Automated Evaluation of Network Intrusion Detection Systems in IaaS Clouds

Thibaut Probst, Eric Alata, Mohamed Kaâniche, Vincent Nicomette

▶ **To cite this version:**

Thibaut Probst, Eric Alata, Mohamed Kaâniche, Vincent Nicomette. Automated Evaluation of Network Intrusion Detection Systems in IaaS Clouds. 11th European Dependable Computing Conference (EDCC 2015), Sep 2015, Paris, France. hal-01212064

HAL Id: hal-01212064
https://hal.science/hal-01212064

Submitted on 6 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Automated Evaluation of Network Intrusion Detection Systems in IaaS Clouds

T. Probst[1,2], E. Alata[1,3], M. Kaâniche[1,4], V. Nicomette[1,3]
[1]CNRS, LAAS, 7 Avenue du colonel Roche, F-31400 Toulouse, France
[2]Univ de Toulouse, INP de Toulouse, LAAS F-31400 Toulouse, France
[3]Univ de Toulouse, INSA de Toulouse, LAAS F-31400 Toulouse, France
[4]Univ de Toulouse, LAAS, LAAS F-31400 Toulouse, France
{probst,ealata,kaaniche,nicomett}@laas.fr

***Abstract*—This paper describes an approach for the automated security evaluation of operational Network Intrusion Detection Systems (NIDS) in Infrastructure as a Service (IaaS) cloud computing environments. Our objective is to provide automated and experimental methods to execute attack campaigns and analyze NIDS reactions, in order to highlight the ability of the NIDS to protect clients' virtual infrastructures and find potential weaknesses in their placement and configuration. To do so, we designed a three-phase approach. It is composed of the cloning of the target client's infrastructure to perform the subsequent audit operations on a clone, followed by the analysis of network access controls to determine the network accessibilities in the cloned infrastructure. Using evaluation traffic we modeled and generated, the last phase of the approach, presented in this paper, focuses on executing attack campaigns following an optimized algorithm. The NIDS alerts are analyzed and evaluation metrics are computed. Our approach is sustained by a prototype and experiments carried out on a VMware-based cloud platform.**

***Keywords*—security, evaluation, attacks, cloud, NIDS.**

I. Introduction

In cloud computing environments, providers offer various resources as services, including infrastructures, platforms and applications. These services are subscribed by clients willing to reduce the deployment and operation costs of their traditional on-premise business. The IaaS cloud service model allows the creation and management of entire virtualized infrastructures, bringing together virtual datacenters (vDCs) hosting virtual machines (VMs), networks and storage. In addition, network security mechanisms are deployed to protect the data hosted on virtualized infrastructures from network attacks. Firewalls are responsible for packet filtering to control the network access. On the other hand, NIDS are in charge of detecting attacks occurring on communication paths.

The objective of security administrators (either on the provider side or the client side) is to prevent and detect attacks, while not disturbing the rest of the cloud computing environment (for example, avoid the consumption of too much resources by the security mechanisms). Making firewalls and NIDS work efficiently is not an easy task. Indeed, the deployed products have to be kept constantly up-to-date, properly configured, and correctly positioned in the network. In addition, cloud environments constantly evolve over time. Existing clients may add or remove VM instances and networks, or modify configurations in their vDCs; new clients subscribe to services and create new infrastructures; providers also administrate and modify some components in the cloud. This dynamicity could have negative impacts on the cloud security. Therefore, it is important for the client and the provider to monitor and analyze at a regular basis the security level of cloud infrastructures, in order to adapt and improve the deployment of the security tools.

Our goal is to be able to highlight the aforementioned issues by conducting automated evaluations of cloud network security mechanisms. These evaluations would aim at analyzing network access controls within a given client's virtual infrastructure, and also assessing NIDS efficiency in monitoring and protecting this infrastructure. Because we want to conduct exprimentally-driven audits but cannot afford to disturb the target client's business, we chose to perform the audits using a clone of the client's infrastructure. However, we chose not to copy the client's VM instances into the cloned infrastructure for the following reasons: 1) we want to preserve the client's privacy; 2) we are not interested in assessing the VMs security but the network security; 3) we need to handle the VMs to perform network audit operations. Instead, only the entire network configuration and firewalls of the initial infrastructure are cloned, and the initial VMs are replaced with customized ones, imported from a template we built and thus provided with the necessary audit tools. The imported VMs are provided with the same network configuration as the original ones. Next, we decided to perform the analysis of network access controls before the evaluation of NIDS. Indeed, we need to know the network communication paths, so-called accessibilities, in order to execute attack campaigns on these paths to assess the NIDS reaction. The network accessibility analysis is done both statically, by parsing the cloud components configuration and using the retrieved information in a logic engine; and dynamically, by sending network packets. The results offered by both methods can be compared to look for discrepancies in the configuration and implementation of access controls. This work has already been presented in [**?**]. This paper focuses on the generation and execution of attack campaigns for NIDS assessments using the network accessibilities. Hence, our main contribution, presented in this paper, is an experimental method to perform automated network attack campaigns in cloud environments with different possible parameters, and monitor the NIDS reaction. As an outcome, we are able to generate traditional evaluation metrics and highlight potential weaknesses in NIDS deployments. Moreover, the cloning followed by the analysis of access controls and the evaluation of NIDS constitute a three-phase audit approach. We leverage cloud

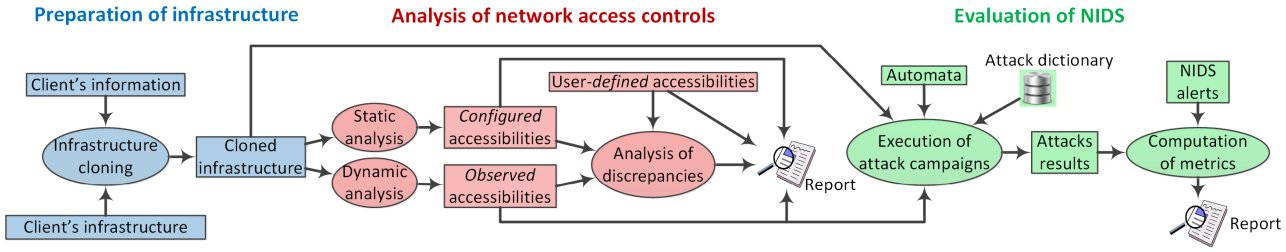

Fig. 1: Overall process of the three-phase approach

benefits to perform quicker and deeper audits, and to make the process fully automated.

The rest of the paper is organised as follows. Section II gives an overview of our three-phase approach. Section III presents how we model, generate and use the evaluation traffic utilized in the attack campaigns execution, presented in Section IV. Section V presents a VMware-based testbed environment used to validate our approach, along with the experimental results. Section VI discusses related work. Finally, conclusion and future work are provided.

II. Overview of the Approach

In this section, we recap the main assumptions we consider in our approach, and then we explain its principles.

