Have it Your Way: Generating Customized Log Datasets With a Model-Driven Simulation Testbed

Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber, *Member, IEEE*

***Abstract*—Evaluations of intrusion detection systems (IDS) require log datasets collected in realistic system environments. Existing testbeds therefore offer user simulations and attack scenarios that target specific use-cases. However, not only does the preparation of such testbeds require domain knowledge and time-consuming work, but also maintenance and modifications for other use-cases involve high manual efforts and repeated execution of tasks. In this article, we therefore propose to generate testbeds for IDS evaluation using strategies from model-driven engineering. In particular, our approach models system infrastructure, simulated normal behavior, and attack scenarios as testbed-independent modules. A transformation engine then automatically generates arbitrary numbers of testbeds, each with a particular set of characteristics and capable of running in parallel. Our approach greatly improves configurability and flexibility of testbeds and allows to reuse components across multiple scenarios. We use our proof-of-concept implementation to generate a labeled dataset for IDS evaluation that is published with this article.**

***Index Terms*—Intrusion detection, log data, model-driven engineering (MDE), testbed.**

I. Introduction

Intrusion detection systems (IDS) are tools that automatically monitor and analyze computer system or network behavior and report suspicious activities. They contribute to system security by recognizing patterns that are known to relate to adversaries within large amounts of complex data or disclose deviations from normal behavior without human intervention.

The ability to measure and compare the performances between IDSs in a representative way is essential for improving their algorithms and providing new research directions [1]. However, many IDSs are designed and configured for deployment in particular environments and focus on the detection of specific types of cyberattacks. Accordingly, objective IDS benchmarking for selection and deployment in real-world applications is nontrivial [2]. For this reason, research groups have developed testbeds that resemble real networks and allow IDS deployment as well as attack execution in controlled environments [3], [4]. In general, it is difficult to make testbeds publicly available, because this would imply sharing network infrastructure comprising complex and possibly closed-source systems in a way that allows others to understand, configure, and conduct experiments. Instead of sharing the testbeds themselves, researchers therefore publish the network traffic or log datasets collected during their simulation runs. Some datasets then become standards for evaluation for some time, however, will at some point be regarded as outdated or criticized for particular aspects, e.g., focus on network traffic rather than log data [5], missing documentation of installed services [6], too simple or unknown simulations of system behavior [1], lack of multi-step attack vectors [7] including long-term advanced persistent threats [8], or too narrow focus that impedes generalization [6]. Eventually, this will encourage other research groups build new testbeds with updated technologies that are relevant for their own use-cases.

Testbeds are essential to validate, evaluate, and compare the capabilities of IDSs. Thereby, testbeds offer analysts environments that yield unbiased results for their use-cases and the real world. Otherwise, it is impossible to reliably assess whether the IDS under test performs with similar efficiency and effectiveness when deployed in productive operation. Furthermore, flawed tests may lead to misconfigurations of IDSs and thus limit their abilities to detect certain attacks. In order to ensure that such requirements are met, high efforts should be spent on preparing and designing the testbed setup.

Setting up testbeds for particular use-cases is usually time-consuming. The most tedious tasks involve adjustments for tests carried out under different conditions as well as follow-up modifications for related use-cases [9]. The main problem is that analysts are stuck with rigid testbeds that are set up a single time by domain experts that could not predict the requirements that became necessary after setup. Such testbeds are difficult to maintain or modify for a number of reasons.

1) Manual work is required to change the testbed in hindsight, e.g., increase the number of network components.
2) Modifications of otherwise identical components have to be repeated multiple times.
3) It is necessary to check and update all components individually to ensure that up-to-date versions are used.

Manuscript received July 31, 2020; revised September 16, 2020; accepted October 12, 2020. Date of publication November 17, 2020; date of current version March 2, 2021. This work was supported in part by the FFG Projects INDICAETING under Grant 868306 and DECEPT under Grant 873980, and in part by the EU H2020 Project GUARD under Grant 833456. Associate Editor: W. K. Chan. *(Corresponding author: Max Landauer.)*

Max Landauer, Florian Skopik, Markus Wurzenberger, and Wolfgang Hotwagner are with the Center for Digital Safety and Security, Austrian Institute of Technology, 1210 Vienna, Austria (e-mail: max.landauer@ait.ac.at; florian.skopik@ait.ac.at; markus.wurzenberger@ait.ac.at; wolfgang.hotwagner@ait.ac.at).

Andreas Rauber is with the Institute of Information Systems Engineering, Vienna University of Technology, 1040 Vienna, Austria (e-mail: rauber@ifs.tuwien.ac.at).

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2020.3031317

TABLE I
TESTBED DEVELOPMENT PHASES

Phase	Involved tasks	Remedies and support
Concept	Define use-cases and goals	-
Design	Component selection	Reuse of code and settings
Deployment	Parameterization and instantiation of components	Tools for automatic system deployment and setup
Utilization	Testing and data collection	Scripts enable automation
Adaptation	Variation of parameters and configuration settings	Automatic selection from predefined ranges or lists

4) Resetting the testbed to a "clean" state is often necessary to remove artifacts that influence subsequent simulations. However, this undoes purposefully inserted changes and thus complicates iterative testbed development.

Another problem is that most existing testbeds are relatively static, because their configuration, e.g., the selection of a user behavior profile from a predefined list relies on manual input and domain knowledge. This impedes fast instantiation of different testbeds with variable configurations. In addition, such parameters usually cover only basic application settings of the testbed environment, but are not extensive enough to fine-tune parameters with less influence, e.g., particular aspects of a specific system behavior profile, and not powerful enough to change the overall testbed setup, e.g., upgrade components to newer versions. However, the possibility to obtain multiple testbeds with variations would be highly beneficial for IDS evaluation, because more available data representing different technical environments would enable generation of separate training, validation, and test datasets, improve robustness of evaluation results, and support validation of approaches that derive reusable attack attributes and patterns for detection of specific attacks across different systems [10].

There is thus a need for a methodology that addresses the aforementioned problems and eases testbed setup and development. Table I gives an overview of the phases typically encountered during testbed development and states ideas for automation of involved tasks. To integrate these strategies in the development procedure, we propose to leverage techniques from model-driven development. In particular, in this article we present an approach that makes use of abstract and testbed-independent models (TIMs) for the testbed infrastructure, system behavior, and attack scenarios, and uses a transformation engine to automatically generate all scripts necessary to set up the infrastructure, configure all components, install services, simulate normal system behavior, and start the attacks. Thereby, our approach is able to generate multiple testbeds instances at once, each with particular characteristics, and produce log datasets that cover variations occurring in different environments. Note that this does not imply that data is generated using model-driven techniques; instead, we propose a model-driven approach for the instantiation of testbeds that are useful for the production of security data.

We implemented a proof-of-concept based on our proposed model-driven methodology where an automatized pipeline allows us to generate arbitrary numbers of testbeds running in parallel. We designed a common real-world use-case, i.e., users that access a mail platform and a web store, and launched two attacks that make use of recently discovered exploits. We use this setup to generate four testbed instances with variations and collect their log data, which is published with this article.[1]

We summarize the contributions of this article as follows:

1) a novel model-driven concept for automatically instantiating arbitrary numbers of parameterized testbeds;
2) adhering to a set of design principles;
3) for the generation of new network and log datasets.

The remainder of this article is structured as follows. Section II reviews testbeds for log data generation or IDS evaluation. In Section III, we first propose a list of design principles and then introduce our approach for automatic testbed generation using model-driven techniques. Section IV contains concrete design decisions regarding testbed infrastructure, simulation of normal behavior, and attacks. Section V validates our approach. Section VI discusses applications and limitations of our approach. Finally, Section VII concludes this article.

II. RELATED WORK

Several testbeds exist for use-cases that include log data generation for IDS evaluation. For example, Sharafaldin *et al.* [4] build a network of real-world components, including servers, firewalls, switches, and user machines. They schedule and launch attacks against the network and then capture and classify the collected network traffic. Instead of building real networks, most existing approaches, including our approach presented in this article, leverage virtualization, since it improves the scalability and flexibility of the testbed. Ring *et al.* [11] design a virtual network that represents a small business environment, containing mail, web, backup, and file servers. They simulate normal behavior by randomized scripts and carry out brute-force attacks, DoS attacks, and security scans, which are subsequently labeled in the collected network traffic by predefined rules. Even though they use parameterized scripts, they limit this functionality to the user behavior and do not consider variations of the system infrastructure or attacker behavior, which is solved by our approach. They also decided to connect their network to the Internet to include real data in their captures. The downsides of this setup are that it is unknown what types of potentially malicious behavior is included in their collected data, the reproducibility is limited since it is difficult to recreate the external influences, and artifacts such as IP addresses linked to real entities have to be anonymized for privacy reasons, thus altering the raw data.

In contrast to this approach, Algorizmi [12] is a framework to generate fully isolated testbeds and launch predefined attacks from a database. The framework is highly configurable and allows to reproduce results by reusing configuration scripts. However, testbed setup is nontrivial, since it relies on manual steps that have to be carried out beforehand. While our approach also generates isolated testbeds, we employ model-driven methods to alleviate setup and configuration difficulties. Maciá-Fernández *et al.* [6] provide another testbed for network-based IDS evaluation and put special emphasis on modeling periodically repeating behavior, such as daily or weekly cycles. Our approach also produces such recurring patterns. Despite being

[1]Log datasets accessible at https://zenodo.org/record/3723083

configurable, most testbeds comprise specific environments and are thus unable to provide evaluation results that are independent of manual settings. TIDeS [13] addresses this issue by varying network load generated by different types of normal behavior profiles derived from real networks.

