Detecting Multi-Step Attacks: A Modular Approach for Programmable Data Plane

Abir Laraba, Jérôme François, Isabelle Chrisment
Université de Lorraine - Inria, France
firstname.lastname@loria.fr

Shihabur Rahman Chowdhury, Raouf Boutaba
University of Waterloo, Canada
{sr2chowdhury|rboutaba}@uwaterloo.ca

***Abstract*—The increasing sophistication of attacks over the last years such as the proliferation of complex multi-steps attacks, calls for new monitoring models and methods for diagnosing the attacks' severity and mitigating them in a timely manner. In this paper, we propose an in-network monitoring approach capable of detecting a set of composed behaviors and consequently triggering different levels of alerts and reactions. Our approach is based on a Petri Net model capable of aggregating individual attacks into a multi-step composition. To this end, we propose a method for deriving a Match-Action Table (MAT) abstraction from a Petri net model. MATs can be then deployed on a P4 programmable data plane, enabling flexible re-composition of attack detection steps at runtime. We demonstrate the feasibility of our proposal by modeling the detection of a multi-step DNS cache poisoning attack and implementing the model on a P4 programmable data plane.**

***Index Terms*—P4, Programmable data plane, Security, SDN, Monitoring, EFSM, Petri Nets, DNS**

I. Introduction

Network attacks are becoming more sophisticated and evolving to become multi-step attacks that very often pass through a set of steps and attackers before impacting the final target [1]. Multi-step attacks improve the efficacy of even common attacks such as DNS cache poisoning attacks as illustrated by the authors in [2]. Both the DNS request transaction ID and the source port are used to avoid brute-forcing of DNS answers. However, the authors demonstrate that the attacker can first guess the source port by performing a port scan and then a brute force on the DNS Transaction ID, which avoids a combinatorial explosion.

Stateful approaches have been shown to be efficient for detecting and mitigating such attacks [3]–[5]. Stateful security monitoring usually requires advanced software or hardware appliances [6], [7], which are expensive and vertically integrated with little programmability. Recently, with the advent of the Software-Defined Networking (SDN) paradigm, the softwarization of network functions has promoted higher flexibility in configuring and monitoring networks from both, control and data plane point of view. Especially, advances in programmable data plane enabled by P4 programming language [8] and Protocol Independent Switch Architecture (PISA) [9] create new opportunities for performing line-rate stateful security monitoring in the data plane.

978-1-6654-0601-7/22/$31.00

P4-based security monitoring solutions have already been proposed in the research literature [10]–[13]. The switches are usually deployed with a dedicated program to handle the attack, since switches are attack-specific, they are very efficient. However, they sacrifice the flexibility offered by programmable switches.

Different multi-step attacks can have one or more common steps such as IP/TCP scanning, brute-forcing, *etc*. The attacker can change or refine their behavior over time, so the composition of steps of an attack can also evolve. For these reasons, being able to compose and recompose a detection and mitigation procedure would be beneficial for flexibility and responsiveness.

In this paper, we address the challenge of efficiently monitoring a compound attack and eventually react against it. We propose to separate the decision module from the detection modules, in this way enabling the network operators to define an appropriate level of mitigation through different compositions of these modules. The decision module can be represented by an abstraction synchronizing the set of detection modules. Therefore, the users can control decisions and reactions on an attack progression at run-time using one model abstraction. This approach is highly configurable, easy to manage and helps in defining multiple security alert levels.

Our solution consists of a two-layer security monitoring approach in P4 programmable data plane. In the first layer, we adopt an Extended Finite State Machine (EFSM) abstraction [13] that we apply to define detection modules. This approach allows to detect individual attacks and thus suppose a perfect knowledge about the attacker behaviors or threats in advance. Our new approach empowers the composition of a set of an attack behaviors. The second layer corresponds to the decision module that gathers information from the first layer detection modules. The decision module is modeled using a Petri net which gives the possibility to compose behaviors of attacks at runtime and to define a set of alert levels based on the composition of behaviors from the first layer. J.P McDermott [14] was the first to introduce the approach of using Petri net for attack modeling and called it attack nets. Tokens moving from place to place indicate the progress of the attack. As described at [14] a Petri based model is efficient for modeling concurrency, attack progress, intermediate and final objectives. These different abstractions, EFSM and Petri nets can be programmed with P4 [8] and

instantiated on programmable PISA switches. Particularly, the concept of Reconfigurable Match-action Table (RMT) [9] allows us to recompose the second layer on the fly, *i.e.* redefine the Petri net. In a nutshell, this paper proposes a modular and configurable approach based on EFSM and Petri net for detecting multi-step attacks in the data plane. Our contributions are threefold:

- A two-layer method for synchronizing individual detection modules based on EFSM in a recomposable model using Petri nets;
- A systematic technique to map this method in the P4 data-plane enabling a switch to detect and mitigate attacks within the network (in-network);
- An experimental validation of our technique on the DNS multi-step cache poisoning attack.