A. Main assumptions

1) Target environment: Our approach focuses on the virtual infrastructure level, that is why the considered cloud service model is IaaS. A virtual infrastructure is defined as a set of vDCs, where a vDC includes VMs, networks, and firewalls.

We assume that the firewalls apply stateful packet inspection, and we consider two different types of virtual firewalls commonly found in the cloud:

- Edge firewall: gateway for client's virtual networks that routes, filters and translates inbound or outbound traffic. It is generally controlled by the client.
- Hypervisor-based firewall: introspects the traffic sent and received by VMs disregarding the network topology. It is generally controlled by the provider. However, some rules can be configured by the clients on their network scopes, which can give them a partial control on it.

As for the NIDS, even if some approaches ([**?**], [**?**]) already suggested a concept of *IDS as a service* offered to the clients, we assume that the NIDS are deployed and controlled by the provider, in charge of detecting the attacks run from, to, and within the clients' networks. We also assume that NIDS apply signature-based detection techniques.

2) Use cases: Our system is designed to evaluate network security of small-sized to medium-sized virtualized infrastructures (< 50 VMs). The security analysis should not disturb the client's business. Also, it is intended to be fully automated (it requires the minimum of human intervention), and the related audit operations are run on behalf of the provider, therefore using administrator privileges on the cloud components. Moreover, provided results correspond to the state of the system at the time when the audit is run. This paper focuses mainly

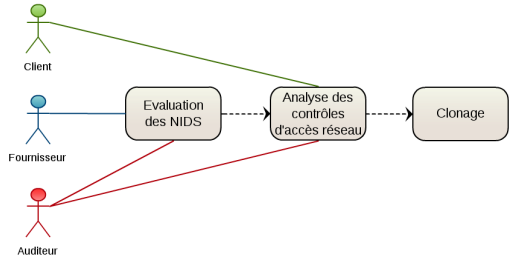

Fig. 2: Use cases of the system

on the approach itself, and we do not constrain the usage of the audit system to specific actors. Indeed, it depends on the Service Level Agreement (SLA) between the provider and its clients. However, considering three possible actors (client, provider and auditor), we recommend to use the features of our system as shown in Figure 2. The client can ask for an analysis of network access controls of its infrastructure (which entails the cloning of it). The provider can conduct an evaluation of NIDS towards a client's infrastructure (which requires an analysis of network access controls, and thus a cloning). As a third-party actor, the auditor can require an evaluation of NIDS, as well as an analysis of network access towards any client's infrastructure.

3) Compliance: We believe that our approach helps implementing the Cloud Security Alliance (CSA) recommendations in terms of security assessments [**?**]. Among the topics addressed in this document, our approach provides assistance in the following areas :

- Network and System Vulnerability Assessments.
- Network/Security System Compliance Assessments.
- Virtual Infrastructure Assessment.
- Web Application Security Assessments.
- Internal/External penetration testing.
- Security Controls Assessments.

B. Principles

Our approach is a three-phase approach, illustrated in Figure 1. Even if this paper focuses on the last phase, we sum up the first two phases, preliminary to the last one.

The first phase consists in cloning the client's virtual infrastructure so the client's privacy is preserved, i.e., its production instances are not accessed, used and disturbed in the following steps. Nevertheless, for the reasons developed in the Introduction, we do not copy the initial VMs instances into the cloned infrastructure. Instead, VMs are imported from

a predefined template[1]. They are placed in the same network location and with the same network configuration as copies of the initial ones would be. The newly created VMs are equipped with our audit tools and ready to run attack campaigns as explained further on. Note that whereas it is aimed at copying a client's infrastructure, the creation of a cloned infrastructure can also be assimilated to the arrival of a new IaaS client in the cloud. It can be summarized as follows:

1) Fetch the necessary configuration information from the client's vDCs: IP addressing, network connections, firewalls configuration, etc.
2) From the fetched configuration information, create new vDCs ready to host new networks, firewalls and VMs.
3) From the fetched configuration information, create new networks and firewalls in the newly created vDCs.
4) From the fetched configuration information and our template, create new VMs in the newly created vDCs.

The second phase analyzes network access controls statically and dynamically to generate a security analysis report containing the end-to-end network accessibilities (between the VMs, and between the VMs and external networks) found by both methods. This phase has been presented in [**?**]. The static analysis parses the cloud configuration (VMs, networks and firewalls configuration), and translates this information into predicates. These predicates are used in an optimized logic engine running an algorithm able to determine the configured network accessibilities. The dynamic analysis consists in sending network packets between the VMs, following an algorithm that performs as many network exchanges in parallel as possible, to be as fast as possible, in order to determine the observed network accessibilities. The accessibilities found by both methods are compared with each other, and also with the user-defined accessibilities to identify potential discrepancies in the results. The network accessibilities are essential because they are used in the third and last phase, which consists in performing attack campaigns. Indeed, no network attack would be possible on a blocked path, that is why we need to know the accessibilities.

The last phase deals with the evaluation of NIDS by executing attack campaigns over the virtual infrastructure protected by the NIDS. The associated algorithm uses three main elements:

- The network accessibitilies as an accessibility matrix: the allowed communication paths derived from dynamic analyis of network access controls. We chose results from dynamic rather than static analysis as they are ascertained from packets sendings, which is more foolproof and reveals the actual network reachability.
- An attack dictionary: a database containing necessary information about the attacks.
- A data set of malicious and legitimate evaluation traffic, with tools to replay this traffic.

After the execution of the attacks, the NIDS alerts are analyzed to compute evaluation metrics. As it is the core contribution of this paper, this phase is developed in details in the next sections.

[1]The template used in our prototype is detailed at the end of Section V-A.

III. Evaluation traffic model

Assessing the efficiency of NIDS deployed in the cloud implies executing network attack campaigns that involves sending specific network evaluation traffic. Model-based approaches are well suited to automate this process. We chose to model this evaluation traffic (malicious and legitimate) with a certain level of abstraction, representing it as a set of automata. This abstraction is helpful to conviniently: 1) generate the automata in an offline stage, prior to the carrying out of the audit process and execution of attack campaigns; 2) replay them easily in a different context during the attack campaigns.

A. Modeling of traffic as automata

A traffic is defined as a sequence of network packets exchanged, being either malicious or legitimate. To conduct attack campaigns as explained later on, we plan to send malicious and legitimate traffic targeting vulnerable applications. Therefore, the VMs in the cloned infrastructure have to be able to send and receive this traffic. Several challenges arise. First, we cannot surmise what applications might be deployed any time in the virtual infrastructures. Thus, we cannot surmise what kind of attacks might be executed and so which have to be covered by the NIDS. Hence, we need a solution to model any evaluation traffic for any application. Another issue we faced is that many targeted applications are Windows-based, which would require a Windows license for every VM hosting those applications. Furthermore, one cannot run several applications listening on the same network port at the same time. Thus, the whole process of acquiring, installing, starting and stopping applications is expensive, tedious and hard to automate. Instead, we decided to model and replay some network behaviors of applications, in order to be able to respond to the exploits requests without installing and running the real applications. We are not interested in modelling complete automata representing entire specifications of network protocols. Indeed, we only model and replay some network flows: the malicious utilization of the application, along with some legitimate usages of it.

