. In the editing
window that comes up, the user can select which protocol to use for the first
four layers of the network stack and if there should be a payload. The options
for the first layer are MAC or none. The options for the second layer are
Ethernet II, 802.3 raw, 802.3 LLC, 802.3 LLC SNAP and none. The options
for the third layer are IPv4, ARP and none. The options for the fourth layer
are TCP, UDP, ICMP and none. After the user is done selecting the protocols,
he can switch to the protocol data tab to edit the parameters of the selected
protocols. In case of IPv4, the user can edit the source IP and port as well as
any flags and payload.

In the next tab, called stream control, the user can choose between
sending packets or bursts and configuring the number of packets or bursts.
Finally, in the packet view tab, the user can review what the end result
packets would look like. If everything is okay the user can press the Ok button
and must press the Apply button in the top right of the original window.

At this point, the user can select the required port group in the list in
the bottom of the window and use the seven buttons right above to control
the packet generation. The last magnifying glass button interfaces with
WireShark to view the generated packets.
5.3.3 Implementation in Soar

Soar is the only other tool in this comparison that shares the general capabilities with Yoshka. However, the biggest problem with Soar is the fact that it is not designed for cyber emulation specifically and thus poses the issue of “incompleteness”. It is not possible to use Soar to complete this mission “out of the box”, in a reasonable amount of time, and requires a lot of additional work to achieve the same goals.

Figure 5.4 shows a screenshot of the Soar graphical user interface. On the top, we have a typical row of buttons for general menu control. On the left is a pane to
indicate the current files in the project, the right pane contains the contents of the files listed and bottom pane displays the output of the program after it is executed.

To implement the first stage of the scenario the user would have to create rules to emulate the necessary actions. A sample Soar rule can be seen in figure 5.5. Every Soar rule must start with “sp (soar production)” and the body of the rule must be encapsulated in curly braces “{ }”. The name of the rule is the first text after the opening curly brace. In this case the name is “hello-world”. On the next line, we have the condition to be evaluated during rule execution followed by the action to be made if the condition passes (separated by the arrow). There can be several conditions and actions specified on subsequent lines as seen in this rule. This rule is supposed to evaluate whether itself exists and if that is true, print “Hello World” to the screen and then halt.

```plaintext
sp {hello-world
   (state <s> ^type state)
  -->
   (write |Hello World|)
   (halt) }
```

*Figure 5.5 A snippet of Soar code showing an example Hello World rule.*

Using such rules the user can construct more complex behaviours that have an inherent logical structure that needs to be implemented in the second
stage of the scenario. To execute and debug these rules, Soar provides the user with a separate tool called the Soar Visual Debugger as seen in figure 5.6.

![Soar Visual Debugger](image)

Figure 5.6 A screenshot of the Soar Visual Debugger user interface.

To run the user model built with soar productions in the previous user interface, the user needs to load that file into the Debugger tool and press the "Run" button located in the bottom left cluster of the window. The user is also able to insert breakpoints and step through the soar productions to find any issues. As we can see in figure 5.6, we run the hello-world rule discussed previously and can see the output on the left pane of the window. It sources
the production, runs it, evaluates the condition, condition succeeds and goes
to run the actions of printing the message and halting.
Chapter 6
Recommendations

The current work flow for deploying behaviours to machines on a testbed is rather convoluted and problematic. We have two separate parts of the system that takes a human in the loop to put together. On one hand, there are the abstract behaviours that should be able to work on any testbed, on the other hand there is the actual network topology and any services that the behaviours might need. The process of merging these two ends to work together is the problematic part that currently requires a tedious amount of human intervention. The process of creating a testbed and provisioning it takes an impractical amount of time and is bound by human errors. With the above described process we intend to make it easier by abstracting creating and provisioning a testbed based on the behaviors defined. This would allow the user to concentrate on the emulations required for their experiment instead of configuring and provisioning the network. We have started looking into how to automate this process and are in the middle of developing such a tool. We are planning to leverage the power of ontologies to help us discover the requirements for experiments and glue the disparate parts together.
Another way we want to streamline the process of creating behaviours is by improving the user interface of the web portal. Currently it displays the behaviour graph as a file tree structure. Instead we would like to display this as an actual tree visualization. We are currently working on another visualization tool to help us track the execution of behaviours in real time. This visualization depicts the user models in the tree graph that we desire. We plan to reconcile the two parts together and make use of this already existing visualization in the web portal.

Another feature we have considered adding to the framework is generating statistically similar behaviour models based on captured traffic data. We have briefly looked into this in the past, but have not made any significant headway. In the future, we hope to add this feature to the core of our framework with support for JSON traffic data that can be transformed into behaviour tree-based user models.
In this paper, we introduced a new adaptive framework for the support of cyber experimentation emulation. First, we surveyed over 20 tools from both academia and industry, and reviewed a handful to showcase why we chose to create our own tool. Next, we laid out the individual components of our framework, our reasoning behind our design choices and how all these parts worked together. Unlike most existing solutions, this framework is adaptable to the changing conditions of an environment, portable across platforms and extensible. We explained how our decisions are motivated by our focus on the balance between capabilities of user modelling and emulation of network traffic. We showcased an example implementation and scenario of using this tool. Finally, we presented an informal qualitative evaluation of other similar tools. In future work, we hope to further automate this process and make behaviour design even easier, especially for those who are not programmatically inclined.
Bibliography


    of the Future (CEF): Catalyzing a New Generation of Experimental 

[3] "MIT Lincoln Laboratory: Cyber Security and Information Sciences: 
    Cyber System Assessments: Projects," Massachusetts Institute of 
    Technology, [Online]. Available: 
    [Accessed August 2016].

