Intrusion Detection using ASTDs

Lionel N. Tidjon, Marc Frappier and Amel Mammar

Abstract In this paper, we show the application of ASTDs to intrusion detection. ASTD is an executable, modular and graphical notation that allows for the composition of hierarchical state machines with process algebra operators to model complex attack phases. Overall, ASTD attack specifications are more concise than industrial tools like Snort, Zeek, and other attack languages in the literature. For intrusion detection, iASTD (the ASTD interpreter) and Zeek provided similar results. iASTD produced less false positives and a smaller number of true positives per attack than Snort, which is an important factor to deal with huge amounts of events. The processing time of iASTD on the real-time testbed is slower than Snort and Zeek, but it can be improved by compiling ASTD specifications into Zeek scripts.

1 Introduction

In Security Operations Center (SOCs), several intrusion detection tools are placed at different levels of the network to ensure the security and privacy of information [1]. Snort [2], a widely used IDS, provides a low-level signature language to express and detect multi-stage Advanced Persistent Threats (APT) attacks. Zeek [3] was proposed to overcome some limitations of Snort by providing an event-driven

Lionel N. Tidjon
GRIF, Université de Sherbrooke, 2500 boul. de l'Université, Sherbrooke, Canada and SAMOVAR, Télécom SudParis, Institut Polytechnique Paris, Palaiseau, France. e-mail: lionel.nganyewou.tidjon@usherbrooke.ca

Marc Frappier
GRIF, Université de Sherbrooke, 2500 boul. de l'Université, Sherbrooke, Canada. e-mail: marc.frappier@usherbrooke.ca

Amel Mammar
SAMOVAR, Télécom SudParis, Institut Polytechnique Paris, Palaiseau, France. e-mail: amel.mammar@telecom-sudparis.eu

scripting language to precisely specify and identify APT attacks. The writing of Zeek scripts is essentially programming using functions and global variables. Eckmann *et al.* [4] have proposed STATL, a stateful and domain-independent language that allows a more abstract representation of attack scenarios than Snort and Zeek using state machines with actions and state variables. Other attack languages like LAMBDA [5] and Chronicle [6] have been proposed. LAMBDA [5] provides an abstract description of an attack operation in terms of conditions and effects, expressed using predicates. Chronicle [6] reconstructs a state machine from event patterns that can be ordered with timing constraints. Barringer *et al.* [7] have introduced quantified event automata (QEA), in which universal and existential quantification are used to quantify parameters of an automaton, allowing to replicate an automaton and efficient execution.

In this paper, we show the application of ASTDs to intrusion detection. ASTD is an executable, modular and graphical notation that allows for the composition of hierarchical state machines using process algebra operators such as flow, sequence, quantified interleaving and parallel synchronization [8, 9]. It allows one to capture "big picture" of complex attacks by graphically specifying their behaviours in a modular fashion, defining complex relationships between events (i.e., event correlation) to model and detect attack phases. ASTDs can be seen as extensions of STATL, since state machines are elementary ASTDs. STATL does not compose state machines using process operators; thus it is less modular. ASTDs offer a more abstract representation of attacks than LAMBDA, since the logical expressions of attacks are low-level mechanisms to deal with APT attacks. ASTD operators can be encoded into Chronicle, but at the expense of loosing abstraction and concision. Quantified versions of synchronization and interleaving ASTDs [9] are generalisations of QEA's quantifications. These quantified versions provide the ability to replicate ASTDs and index them with a quantified variable (e.g., address, port), which is necessary to construct specifications which are more resilient to attack mutations and variants.

Our specification approach using ASTDs is based on attack pattern databases like MITRE's Common Attack Pattern Enumeration and Classification (CAPEC) [10] and ATT&CK (Adversarial Tactics, Techniques & Common Knowledge) [11]. We propose to specify a case study of ransomwares with different Snort, Zeek, and ASTD in the litterature to identify their weaknesses and strengths. The specification of recent malwares including ransomwares has been conducted in collaboration with Nokia Threat Intelligence Centre. The aim was to get feedbacks from cybersecurity experts and improve the ASTD language for intrusion detection.

Among existing tools, we have selected IDSs like Snort and Zeek for comparison; because they are widely used and well maintained by the cybersecurity community. Tools related to other attack languages in the literature were either no longer available, or deprecrated i.e. not able to run on current operating systems, or not working operationally on real world environments (essentially worked on old datasets). Our results show that the ASTD notation is more abstract, modular and concise than Snort and Zeek, while achieving good performance on heterogeneous event sources, thanks to its advanced event correlation capabilities [9]. Attack detection is done by

executing ASTD specifications on online and offline events using the iASTD tool, whose processing time is slower than Snort and Zeek, but it can be improved by compiling ASTD specifications into Zeek scripts or other programming languages.

The rest of this paper is structured as follows. The methodology for ASTD attack specification is described in Section 2. In Section 3, we present the specification of a case study using Snort, Zeek, and ASTD. Section 4 describes the execution of attack specifications by the tools. In Section 5, we compare and discuss the results of the iASTD tool against Snort and Zeek. Section 6 concludes with some perspectives.