So far, we only considered application-based network attacks, running above TCP or UDP. We model the network exchanges between the attacking host and the target application as a finite-state automaton, where each state is represented as a sextuple including:

- The direction of the packet: received or sent by the application.
- The protocol: TCP or UDP.
- The port number.
- The identifier of the application.
- The TCP or UDP payload.
- Pointers to the child states.

The initial state of the automaton has empty values. Figure 3 shows an example of an automaton representing the network behavior of a FTP server application with two usages: a malicious one (in red) and a legitimate one (in green). The automaton actually merges two automata : the malicious one and the legitimate one. They could be merged because they are related to the same application, so the identical states (generally the first ones) are combined, as we can see on the figure, where the two behaviors share several states before

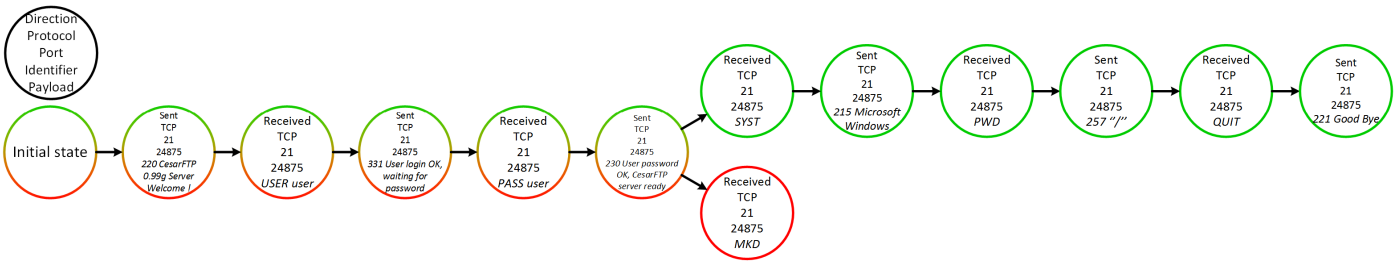

Fig. 3: Example of automaton modelling network exchanges of a legitimate (in green) and a malicious (in red) usage of a FTP application

splitting up. The shared states are related to the login to the FTP server, and the next legitimate states are basic legitimate FTP commands, whereas the malicious state corresponds to a command buffer overflow.

B. Generating the automata

Although it requires a human effort, the process of generating an automaton remains much less tedious than deploying applications on the fly, as it has to be done once and for all. To generate a set of automata, we set up a quite simple platform with two VMs directly connected. The target VM hosts the vulnerable applications and captures the network traffic, while the attacker VM is responsible for generating the requests to the application. The target is either a Windows-based VM or a Linux-based VM (depending on the application to model) running Wireshark [**?**] to capture network traffic. The attacker is a Linux-based VM, running Metasploit [**?**] and some command-line programs to generate network requests:

- `wget` and `curl` as HTTP clients.
- `ftp` as FTP client.
- `telnet` as SMTP and IMAP client.

We use Shell scripts to generate several requests (done by `wget` or `curl`) for non interactive network protocols like HTTP. On the other hand, for interactive ones like FTP, SMTP or IMAP, we use *autoexpect* [**?**] to record the interactive sessions (conducted using `ftp` or `telnet`) and generate associated Expect scripts. The generation process is quite easy:

1) Install the application on the target and launch it.
2) Run Wireshark on the target and filter the capture.
3) Execute the exploit using Metasploit on the attacker.
4) Save the Wireshark capture file and use it to generate a pickle file automaton with our Python generation tool.
5) Execute the legitimate scripts on the attacker.
6) Save the Wireshark capture file and use it to generate a pickle file automaton with our Python generation tool.

Figure 4 illustrates the process of generating automata on our platform. The Python generation tool reads packets in the PCAP files using the Scapy [**?**] Python API, and creates automata as Python lists. These lists are saved as pickle files that could be easily loaded and merged as shown in the example of Figure 3.

C. Replaying the automata

Our goal is to get rid of the installation and configuration of the initial vulnerable application by replaying attacks and

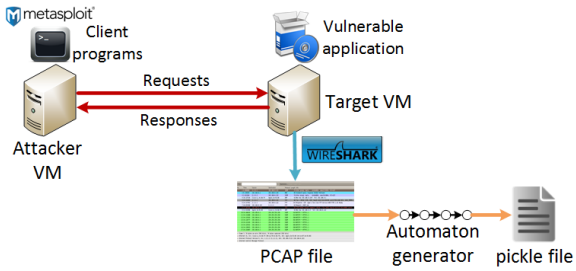

Fig. 4: Automata generation process

legitimate usages we already performed in the generation process. To do so, we designed a second Python tool (called the automata player), very lightweight and easily executable in any VM, which listens for incoming network packets to send the appropriate responses, according to the automata it loaded. Thus, we allow the exploit programs to send as many network requests as they did during the generation process, and as they would if targeting the real applications.

The automata player tool first reads the pickle files and loads the automata of the application, stored in the pickle files. Then it creates two threads:

- The first one creates a TCP socket to listen for incoming connections, and spawns a new thread everytime a new TCP connection is initiated. Those threads listen for incoming packets within their connection.
- The second one listens for incoming UDP packets.

The principle of response selection, applied upon the arrival of a request by calling the $select\_responses()$ function, is illustrated in Algorithm 1. The current state of the automaton is initialized to the initial state, and updated during the execution of the algorithm. Given a received payload, the goal is to match this received payload with the closest one from the current state or initial state of the automaton (function $find\_payload()$), and to look for the next response to send (function $find\_next\_payload()$). Note that the research of the closest payload in the automata is performed on the child states of the current state so that we can follow the packet exchanges. It is also performed on the child states of the initial state so that we can react to a new usage of the application. Therefore, we do not need to reload the automata every time a new behavior (legitimate or malicious) of an application is requested.