Similar to our approach, Čeponis and Goranin [5] create a testbed for host-based IDS evaluation, i.e., log data rather than network traffic is collected. In particular, they collect system call logs from malware injected on a host system. However, they do not simulate normal behavior, because their main focus is to extract the malware manifestations rather than detecting them within normal data. Creech and Hu [14] on the other hand perform normal web browsing or document editing while collecting system call data. Their attack exploits a vulnerability that was purposefully installed on an otherwise fully patched server. Similarly, Skopik *et al.* [3] design a virtual testbed that imitates a real web platform. They use scripts to simulate user behavior and validate their approach by comparing simulated page visit frequencies with log data collected on a real system. We also use scripts to simulate normal user and attacker behavior, but generate different profiles automatically following a model-driven approach.

ViSe [15] is a testbed for both network-based and host-based IDS evaluation. It provides snapshots of virtual machines that allow easy instantiation of predefined testbeds suitable for running attacks against already installed IDSs. Another comprehensive framework for IDS evaluation is LARIAT [16] that comes with tools for normal behavior simulation and attack injection. Profiles for normal behavior are thereby derived from a real network and attacks involve sequences of malicious actions carried out in a particular order. In our approach, it is also possible to parameterize the models with ranges and lists of expected values that may be derived from expert knowledge or real observations.

Beside IDS evaluation, testbeds are frequently used for malware analysis and experiments. Due to the fact that requirements on the testbeds, particularly isolation, repeatability, and the presence of monitoring tools, are similar, application for log data collection and IDS evaluation is usually reasonable. For example, DETER [17] has been used to test Distributed Denial-of-Service attacks, worms, and malicious code. When experimenting with worms, Jiang *et al.* [18] also include and test countermeasures and measure their effectiveness. Experimental testbeds are related to educational simulations, for example, Frank *et al.* [19] describe the design process of a security training testbed that enables automatic deployment of infrastructure and services.

While several of the aforementioned approaches emphasize configurability of their networks, they do not make use of model-driven development for preparing testbed setup, normal behavior, or attacks. Accordingly, variations of configurations imply manual work and domain knowledge about the deployed technologies and the testbed setup itself. This issue is addressed by Cappos *et al.* [20], who propose a meta-framework called Tsumiki that specifies guidelines for testbed design as reusable components and interfaces. Galan *et al.* [9] on the other hand leverage model-driven techniques to enable testbed design on an abstract level and automatic generation of testbed instances. While they focus solely on networking aspects, our approach extends the idea of abstract testbed modeling to its infrastructure setup, dynamic behavior, and attacks.

III. Testbed Design Methodology

In this section, we introduce our proposed methodology for model-driven testbed development. Before presenting the model-driven workflow, we propose a set of design principles that act as requirements for subsequent design decisions.

A. Design Principles

Most of the issues with existing log datasets generated in testbeds are attributable to shortcomings of the system infrastructures and environments where the data was collected. Accordingly, it is necessary to align the design process of the testbed with requirements on the data to be generated. We therefore pursue a number of design principles that form the basis of our testbed generation methodology. In the following, we briefly discuss each of the principles.

1) Authenticity: Log data should be collected within realistic scenarios to ensure a representative evaluation of the capabilities of IDSs. Thereby, several aspects must be considered: First, all involved components, e.g., servers or clients, have to be selected and arranged within a network that is representative for a well-defined use-case, e.g., a small enterprise. This includes network complexity, i.e., diversity of involved components, as well as scale, i.e., total number of components. Second, components must act and react like their real counterparts. This includes automatic behavior, e.g., scheduled tasks, as well as user behavior that may be erratic, unpredictable, and dependent on user roles. Third, attacks carried out on the testbed should be related to recently discovered vulnerabilities to ensure that the detection capabilities are not measured on outdated exploits that possibly have lost relevance in modern infrastructures. Accordingly, all services should be set up with fully patched and up-to-date software. Moreover, the attacks should affect common technologies in order to be relevant for a large number of people and organizations. Fourth, the collection of the log data and network traffic has to take place in a realistic manner. This means that only commonly available log sources should be used and that logging should be configured on a level that is adequate for the use-case.

2) Flexibility: Setting up a testbed encompasses manual time-consuming work [9]. It is therefore economically reasonable to design a testbed that is flexible in the sense that it supports adjustments and extensions and enable iterative development. There are mainly three dimensions of modifying the testbed. First, enlarge or shrink the scale of the network by adding or removing components. Thereby, we suggest to initially create a number of predefined components that act as building blocks that can be arbitrarily duplicated and set into relation with each other. Second, the configurations of these components, including all installed services and their versions, are subject to modification. Third, it should be possible to change the dynamic behavior and interactions between the components, e.g., the types of services accessed by clients.

3) Reproducibility: The ability to reproduce the generated log dataset requires that it is possible to reset the testbed to a past state. This is particularly useful when the effects of an attack on modified versions of the testbed are subject of investigation, for example, comparisons of patched and nonpatched services. It is important to note that it is usually impossible to guarantee that the reproduced dataset is identical to the original dataset, but rather only conform in their main characteristics, such as the overall user behavior. This is due to the fact that it is difficult to avoid that latencies during communication of components as well as arbitrarily occurring events or failures result in nondeterministic behavior. In order to ensure reproducibility, it is necessary to isolate the testbed from all external sources that may have unpredictable influence on the outcome of the simulation and are not under control of the analyst, for example, publicly accessible connection over the Internet may cause unknown and potentially malicious behavior manifesting itself in the logs, making subsequent evaluations on the captured data less reliable.

Another important aspect of reproducibility is that only freely available or open-source services are used within the testbed. The reason for this is that the use of commercial products or services that are not publicly available may prevent others from rebuilding the same system.

4) Availability: To enable IDS benchmarking and comparison of detection capabilities with other approaches, it is important to make generated datasets publicly available. In addition, the dataset has to be accompanied with appropriate documentation, including the overall purpose of the dataset, the infrastructure setup, and a description of the normal and attacker behavior. If such a documentation is missing, it is difficult for others to understand certain artifacts in the data, interpret evaluation results, or reproduce the dataset.

5) Utilizability: The availability of a dataset alone is not sufficient to enable evaluation of IDSs. In order to obtain comparable evaluation results, a ground truth that defines the malicious behavior in a quantifiable way is needed. Thereby, several levels of labeling the data are possible. The most superficial approach is to label all log events generated during time intervals of attacks as malicious. Note that all other events can be considered benign, because the testbed is a simulation that runs isolated from a productive system that may be affected by unknown processes or attacks. While this form of labeling is easy to accomplish since it is possible to derive anomalous time windows from attack scenario descriptions, it has the disadvantage of also labeling normal events that occur during attacks as anomalous. However, since most IDSs report individual events as anomalies, a more in-depth evaluation is enabled by labeling only events that actually correspond to malicious behavior as anomalous. Even better are labels that differentiate between different types of attacks or attack steps, enabling in-depth evaluation of anomaly detection systems that support attack classification or focus on multistep attacks.

Anomaly detection systems or other self-learning approaches additionally require that the generated log data covers a sufficiently large duration of the normal behavior in order to be adequately utilized [6]. In particular, the data has to span over multiple cycles of normal behavior, i.e., all repeating processes should be at least once fully present in the data. Incomplete training sets may lead to misclassifications, e.g., false alarms, during evaluation. In addition, log data should always be published in raw format, because any modification such as anonymization or pseudonymization possibly distort evaluation results [4].

B. Model-Driven Testbed Setup Methodology

The usual setup process of a testbed that adheres to the outlined design principles involves time-consuming and nontrivial work. In particular, ensuring flexibility of the testbed, i.e., enabling arbitrary changes of the size of the represented network while at the same time allowing the user to steer component configurations, is technically difficult and involves tedious tasks, such as repeatedly setting up or modifying similar components in slightly different environments. This procedure becomes especially nerve-racking when settings of the system configuration have to match simulated user behavior or are dependent on the type of attack [3].

We suggest to use techniques from model-driven engineering (MDE) [21] to alleviate these issues. MDE is a methodology that aims at simplifying software development by providing programmers a framework to design solutions on a higher level of abstraction, thereby allowing them to focus on the actual problem at hand independent of technical details and complex implementations on specific platforms. This results in applicable models that support development on multiple different platforms. MDE further makes use of transformation engines that automatically process platform-independent models and generate code for specific platforms.

For our proposed approach, we adopt these concepts from MDE and apply them for testbeds rather than for software platforms. Fig. 1 shows an overview of the layers of abstraction and workflow that we use for testbed design and automated testbed deployment. Our model-driven approach thereby differentiates between 1) the technical infrastructure, 2) the normal system behavior, and 3) the modeled attack.