The rest of the paper is structured as follows. We present an overview of the proposed approach in Section II. We then provide details of the proposed Petri net model in Section III and the mapping into a P4 programmable data plane in Section IV. We describe the DNS cache poisoning and our application to this multi-step attack in Section V. We report our findings from the experimental evaluation in Section VI. Some deployment considerations are discussed in Section VII. After a summary of the related work in Section VIII, the paper draws conclusions and points out future research perspectives in Section IX.

II. Proposed Approach

Multi-step and sophisticated attacks are based on a combination of behaviors. Every individual step contributes to the effectiveness of the attack. These steps can be sequential, parallel, optional or alternative. Detecting such attacks at an early stage can be done by correlating observations representing the different attack steps. In this paper, we assume that each attack step aims a particular sub-goal or stage. Based on the attack progression and on reached sub-goals, we can define different levels of alerts and reactions.

The proposed approach illustrated in Figure 1 models the relations between the attack sub-goals and their combinations. It is based on a two-layer modeling. A first layer (low layer) relies on a set of detectors in charge of monitoring individual sub-goals or stages of an attack. A second layer (high, global layer) represents a decision model which is synchronized with the set of behaviors detected from the first layer modules and defines possible decisions and alert levels. This high-level module checks whether individual detectors have reached their sub-goals to decide on the attack progression based on a user-given attack representation. Indeed, our approach allows the user to compose the different behaviors according to his own needs.

Since our method is defined to be deployed on P4-based programmable switches and run at line-rate, we have faced a number of constraints. A P4 program [15] defines the packet processing starting with the parsing phase. Once a packet is parsed, the retrieved headers can be then used to apply the processing logic and actions (drop, forward, etc.) with the

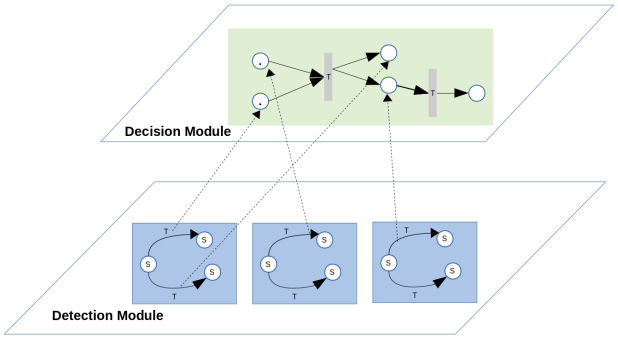

Fig. 1. Approach overview

support of re-configurable Match-Actions tables (MAT). On such platforms, complex processing is heavily limited [16]. For instance, stochastic models should be prohibited, state management is not supported for recursive functions, no loop is allowed, etc. In our approach, we mainly rely on:

- **Match-Actions tables (MAT):** match lookup keys with packet fields or computed metadata with actions.
- **metadata:** data generated and computed during P4 program execution (*e.g.* switch queue size).
- **registers:** stateful constructs (across packets) read/written by actions.

We combine a Petri net model with a set of EFSMs models [13] to be able to detect sophisticated attacks. We explicitly separate the attack decision (monitoring) module from the attack detection modules.