A received payload is delivered by a socket, and could come from one UDP packet, or several TCP packets. Indeed, the payload can be split over several packets in the case of TCP. Unlike a message-oriented protocol like UDP, TCP is a

Algorithm 1 Selection of responses

Require: $initial\_state\_automaton$: pointer to the initial state of the loaded automaton.
Require: $current\_state\_automaton$: pointer to the current state of the loaded automaton, initialized to the initial state.

function PAYLOAD_MATCH($state, previous\_match\_score, payload1, payload2$)
    return $(levenshtein(payload1, payload2) > previous\_match\_score\ ?\ levenshtein(payload1, payload2) : False)$
end function
function FIND_PAYLOAD($state, payload, previous\_match\_score, tcp$)
    $score \leftarrow previous\_match\_score$
    for $s \in state["child\_states"]$ **do**
        if $s["received"]$ and $s["proto"] == tcp$ **then**
            $concat\_payloads \leftarrow s["payload"]$
            $last\_state \leftarrow s$
            if $s["proto"]$ **then** $(last\_state, concat\_payloads) \leftarrow concat\_tcp\_payloads(s, concat\_payloads)$ **end if**
            $match\_score \leftarrow payload\_match(score, concat\_payloads, payload)$
            if $match\_score$ **then**
                $score \leftarrow match\_score$
                $current\_state\_automaton \leftarrow last\_state$
            end if
        else
            $score \leftarrow find\_payload(s, payload, score, tcp)$
        end if
    end for
    return $score$
end function
function FIND_NEXT_PAYLOAD_TO_SEND($state, previous\_selected\_payload, tcp$)
    $payload\_to\_send \leftarrow previous\_selected\_payload$
    for $s \in state["child\_states"]$ **do**
        if $s["proto"] == tcp$ and ! $s["received"]$ **then**
            $payload\_to\_send \leftarrow s["payload"]$
            $current\_state\_automaton \leftarrow s$
        else if ! $s["proto"] == tcp$ **then**
            $payload\_to\_send \leftarrow find\_next\_payload(s, payload\_to\_send, tcp)$
        end if
    end for
    return $payload\_to\_send$
end function
function SELECT_RESPONSES($received\_payload, tcp$)
    $score \leftarrow 0$; $responses \leftarrow \emptyset$; $next\_payload \leftarrow$ ""
    $score \leftarrow find\_payload(current\_state\_automaton, received\_payload, score, tcp)$
    $score \leftarrow find\_payload(initial\_state\_automaton, received\_payload, score, tcp)$
    if ! $automaton\_finished(current\_state\_automaton)$ **then**
        $next\_payload \leftarrow find\_next\_payload\_to\_send(current\_state\_automaton, next\_payload, tcp)$
        while $next\_payload$ **do**
            $responses \leftarrow responses \cup next\_payload$
            $next\_payload =$ ""
            if ! $automaton\_finished(current\_state\_automaton)$ **then**
                $next\_payload \leftarrow find\_next\_payload\_to\_send(current\_state\_automaton, next\_payload, tcp)$
            end if
        end while
    end if
    return $responses$
end function

byte-oriented protocol that does not respect the initial payload message boundary and can fragment it in several packets. Thus, we considered two cases for selecting a response upon receiving a payload.

In the case of UDP, the received payload is compared with the payload of the child states of the current state, and also with the payload of the child states of the initial state. Then, from the state containing the matched payload, the next contiguous payloads of the same protocol (UDP) and sent in the opposite direction (packet sent by the application) are selected and sent as a response in UDP packets.

In the case of TCP, as long as the direction is the same (packet received by the application), the contiguous payloads of the child states of the current state and those of the initial

state are concateneted and compared with the received payload. Then, from the state containing the last matched concateneted payload, the next contiguous payloads of the same protocol (TCP) and sent in the opposite direction (packet sent by the application) are selected and sent as a response in a TCP flow.

Eventually, once a TCP or UDP response is sent, the current state of the automaton is updated to the state containing the last payload sent, and the automaton is checked to know if it has reached its final state. Note that we used an implementation [**?**] of the Levenshtein distance to compare the payloads.

In addition to be portable and command-line oriented, we believe that the implemented automata player tool is adequate for our needs. A string metric algorithm (like the Levenshtein distance) is sufficient to identify the payloads even if some elements of the received payload vary in the matched payload (i.e., the similarity is not perfect). As for binary protocols, the string metric algorithm applied to binary data payloads gives a fairly good performance from what we experimented. The recognition of the payload is quite simple given the restricted number of states in the automata, as we do not model entire network protocols. Also, from those we studied, the exploit codes do not perform a strong response analysis, that is why we do not conduct an analysis and modification of the payloads sent as responses according to the context of the execution. As for the NIDS, the signatures are generally triggered by the requests. Besides, even in the cases where the responses raised the alerts, we observed that variable parameters (timestamps, IP addresses, hostnames, versions, etc.) in the responses do not affect the way to trigger the alerts. This is actually coherent as signatures cannot depend on the context.

IV. Attack campaigns

This section presents our method to execute attack campaigns within the virtual infrastructure in an automated fashion. We first introduce the attack dictionary used in the algorithm presented subsequently. Then, we explain how IDS evaluation metrics are computed following the execution of attacks.

A. Attack dictionary

We needed an attack dictionary to store the necessary information about the attacks we are able to launch. An attack is characterized by the exploit used, its date, the targeted vulnerability, the associated network protocol and port, and its description. Thus, we implemented the attack dictionary as a simple SQLite database with one table, defined in by the attributes of Table I. To fill the data into the dictionary, we requested the free Exploit Database [**?**] website which allows us to get all the information we needed, sometimes along with the vulnerable application itself. The $automata$ field is the only field we update manually after the generation of automata associated to the application (cf. Section III). Note that $exploit\_id$ is used to name the automata (the pickle files).

B. Attack sessions

An attack campaign consists of a set of attack sessions. An attack session is the set of attacks executable between two hosts. An attack is the execution of an exploit from a host targeting another host on a given protocol and port. It is feasible if there is an accessibility between the two hosts on the given port, and if the exploit and the automaton are available. Attack sessions are accessibility-driven, so no attacks are launched if the network does not allow them, which regulates the total execution time of attack sessions. A standard attack is an attack occurring on a port the vulnerable application is generally supposed to run, for instance a Web attack targeting a Web server on port 80. A non-standard attack is an attack occurring on a port the vulnerable application is generally not supposed to run, for instance a Web attack targeting a Web server on port 67.

Different strategies of attack campaigns can be considered, by defining the proportion of standard and non-standard attacks executed on the accessibilities found. Thus, the current prototype provides three attack modes for the execution of standard attacks or non-standard attacks:

- Launch all the available standard/non-standard attacks per accessibility.
- Launch the R most recent available standard/non-standard attacks per accessibility.
- Launch R randomly chosen available standard/non-standard attacks per accessibility.

An attack session between two VMs (controlled by the hypervisor API) is performed by running, for each accessibility and following the attack modes, the standard and non standard attacks, as well as the legitimate scripts, between the attacker and the target. The VMs are synchronized : when the attacker VM is ready to execute an exploit or execute legitimate scripts, the automata player is launched to load the associated automata. The attacker runs a Ruby script in charge of: 1) looking in the attack dictionary for the exploits to execute according to the protocol, port and the attack modes; 2) waiting for signals before executing each exploit with Metasploit; 3) notifying when each exploit is done. The hypervisor API allows us to launch the programs embedded inside the VMs, and retrieve the results of the attacks: the completed exploits (where the automaton reached its final state).