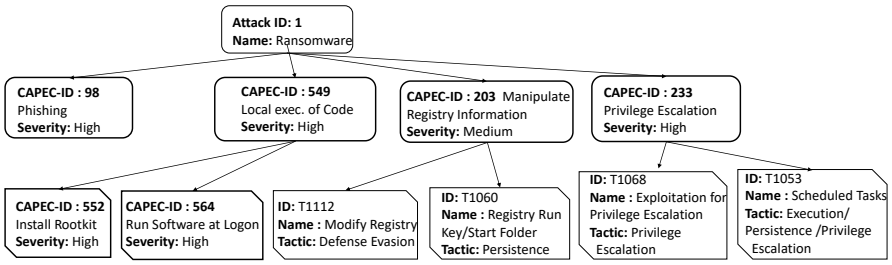

Fig. 1: Ransomware attack pattern from CAPEC and ATT&CK

2 Attack Specification Methodology using ASTDs

CAPEC [10] is one of the most well-known attack pattern database. Cybersecurity companies use it to figure out how cyber actors exploit weaknesses in applications and platforms. Another interesting database is ATT&CK [11], which complements CAPEC in providing more details about attacker actions and techniques. Attack patterns are hierarchical descriptions: they are decomposed into phases, which are further decomposed into steps. A step is realized using a combination of events. A phase or a step may appear in several attack patterns, so there is an interest in describing phases and steps independently and to compose them to build an attack specification.

The ASTD notation provides the necessary operators to construct modular formal models of attack patterns. Each phase and step can be defined by its own ASTD, properly encapsulated and parameterized, in order to allow its reuse in several attack patterns. Attack patterns can be composed together to create a global ASTD specification of an IDS.

To illustrate our approach, we show an attack pattern for ransomwares in Fig. 1, extracted from CAPEC and ATT&CK. The pattern states that the attacker starts by delivering an attached file to the victim machine through email phishing. The victim downloads the malware by clicking on the malicious link in the email (CAPEC-98). The malware locally executes and encrypts user files (CAPEC-549). It also mod-

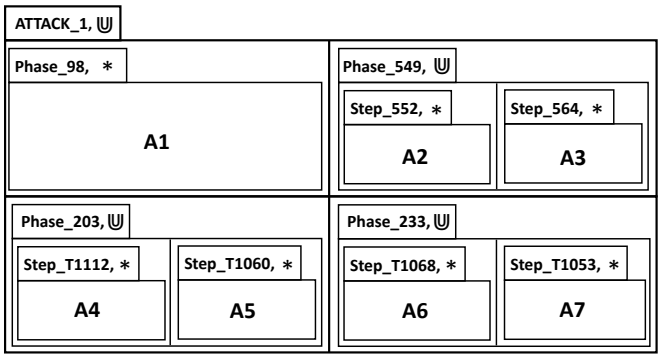

Fig. 2: ASTD specification of attack pattern of Fig. 1

ifies the registry by adding an entry to the ”run keys” in the registry to be persistent (CAPEC-203). Concurrently, the malware spreads itself and executes malicious scheduled tasks on the local or remote system (CAPEC-233).

Fig. 2 shows the top-level ASTD specification of the attack pattern of Fig. 1. Attack phases and steps are composed using the flow operator ⋃ to execute them in parallel. Phases are intuitively perceived as sequential, but in practice, they may overlap, thus their composition is better represented by a flow. The Kleene closure "$\star$" allows iterating on each attack step as the attacker can execute the same step several times. In each step, A_i $(i \in 1..7)$ denotes a call to an ASTD which represents the ASTD step, typically in terms of an automaton.

3 Specification of a case study

We have specified more than 65 malware variants (including ransomwares) from Nokia, targeting different operating system (Windows, Linux, Android, iOS) and 20 other malware variants from theZoo github project using the ASTD, Zeek, and Snort languages. In this section, we specify a recent variant of ransomwares called Gandcrab using ASTD, Snort, and Zeek. The specification of this variant in other attack languages like STATL can be found in [12]. Gandcrab can be resumed into 6 actions. In action 1, the attacker delivers an email containing an embedded file. Once the victim runs the attached file, it downloads and executes Gandcrab (action 2). In action 3, Gandcrab gets the IP address of the victim host by sending a DNS request to the website *ipv4bot.whatismyipaddress.com*. In action 4, Gandcrab replicates by creating a malicious file in the AppData folder (e.g., *yxvace.exe*). This malicious file checks-in multiple command and control (C&C) sites using the command *nslookup site_name dns_server*. During C&C check-in, Gandcrab also sends a HTTP GET request to its C&C site (action 5). Next, Gandcrab encrypts collected data in action 3 and post it to the C&C server (action 6).

Actions 1 and 2 are done in phase CAPEC-98. The remainders are done in phase CAPEC-549. Hereafter, we specify the Gandcrab case study using the Snort, Zeek and ASTD languages in order to compare their weaknesses and strengths.