The top of the figure depicts preselected relevant aspects of the real world that is simulated in the testbed. In particular, we seek for commonly available infrastructures that are frequently subject to attacks, such as servers that are accessible over a network. We also look for frequently installed packages and examine the settings of the logging services. Given a real infrastructure, it is also possible to monitor the exhibited behavior and derive relevant characteristics of normal system usage, such as usage distributions over a period of time. Finally, attacks are either observed on the real infrastructure or exist in the documented form in online threat databases.[2]

We then define TIMs. Regarding the infrastructure, this implies declarations of the setup routines for all involved components and services without specifying any concrete parameters. For example, we define how a component is connected to the network, but do not allocate IP addresses, assign names, or specify the number of users, but only the type and range of these parameters. Similarly, we design a model of the system

[2]For example, https://www.metasploit.com/

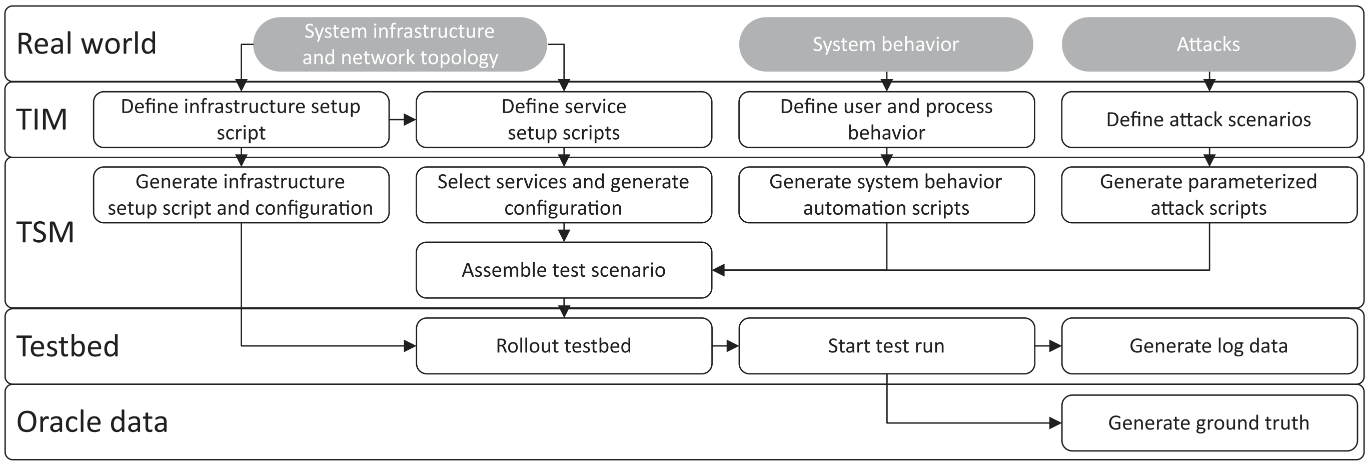

Real world
System infrastructure and network topology
System behavior
Attacks
TIM
Define infrastructure setup script
Define service setup scripts
Define user and process behavior
Define attack scenarios
TSM
Generate infrastructure setup script and configuration
Select services and generate configuration
Generate system behavior automation scripts
Generate parameterized attack scripts
Assemble test scenario
Testbed
Rollout testbed
Start test run
Generate log data
Oracle data
Generate ground truth

Fig. 1. Model-driven testbed generation approach. TIMs are derived from real scenarios and transformed to testbed-specific models (TSMs) that instantiate the testbed for labeled network and log data generation.

behavior as a state machine without fixed transition probabilities between its states, and a model for the attack scenario that consists of the basic steps that are necessary for carrying out the attack. All TIMs function as templates, i.e., they are scripts that represent specific routines, but are configurable through consciously placed parameters throughout the code.

Our transformation engine that generates TSMs processes the templates and inserts all parameters to produce executable code. The parameters are thereby selected randomly based on their type specified in the TIM. For example, the number of simulated users is selected from a predefined range, their names and passwords are picked from predefined lists, and IP addresses are automatically assigned from a pool. For the user behavior, we specify a number of profiles with ranges for transition frequencies that the transformation engine translates into probabilities. Regarding the attack scenario, optional parameters of individual steps as well as their order and delays are randomly selected. Note that modeling may be based either on attacks or vulnerabilities, i.e., an attack model scenario may focus on a single malicious action or involve several vulnerability exploits and diverse attack vectors.

Since transformation of TIMs to TSMs is fully automatic, it is possible to generate arbitrary amounts of TSMs at the same time, where each TSM exhibits variations depending on the settings for random selection. For each TSM, we first run the infrastructure setup scripts to build the virtual machines and set up the network of the testbed instance. We then gather, allocate, and run all generated scripts for component setup, user simulation, and attack execution, on the respective machines. After completion, we use another script to copy all logs from the virtual machines and label them according to the outcomes of the simulation. In MDE terminology, such labeled data are usually referred to as oracle data [22].

IV. Testbed Models

The previous section outlines the idea of generating testbeds using a model-driven approach. In this section, we discuss selected design aspects of our implementation in more detail.

```
<for mailID in 0:numMails>
-hosts: mail-<mailID>
 hostname: <random(hosts.txt)>
 ip: <random(ip-pool)>
 user_list:
 <for u in random(users.txt)>
  -name: “<u.name>“
   password: “<crypt(u.pw)>“
 <end for>
<end for>
<for userID in 0:numUsers>
-hosts: user-<userID>
 ip: <random(ip-pool)>
 connectedTo: “<random(mail)>“
<end for>
```

```
-hosts: mail-0
 hostname: mail.abc.com
 ip: 192.168.10.175
 user_list:
  -name: “alice“
   password: “OyHkSWj“
  -name: “bob“
   password: “nqzkLYY“
-hosts: user-0
 ip: 192.168.10.229
 connectedTo: “mail-0“
-hosts: user-1
 ip: 192.168.10.190
 connectedTo: “mail-0“
...
```

Fig. 2. Simplified sample transformation of TIM (left) to TSM (right) for infrastructure setup.

A. System Infrastructure

To evaluate our model-driven concept for generating testbeds, we were aiming to create a simulation of a system infrastructure that is common in many organizations. After reviewing usage statistics of well-known technologies, we decided to model an Apache web server hosting a mail platform and content management system (CMS) that are accessed and used by an arbitrary number of users. In particular, we selected Horde Groupware Webmail[3] and a webshop provided by OkayCMS,[4] because both platforms are available open source and have recently been affected by vulnerabilities. Each generated testbed should consist of one web server with a database and a variable number of connected host machines, each representing one or more users.

We designed TIMs in YAML syntax for setup of a web server and a user host machine. Fig. 2 shows a simplified and shortened version of such a template on the left side, where “mail” refers to the web server and “user” to the user host machine. A transformation engine is able to process such a

[3][Online]. Available: https://www.horde.org/
[4][Online]. Available: https://okay-cms.com/

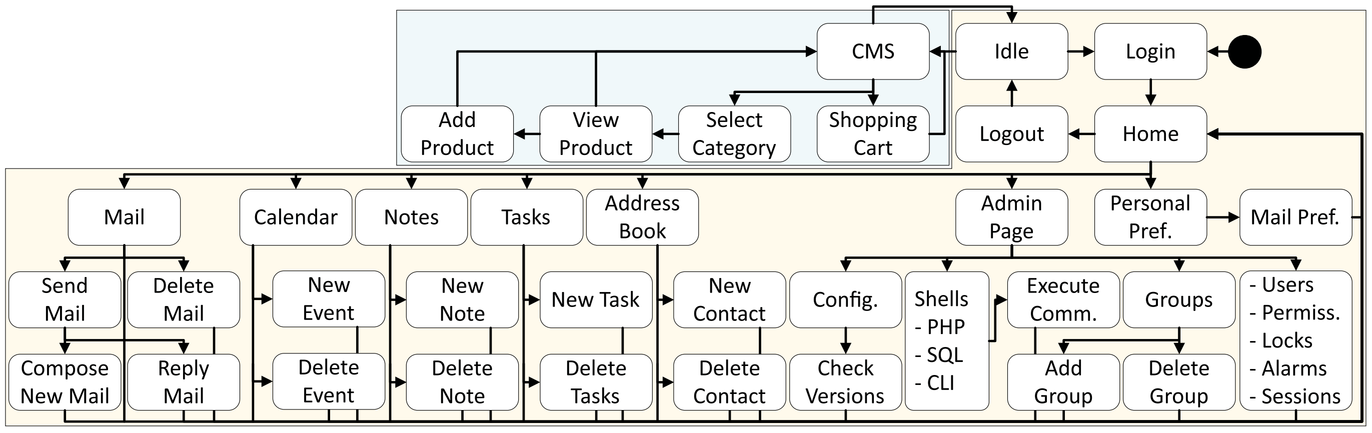


Fig. 3. Simulated normal user behavior on the Horde Webmail (yellow) and OkayCMS (blue) modeled as a state machine.

TIM and generate the TSM on the right side of the figure that acts as a configuration file for the setup procedure. The engine thereby executes the code within the arrow brackets specified in the TIM to fill the gaps of the template with parameters that are subject to change in every testbed instance. For example, the server hostname is randomly selected from a predefined list of names, IP addresses are automatically assigned during setup, and accounts for a random set of users are created.