C. Attack campaign algorithm

The attack campaign algorithm, illustrated in Algorithm 2, consists in running as many attack sessions (i.e., execute the $run\_attacks()$ function for each accessibility of the session) in parallel as possible, while respecting some constraints so as not to compromise the correct execution of the attacks. Indeed, a VM can be part of only one attack session at the same time (i.e., attacker or target of the session), otherwise the embedded tools would overlap and disturb one another. To do so, threads are spawned to continuously parse the accessibility matrix (set of all the accessibilities) and look for available attack sessions (at least one accessibility between two VMs). When an attack session is found, if the VMs are available, the

Name	Type	Description
$exploit\_id$	Integer	Identifier of the exploit
cve	String	Identifier of the vulnerability
$proto$	String	Associated network protocol
$port$	Integer	Associated network port
$date$	Date	Date of the exploit
$description$	String	Description of the vulnerability
$automata$	Boolean	Indicates if automata are available

TABLE I: Attributes of the attack dictionary

Algorithm 2 Attack campaigns execution

Require: AM: the accessibility matrix.
Require: $nb\_sessions$: the total number of attack sessions according to the accessibility matrix.
Require: $attack\_modes$: the attack modes for standard and non-standard attacks.
Require: $sessions$: shared variable to record the done sessions and the sessions in progess, initialized to $\emptyset$.
Require: S: shared binary semaphore manipulated with traditional $P()$ and $V()$ operations, initialized to 1.

$still\_sessions \leftarrow True$
while $still\_sessions$ **do**
  $still\_sessions \leftarrow False$
  $P(S)$
  if $sessions.length == nb\_sessions$ **then** $still\_sessions \leftarrow True$ **end if**
  $V(S)$
  $n \leftarrow 0$
  for $attacker \in AM.keys()$ **do**
    for $target \in AM[attacker].keys()$ **do**
      for $ip \in AM[attacker][target].keys()$ **do**
        $session\_available = False$
        for $proto \in AM[attacker][target][ip].keys()$ **do**
          if $AM[attacker][target][ip][proto]$ **then**
            $session\_available \leftarrow True$; $n \leftarrow n + 1$
          end if
        end for
        if $session\_available$ **then**
          $P(S)$
          for $s \in sessions$ **do**
            if $s["n"] \neq n$ or ($s["in\_progr"]$ and $((attacker$ or $target) \in (s["attacker"]$ or $s["target"]))$ **then**
              $session\_available \leftarrow False$
            end if
          end for
          if $session\_available$ **then**
            $sessions \leftarrow sessions \cup \{"num" : n, "attacker" : attacker, "target" : target\}$
            $V(S)$
            for $proto \in AM[attacker][target][ip].keys()$ **do**
              for $port \in AM[attacker][target][ip][proto]$ **do**
                $run\_attacks(attacker, target, proto, port, attack\_modes)$
              end for
            end for
            $P(S)$
            for $s \in sessions$ **do**
              if $s["n"] == n$ **then** $s["in\_progr"] \leftarrow False$ **end if**
            end for
            $V(S)$
          else
            $V(S)$
          end if
        end if
      end for
    end for
  end for
end while

session is executed following the attack modes provided. If the VMs are already busy, it looks for another attack session to run, until all the attack sessions are done.

D. Computation of evaluation metrics

Building a system able to automatically compute accurate evaluation metrics for IDS is quite arduous in practice. The associated theory, based on confusion matrices, is straightforward to understand, and provides many performance metrics, as summarized in [**?**]. Nevertheless, the values of metrics can vary considerably according to the way to compute them. The four metrics traditionally used in the evaluation of IDS are:

- True Positive (TP): malicious activity classified as malicious (raised an alert).
- False Positive (FP): legitimate activity classified as malicious (raised an alert).
- True Negative (TN): legitimate activity classified as legitimate (did not raise any alert).

- False Negative (FN): malicious activity classified as legitimate (did not raise any alert).

Moreover, one can compute the Detection Rate (DR), noted as: $DR = \frac{TP}{TP+FN}$, and the Precision (PR), noted as: $PR = \frac{TP}{TP+FP}$. In practice, automatically associating an alert to the corresponding attack (i.e., find a TP) is not an easy task, even manually. According to the alerts we studied from several NIDS products, the information provided often consists of: the date of the alert, the IP addresses, protocol, ports, the identifier of the associated signature, and a textual description. To know whether every attack executed in the campaigns raised an alert, the audit system should retrieve and parse the file containing the alerts generated during the campaign. An alert is associated to an attack if the following conditions are satisfied:

- Considering the clocks synchronized, the delay between the date of the alert and the date of the attack is shorter than a given duration. We call this delay the detection accuracy tolerance window (denoted W).
- The IP addresses, protocol and ports involved in the alert are also present in the attack.

Optionally, the following conditions can also be taken into account to validate a TP:

- The alert is a severe alert.
- The CVEs referenced in the signature of the alert include the CVE of the attack.

Therefore, the number of TP strongly depends on the good synchronization of the clocks and the detection accuracy tolerance window. The two optional conditions are activated depending on the user that has to consider the evaluated NIDS deployment. Indeed, users of the audit system might want to only consider severe alerts. In addition, some NIDS do not reference CVEs in their signatures, or do not reference the same CVEs as those referenced in our attack dictionary. Note that several alerts may be associated to an attack. The FP are deduced by looking at alerts that are not part of a TP and affect the IP addresses involved in the attack campaign. The FN are determined by checking the attacks that are not part of a TP.

As we know the detected and non detected attacks, we are able to identify the non detected attacks that used the same exploits as detected attacks involving other IP addresses or ports. This is important to highlight, because it can result from a misconfiguration of the NIDS.

So far, we developed JSON and CSV alert parsers for Suricata [**?**] and Snort [**?**] IDS products.

V. Testbed environment and experiments

In this section, we present the VMware-based environment we used to host experiments in order to validate the first prototype which implements our approach. Then, we provide details on the data set of attacks we used in these experiments. Finally, the experimental results are presented and discussed.

A. Experimental platform

To validate the feasibility and efficiency of our approach, we run our experiments on a cloud platform based on VMware, which is widely used for the deployment of IaaS clouds. This platform includes two physical rack servers (Dell PowerEdge R620 with 2 Intel Xeon E5-2660 and 64GB of RAM) connected on a physical network switch (HP 5120-24G EI). The servers run VMware vCloud Suite (VMware's IaaS solution). The latter includes a hypervisor (VMware ESXi), cloud management software (vCenter, vCloud Director, Microsoft SQL Server) and cloud security management sofware (vShield Manager). A distributed virtual switch connects all the VMs of the platform. Virtual networks are represented as port groups on the virtual switch. Hence, there is a port group for every client's virtual network. In addition, there is one virtual network, the management network, that interconnects the cloud management VMs and the gateway of every client's network. On a powerful VM with 4 CPUs and 8GB of RAM, we deployed two open-source NIDS solutions: Suricata 2.0.5 and Snort 2.9.2.2. The NIDS VM is connected to the management network. We kept the default configuration of the NIDS products (we just configured them to monitor the right network), and populated them with the latest set of rules from [**?**]. On the virtual switch, we applied port mirroring to duplicate all the traffic to and from the clients' VMs and send it to the NIDS VM.