3.1 ASTD specification

As Gandcrab operates at both the host and the network levels, we build one attack model for each and compose them using the flow operator, following the specification methodology. In Fig. 3, the main ASTD *Gandcrab_Ransom* is of type flow and declares two variables *v1* and *v2*, each of type boolean. These variables can be modified by actions. *Gandcrab_Ransom* composes phases *Phishing* (CAPEC-98) and *Exec_Code* (CAPEC-549) using the flow operator ⋓.

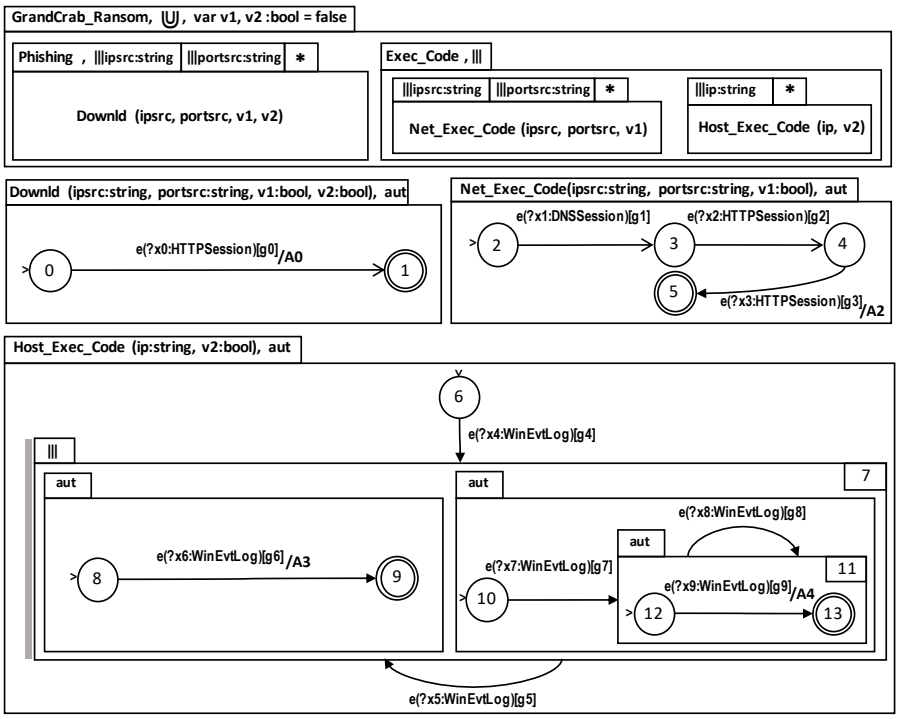

Fig. 3: Gandcrab crypto-worm specification

The first phase (i.e., *Phishing*) is a quantified interleaving ASTD that allows detecting the attacker's action 2. *Phishing* and its nameless sub-ASTD (i.e., |||*portsrc :string*) interleave instances of its sub-ASTD, indexed by variables *ipsrc* (source address) and *portsrc* (source port). Its sub-ASTD is a nameless Kleene closure and it is identified by $\star$. It allows iterating on an ASTD call that refers to the ASTD

definition of *Downld* described in Fig. 3. *Downld* is an ASTD automaton that receives parameters *ipsrc*, *portsrc*, *v1*, and *v2* from its calling ASTDs. Being called within two quantified interleaves, there is an instance of this automaton for each pair of values of *ipsrc* and *portsrc*. The initial state 0 has an outgoing transition labeled by the event *e(?x0:HTTPSession)* and a guard *[g0]*. The local variable *x0* has a user-defined type *HTTPSession*. *HTTPSession* is described in an ontology to process multiple HTTP sessions from the network traffic.

From the initial state 0, the transition executes action *A0* (e.g., A0 = { v1 = true; v2 = true;}) when the guard *g0* is *true*. The guard *g0* checks if HTTP sessions contain signatures of the malicious file during downloading (e.g., GET /js/kukul.exe). When transition 0→1 is executed, *v1* and *v2* take value *true* to notify other ASTDs that the phishing phase is done. The second phase (i.e., *Exec_Code*) interleaves two nameless ASTDs. Each one respectively interleaves host and network events to identify Gandcrab actions. The first one (i.e., |||*ipsrc:string*) and its nested component (i.e., |||*portsrc: string*) allow multiple instances of the ASTD automaton *Net_Exec_Code*, indexed by *ipsrc* and *portsrc*. Within *Net_Exec_Code*, the transition 2→3 tracks the Gandcrab action 3 in the network traffic. Next, the transition 3→4 detects the Gandcrab's action 5 when *g2* holds. From state 4, the transition 4→5 checks the malicious action 6. The action A2 (e.g., A2 = {**if** v1 **then** print "GandCrab CnC"; }) shows an alert only if the phishing phase tooks place.