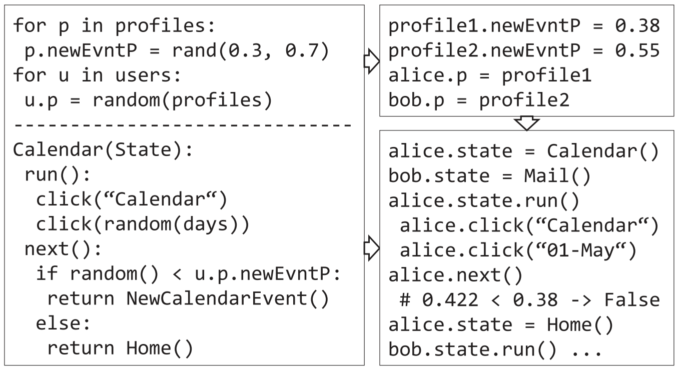
Before running the transformation engine, it is necessary to specify the total amount of web servers and user host machines to be generated from the TIM. These numbers are critical since there is usually a limited amount of computational resources available for the virtual machines. The transformation engine then allocates the host machines randomly to web servers (parameter "connectedTo" in Fig. 2) in accordance with predefined values for the minimum and maximum number of host machines per web server.

There are two main advantages of designing the infrastructure on this abstract level. First, it is simple to generate large numbers of different testbeds that run in parallel. It is thereby easy to steer the degree of variation by adjusting the predefined ranges in the TIM. Second, changes that affect all components of a particular type only have to be carried out once in the TIM since these modifications will automatically propagate to all TSMs when running the transformation engine again.

B. Normal System Behavior

The purpose of the testbed is to generate log data for evaluating attack detection tools. However, executing malicious actions on an idle system makes their detection relatively easy, since almost all generated logs are likely to be related to the attack. This scenario is not authentic, because web servers in the real world are almost always actively used. Furthermore, IDSs based on anomaly detection usually rely on a training phase that represents normal and anomaly-free behavior in order to disclose deviations from the learned patterns.

We therefore simulate normal system behavior by modeling typical user accesses. For this, we created a state machine that covers all relevant functions of both the Horde Webmail and



```
for p in profiles:
 p.newEvntP = rand(0.3, 0.7)
for u in users:
 u.p = random(profiles)
------------------------------
Calendar(State):
 run():
  click(“Calendar“)
  click(random(days))
 next():
  if random() < u.p.newEvntP:
   return NewCalendarEvent()
  else:
   return Home()
```

```
profile1.newEvntP = 0.38
profile2.newEvntP = 0.55
alice.p = profile1
bob.p = profile2
```

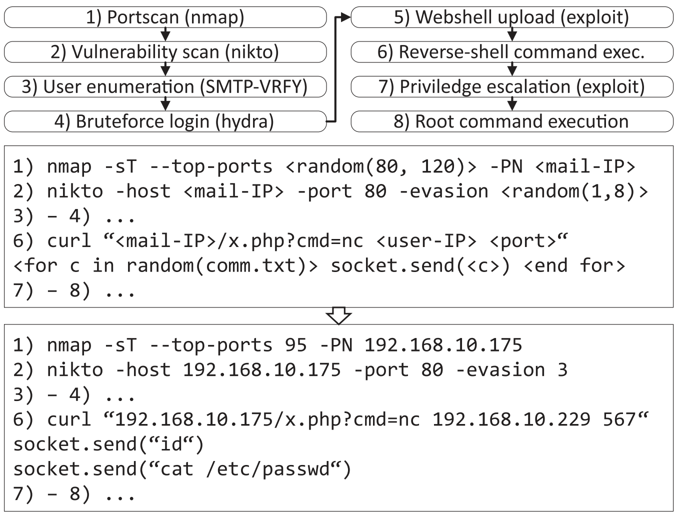
```
alice.state = Calendar()
bob.state = Mail()
alice.state.run()
 alice.click(“Calendar“)
 alice.click(“01-May“)
alice.next()
 # 0.422 < 0.38 -> False
alice.state = Home()
bob.state.run() ...
```

Fig. 4. Randomized user profiles and state machine of system behavior TIM (left) transformed to TSM (top right) and testbed execution (bottom right).

OkayCMS platforms using the well-known web automation framework Selenium.[5] Fig. 3 shows a graphical overview of all subpages and activities that are covered by the state machine and users are thus able to visit. On Horde Webmail, the users are capable of changing their preferences, writing mails to other users and responding to received mails, and creating and deleting entries in the calendar, notebook, list of tasks, and address book, where fields are filled out with random values or dummy text. Users with administrator privileges are further able to access the admin page and its subpages. On OkayCMS, users browse the articles available on the webshop and add or remove products from their shopping carts.

Fig. 4 shows a sample transformation from TIM to TSM and further an exemplary execution of a parameterized system behavior script. The left side of the figure shows the state machine as well as the instantiation of a predefined number of profiles, each containing ranges of transition frequencies between the states, and their random allocation to users. The profiles also specify the browser used to access the websites. Moreover, the mail recipients are selected based on a randomly generated small-world network, i.e., most users communicate in small groups rather than randomly sending mails to every other

[5][Online]. Available: https://selenium.dev/


1) Portscan (nmap)
2) Vulnerability scan (nikto)
3) User enumeration (SMTP-VRFY)
4) Bruteforce login (hydra)
5) Webshell upload (exploit)
6) Reverse-shell command exec.
7) Priviledge escalation (exploit)
8) Root command execution

```
1) nmap -sT --top-ports <random(80, 120)> -PN <mail-IP>
2) nikto -host <mail-IP> -port 80 -evasion <random(1,8)>
3) – 4) ...
6) curl “<mail-IP>/x.php?cmd=nc <user-IP> <port>“
<for c in random(comm.txt)> socket.send(<c>) <end for>
7) – 8) ...
```

```
1) nmap -sT --top-ports 95 -PN 192.168.10.175
2) nikto -host 192.168.10.175 -port 80 -evasion 3
3) – 4) ...
6) curl “192.168.10.175/x.php?cmd=nc 192.168.10.229 567“
socket.send(“id“)
socket.send(“cat /etc/passwd“)
7) – 8) ...
```

Fig. 5. Multistep attack on Horde Webmail (top) and its transformation from TIM (center) to TSM (bottom).

user with the same probability [23]. In addition, users regularly log out and go idle for random amounts of time, and stay inactive during night time to simulate daily routines.

The TSM generated by the transformation engine yields a configuration that is exemplarily displayed in the top right of Fig. 4. Note that probabilities for choices are normalized to ensure that they sum up to 1. All users exhibit different behavior, even though their actions are based on the same independent behavior model. The bottom right of the figure shows the execution log of users “alice” and “bob.” In this sample, user “alice” views a random day from the calendar, but returns to the home page rather than adding a new event, because a randomly selected value is below the required threshold. This sample also shows multiple users using the system at the same time causing interleaving processes.

C. Attacker Behavior

We prepared two attacks to be executed on the testbed. The first one is a multistep intrusion that involves several tools commonly used by adversaries and exploits two well-known vulnerabilities to gain root access on a mail server. The top of Fig. 5 shows an overview of the attack steps. The first two steps involve scans for open ports[6] and vulnerabilities.[7] Then, the attacker uses the smtp-user-enum tool[8] for discovering Horde Webmail accounts using a list of common names and the hydra tool[9] to brute-force log into one of the accounts using a list of common passwords. The attack proceeds with an exploit in Horde Webmail that allows to upload a webshell (CVE-2019-9858) and enables remote command execution. We simulate the attacker examining the web server for further vulnerabilities by executing several commands, such as printing out system info. In our scenario, the intruder realizes that a vulnerable version of the Exim package is installed and thus uploads an exploit (CVE-2019-10149) to obtain root privileges through another reverse connection.

Fig. 5 shows how we model this attack procedure as TIM and one possible transformation to TSM. As visible in the TIM, we use a sequence of predefined commands, but do not specify values that are only known after instantiating the testbed, such as the IP addresses of the web server (“mail-IP”) and user host (“user-IP”), as well as parameters that are varied in each simulation, such as port numbers, evasion strategies, or commands executed after gaining remote access. This attack was purposefully designed as a multistep attack with variable parameters to evaluate the ability of IDSs to disclose and extract individual attack steps and their connections, and recognize the learned patterns in different environments despite variations.

The second attack targets the web shop. A recently discovered flaw in OkayCMS allows an attacker to inject a malicious php-object via a crafted cookie (CVE-2019-16885). In this scenario, the attacker uses the exploit to upload a webshell and is then again able to execute commands through the remote interface. Since no user credentials are required for authentication, this attack consists of only a single step and does not involve variations. Instead, it is designed to evaluate whether IDSs are able to detect and classify the injection of the php-object, since it only manifests itself in slightly different library calls that are difficult to detect.

Both attacks are carried out at a random point in time within a predefined period on randomly selected user host machines. Since the attacks are carried out independent of each other, they may be executed at the same time. During execution, the outcomes of the commands are automatically searched for keywords that indicate successful execution. We log this information together with the start and end times of each attack step, which is useful for labeling the recorded log data.

D. Ground Truth

Labeling data is essential for appropriately evaluating and comparing the detection capabilities of IDSs. However, generating labels is difficult for several reasons, such as follows.

1) Log data is generated in large volumes and manual labeling all lines is usually infeasible.
2) Single actions may manifest themselves in multiple log sources in different ways.
3) Processes are frequently interleaving and thus log lines corresponding to malicious actions are interrupted by normal log messages.
4) Execution of malicious commands may cause manifestations in logs at a much later time due to delays or dependencies on other events.
5) It is nontrivial to assign labels to missing events, i.e., log messages suppressed by the attack.