For our experiments, we deployed and configured a small-sized virtual infrastructure for a hypothetical client, composed of 1 vDC, 3 virtual networks, 2 virtual Edge firewalls and 3 VMs (2 VMs with 1 network interface, 1 VM with 2 network interfaces). The audit VM (4 CPUs and 4GB of RAM) orchestrates the audit operations of our three-phase approach. It is placed on the management network so it can interact with VMware APIs and the NIDS. Note that the clocks of the NIDS VM and the audit VM are synchronized through NTP. Moreover, the audit VM is external to the client's networks, and thus it can turn into an attacker and target so we can verify the accessibilities and try attacks to and from its location. As the first phase of our approach, we automatically cloned the deployed client's virtual infrastructure to obtain the corresponding cloned virtual infrastructure that will be used in the next phases. The template we use to import VMs in the cloned infrastructure is a lightweight Open Virtualization Appliance (OVA) package, provided with 1 CPU and 1GB of RAM, and the following tools : Python, the dynamic accessibility analysis Python tools, the Python automata player, Metasploit, Ruby, the Ruby script that uses Metasploit, SQLite3, and the attack dictionary as a SQLite3 database. Our platform with this deployment is illustrated on Figure 5.

B. Evaluation traffic data set

In our first prototype, we focused on four different protocols: HTTP, FTP, SMTP and IMAP. We chose 16 exploits from the Exploit Database that are also available in Metasploit and target available applications. The selection covers a mixture of vulnerabilities published between 2002 and 2014. We generated the associated automata following the process presented in Section III-B, as well as the associated scripts in charge of sending legitimate requests. We manually tested their functioning to make sure that we would be able to replay them during the attack campaigns. Table II shows the related entries in the attack dictionary, along with the number of states of the generated legitimate automaton (noted as N_{sl}) and the number of states of the generated malicious automaton (noted as N_{sm}). Note that the number of states of the automata is

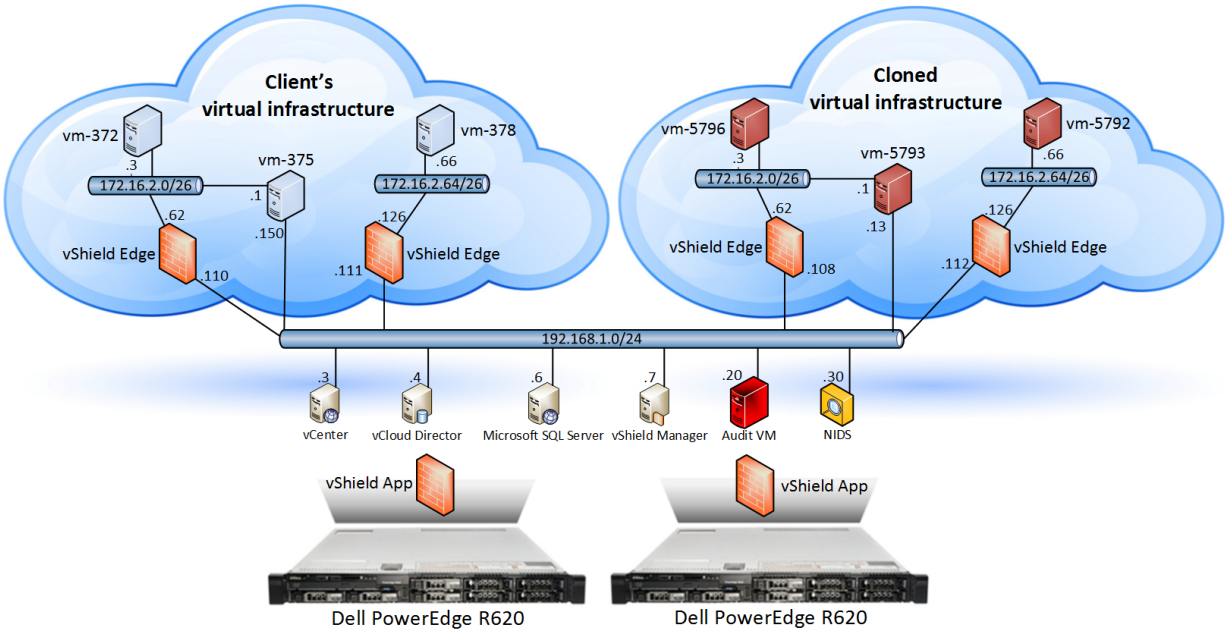

Fig. 5: Experimental platform

exploit_id	*cve*	*proto*	*port*	*date*	*description*	N_{sl}	N_{sm}
35660	2014-9567	TCP	80	2014-12-02	ProjectSend Arbitrary File Upload	3	3
34926	2014-6287	TCP	80	2014-09-11	Rejetto HttpFileServer Remote Command Execution	12	12
33790	N/A	TCP	80	2014-05-20	Easy File Management Web Server Stack Buffer Overflow	7	8
25775	2013-2028	TCP	80	2013-05-07	Nginx HTTP Server 1.3.9-1.4.0 - Chuncked Encoding Stack Buffer Overflow	3	8
16970	2002-2268	TCP	80	2010-12-26	Kolibri 2.0 - HTTP Server HEAD Buffer Overflow	2	2
16806	2007-6377	TCP	80	2007-12-10	BadBlue 2.72b PassThru Buffer Overflow	9	3
16772	2004-2466	TCP	80	2007-08-14	EFS Easy Chat Server Authentication Request Handling Buffer Overflow	4	10
28681	N/A	TCP	21	2013-08-20	freeFTPd PASS Command Buffer Overflow	11	4
24875	N/A	TCP	21	2013-02-27	Sami FTP Server LIST Command Buffer Overflow	11	3
17355	2006-6576	TCP	21	2011-01-23	GoldenFTP 4.70 PASS Stack Buffer Overflow	13	7
16742	2006-3952	TCP	21	2006-07-31	Easy File Sharing FTP Server 2.0 PASS Overflow	11	6
16713	2006-2961	TCP	21	2006-06-12	Cesar FTP 0.99g MKD Command Buffer Overflow	11	6
16821	2007-4440	TCP	25	2007-08-18	Mercury Mail SMTP AUTH CRAM-MD5 Buffer Overflow	9	8
16822	2004-1638	TCP	25	2004-10-26	TABS MailCarrier 2.51 - SMTP EHLO Overflow	6	6
16476	2006-1255	TCP	143	2006-03-17	Mercur 5.0 - IMAP SP3 SELECT Buffer Overflow	7	4
16474	2005-4267	TCP	143	2005-12-20	Qualcomm WorldMail 3.0 IMAPD LIST Buffer Overflow	2	2

TABLE II: Entries in the attack dictionary and number of states of the generated automata

quite limited (less or equal to 13).