- **First layer (Detection modules).** This layer includes a set of individual EFSMs models. EFSMs have been proved to be P4-compatible either within the programmable logic or using Match-Action tables [13]. However, representing an EFSM with only MATs consumes a lot of tables entries and is less efficient than a compiled program running the EFSM model. Obviously, compiled programs discard the possibility to modify the EFSM model. Therefore, every detector has to represent an EFSM of an individual and generic attack stage or sub-goal (scan, connection attempt, etc.). It can be then pre-deployed on the switch and leveraged by the user through the combination of behaviors in the second layer. It is worth mentioning that non EFSM models could also be programmed and compiled with the P4 programmable logic to define other types of attack detection modules but this is out of the scope of this paper.
- **Second layer (Decision module).** In this layer a Petri net models the possible reactions and alerts based on the behavior combination information gathered from the detection modules. The relations between the first layer elements (composition, sequence, parallelism) are defined. The main advantage of the Petri net model is its simplicity, which can be implemented using Match-Action Tables (MATs). The MATs are reconfigurable at runtime and so they are supporting the flexibility of our

approach. Therefore, a user can define and redefine the composition of the behaviors on the fly.

The Petri net describes the semantics of an attack event combination, synchronization, and composition during a multi-step attack. Depending on the behavior detected at each EFSM model, the tokens of the Petri Nets indicate whether a certain sub-goal has been achieved.

As a summary, on a target P4 switch, EFSM models are compiled to a set of conditional statement programs which can not be modified at run-time but generate tokens in a Petri net. The fundamental advantage of the proposed approach is that a Petri net enables a modular monitoring by mapping it into a MAT that is more configurable at run-time via a control plane. The Match Action Table is configurable, based on the attack risk levels and the composition of the set of an attack's sub-goals.

III. Composition Model

A. Petri Net to compose attacks

We suppose that a multi-step attack is composed of a set of sub-goals $SG = \{sg_1, sg_2, .., sg_n\}$ and stages $AS = \{as_1, as_2, .., as_m\}$; each stage represents the composition of a set of sub-goals to achieve an intermediate objective. The set of sub-goals and stages are combined to reach a final attacker objective. Therefore, an attack ATK is defined as follows: $ATK = \{SG\} \cup \{AS\}$

Figure 2 illustrates an example of our Petri net model, where an attacker can reach three subgoals, $SG = \{1, 2, 3\}$, and 3 stages, $AS = \{A, B, C\}$.

We define a Petri Net PN by a 5-tuple $PN = (P, T, M, M_F, R)$ with:

- P, the set of places, represents the set of an attack's sub-goals SG and stages AS, such that, $\forall sg_i \in SG : \exists p_i \in P$, $\forall as_i \in AS : \exists p_i \in P$.

  A set of reaction actions, namely $decision$, can be associated with the set of places representing an attack stage, such that, $\forall p_i \in AS \ \exists decision = \{\emptyset, drop, alert, .., etc.\}$. The decision can be, for example, doing nothing (empty), an alert, or a drop action. It is worth mentioning that this set of actions is not fixed and depends on the switch capabilities which can be leveraged as actions in a MAT, e.g., placing packets in different QoS queues or modifying them. However, this assumes the reaction type is available to be used in MATs (either native or implemented beforehand similar to EFSMs). In our example $P = \{1, 2, 3, A, B, C\}$
- A set of tokens M. The presence of one token in a place indicates that an attack sub-goal $sg_i \in SG$ or an attack stage $as_i \in AS$ has been achieved.
- A marking function $M_F : P \times M$ denotes the marking of a place p with a token. This action is triggered by a module of the first layer (e.g., $EFSM$ model) to inject tokens into the Petri model when executed. This is different to the usual Petri net models where tokens are initiated with an initial marking. In our case, tokens can be injected externally and the Petri net execution can be changed based on external events (modeled by EFSM).
- A transition $t \in T$ is connected to input and output places with arcs. A transition is enabled if each input place contains at least one token and it produces one token in each output place. The transition represents a conditional set of an attack's sub-goals or stages to be validated to reach another stage. In our model, unlike the semantics of transitions in common Petri nets, a token is not consumed from a place by a transition to another place. Hence, when attacker reaches an attack sub-goal or stage, this information is voluntary made persistent over time. In the example in Figure 2, a token in place A will be added if there is at least one token in places 1 and 3.
- The arc $r \in$R if r connects places and transitions and is formally defined as the couple (i, o) where $i \in P$ and $o \in T$ or $i \in T$ and $o \in P$ exclusively. Hence, places cannot be connected directly.