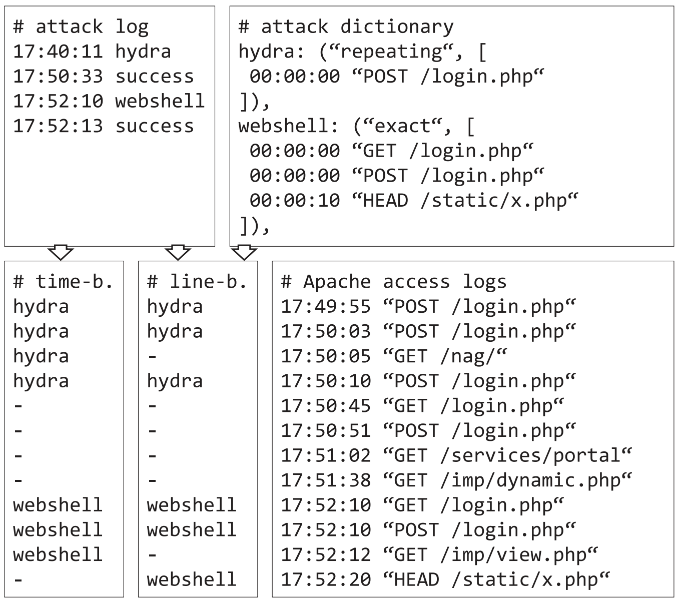
We attempt to alleviate most of these problems by automatically labeling logs on two levels. First, we assign time-based

[6][Online]. Available: https://nmap.org/
[7][Online]. Available: https://cirt.net/Nikto2
[8][Online]. Available: https://tools.kali.org/information-gathering/smtp-user-enum
[9][Online]. Available: https://tools.kali.org/password-attacks/hydra

labels to all collected logs. For this, we make use of the attack execution log mentioned in the previous section. We implemented a script that processes all logs, parses their time stamps, and labels them if their occurrence time lies within the time period of an attack stage. Under the assumption that attack consequences and manifestations are not delayed, it is then simple to check whether anomalies reported by IDSs lie within the expected attack time phases. Since exact times of malicious command executions are known, it is even possible to count correctly reported missing events as true positives.

While time-based labeling is simple and effective, it cannot differentiate between interleaved malicious and normal processes and does not correctly label delayed log manifestations that occur after the attack time frame. Therefore, our second labeling mechanism is based on lines that are known to occur when executing malicious commands. For this, we carry out the attack steps in an idle system, i.e., without simulating normal user behavior, and gather all generated logs. We observed that most attack steps either generate short event sequences of particular orders (e.g., webshell upload) or large amounts of repeating events (e.g., scans). We assign the logs to their corresponding attack steps and use the resulting dictionary for labeling new data. For the short ordered sequences, we pursue exact matching, i.e., we compute a similarity metric [24] based on a combination of string similarity and timing difference between the expected and observed logs and label the event sequence that achieves the highest similarity. For logs that occur in large unordered sequences, we first reduce the logs in the dictionary to a set of only few representative events, e.g., through similarity-based clustering [24]. Our algorithm then labels each newly observed log line that occurs within the expected time frame and achieves a sufficiently high similarity with one of the representative lines. These strategies enable correct labeling of logs that occur with a temporal offset or are interrupted by other events, but obviously suffer from misclassifications when malicious and normal lines are similar enough to be grouped together during clustering.

Fig. 6 shows an example of our labeling procedure that involves two sample attack steps, the brute-force login tool "hydra" and the "webshell" upload. The top left of the figure shows start and end times of both attacks logged during attack script execution. The top right of the figure shows a dictionary that lists the log lines that are expected to occur in the Apache access log at attack execution. Note that the "hydra" logs are marked as "repeating," i.e., they represent a large number of similar lines, whereas the "webshell" logs are marked as "exact," i.e., they correspond to ordered individual lines. The bottom left of the figure displays the time-based and line-based labels for the Apache access logs in the bottom right. As visible in this example, the time-based labels are assigned to the lines solely by their occurrence timestamps. Due to the interleaving user actions, this means that lines generated by actions other than the attack (e.g., viewing Horde task list "nag"), but occurring in the same time frame, are also labeled accordingly. These lines remain correctly unlabeled by the line-based method. In particular, the "repeating" technique labels all lines within the "hydra" attack time frame that achieve a minimum string similarity to the message "POST /login.php." In this simplified example, these



```
# attack log
17:40:11 hydra
17:50:33 success
17:52:10 webshell
17:52:13 success
```

```
# attack dictionary
hydra: ("repeating", [
 00:00:00 "POST /login.php"
]),
webshell: ("exact", [
 00:00:00 "GET /login.php"
 00:00:00 "POST /login.php"
 00:00:10 "HEAD /static/x.php"
]),
```

```
# time-b.   # line-b.   # Apache access logs
hydra       hydra       17:49:55 "POST /login.php"
hydra       hydra       17:50:03 "POST /login.php"
hydra       -           17:50:05 "GET /nag/"
hydra       hydra       17:50:10 "POST /login.php"
-           -           17:50:45 "GET /login.php"
-           -           17:50:51 "POST /login.php"
-           -           17:51:02 "GET /services/portal"
-           -           17:51:38 "GET /imp/dynamic.php"
webshell    webshell    17:52:10 "GET /login.php"
webshell    webshell    17:52:10 "POST /login.php"
webshell    -           17:52:12 "GET /imp/view.php"
-           webshell    17:52:20 "HEAD /static/x.php"
```

Fig. 6. Example of our labeling procedure. Information on attack execution (top left) and expected attack logs (top right) are used to create labels (bottom left) using time-based and line-based techniques for log data (bottom right).

lines are identical and thus achieve a perfect similarity score. The "exact" technique matches the three expected lines of the "webshell" attack step within all lines occurring in the attack time frame to find and label their counterparts. Note that it is possible to specify the temporal offset through the timestamp in the attack dictionary, e.g., "HEAD /static/x.php" is expected to occur 10 s after the first two lines of the "webshell" attack step.

E. Implementation

We implemented the outlined concept for the automatic generation of testbeds using model-driven techniques. Fig. 7 shows an overview of the typical workflow for testbed and log data generation. As visible in the figure, we use the infrastructure-as-a-service tool Terraform[10] to instantiate the testbed infrastructures as virtual machines on an Openstack[11] cloud platform using our predefined setup scripts. Configurations at this point involve the total number of machines, operating systems, and computational resources, e.g., memory.

Building the machines with Terraform yields a so-called state file that contains deployment information, such as IP addresses. The testbed script generator implemented in Python that acts as the transformation engine of our model-driven proof-of-concept implementation imports the state file together with a configuration file, system behavior and attack TIMs, and thesauri, i.e., word lists arranged by topics such as usernames, passwords, and host names. The configuration contains lower and upper limits for parameters that are randomly chosen when generating TSMs, i.e., files and executable scripts. Moreover, the transformation function generates a playbook that specifies the services to be

[10][Online]. Available: https://www.terraform.io/
[11][Online]. Available: https://www.openstack.org/

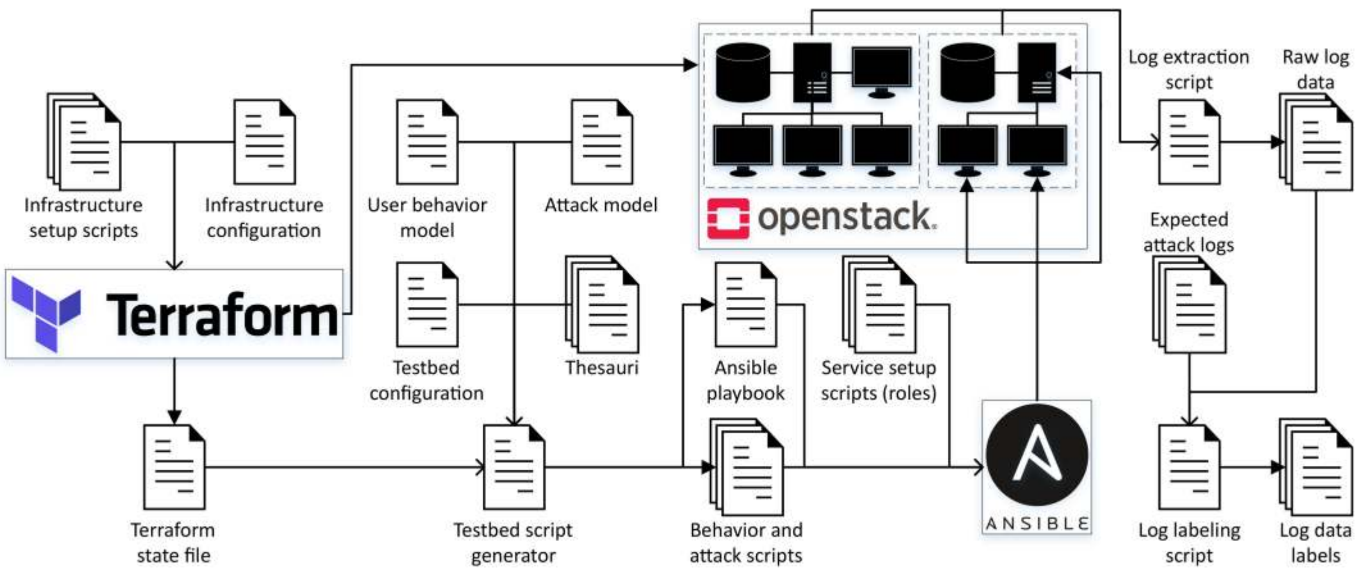


Fig. 7. Technical implementation of the model-driven testbed and log data generation approach. Simple arrows indicate imports and filled arrows indicate generation of resources such as scripts, configuration files, or machines.

installed, which are referred to as roles. Examples for such roles are PHP, Apache for web server setup, MariaDB for database setup, suricata IDS, or Internet browsers. Each role requires a setup script that states a list of tasks to be carried out. Thereby, it is possible to use variables in the playbook to specify random modifications of the setup process, e.g., install different versions, or replace them with alternative roles altogether. We then use the application-deployment tool Ansible[12] to distribute all generated files, set up services, and start the execution of user and attack scripts.