Concurrently, the transition 6→7 in ASTD *Host_Exec_Code* also tracks the Gandcrab action 4 in host events. It is labeled by the event *e(?x4:WinEvtLog)* where *x4* is a transition local variable of type *WinEvtLog*. Type *WinEvtLog* has a structure similar to Windows event logs. The guard *g4* is *true* when the Gandcrab file creates a process that checks-in its C&C domains using the Windows command (e.g., *nslookup carder.bit ns1.wowservers.ru*). Once *g4* is *true*, the transition moves to the shallow final state 7. The state 7 is a complex state (i.e., an interleaving ASTD) that composes two ASTD automatons. Within the first automaton, the transition 8→9 checks if the previous nslookup command successfully established a DNS connection to C&C servers (e.g., ns1.wowservers.ru). Concurrently, the transition 10→11 detects when the nslookup command process forks into another process *svchost.exe* to leak system information over the open port 3389.

3.2 Snort specification

From the case study, we can deduce 4 detection signatures [12], each referring to phases *CAPEC-98* and *CAPEC-549*. These signatures are low-level representations of transitions in ASTD automatons *Downld* and *Net_Exec_Code* (see Fig. 3). For example, the following signature

```
alert tcp $HOME_NET any -> $EXTERNAL_NET 80 (msg:"Malicious Software Downloading";
flow:established, from_client; content:"GET"; http_method; content:".exe HTTP/1.";
fast_pattern:only; content:"Connection: Keep-Alive"; http_header; content:"Accept
|3a 20|";http_header;content:"User-Agent: Mozilla";http_header;content: "Host|3a|";
pcre:"/Host\x3a\x20(:?[0-9]{1,3}\.){3}[0-9]{1,3}/H"; reference: capec,CAPEC-98;
classtype:Downloader; sid:100000004; rev:1;)
```

detects the Gandcrab downloading through the attached file (i.e., Action 2). This signature corresponds to the ASTD *Downld*. It targets the inbound HTTP traffic (*$HOME_NET*) directed to the C&C server (*$EXTERNAL_NET*). The clause *any* means that the signature accepts all connection ports from the inbound traffic. Within packet flows, it checks if the HTTP method is *GET*, the HTTP uri contains pattern *.exe HTTP/1.*, the HTTP connection is kept alive, the Accept header is used, the user-agent is Mozilla and the Host header contains the ip address of the C&C server. The unique pattern *.exe HTTP/1.* precisely characterizes the downloader trojan.

3.3 Zeek specification

We have specified the Gandcrab phases using Zeek scripts and signatures [12]. Zeek signatures [3] do essentially pattern matching like Snort. The strength of Zeek is shown using Zeek scripts. The script below is a low-level representation of ASTD automatons *Downld* (CAPEC-98) and *Net_Exec_Code* (CAPEC-549).

```
...
#BLOCK 0
export {
  ...
  global v1: bool = F;
  global v2: bool = F;
  global state_downld: int = 0;
  global state_net_exec_code: int = 2;
  ...
}
...
event http_request (c: connection, ...)
{
 #BLOCK 1 (CAPEC-98)
 local g0: bool = (c$http$method == "GET")
  && (/\/[a-z]+\.(bin|exe)/ in c$http$uri)
  && (/Mozilla/ in c$http$user_agent)
  && (/([0-9]{1,3}\.){3}[0-9]{1,3}/
   in c$http$host);

 if (state_downld == 0 && g0)
 {
   A0(v1, v2);
   state_downld = 1;
 }
 #BLOCK 3 (CAPEC-549)
 local g2: bool = (c$http$method=="GET")
  && (|c$http$uri| == 1)
  && (/[a-z]+\.(bit|ru)/ in c$http$host);
 if (state_net_exec_code == 3 && g2)
 {
   state_net_exec_code = 4;
 }
 #BLOCK 4 (CAPEC-549)
 local g3: bool=(c$http$method=="POST")
  && (/[a-z]+\.(bit|ru)/ in c$http$host)
  && (/Mozilla/ in c$http$user_agent);

 if (state_net_exec_code == 4 && g3)
 {

  # Issue an alert
  A1(v1, v2);
  state_net_exec_code = 5;
 }
} # end http_request

event dns_request (c: connection, ...)
{
 #BLOCK 2 (CAPEC-549)
 local g1: bool =(/\x00\x01\x00\x00/
  in c$dns$query)
  && (/whatismyipaddress/
  in c$dns$query);
 if (state_net_exec_code == 2 && g1)
 {
   state_net_exec_code = 3;
 }
} # end dns_request
```

Similar to ASTD *Gandcrab_Ransom*, the header of the script contains two global variables *v1*, *v2* to notify that the downloading phase is done (see block 0). The header also declares two state variables *state_downld* and *state_net_exec_code* for maintaining states during the correlation. In the body of the script, we have two Zeek event functions: **event** *http_request(c: connection, ...)* and **event** *dns_request (c: connection, ...)*. They respectively catch HTTP and DNS requests. Within the *http_request* event, the block 1 detects the Gandcrab's action 2. The action function A0 is executed to set variables *v1* and *v2* to true. Within the *dns_request* event, the block 2 targets action 3. Within the *http_request* event, the block 3 detects action 5. Next, the block 4 checks the malicious action 6.