    Sciences Institute, 2014. [Online]. Available: 

    2016].


[38] D. D. Salvucci and F. J. Lee, "Simple cognitive modeling in a complex
cognitive architecture," in *SIGCHI Conference on Human Factors in
Appendix A

23:06:40.653    edu.fit.hiai.yoshka.core.Engine - Step #1 Running
node - name: readFile, task: generics.ReadFileTask, args:
{name=readFile, file=distribution.txt}, duration: 5-15,
23:06:40.654    edu.fit.hiai.yoshka.tasks.generics.ReadFileTask -
ReadFileTask.run: File distribution.txt not found
23:06:40.654    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
readFile className = ReadFileTask status = false
23:06:52.650    edu.fit.hiai.yoshka.core.Engine - Processing time:
4, Duration: 12000
23:06:52.651    edu.fit.hiai.yoshka.core.Engine - Step #2 Running
node - name: readFile, task: generics.ReadFileTask, args:
{name=readFile, file=distribution.txt}, duration: 5-15,
23:06:52.651    edu.fit.hiai.yoshka.tasks.generics.ReadFileTask -
ReadFileTask.run: File distribution.txt not found
23:06:52.651    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
readFile className = ReadFileTask status = false
23:07:07.652    edu.fit.hiai.yoshka.core.Engine - Processing time:
0, Duration: 15000
23:07:07.652    edu.fit.hiai.yoshka.core.Engine - Step #3 Running
node - name: readFile, task: generics.ReadFileTask, args:
{name=readFile, file=distribution.txt}, duration: 5-15,
23:07:07.652    edu.fit.hiai.yoshka.tasks.generics.ReadFileTask -
ReadFileTask.run: File distribution.txt not found
23:07:07.652    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
readFile className = ReadFileTask status = false
23:07:12.653    edu.fit.hiai.yoshka.core.Engine - Processing time:
0, Duration: 5000
23:07:12.653    edu.fit.hiai.yoshka.core.Engine - Step #4 Running
node - name: echo, task: generics.ExecuteCommandTask, args:
{name=echo, command=echo message}, duration: 5-15,
echo className = ExecuteCommandTask status = true
23:07:22.653    edu.fit.hiai.yoshka.core.Engine - Processing time:
7, Duration: 10000
23:07:22.656    edu.fit.hiai.yoshka.core.Engine - Step #5 Running
node - name: echo, task: generics.ExecuteCommandTask, args:
{name=echo, command=echo message}, duration: 5-15,
23:07:22.659    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
echo className = ExecuteCommandTask status = true
23:07:29.656    edu.fit.hiai.yoshka.core.Engine - Processing time:
3, Duration: 7000
23:07:29.656    edu.fit.hiai.yoshka.core.Engine - Step #6 Running
node - name: readFile, task: generics.ReadFileTask, args:
{name=readFile, file=distribution.txt}, duration: 5-15,
23:07:29.657    edu.fit.hiai.yoshka.tasks.generics.ReadFileTask -
ReadFileTask.run: File distribution.txt not found
23:07:29.657    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
readFile className = ReadFileTask status = false
23:07:44.656    edu.fit.hiai.yoshka.core.Engine - Processing time:
1, Duration: 15000
23:07:44.657    edu.fit.hiai.yoshka.core.Engine - Step #7 Running node - name: echo, task: generics.ExecuteCommandTask, args: {name=echo, command=echo message}, duration: 5-15,
23:07:44.659    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:07:49.656    edu.fit.hiai.yoshka.core.Engine - Processing time: 3, Duration: 5000
23:07:49.658    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:07:58.660    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:08:09.658    edu.fit.hiai.yoshka.core.Engine - Processing time: 1, Duration: 9000
23:08:09.658    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:08:24.690    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:08:36.659    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:08:36.659    edu.fit.hiai.yoshka.core.Engine - Processing time: 3, Duration: 13000
23:08:36.659    edu.fit.hiai.yoshka.core.Engine - hostname = icis19
23:09:03.660    edu.fit.hiai.yoshka.core.Engine - Processing time: 3, Duration: 13000
23:09:03.660  edu.fit.hiai.yoshka.core.Engine - Step #14 Running
node - name: echo, task: generics.ExecuteCommandTask, args:
{name=echo, command=echo message}, duration: 5-15,
23:09:03.663  edu.fit.hiai.yoshka.core.Engine - hostname = icis19
echo className = ExecuteCommandTask status = true
23:09:15.660  edu.fit.hiai.yoshka.core.Engine - Processing time:
3, Duration: 12000
23:09:15.661  edu.fit.hiai.yoshka.core.Engine - Step #15 Running
node - name: checkProcesses, task: generics.ExecuteCommandTask,
args: {name=checkProcesses, command=ps aux | grep yoshka},
duration: 5-15,
23:09:15.689  edu.fit.hiai.yoshka.core.Engine - hostname = icis19
checkProcesses className = ExecuteCommandTask status = true
23:09:27.661  edu.fit.hiai.yoshka.core.Engine - Processing time:
28, Duration: 12000
23:09:27.662  edu.fit.hiai.yoshka.core.Engine - Change threshold
triggered. Overriding distribution selection policy to execute next
Task instead.
23:09:27.662  edu.fit.hiai.yoshka.core.Engine - Step #16 Running
node - name: echo, task: generics.ExecuteCommandTask, args:
{name=echo, command=echo message}, duration: 5-15,
23:09:27.664  edu.fit.hiai.yoshka.core.Engine - hostname = icis19
echo className = ExecuteCommandTask status = true
23:09:39.663  edu.fit.hiai.yoshka.core.Engine - Processing time:
2, Duration: 12000
23:09:39.663  edu.fit.hiai.yoshka.core.Engine - Step #17 Running
node - name: echo, task: generics.ExecuteCommandTask, args:
{name=echo, command=echo message}, duration: 5-15,
echo className = ExecuteCommandTask status = true
23:09:50.664  edu.fit.hiai.yoshka.core.Engine - Processing time:
3, Duration: 11000
23:09:50.664  edu.fit.hiai.yoshka.core.Engine - Forgetfulness
triggered. Restoring previously forgotten Task for execution on
selection.
23:09:50.664  edu.fit.hiai.yoshka.core.Engine - Step #18 Running
node - name: readFile, task: generics.ReadFileTask, args:
{name=readFile, file=distribution.txt}, duration: 5-15,
23:09:50.665  edu.fit.hiai.yoshka.tasks.generics.ReadFileTask -
ReadFileTask.run: File distribution.txt not found
readFile className = ReadFileTask status = false
23:09:55.664  edu.fit.hiai.yoshka.core.Engine - Processing time:
1, Duration: 5000
Appendix B