Note that roles have dependencies that have to be deployed before initiating the installation of the dependent role. Fig. 8 shows an overview of all roles currently available and their dependencies. As visible in the figure, the roles for installing Horde Webmail and OkayCMS require several other roles for database setup, user management, mailing services, etc. The user automation scripts as well as the attacks on the respective web services in turn require the availability of Horde Webmail and OkayCMS to access the web pages. In addition, some services are depending on specific versions, e.g., the vulnerable Exim version requires a specific Debian snapshot.

The right side of Fig. 7 shows that once the simulation is complete, another script collects all log files from the virtual machines and stores them on disk. As outlined in the previous section, we automatically label the logs using attack execution information extracted together with the other logs as well as a predefined dictionary of expected log lines for each attack step. We store the lists of generated labels in separate files.

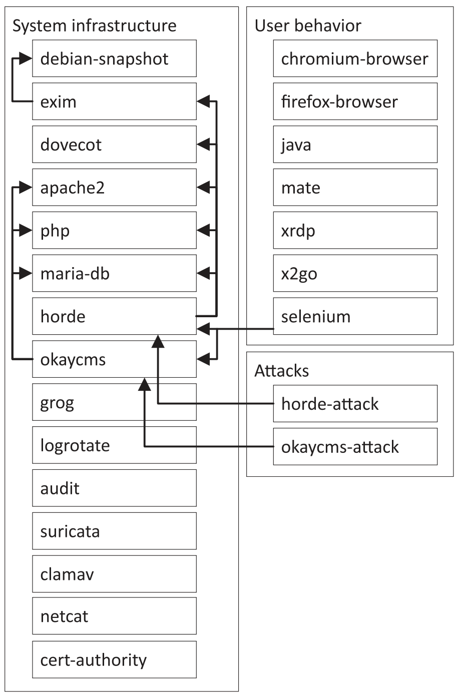


Fig. 8. Overview of role dependencies. To install any role, it is necessary that all roles that the attached arrows point to are available and correctly installed.

V. Validation

We devote this section to the validation of our approach and discovery of limitations. We first evaluate whether our approach adheres to the design principles defined in Section III-A. We then analyze the collected log data and show the effects of automatically selected parameters on the system behavior. In addition, we discuss selected case examples to demonstrate the simplified process of iterative testbed development.

A. *Fulfillment of Design Principles*

We selected a simple web server to target a realistic and common use-case. However, real web servers may be accessed

[12][Online]. Available: https://www.ansible.com/

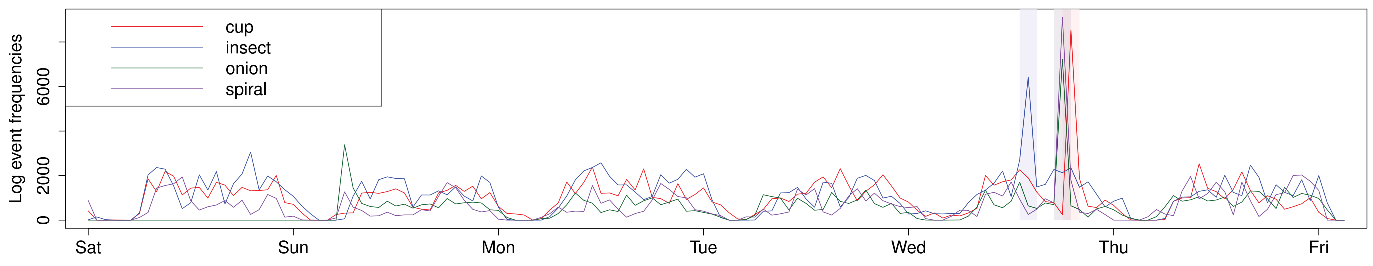


Fig. 9. Event frequencies of Apache access logs. Scans executed as part of a multistep attack manifest themselves as peaks (shaded intervals).

by humans as well as bots with extremely high frequency and diverse behavior patterns. While our approach theoretically allows to add arbitrary numbers of user hosts, the total amount of machines is limited by the available computational resources and may thus not represent the heavy loads present in real networks. In addition, we did not use real user activity measurements to define behavior parameters, but argue that our model-driven approach makes it easy to adjust the TIM appropriately if such data is available. The prepared attacks are realistic, relevant, and make use of recently discovered exploits. Finally, all used log sources were either left in their standard configurations or were realistically adapted.

Our approach fulfills all three dimensions of the flexibility principle due to the incorporation of model-driven techniques. The number of testbeds and sizes of the networks only depend on the predefined amount of machines. Changing components or user and attack behavior is easy by modifying TIMs. For example, it is simple to extend the state machine that represents an independent model of the user behavior by adding new states for particular actions while leaving everything else untouched. Since all the configuration files are reusable, it is possible to recreate the overall system behavior multiple times and thus reproduce the results. In addition, all technologies used in our scenario as well as the tools used to generate the testbed (Terraform, Openstack, Ansible) are open source.

We made all produced log data available online in documented form. In addition, we provide labels for the logs created by time-based and line-based methods. The labels are on the level of attack steps and thus support the evaluation of IDSs. Finally, our generated data covers several days and thus contains several periods of repeating patterns, which allows anomaly detection tools to learn a baseline of normal behavior.

B. *Manifestations of Testbed Variations*

Depending on the types and characteristics of variations, different log files are affected in particular ways, e.g., by event appearances or changed parameters. In the following, we focus on event frequencies as a measure to compare testbeds. We analyze Apache access logs, because they keep record of page visits on Horde Webmail and OkayCMS, and thus allow to reconstruct user behavior, which is subject to variation.

Fig. 9 shows user access frequencies on four web servers *cup*, *insect*, *onion*, and *spiral*, aggregated in time windows of 1 h over six days. The plot depicts that users access the server more frequently during the day than at night, resulting in a daily cycle. The peaks (shaded intervals) are caused by the scans as part of the multistep attacks. Note that additional detection techniques are required to disclose manifestations of the remaining attack steps. Since the amount of users per web server is selected randomly in order to increase variation, the average access rates differ among the web servers.

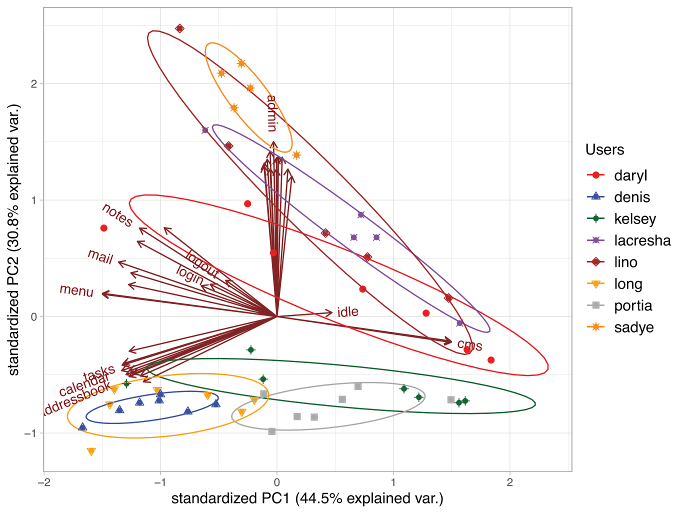


Fig. 10. Biplot of user page visit frequencies aggregated in daily intervals.

We further retrieve activity logs from each user. Since transition probabilities of the behavior state machines are specific to each user and remain constant over time, it is possible to relate observed behavior to users. For this, we compute the relative frequencies of accessed web pages for each user in time intervals of one day and use principal component analysis to scale down the resulting high-dimensional data. Fig. 10 shows a biplot[13] containing daily user behavior as scores (visualized as points) and the influence of visited pages on principal components as loadings (visualized as vectors). The ellipses represent the normal distributions of the daily user activities and show that each user follows a distinct pattern. The behavior spectrum includes admin users (*daryl*, *lacresha*, *lino*, and *sadye*) and shows overlaps between users following similar behavior profiles, e.g., *denis* and *long*.