4 Execution of attack specifications

The execution process of Snort signatures is described in [2], and Zeek scripts in [3]. In Fig. 4, the ASTD-based detection process is shown. In a corporate network, cyber-analysts specify attacks using the graphical editor *e*ASTD and following the attack strategy methodology provided in 2. They also create new custom event types (e.g., sFlow, DNP3 Session) using the ontology editor Protégé. These event types are parsed into JSON to feed *e*ASTD and iASTD [1]. ASTD attack specifications are saved in the local host in a specific repository, depending of the attack domain (network, host, both). A watcher agent automatically synchonizes local specifications and new event types to a remote network node, where iASTD is installed. iASTD has five modules.

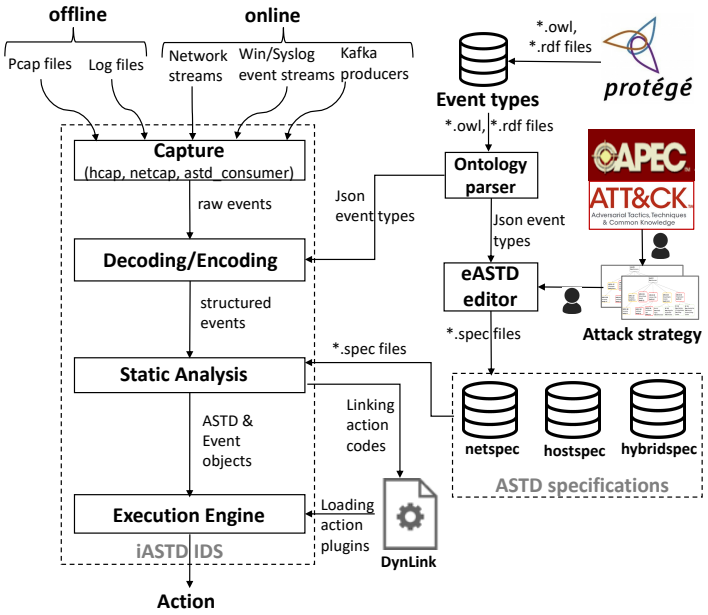

Fig. 4: ASTD-based detection process

The *capture* module collects different event sources in offline and online mode. In offline mode, it reads pcap files (option *-pcap*) and log files (option *-i*). Only log files containing Windows/Syslog traces in the ontology format are recognized. The attack specfication can be run from command line (e.g., *./iASTD -s my.spec -pcap my.pcap*), or in fully automated mode. In online mode, the *capture* module collects network streams (HTTP sessions, DNS sessions, or custom sessions) from network interfaces provided in a YAML configuration file. It also gathers Win-

[1] The tools are available at https://depot.gril.usherbrooke.ca/fram1801/iASTD-public

dows/Syslog event streams on endpoints using the shipping agent *hcap*. The agent *hcap* is installed on multiple endpoints, where it scans log files and sends real-time events to iASTD on port 9092. The *capture* module also collects network flows/sessions from *astd_producer*, a shipping agent based on the rdkafka library. The agent *astd_consumer* allows one to consume Kafka events to feed iASTD.

The *decoding/encoding* module essentially identifies which type of event stream is being read (decoding) and structures it in the corresponding ontology format (encoding). The *static analysis* module is based on the Flex/Bison analyzer and it checks if the input structured events and attack specifications are syntactically and semantically corrects i.e., well parenthesized, structured and not containing unknown keywords. Next, this module extracts the hierachichal structure of ASTD specifications and stores into ASTD objects. It also stores event contents into event objects. Since transition actions contain executable code, they are compiled and linked at runtime using the DynLink library. Next, they are loaded as plugin modules in the execution engine at runtime for intrusion detection.

The *execution engine* runs ASTD objects on events using semantic rules. The semantic rules for ASTDs are provided in [9]. The iASTD detector can efficiently execute ASTD specifications on event streams in $n \log m$, where n is the size of the ASTD specification and m the size of the quantification variable types [13]. Once an attack behavior is detected, it executes real-time actions such as alerting, blocking of a port, or dropping of a malicious traffic.

5 Experiments

We have selected two existing real-world datasets (i.e., CSE-CIC-IDS2018 [14] and CTU [15]) and we have built a testbed close to a real world environment for evaluation. CSE-CIC-IDS2018 [14] is a huge dataset of terabytes of packet captures and audit traces (normal, attack), where 18 attacks were executed (including GoldenEye, HOIC DDoS HTTP, LOIC DDoS UDP, SQL Injection, XSS, FTP and SSH BruteForcing) on the Amazon Web Service (AWS) platform. CTU [15] is a dataset of gigabytes of botnet traffic (normal, background), where 7 botnets have been executed (Neris, Virut, Donbot, Sogou, Qvod, Rbot, NSIS.ay). For the real-time testbed, 20 attacks (including Gandcrab, TeslaCrypt, WannaCry and Petya) were specified in the IDS tools and executed on the AWS platform.

5.1 Traffic and audit data generation

For the real-time testbed, we have selected 14 services to simulate random normal user and attacker behaviors including HTTP, HTTPS, SSH, SMTP, TELNET, and FTP. Normal users perform benign activities including accessing HTTP/HTTPS pages and sending/consulting emails. Concurrently, we semi-randomly run each at-

tack in different time frames to ensure that it is close to real-world attacks. This means that attacker can repeat the same phase or the previous one in another phase in different time intervals.