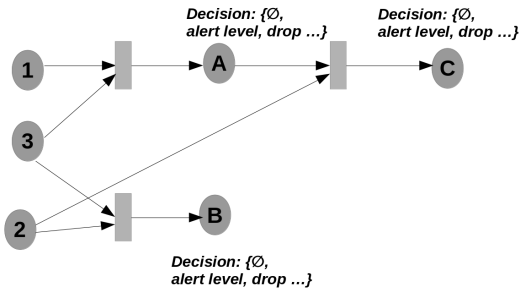

Fig. 2. Petri Net Example

Once the Petri Net model PN is defined, the execution of its transition system will change tokens through places. The state of its execution can be represented by the presence of tokens in each place and is formally denoted as $PN' = (p_1(m), p_2(m), \ldots, p_n(m)$. By analogy, $SG' = \{sg_1(status), sg_2(status), \ldots, sg_n(status)\}$ represents the different status of each sub-goal sg_i where $sg_i = 1 \iff p_i(m) > 0$ (sg_i is binary).

B. Attack sub-goals

The global model based on Petri net is independent of detection modules because the marking function M_F can be instantiated by any type of modules. However, in this paper, we adopt the EFSM abstraction for modeling the detection of each sub-goal related to an attack. This abstraction has been proven to support the right level of expressiveness to capture stateful attacks without the explosion of the number of states and, most of all, it can be mapped into the P4 primitives as proposed in [13]. An EFSM is formally defined as a 7-tuple (S, E, A, I, V, C, T), where : S is the set of states, E the set of events, A the set of actions, I the set of initial states, V the set of variables, C the set of conditions and T the set of transitions. At the switch level, packets are parsed and the extracted information from headers is used to change the

current state of the EFSM (*e.g.* TCP flags) and/or apply actions (*e.g.* update a variable like the last seen TCP sequence number, modify packet header, etc.). To cope with the proposed layered model, the decision module (Petri net) must synchronise with the detection module (the set of EFSMs). Hence, the action set is augmented with the marking function M_F to add a token into the Petri net at a specific place.

IV. Mapping Petri Net model into P4

A. Match-Action Table representation

To map and execute the Petri net model into a P4 programmable pipeline, the Petri model needs to its particular constructs. As highlighted in Section III-A, a transition is triggered when all input places are marked. It is worth mentioning that a transition requires at least one token in the input places and, in our case, tokens are maintained and made persistent once set in a place. Therefore, keeping track of the number of tokens can be omitted. For the same reason, the attack stages do not need to be maintained because they can be recovered from the persistent tokens in places representing sub-goals.

Hence, a single MAT is used to model the Petri net where the lookup keys represent the places of the sub-goals $p \in SG$, one bit for each $p \in SG$. For example, 8 places are represented as a single byte and 10010000 is the match key when tokens are present on p_0 and p_3. To keep in memory the state of the Petri net, *i.e.* the token, we rely on P4 registers that are stateful constructs written and read by P4 actions. However, they cannot be used directly in MAT lookup keys. Note that registers are extern functions, *i.e.* vendor specific, but they are assumed to be present and usable as they are also a basic functionality. When processing a packet, P4 allows the use of metadata to get contextual information about the switch. For example, queue occupancy can be retrieved in a metadata but there is also user-defined metadata. It is thus possible to read register values as metadata similarly to parse a packet into header structure and save metadata in a register.

As a result, we define a set of metadata representing the places $p \in SG$ and use the registers to maintain them persistent over the processing of packets subsequently: $Meta = \{meatadata_1, metadata_2, .., metadata_n\}$ where $\forall sg_i \in SG : \exists m_i \in MP$ with $m_i = 1$ when sg_i has been reached (a token is in the place), 0 otherwise.

Each entry in the MAT with match values ($\in Meta$) set to 1 represents the set of sub-goals composing a stage achievement $as_i \in AS$, and so the action of this match is the reaction defined for as_i.

Table I corresponds to the Petri net depicted in Figure 2:

- The first line represents the achievement of the sub-goal 1 and 3, the applied action (e.g., alert level 4) is the one associated with the stage $A \in AS$.
- The second line represents the achievement of the stage B composed of sub-goal 2 and 3.
- The third line represents the stage C corresponding to the achievement of the sub-goal 2 and stage 2, which in turn corresponds to sub-goals 1 and 3. As highlighted, tokens are persistent, so the attack stages are easily recovered from sub-goals without maintaining the individual states of each stage. As a result, it is more memory-efficient.