Questionnaire

On a scale of 1 to 5 (with 1 being the hardest and 5 being the easiest)
You should use the additional comments section to briefly explain why it was easy or difficult.

1. How difficult was it to create the necessary tasks for the mission?
   
   1  2  3  4  5

   Additional comments:

2. How difficult was it to implement the mission structure?

   1  2  3  4  5

   Additional comments:

3. How difficult was it to implement the execution of the mission?

   1  2  3  4  5

   Additional comments:
Appendix C

Your mission is to use the selected tool to implement up to five actions and generate the traffic representative of executing these actions. The actions that you need to implement are the following:

**Stage 1:** Creating and implementing user actions.
1. Creating a file. This action must generate a file on the local filesystem.
3. Writing to a file. This action must write arbitrary data to the previously created file.
4. Indicating mission success. This action must rm the user that the mission has succeeded.
5. Indicating mission failure. This action must rm the user that the mission has failed.

**Stage 2:** Creating and implementing the logical structure of the mission.
These actions need to be organized in such a way that would represent the following flow of logic.
The mission to be executed is a simple conditional sequence of actions that starts with creating a file, on success, sending a get request or writing to that file and on failure failing the whole mission. How you choose between the default task and the alternative task is as follows:

By default, always or based on a distribution execute the default task. If the execution of this task fails, resort to the alternative task. If this task also fails, fail the mission.

The duration of the execution of Actions 2 and 3 must take 5 seconds each.

Stage 3: Execution and debugging of mission and results.
Verify whether the mission was successful or not by executing your user model. Check to make sure that the files have been created/modified and that network traffic has been generated.
Appendix D

Thank you for volunteering to participate in this rmal study. This is a study meant to compare the ease of use and capabilities of three tools implementing user models in a cyber emulation scenario. The tools you will be comparing are: Yoshka, Ostinato and Soar.

All tools, documentation and the software environment will be set up and provided to you ready for use. You will be trying these three tools one after the other. You are permitted to take up to a 15-minute break between each tool.

You will be provided with a document outlining the goals to be achieved with each tool. Prior to using any of the tools, you will have 15 minutes to read the documentation and familiarize yourself with the software environment of the respective tool. Following that, you will have 30 minutes to complete the task outlined in the goals document. I will periodically let you know how much time you have remaining.

The goals document is divided into three stages. After you complete a stage, please let me know so I can take note of the time taken. Please take the time to read this goal document now.

[Wait for volunteer to read goals document and answer any questions]

If you get stuck implementing a certain stage, feel free to use the documentation and/or the internet for help. You can try moving onto a different stage, but know that all the stages are directly dependent on each other.

At the end of the allotted 30 minutes you will be given a questionnaire. Please be as objective as possible. Since these tools are compared with each other, you may go back to change your answers after you have completed usage of all three tools.

If you have any questions, please let me know right now as I will not be able to help you after the timer starts.