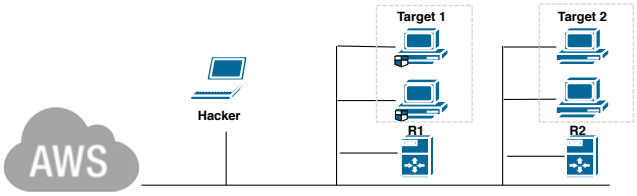

Fig. 5: AWS Testbed

The simulation network consisted of 2 work groups, each connected through 2 router servers (i.e., *R1* and *R2*), running on Ubuntu 16.04 (see Fig. 5). Each work group had 2 local machines running on Windows 10. The first work group (containing *Target 1*) had been patched with the latest Windows updates while the second (containing *Target 2*) was running without Windows patches. The system monitor (Sysmon) has been installed on *Target 1* and *Target 2*. Snort (version 2.9.15) and Zeek (version 3.0.0) were installed on *R1* and controlled the inbound traffic directed to the first work group. Kafka and iASTD were installed on *R2* to collect and analyze Windows/Syslog events from work groups and router servers.

The network infrastructure was built on Amazon Elastic Compute Cloud (Amazon EC2) using T2 Small and Medium instances. The built-in network had a bandwidth of 550 Mbits/s while all the aforementioned services were running.

5.2 Results

We consider two metrics to compare the accuracy and performance of IDS tools: detection rate (DR) and false positive rate (FPR). The detection rate (DR) is the probability that the IDS outputs an alert when there is an intrusion. The false positive rate (FPR) is the probability that the IDS outputs an alert although the behaviour of the system is normal. These metrics are expressed of the form,

$$\mathsf{DR} = \frac{\mathsf{TP}}{\mathsf{TP}+\mathsf{FN}} \qquad\qquad \mathsf{FPR} = \frac{\mathsf{FP}}{\mathsf{FP}+\mathsf{TN}}$$

where False Positive (FP) is the number of normal alerts misclassified as malicious, True Positive (TP) is the number of malicious alerts correctly classified as malicious, False Negative (FN) is the number of malicious alerts misclassified as normal, and True Negative (TN) is the number of normal alerts correctly classified as normal.

Existing datasets. Results for CSE-CIC-IDS2018/CTU datasets and the real-time testbed are reported in Table 1. The notation *Zeek-sig/Zeek-script* is used to distin-

guish results of two specification versions for Zeek. Zeek-sig denotes a specification that uses only signatures while Zeek-script denotes a specification using scripts.

Table 1: Evaluation of IDS tools

CSE-CIC	Zeek-sig/Zeek-script						Snort						iASTD					
-IDS2018	TP	TN	FP	FN	**DR(%)**	**FPR(%)**	TP	TN	FP	FN	**DR(%)**	**FPR(%)**	TP	TN	FP	FN	**DR(%)**	**FPR(%)**
LOIC DDoS UDP	0/1	0/0	164/0	0/0	0/100	100/0	8904	4914	912	0	100	15.65	1	0	0	0	100	0
HOIC DDoS HTTP	0/1	0/0	289342/0	0/0	0/100	100/0	197	5071	755	0	100	13.31	1	0	0	0	100	0
SSH BruteForce	0/1	0/0	94216/0	0/0	0/100	100/0	67	94	0	0	100	0.00	1	0	0	0	100	0
FTP BruteForce	0/1	0/0	193392/0	0/0	0/100	100/0	3868	94	0	0	100	0.00	1	0	0	0	100	0
GoldenEye DoS	0/1	0/0	27751/0	0/0	0/100	100/0	1	96	8	0	100	7.84	1	0	0	0	100	0
Web BruteForce	71/1	0/0	0/0	0/0	100/100	0/0	3	2682	73	0	100	2.64	1	0	0	0	100	0
XSS BruteForce	19/1	0/0	0/0	0/0	100/100	0/0	1	2482	0	0	100	0.00	1	0	0	0	100	0
SQL Injection	15/1	0/0	0/0	0/0	100/100	0/0	2	2482	0	0	100	0.00	1	0	0	0	100	0
CTU																		
Neris	3/1	0/0	0/0	0/0	100/100	0/0	1	131	2	0	100	1.50	1	0	0	0	100	0
Rbot	2/1	0/0	0/0	0/0	100/100	0/0	10	5249	77	0	100	1.45	1	0	0	0	100	0
Rbot DoS	4/1	0/0	0/0	0/0	100/100	0/0	3	6684	20	0	100	0.30	1	0	0	0	100	0
Virut	2/1	0/0	0/0	0/0	100/100	0/0	1	11	0	0	100	0	1	0	0	0	100	0
Donbot	178/1	0/0	0/0	0/0	100/100	0/0	91	92	10	0	100	9.8	1	0	0	0	100	0
Sogou	2/1	0/0	0/0	0/0	100/100	0/0	1	15	0	0	100	0	1	0	0	0	100	0
qvod	2/1	0/0	0/0	0/0	100/100	0/0	2	501	36	0	100	6.7	1	0	0	0	100	0
NSIS.ay	3/1	0/0	0/0	0/0	100/100	0/0	3	97	0	0	100	0.00	1	0	0	0	100	0
Real-time																		
WannaCry	6/3	0/0	0/0	0/0	100/100	0/0	6	2434	58	0	100	2.32	2	0	0	0	100	0
Petya	13/4	0/0	0/0	0/0	100/100	0/0	22	1336	47	0	100	3.40	2	0	0	0	100	0
TeslaCrypt	4/1	0/0	0/0	0/0	100/100	0/0	8	20	0	0	100	0	2	0	0	0	100	0
Gandcrab	4/1	0/0	0/0	0/0	100/100	0/0	10	39	0	0	100	0	2	0	0	0	100	0
Normal	0/0	0/0	0/0	0/0	0/0	0/0	0	0	0	0	0	0	0	0	0	0	0	0