[13]R package available at https://github.com/vqv/ggbiplot

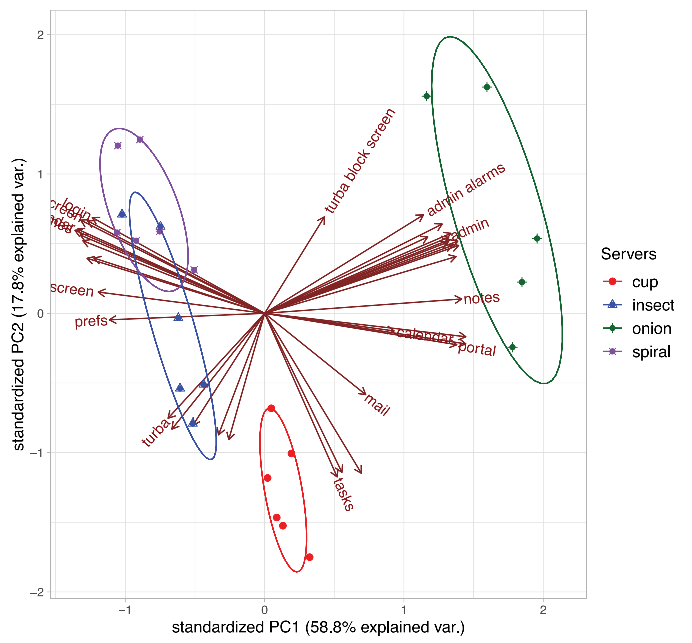


Fig. 11. Biplot of Apache access logs collected from different testbeds.

We also extract page visits from Apache access logs collected at the web servers. Note that we do not attempt to trace individual accesses to specific users; instead, we analyze differences of the overall behavior observed at the web servers. Fig. 11 is another biplot that shows groups of daily page visits on the web servers. Despite the aggregation of different user behavior patterns, the activities on the web server form distinct groups, for example, *onion* is located far away due to its high activity of admin users. The figures suggest that our approach achieved to generate testbeds with variations.

C. Case Examples of Testbed Extensions

We designed the presented approach to simplify iterative testbed development and support variations across generated testbeds. Our experiences during the development of our proof-of-concept implementation (cf., Section IV) endorsed the achievement of this target. In particular, we proceeded by adding new TIMs as reusable modules and repeatedly built and destroyed testbeds to test new features. In the following, we discuss three case examples of such extensions in detail and measure the required manual work in lines-of-code that were adapted.

1) Tool: System administrators install different tools on web servers based on domain knowledge and personal preference. We selected the Clam AntiVirus software[14] as an exemplary tool to be installed on some testbeds. Since there are no dependencies to other modules, another infrastructure TIM with ten lines of code is required to define a new role that contains two tasks that install the software and set up a cron job that regularly performs scans. Thereby, we leave the scheduled scan time as a variable. In the transformation engine that generates and populates the testbed setup scripts, we add 14 lines of code to specify the probability for installing Clam Antivirus, set the scan intervals, and add the resulting parameters to the Ansible playbook.

2) Browser: Since real users prefer different browsers for accessing web platforms, we planned to add Firefox[15] as an alternative to Chromium.[16] Similar to the antivirus tool, this implies creating a role with a single task consisting of 6 lines that specify the installation details. However, it is further necessary to change the existing TIM of the user behavior by adding a task that copies the required browser drivers for web automation (four lines of code) and adapting the user behavior script to support the new browser (five lines of code). Finally, a single line is edited in the transformation engine that randomly assigns one of the available browsers to each user profile.

3) Web Platform: At first, only Horde was implemented in the testbed. To increase diversity of the generated log data, we then decided to extend the simulation to also include OkayCMS. For this, we first set up and configured an OkayCMS instance. Once this was accomplished, a role with only 23 lines of code was required to specify three tasks that copy the OkayCMS instance in the appropriate webroot and set up the database to make the web store accessible to the users. Around 120 lines of code were necessary to update the state machine in the user behavior TIM so that users are able to navigate four pages of the website and perform adequate actions. Finally, three lines of code in the transformation engine specify the transition probabilities between the states.

VI. Discussion

In this article, we propose to shift from traditional testbed setup to model-driven testbed design in order to overcome common issues, including high manual efforts and repeated work when adjusting or upgrading components. In the previous sections, we discuss several design aspects and show examples of TIMs and automatically generated TSMs. In the following, we will outline possible use-cases and review limitations that could provide ideas for future work.

A. Applications

There are several promising use-cases for our model-driven testbed generator. Foremost, our main intention is to automatically build testbeds for generating log datasets suitable for IDS evaluation without the need to start from scratch for every new use-case, but instead reuse existing components and develop testbeds iteratively. For example, starting from our proof-of-concept, it is possible to introduce and exploit new vulnerabilities by changing only the affected components and attacks while leaving everything else untouched. Another idea is to model account hijacking by changing a user profile at some particular point in time, which could be the focus of detection tools based on user profiling.

Since testbeds are isolated from real networks and thus do not produce sensible data that could raise privacy concerns, the

[14][Online]. Available: https://www.clamav.net/

[15][Online]. Available: https://www.mozilla.org/en-US/firefox/

[16][Online]. Available: https://www.chromium.org/

generated log data is always suitable to be shared with others. Moreover, it is simple to create multiple variants of the same testbed in parallel and generate several datasets that represent different environments. This improves the robustness of results from IDS evaluation and allows researchers to measure the variation of the detection capabilities of their IDSs.

Alternatively, it is possible to deploy IDSs directly in the generated testbeds by adding an appropriate setup script to the infrastructure TIM. In this case, the generation of log data is less relevant, and instead analysts are able to observe and measure the detection capabilities in real-time. This application scenario could be especially useful for experiments and live demonstrations, where attacks are injected manually.

Another relevant application case is malware analysis. Since it is easy to generate many testbeds with variabilities, inserting the code to deploy malware in the TIM allows to observe their behavior in different environments. This enables analysts to derive insights on the behavior of the malware without much effort spent on setting up the necessary machines. Then, the same attack can be deployed in testbeds with patched services to ensure that the intrusions fail in every case.

Finally, our provided datasets contain log data rather than network traffic and thus enable evaluation of host-based IDSs, a field where datasets are urgently needed [5]. In addition, since one of our injected attacks involves the execution of several steps, the resulting dataset is a great benefit for the research community around multistep attacks, where publicly available datasets are rare [7]. Even more so, the variations of these multistep attacks across our generated datasets enable evaluation of algorithms that extract attack patterns independent of the environment and transform them into reusable cyber threat intelligence [10].

B. Limitations and Future Work

Despite the aforementioned benefits, we recognize some drawbacks of our method. First, it is necessary to point out that model-driven testbed design requires more effort than setting up a single static testbed, because all installation procedures have to be formalized and separated into fixed and variable parts that are subject to change, e.g., IP addresses have to be dynamically retrieved whenever they are necessary for a command. However, we argue that this increased initial effort pays off when testbeds are reused multiple times, especially when application scenarios are subject to change or multiple instances and variations of testbeds are required.

We further encountered that the ability to automatically upgrade all components to their newest versions in each rollout comes handy to ensure that the testbed is relevant to real-world scenarios, but possibly causes problems when services are dependent on each other or rely on version-specific configurations. In such cases, there is no way around manually fixing the TIMs, because such requirements of future versions cannot be foreseen. For critical components, it is possible to always install a fixed version, despite the downside that the service will eventually be outdated.

It is also important to note that generating data in our generated testbed requires the users to run in real-time. The reason for this is that it is infeasible to speed up the actions carried out by users, e.g., decreasing the sleep time between commands in the user behavior or attack scripts, since also the timestamps have to be adopted accordingly, i.e., the generated log data needs to be modified in hindsight. In addition, properties of the infrastructure, e.g., latencies and loading times, may have unrealistic influence on the log data when timestamps are changed, and eventually limit the possibility to increase the speed of the publication. Thus, it is not simply possible to simulate long timespans in a short amount of time.

Another limitation of our approach is that our line-based method for automatically labeling log messages corresponding to malicious activity is not guaranteed to always yield correct results and should thus only be seen as a complementary approach to the time-based method that provides additional confidence to the labels. The reason for this is that this method is based on string similarity, and as such is unable to differentiate between messages that are not sufficiently distinct, which leads to incorrect labeling. In addition, selecting the similarity threshold is nontrivial, since it depends on the overall structures of all possible log events. At the moment, gathering the expected logs for each attack step involves manual work, in particular, executing each attack step separately to populate the attack dictionary. Introducing new attack steps or changes of the logging infrastructure require to repeat this process. For future work, we are therefore planning to automatize this task so that it is executed before each simulation run. Nevertheless, attack steps that involve random or otherwise variable manifestations will remain difficult to label correctly. A possible solution could be to reconstruct the links between processes and their manifestations by analyzing system traces. In addition, it is unclear how overlapping attacks should be labeled, since this would require a more complex syntax, e.g., lists of labels for each line.

Regarding the log dataset produced in our proof-of-concept, we see a number of extensions that could improve future simulations. First of all, the authenticity of the user behavior can be improved by deriving parameters from real system usage, which was omitted due to lack of such real data. In addition, randomness is usually based on uniform distributions, however, actual user behavior could be better represented by other distributions, such as the normal distribution.

Finally, it would be interesting to develop a formal modeling language for generating testbeds. Thereby, the transformation engine would work as a function that selects properties of infrastructure components, user behavior, and attacks, from the predefined ranges of allowed values. This would help to define a metric that makes testbeds comparable by measuring their similarity through their common properties. Aggregating such a testbed similarity metric over all generated testbeds, it would be possible to provide the analyst with a feedback on the diversity of the testbeds, i.e., a measure on the coverage of possible combinations of model parameters. Ultimately, the resulting aggregated metric could be used to determine whether an appropriate amount of testbeds have been generated to represent most possible testbed configurations, or to calculate an estimation for the number of testbeds required.