TABLE I
Petri Net as a MAT

meta 1	meta 2	meta 3	actions
1	0	1	$decision \in A$ (e.g., alert level 4)
0	1	1	$decision \in B$ (e.g., alert level 3)
1	1	1	$decision \in C$ (e.g., drop)

B. Petri net execution

Algorithm 1 describes how the first and second layer are synchronized in a minimal example, in particular without the detection program. Lines 2-4 actually define only the synchronization part of the first layer. If an attack sub-goal $sg_i \in SG$ is achieved (line 2), a token is set on a place p_i by the action $setToken(p_i) \in EFSM$, that uses the marking function.

In the data plane, marking tokens on places is represented by setting a $metadata_i \in Meta$ to 1 to indicate the presence of a token on a place p_i (line 6).

The Petri net tuple is abstracted to a MAT in the data plane, the set of match fields are the set of metadata $\in Meta$ and the $decision$ is a predefined action with respect to the achieved attack stage (line 7).

Algorithm 1: EFSM and Petri net synchronization algorithm

1 **Definitions:**
- n: number of the attack sub-goals SG
- $setToken(p)$: function triggered by an external model to set tokens

1: **Synchronization between** $EFSM$ **and** PN
2: **if** $sg_i(status) = achieved$ **then**
3: $\quad setToken(p_i) \Longrightarrow M_F : P \times M$
4: **end if**
5: **Synchronization between** PN **and** MAT
6: $M_F : P \times M \Longrightarrow metadata_i = 1$
7: $PN : (p_1(m), p_2(m), .., p_n(m)) \Longrightarrow$ $table(match_1(metadata_1), match_2(metadata_2),$ $..., match_n(metadata_n) \mid decision)$

V. Application to multi-stage DNS cache poisoning Attack

In this section, we demonstrate the viability of our proposed layered approach to detect the recent DNS multi-stage cache poisoning attack [2].

To poison the DNS cache, an attacker starts with a DNS request sent to the victim DNS server as shown on left of Figure 3 (step 1). This server is in charge of the recursive resolution by requesting other DNS servers within the Internet (step 2). The objective of the attacker is to reply first with a

valid answer to one of these requests. To be valid an answer must match the domain initially requested but also the source port and query ID used by the DNS server. In the past, the source port was not randomized and it was feasible to brute force the ID [17]. Due to this flaw, the UDP source port is now randomized making the brute-forcing impossible (quadratic complexity). An attacker can alleviate this issue by doing a UDP port scan and expecting the DNS server to respond with ICMP unreachable messages for all closed ports except for the open one (steps 3 and 4). Thanks to this strategy, the attacker can first guess the source port and then brute-force the query ID in step 5 (linear complexity).

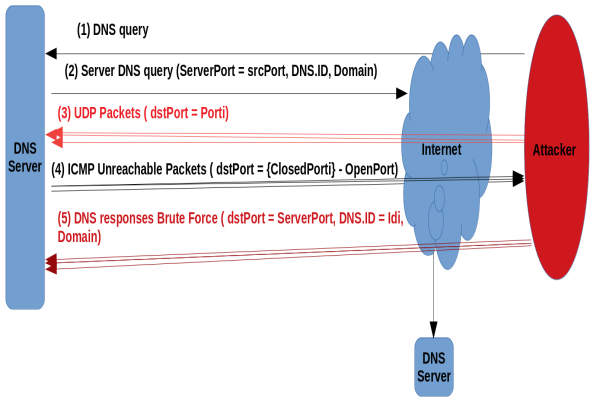

Fig. 3. Multi-stage DNS cache poisoning attack

A. Attack Petri Net

We decompose the attack into 3 sub-goals $SG = \{sg_1, sg_2, sg_3\}$: sg_1 corresponds to the reception of a valid response (valid domain and valid ID) that matches the server outgoing query and the port used in the response is identified as open; sg_2 represents the UDP port scan; sg_3 is the DNS brute force attack. As highlighted in Figure 4, there are three attack stages with different alert severity levels. For example, when an attacker does a port scan sg_2 and a DNS brute force for sg_3, the Petri net enters into attack stage 5 (alert level 5). However, to be effective, the attacker has also to send a DNS valid response sg_1. So, if all sub-goals are reached, all attack stages are reached too, including the most critical with the action *alert level 6*. In this attack, we have to jointly monitor two types of traffic: DNS traffic and generic UDP traffic (combined with ICMP). In the DNS traffic, we have to observe both the brute force and a valid answer. Therefore, sg_1 and sg_3 will be monitored by a DNS-specific module while sg_2 is handled apart. Furthermore, a valid answer is also expected from the legitimate DNS server. Hence, observing sg_1 only is not helpful to distinguish an attacker and so we must rely on observing other side attacks.