Overall, Zeek-script/Snort/iASTD sucessfully detected CSE-CIC-IDS2018 and CTU attacks with a DR of 100%. Zeek-script produced less TPs than Zeek-sig per attack thanks to its correlation capabilities using global state variables. In Zeek-sig, we have attempted to specify signatures for Distributed DoS (DDoS) and SSH/FTP Brute attacks, essentially based on protocols and weak observed contents (e.g., `GET / HTTP/1.`, `User-Agent: Mozilla`) that were not unique enough (FPR = 100%). In addition, Zeek-sig detected SQL injection, XSS and Web BruteForce with a DR of 100%, because more precise and unique patterns were found (e.g., `.php?id=3+`, `script\x25\x33\x45`).

Snort generated a significant FPR for LOIC DDoS UDP (15.65%) and HOIC DDoS HTTP(13.31%). Since these attacks have not unique signatures (e.g., `GET /`) , one must rely on the statistical distribution of packets per second. We have used Snort features like *threshold* to reduce the number of alerts. Snort also generated a significant number of TPs for LOIC DDoS UDP (8904), HOIC DDoS HTTP (197), FTP BruteForce (3868), SSH BruteForce (67), and Donbot (91).

Overall, Zeek-script and iASTD achieved better detection performance than Zeek-sig and Snort with a high DR of 100% and no FP. The tools were able to correlate multiple HTTP and DNS connections from CSE-CIC-IDS2018 and CTU attacks.

Real-time testbed. In Table 1, Zeek/Snort/iASTD detected ransomware attacks with a DR of 100%. Zeek-script produced less TPs than Zeek-sig and no FP for WannaCry and Petya attacks. For Zeek-script, we got 3 TPs for WannaCry and 4

TPs for Petya using a behavioral analysis over SMB based on the entropy [12]. In addition, Zeek-script did not generated FPs on the normal traffic.

Like Zeek-sig, Snort matched network sessions only on a stream-by-stream basis and generated redundant TPs per attack (e.g., 22 TPs for Petya). In addition, Snort produced a significant FPR compared to Zeek-script and iASTD for WannaCry and Petya attacks (2.32% for WannaCry, 3.4% for Petya).

Oppositely to Snort and Zeek, iASTD can correlate both network sessions and host logs by generating no FP and 1 TP per attack for each environment (i.e., 1 TP for host and 1 TP for network). Similar to Zeek and Snort, iASTD did not generate FPs on the normal traffic.

5.3 Discussion

Snort is a low level, stateless, event pattern language. Zeek is a scripting language that is essentially a programming language. Its composition mechanisms are those of imperative programming languages: procedural abstraction and programming composition using if-then-else, case analysis, and state variables. Creating Zeek script is a daunting, complex and error-prone task. ASTD is a more abstract language. Its state machines offer a graphical, deep representations of attack behavior and state transitions. Its process algebra operators free the specifier from dealing with low level composition of attack specification elements. An attack can be specified in a modular fashion, following the natural language description of an attack's structure into phases and steps. Attack specifications can be easily composed to create a global IDS specification. Snort and Zeek signatures can be easily represented by ASTD specifications using automatons and quantified interleaves.

Verification tools [9] can be used to check the correctness of ASTD specifications and check properties about them. ASTDs are also extensible, portable and heterogeneous as they are domain-independent. This means that they can be executed in any environment and on any source (e.g., network/host events, natural events). It is an important factor to deal with complex attacks that operate on common networks but also on cyber-physical systems. Snort and Zeek languages require additional updates of the source code to be extended in new environments (e.g. host). In addition, Zeek and Snort signatures refer to unique strings (e.g., `GET /wordpress/?ARX8`) that can easily be changed by the attacker and make them obsolete. ASTD operators like quantified interleaving allow one to abstract from a particular machine (ip address, name) and to specify any ordering constraint, at any level of abstraction (e.g., ip, host name, uuid, etc.) and in any combination.

Zeek is stateful using state variables and event functions. It can correlate multiple network events and latterly host events after some manual updates of the source code. Being abstract and domain-independent, the stateful language ASTD correlates multiple diverse events in any environment, thanks to process algebra operators and ontologies that are used on transitions of ASTD automata to structure the knowledge about events. Snort cannot correlate different network connections (e.g.,

HTTP, DNS, SSH). Snort features like *flowbits* can only handle packets within the same connection.