C. Experiments, results and discussion

On the cloned virtual infrastructure illustrated in Figure 5, we performed the dynamic analysis[2] of network access controls and obtained an accessibility matrix, represented by its accessibility graph shown in Figure 6. A total of 13 observed

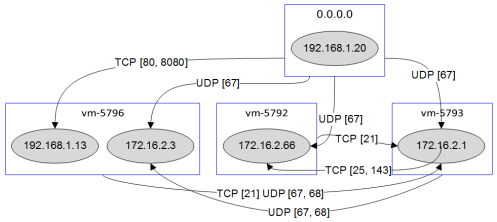

Fig. 6: Accessibility graph generated from dynamic analysis

[2]We tested 587 known services from the DARPA Internet network, commonly found under the `/etc/services` file in Linux-based distributions

accessibilities were found, implying 8 possible attack sessions. From the accessibility matrix, we performed some attack campaigns to evaluate the deployment of the NIDS solutions, using our prototype and applying different parameters. Remember that our objective is to analyze the feasibility of our approach, and not to provide a benchmark evaluation of some NIDS products. To execute diversified attacks, we used the following attack modes (cf. Section IV-B):

- Launch all the standard attacks per accessibility.
- Launch R randomly chosen non-standard attacks per accessibilitiy.

The different parameters of the attack campaigns are listed as:

- W: the NIDS detection accuracy tolerance window (in seconds).
- S: if only severe NIDS alerts are treated or not.
- R: the number of randomly chosen non-standard attacks per accessibility.
- N_s: the total number of standard attacks launched.

Parameters														
S	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
R	0		1		2		3		4		5		6	
N_s	21		21		21		21		21		21		21	
N_n	0		13		26		39		52		65		78	
N_l	21		34		47		60		73		86		99	
Efficiency														
N_c	20		31		43		55		66		79		89	
C	95.24%		91.18%		91.49%		91.67%		90.41%		91.86%		89.9%	
Duration	5mn41s		11mn57s		16mn16s		22mn18s		23mn37s		29mn22s		30mn09s	
Suricata														
# of TP	8	7	11	9	15	10	19	13	22	17	22	18	24	17
# of FN	12	13	20	22	28	33	36	42	44	49	57	61	65	72
# of FP	1	1	4	1	7	2	11	6	11	5	13	8	26	15
DR	40%	35%	35.48%	29.03%	34.88%	23.26%	34.55%	23.64%	33.33%	25.76%	27.85%	22.78%	26.97%	19.1%
PR	88.89%	87.5%	73.33%	90%	68.18%	83.33%	63.33%	68.42%	66.67%	77.27%	65.86%	69.23%	48%	53.13%
Snort														
# of TP	10		16		17		24		27		29		34	
# of FN	10		15		26		31		39		50		55	
# of FP	12		27		51		57		81		103		143	
DR	50%		51.61%		39.53%		43.64%		40.91%		36.71%		38.20%	
PR	45.45%		37.21%		25%		29.63%		25%		21.97%		25.5%	

TABLE III: Experimental results of attack campaigns for different parameters

- N_n: the total number of non-standard attacks launched.
- N_l: the total number of attack and legitimate activity pairs launched.

As a result, we are interested in first analyzing the efficiency of our prototype system, characterized by:

- N_c: the total number of attack and legitimate activity pairs completed, those where the associated automata reached their final state.
- C: the completeness ratio given by $\frac{N_c}{N_l}$.
- The duration of each attack campaign.

Then, we report the evaluation metrics to assess the efficiency of the NIDS. We executed 7 different attack campaigns, increasing the value of R from 0 to 6. The value of N_s remains constant as we always launch all the available standard attacks per accessibility. The values of N_n, N_l and N_c increase as R is increased. W depends on the target environment. The smaller W, the higher the risk to miss alerts. The larger W, the higher the risk to increase the number of false alarms. In our environment, given the extra time needed to perform network port mirroring and packets transmission to the NIDS, and the time needed for the NIDS to process every packet and report the alerts, we chose 10 seconds for W. It is large enough so no alert is missed, and not too large so no additional false alert is taken into account. We did not apply CVE reference verification not to be too restrictive in the treatment of the alerts. However, for each attack campaign, we changed the value of S to process only severe alerts or all alerts. Note that Snort does not provide a severity indicator in its alerts and that is why they are all considered as severe ones. The overall results of the experiments are presented in Table III.

From our results, we can see that our prototype system is able to conduct attack campaigns with a good efficiency and an acceptable duration. Indeed, C is always at least above 89.9%, so only less than 10.1% of the attacks launched failed. Over the different campaigns, the failed attacks were always different ones. Indeed, as we also manually ran the attack sessions to verify their possible execution, we know that the few failures were due to random software crashes, either from the VMware API (that we request intensively) or the Metasploit framework. Also, our prototype is able to execute approximately 3 to 4 attacks per minute. Overall, the attack campaigns have a linear duration that depends on N_c. This is encouraging, considering that we did not make the time performance a priority when implementing the first prototype.

We observe that overall and for both NIDS products, increasing R makes TP and FP increase, and thus makes DR decrease. Note that the false positives are generated by the legitimate activities we generated, but also by other existing background traffic. Also, less TP and FP are reported when considering only severe alerts for Suricata, and thus DR decreases. In fact, the detection rate is lowered as the number of non-standard attacks is increased, which is due to the configuration of the NIDS we deployed. Indeed, as we kept the default configuration, we know that some alerts are only raised for certain standard network ports. As for the precision, PR tends to decrease for both NIDS when R is increased. It is lowered because the number of false alarms increases faster than the number of true positives. Globally, Suricata exhibited a lower detection capability than Snort, but is much more precise. These observations are illustrated on Figure 7 and Figure 8. Although the goal of this paper is not to provide a thorough evaluation of Snort and Suricata NIDS products, we were surprised by the values of DR and PR. After having manually run the attacks and checked the results, we know that this is mainly due to the lack of signatures (only about half of the malicious payloads sent by the exploits we chose are actually related to a NIDS signature). Note that low detection rates and precision rates for NIDS products including Snort were also reported in other works (e.g., [**?**], [**?**], [**?**]).