It is worth mentioning that the different detection modules require to share some information to be effective. For instance, valid answers can be emitted by a DNS server. To distinguish between a benign and malicious valid answer, our second

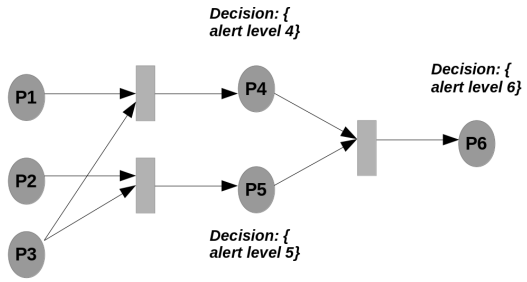

Fig. 4. Petri Net of DNS cache poisoning

layer model relies on detecting the port scan. This implicitly assumes that the port scan achieved (sg_2) by the attacker has discovered the port used for the valid answer (sg_1). Independently of the second layer performing Petri net execution, the first layer modules may need to share information. Similarly to token management, shared data between detection modules is performed using registers denoted as context variable hereafter.

B. Port scan detection

Figure 5 represents the detection module of the port scan attack sub-goal sg_2. The EFSM is defined as follows:

- $S = \{ListenUDPQueryICMPResponse, ScanAttempt, AlertPortScan\}$;
- $E = \{IP.dst = DNSserverIP, IP.protocol = 17, ICMP.type = 3\}$
- $V = \{MetaQR, DNSserverIP, THscan\}$;

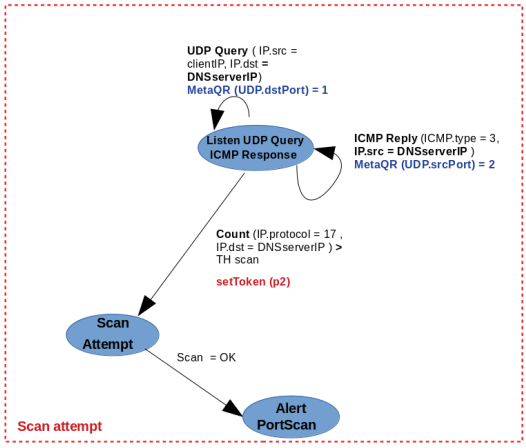

Fig. 5. Port scan detection

This EFSM monitors if an attacker does a port scan. Therefore, the EFSM maintains a counter of UDP packets sent from the potential attacker to the server[1]. A register, $CountScan$, is used to track this value in a stateful manner. It must be managed for each potential victim, *i.e.* IP addresses of DNS servers to be protected. The source IP addresses are not taken into account in our model as we assume the attacker has the capacity to perform the attack from distributed hosts.

[1] stateful variables used in EFSM are mapped to registers

Reaching a predefined threshold triggers a transition to the state $s = ScanAttempt$. Once this attack sub-goal sg_2 has been met, a token in the Petri Net is set on the place $p2$ representing the attack sub-goal ($setToken(p2)$). To keep track of tested and open ports, we introduce the context variable $MetaQR$ that is a hashmap with the UDP port as a key. When the UDP probe is seen, the value is set to 1 meaning that the port is assumed open until an ICMP unreachable (type 3) reply is received. In that case, $MetaQR$ is set to 2 (port closed).

C. DNS brute force detection

Figure 6 describes the DNS brute force stage detection module composed of the sub-goals sg_1 and sg_3, the EFSM is defined as follows:

- $S = \{ListenDNSQueryResponse, Bruteforceconfirmed, MatchedResponse, Attackconfirmed\}$;
- $E = \{IP.dstIP = DnsServerIP, IP.srcIP = DnsServerIP, UDP.dstPort = 53, UDP.srcPort = 53, DNS.ID, DNS.Domain\}$
- $V = \{MetaServerQR, DNSserverIP, THbrute\}$;