As measured in our experiments, Snort has a significant FPR and a high DR on average. The processing time of Snort on the real-time testbed is also very low on average (2.336s for 1Gb packets, 8.201s for 10Gb packets). Zeek-script has a very low FPR and a high DR on average. Its processing time on the real-time testbed is low on average (7.480s for 1Gb packets, 29.766s for 10Gb packets). iASTD has a very low FPR and a high DR average but its processing time on the real-time testbed is relatively medium on average (20.184s for 1Gb packets, 96.778s for 10Gb packets). For network intrusion detection, Snort is faster than Zeek and iASTD since it matches each single network connection without correlating them. Zeek can correlate multiple network connections and it is faster than iASTD. For network and host intrusion detection, iASTD was able to correlate both network connections and Windows/Syslog events but the huge amount of events affected considerably the detection time (188.667s for 10Gb packets and 87 450 mixed Windows/Syslog events).

To improve the processing time of iASTD for network detection, we are currently working on translation rules to generate Zeek scripts from ASTD specifications [2]. Hence, one could use the Zeek engine to run ASTD specifications on network event streams. The Zeek scripts generated from ASTD specifications are as efficient as manually written Zeek scripts. Another way is to generate Snort signatures from ASTD specifications to process network events faster. This approach involves several FPs due to the aforementioned limitations of Snort. Thus, it is not suitable for network detection.

6 Conclusion

We have compared Zeek, Snort, and ASTD for intrusion detection. Snort is a stateless language that offers very limited event correlation capabilities. Both Zeek scripts and ASTD are stateful and thus better support event correlation. Consequently, Snort produces more redundant true positives and false positives than Zeek and ASTD. Zeek scripts and ASTD are equivalent in terms of detection and correlation capabilities. However, Zeek being a scripting language, it is less abstract than ASTD. Thanks to its process algebra composition operators, ASTD makes it easier to create, reuse, compose and maintain attack specifications. Snort is the most efficient IDS in terms of processing time because it does not support correlation. Zeek is faster than iASTD, but it should be possible to compile ASTD specifications into Zeek scripts to execute ASTD specifications more efficiently while benefitting from the features of the Zeek execution environment.

[2] The translation rules and the compiler are available at https://depot.gril.usherbrooke.ca/lionel-tidjon/castd

Acknowledgements This work was supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada). We thank Felix Vigneault and Jonathan Martineau for their contribution to the development of the iASTD tool. We thank Nokia Canada and CSE (Communications Security Establishment) of Canada for their support.

References

1. L. N. Tidjon, M. Frappier, and A. Mammar, "Intrusion detection systems: A cross-domain overview," *IEEE Communications Surveys & Tutorials*, 2019. [Online]. Available: https://doi.org/10.1109/COMST.2019.2922584
2. M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*, ser. LISA '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 229–238.
3. V. Paxson, "Bro: A system for detecting network intruders in real-time," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 3–3.
4. S. T. Eckmann, G. Vigna, and R. A. Kemmerer, "Statl: An attack language for state-based intrusion detection," *J. Comput. Secur.*, vol. 10, no. 1-2, pp. 71–103, Jul. 2002.
5. F. Cuppens and R. Ortalo, "Lambda: A language to model a database for detection of attacks," in *Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 197–216.
6. B. Morin and H. Debar, "Correlation of intrusion symptoms: An application of chronicles," in *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 2003, pp. 94–112.
7. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, "Quantified event automata: Towards expressive and efficient runtime monitors," in *FM 2012: Formal Methods*. Springer Berlin Heidelberg, 2012, pp. 68–84.
8. M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. St-Denis, "Extending statecharts with process algebra operators," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 285–292, Oct 2008.
9. L.N. Tidjon, M. Frappier, M. Leuschel, and A. Mammar, "Extended algebraic state-transition diagrams," in *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, Dec 2018, pp. 146–155.
10. T. M. Corporation, "Common attack pattern enumeration and classification (capec)," Tech. Rep., 2013, http://makingsecuritymeasurable.mitre.org/docs/capec-intro-handout.pdf.
11. B. E. Strom, J. A. Battaglia, M. S. Kemmerer, W. Kupersanin, D. P. Miller, C. Wampler, S. M. Whitley, and R. D. Wolf, "Finding cyber threats with att&ck-based analytics," Tech. Rep., 2017, https://www.mitre.org/sites/default/files/publications/16-3713-finding-cyber-threats%20with%20att%26ck-based-analytics.pdf.
12. iASTD repository, "Universite de sherbrooke," https://depot.gril.usherbrooke.ca/fram1801/iASTD-public, 2019.
13. B. Fraikin and M. Frappier, "Efficient symbolic computation of process expressions," *Science of Computer Programming*, vol. 74, no. 9, pp. 723 – 753, 2009, special Issue on the Fifth International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA?06).
14. I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, January 22-24, 2018.*, 2018, pp. 108–116.
15. S. Garcia, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *Computers & Security*, vol. 45, pp. 100 – 123, 2014.