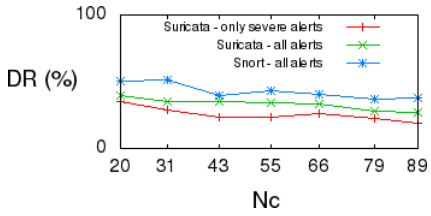

Fig. 7: Detection Rate values of the NIDS

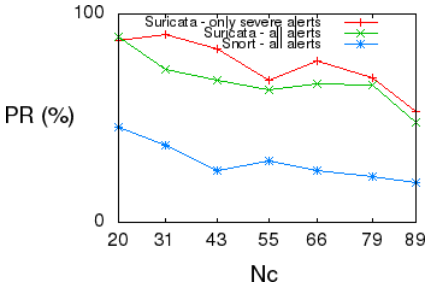

Fig. 8: Precision values of the NIDS

VI. Related work

The work we present in this paper is associated to two main research fields. The first one is protocol reverse engineering, and this is related to the modeling, generation and replay of evaluation traffic. Although we designed and implemented a convenient, lightweight and portable tool to replay network flows in the attack campaigns, we do not claim to propose an full protocol reverse engineering solution, and it is not our main contribution. However, we needed to know the existing tools and understand the problem of protocol reverse engineering to design our own. Thus, we provide an overview of some relevant existing tools we studied and also explain why they do not exactly meet our needs, although having generally rigorous formal models and working solutions. The second research field is the evaluation of IDS, related to our main contribution, about the injection of attack campaigns and computation of evaluation metrics. Moreover, considering the general topic of cloud security audits, we can mention the approaches presented in ([**?**], [**?**]), and the recommandations and guidance on security assessments from the CSA [**?**].

A. Protocol reverse engineering

Protocol reverse engineering is generally used to infer protocol specifications by observing and analyzing the behavior of entities using this protocol. The inferred specifications can then be used to replay traffic and simulate application behaviors. A common application of traffic replay is the evaluation of NIDS. According to the survey in [**?**], the lightweight tools we designed to generate and replay application traffic are classified in the network-based category. This category includes the tools that analyze network traces to infer protocols.

Among such tools, we can cite Netzob [**?**], a well-documented tool that infers network communication protocols from network traffic. It also proposes a simulation module to generate traffic from the inferred protocol and spawns the associated client and server instances. The goal of this tool is to infer entire communication protocols, which is not our objective (we are only interested in some network flows of the applications needed to run the exploits). Furthermore, the simulation module does not provide independent lightweight client and server instances, or any scriptable API easy to control and automate as we want.

Interesting precursor in this field, Scriptgen [**?**] aims at automatically inferring unknown protocol vocabulary to populate the scripts used in the Honeyd [**?**] project. From honeypot traffic, the sequence of messages are analyzed to derive the corresponding state machine then translated into a script. The proposed approach is dedicated to the Honeyd project, which is not publicly updated anymore, and thus does not cover recent attacks. Similar to Scriptgen, PRISMA [**?**] infers stateful markov models of botnet protocols from network traffic captures using a probabilistic approach.

Some application-based tools, i.e., analyzing the execution of the application to infer protocols, also propose interesting formal definitions. In particular, Replayer [**?**] proposes a formal definition of the replay problem, along with a solution using dynamic binary analysis to replay application dialogs.

The work done in [**?**] explores the problem of network attack injection, from the generation of protocol specifications to the injection of attack campaigns on target servers and their monitoring. Again, the goal is to infer entire protocol specifications, while we just model and replay some specific flows. From these specifications, they generate various network attacks in order to discover new vulnerabilities.

B. Evaluation of IDS

There are two main categories of evaluation methods: 1) white-box description and analysis-based evaluation; 2) black-box test-based experimental evaluation. There also exist some hybrid approaches. Since we evaluate NIDS instances in production, our work falls into the second category.

The experimental evaluation of IDS has been addressed in several works, considering different application targets, for instance in [**?**], or by the Defense Advanced Research Projects Agency (DARPA) and the Lincoln Lab ([**?**], [**?**]), though criticized on several aspects in [**?**]. However, no up-to-date data sets are publicly available, and traditional evaluation approaches need to be extended to take into account the specific constraints and the complex operational characteristics of cloud environments.

The methodology presented in [**?**] is a starting point when one wants to elaborate a strategy to assess IDS. They defined three test procedures: one to measure the detection ability, one to measure the resource usage, and one to check if the IDS functions correctly under stressful conditions. Moreover, in their experimental platform, they used Expect as well, to write and run scripts that simulate user's activity to test IDS.

The authors in [**?**] provide a strategy to address the lack of documented data sets for the evaluation of IDS, along with a tool showing its usage. However, unlike us, their evaluation approach is an off-line one. Moreover, their framework does not really assess FP. Indeed, as their work only deals with malicious traffic, they categorize FP as alarms that are raised when the associated attacks failed. They do not execute legitimate traffic to thoroughly measure FP.

The work presented in [**?**] is about the generation of traffic for signature-based NIDS. The authors explain that they are able to generate application-specific traffic from a workload model, and present a capacity estimation method.

In [**?**], the authors create their set of attacks by applying a combinatorial generation of test cases, or recycling test cases from several sources. The attacks are injected to discover new vulnerabilities and thus are not necessarily successful, while we exploit known vulnerabilities. Indeed, we assess the network protection mechanisms and not the security of the targeted hosts. If we take a closer look at their attack campaign injection methods, we see that they faced the same issues as we did, but working in a traditional environment unlike us, these issues remain present. For instance, the attacks cannot always be run on production systems, as well as the recovery

and backup features they adopted to mitigate this problem. Also, it is more difficult for the monitor of the attacks to be well located and work efficiently when the targets are not controlled, while we can easily follow the attack processes in our automata.

VII. Conclusion and Perspectives

We have elaborated and presented our approach to automatically evaluate NIDS deployed in cloud computing environments and in charge of protecting the clients' virtual infrastructures. The first phase of this approach consists in cloning the targeted client's infrastructure while preserving its privacy. Indeed, although the network configuration and equipments are cloned, the new VMs are imported from a template into the cloned infrastructure, instead of copying its VMs. The second phase includes the static and dynamic analysis of network access controls managed by the virtual firewalls. The accessibilities determined by both methods are compared in order to look for discrepancies. The last phase, presented in details in this paper, aims at performing network attack campaigns using the accessibilities found within the cloned infrastructure and allowing different parameters. We modeled and generated evaluation traffic to easily simulate malicious and legitimate network exchanges. As an outcome, evaluation metrics are computed using the evaluated NIDS output alerts. Experiments have been carried out on a VMware-based cloud platform to illustrate its feasibility. The results are encouraging and sustain the first prototype that implements our approach. They validate the combined use of network accessibilities, attack dictionary and automata in our approach to conduct automated attack campaigns in cloud virtual infrastructures.

Based on our generic approach, we can plan to extend our VMware-based prototype to other IaaS solutions, and be able to assess other kind of deployed NIDS products. This would result in the customization of the use of provided APIs to manipulate the cloud resources, and the development of adapted NIDS alert parsers. Ongoing work also includes the generation of new automata to be able to execute more exploits in the attack campaigns. We are aware that it requires a certain human effort, but this would just be a community engineering effort like the populating of exploits in Metasploit or signatures in the NIDS. Furthermore, although our first prototype provided good results on our testbed, we intend to make it scalable for larger virtual infrastructures.

Acknowledgment

This research is partially funded by the Secured Virtual Cloud (SVC) project of the French program Investissements d'Avenir on Cloud Computing.