VII. Conclusion

In this article, we proposed a methodology for creating testbeds for log data generation using techniques from MDE. For this, we designed abstract models for the testbed infrastructure, the simulated system behavior, and the injected attacks, and used a transformation engine to automatically translate these TIMs into testbed-specific scripts and configuration files that allow deployment. This increases the required initial effort, but largely reduces the amount of work required to maintain and modify testbeds for different application scenarios. Due to the fact that TIMs only define parameters as discrete lists or ranges of allowed values, we were able to generate arbitrary numbers of testbeds with variations.

The testbeds aligned with our design principles authenticity, flexibility, reproducibility, availability, and utilizability. We demonstrated our approach by a proof-of-concept implementation that involves users accessing a webmail platform and online store as well as one multistep attack and one complex vulnerability exploit. We used our demonstrator to generate four testbeds and collect log datasets. Log messages related to attack steps were then labeled based on their time of occurrence and their similarity to predefined log patterns, and were thus useful for the evaluation of host-based IDS.

References

[1] C. Thomas, V. Sharma, and N. Balakrishnan, “Usefulness of DARPA dataset for intrusion detection system evaluation,” in *Proc. SPIE*, vol. 6973, 2008, Art. no. 69730G.

[2] M. Wurzenberger, F. Skopik, G. Settanni, and W. Scherrer, “Complex log file synthesis for rapid sandbox-benchmarking of security- and computer network analysis tools,” *Inf. Syst.*, vol. 60, pp. 13–33, 2016.

[3] F. Skopik, G. Settanni, R. Fiedler, and I. Friedberg, “Semi-synthetic data set generation for security software evaluation,” in *Proc. 12th Annu. Int. Conf. Privacy, Secur. Trust*, 2014, pp. 156–163.

[4] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, “Toward generating a new intrusion detection dataset and intrusion traffic characterization,” in *Proc. Int. Conf. Inf. Syst. Secur. Privacy*, 2018, pp. 108–116.

[5] D. Čeponis and N. Goranin, “Towards a robust method of dataset generation of malicious activity for anomaly-based HIDS training and presentation of AWSCTD dataset,” *Baltic J. Modern Comput.*, vol. 6, no. 3, pp. 217–234, 2018.

[6] G. Maciá-Fernández, J. Camacho, R. Magán-Carrión, P. García-Teodoro, and R. Theron, “UGR 16: A new dataset for the evaluation of cyclostationarity-based network IDSs,” *Comput. Secur.*, vol. 73, pp. 411–424, 2018.

[7] J. Navarro, A. Deruyver, and P. Parrend, “A systematic survey on multi-step attack detection,” *Comput. Secur.*, vol. 76, pp. 214–249, 2018.

[8] B. Stojanović, K. Hofer-Schmitz, and U. Kleb, “APT datasets and attack modeling for automated detection methods: A review,” *Comput. Secur.*, vol. 92, 2020, Art. no. 101734.

[9] F. Galan, J. E. L. de Vergara, D. Fernandez, and R. Munoz, “A model-driven configuration management methodology for testbed infrastructures,” in *Proc. Netw. Oper. Manage. Symp.*, 2008, pp. 747–750.

[10] M. Landauer, F. Skopik, M. Wurzenberger, W. Hotwagner, and A. Rauber, “A framework for cyber threat intelligence extraction from raw log data,” in *Proc. Int. Workshop Big Data Analytics Cyber Threat Hunting*, Dec. 2019, pp. 3200–3209.

[11] M. Ring, S. Wunderlich, D. Grüdl, D. Landes, and A. Hotho, “Flow-based benchmark data sets for intrusion detection,” in *Proc. 16th Eur. Conf. Cyber Warfare Secur.*, 2017, pp. 361–369.

[12] K. Ali, “Algorizmi: A configurable virtual testbed to generate datasets for offline evaluation of intrusion detection systems,” Master’s thesis, Department of Computer Science, Univ. Waterloo, Waterloo, ON, Canada, 2010.

[13] G. Singaraju, L. Teo, and Y. Zheng, “A testbed for quantitative assessment of intrusion detection systems using fuzzy logic,” in *Proc. 2nd Int. Inf. Assurance Workshop*, 2004, pp. 79–93.

[14] G. Creech and J. Hu, “Generation of a new IDS test dataset: Time to retire the KDD collection,” in *Proc. Wireless Commun. Netw. Conf.*, 2013, pp. 4487–4492.

[15] M. Richmond, “ViSe: A virtual security testbed,” Master’s thesis, Dept. Comput. Sci., Univ. California–Santa Barbara, Santa Barbara, CA, USA, 2005.

[16] L. M. Rossey *et al.*, “LARIAT: Lincoln adaptable real-time information assurance testbed,” in *Proc. Aerosp. Conf.*, 2002, vol. 6, p. 6.

[17] T. Benzel *et al.*, “Design, deployment, and use of the DETER testbed,” in *Proc. DETER Community Workshop Cyber Secur. Exp. Test*, 2007, pp. 1–8.

[18] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford, “Virtual playgrounds for worm behavior investigation,” in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2005, pp. 1–21.

[19] M. Frank, M. Leitner, and T. Pahi, “Design considerations for cyber security testbeds: A case study on a cyber security testbed for education,” in *Proc. IEEE 15th Int. Conf. Dependable, Autonomic Secure Comput., 15th Int. Conf Pervasive Intell. Comput., 3rd Int. Conf. Big Data Intell. Comput. Cyber Sci. Technol. Congr.*, 2017, pp. 38–46.

[20] J. Cappos, Y. Zhuang, A. Rafetseder, and I. Beschastnikh, “Tsumiki: A meta-platform for building your own testbed,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2863–2881, Dec. 2018.

[21] D. C. Schmidt, “Model-driven engineering,” *IEEE Comput. Soc.*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[22] B. P. Lamancha, M. Polo, D. Caivano, M. Piattini, and G. Visaggio, “Automated generation of test oracles using a model-driven approach,” *Inf. Softw. Technol.*, vol. 55, no. 2, pp. 301–319, 2013.

[23] S. J. Stolfo, S. Hershkop, C.-W. Hu, W.-J. Li, O. Nimeskern, and K. Wang, “Behavior-based modeling and its application to email analysis,” *Trans. Internet Technol.*, vol. 6, no. 2, pp. 187–221, 2006.

[24] M. Wurzenberger, F. Skopik, M. Landauer, P. Greitbauer, R. Fiedler, and W. Kastner, “Incremental clustering for semi-supervised anomaly detection applied on log data,” in *Proc. 12th Int. Conf. Availability, Rel. Secur.*, 2017, pp. 1–6.

Max Landauer received the bachelor’s degree in business informatics from Vienna University of Technology, Vienna, Austria, in 2016. Since 2018, he has been working toward the Ph.D. degree in business informatics with the Austrian Institute of Technology (AIT), Vienna.

In 2017, he joined the AIT, where he carried out his master’s thesis. He is currently a Junior Scientist with the AIT. His research interests include log data analysis, anomaly detection, and threat intelligence.

Florian Skopik received the Ph.D. degrees in computer science from the Institute of Information Systems Engineering, Vienna University of Technology, Vienna, Austria, in 2010 and 2013, respectively.

He joined the Austrian Institute of Technology (AIT), Vienna, in 2011 and is the Thematic Coordinator of AIT’s Cyber Security Research Program. He coordinates national and international (EU) research projects, as well as the overall research direction of the team. He authored/coauthored more than 100 scientific conference papers and journal articles and holds some 20 industry-recognized security certifications. The main topics of his projects focus on smart grid security, the security of critical infrastructures, and national cyber security.

Dr. Skopik is a member of various conference program committees and editorial boards, as well as standardization groups, such as ETSI TC Cyber and OASIS CTI.

Markus Wurzenberger received the bachelor's degree in science and technology from the Department of Mathematics, Vienna University of Technology, Vienna, Austria, in 2013, and the master's degree in technical mathematics from Austrian Institute of Technology (AIT), Vienna, Austria, in 2015. He is currently working toward the Ph.D. degree with the Institute of Computer Engineering, Vienna University of Technology.

In 2014, he joined AIT as a Freelancer. In the end of 2015, he joined AIT as a Junior Scientist, where he is currently working on national and international projects in the context of anomaly detection.

Wolfgang Hotwagner received the bachelor's degree in information and communication systems from the Department of Computer Science, University of Applied Sciences Technikum Wien, Vienna, Austria, in 2017.

He worked as a Linux System Administrator, where he gained experiences in automation and blue teaming. He completed the certifications "Offensive Security Certified Professional," "Offensive Security Certified Expert," "Offensive Security Web Expert," and "AWS Certified Solutions Architect." Furthermore, he was significantly involved in the development of Austrian Institute of Technology's Cyber Range. He has expertise in algorithm testbed design, software development, vulnerability mining, DevOps, and system and cloud architectures.

Andreas Rauber (Member, IEEE) received the Ph.D. degree in computer science from the Vienna University of Technology, Vienna, Austria, in 2001.

He is the Head of the Information and Software Engineering Group, Department of Information Systems Engineering, Vienna University of Technology. He is also the President of the Austrian Association for Research in IT and a Key Researcher with the Secure Business Austria. He has authored/coauthored numerous papers in refereed journals and international conferences.

Dr. Rauber was the PC member and Reviewer/Editorial Board Member for several major journals, conferences, and workshops. He is a member of the Association for Computing Machinery and the Austrian Society for Artificial Intelligence (GAI).