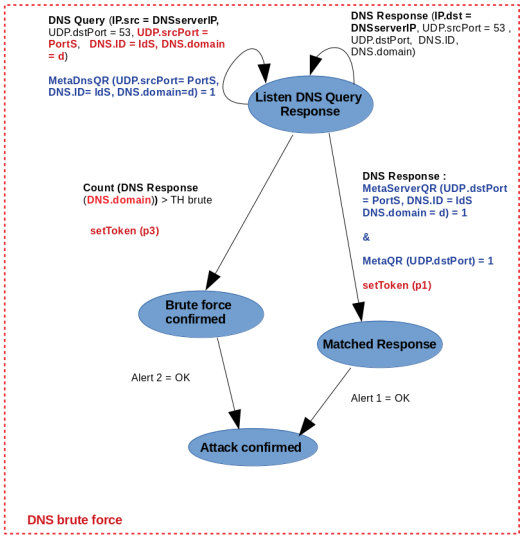

Fig. 6. Multi-stage DNS cache poisoning EFSM (DNS Brute Force)

In the DNS cache poisoning attack, a set of invalid responses are expected before a legitimate response that matches the server outgoing query. Therefore, counting unmatched queries and comparing this counter with a specific threshold can be effective for the attack detection. To detect the match and mismatch of DNS responses, a context variable $MetaServerQR$ associated with a key is used to track the server queries:

- When a server query is received, $MetaServerQR$ is set to a value (e.g, set to 1) with a key identifier composed of $(UDP.srcPort, DNS.ID, DNS.domain)$.
- When a DNS response is received, the value of the $MetaServerQR$ is verified using the reverse port in the key $(UDP.dstPort, DNS.ID, DNS.domain)$ to check the match with the server query.
- Receiving a valid response that matches the server query and the port in a scan attempt $MetaQR = 1$ (shared context variable) results in setting a token in the Petri Net at the place p1 $setToken(p1)$ associated with the sub-goal sg_1.
- Reaching a threshold of invalid responses results in setting a token on the place $setToken(p3)$ corresponding to the sub-goal sg_3. Similar to port scan, a register $CountDNSResponse$ is used to maintain the number of received DNS responses to detect the brute force.

VI. Evaluation

We have implemented the DNS multi-stage cache poisoning attack detection and mitigation model using P4-16 [18] and the bmv2 software switch target [19]. We have used mininet [20] and p4app docker image [21] for conducting the experiments. The control plane provided by the bmv2 CLI is used to configure the MATs.

A. DNS attack mitigation

In this experiment, we assume a topology composed of one switch, one server, and an attacker. We evaluate the effectiveness of the proposed approach when changing the Petri Net MAT configurations: (a) only drop packets when all sub-goals are achieved (P6 in Figure 4), (b) when at least sg_1 and sg_3 are achieved (Petri net P4 or P6 in Figure 4) and (c) when at least sg_3 and sg_2 are achieved (Petri net in P5 or P6 in Figure 4). Therefore, the Petri net-related MAT is configured accordingly.

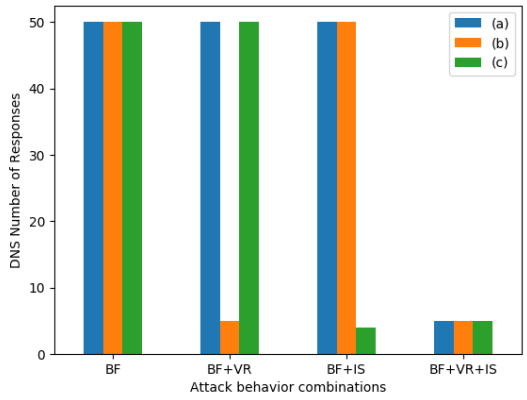

Fig. 7. DNS number of responses for different behavior combinations

To assess its proper functioning, we also make the assumption that an attacker runs different behaviors when generating 50 DNS responses as shown in Figure 7:

- BF: a DNS brute force attack only (sg_3)
- BF+VR: a DNS brute force attack combined with the reception of a response that matches the server query $(sg_3 + sg_1)$

- BF+IS: a DNS brute force attack combined with a port scan $(sg_3 + sg_2)$
- BF+VR+IS: full malicious behaviours composed of a port scan, brute force and a valid response $(sg_1 + sg_3 + sg_2)$

The bars in Figure 7 represent the number of DNS responses. According to the different attacker behaviors and the MAT configurations, the number of packets to filter varies. For example, the configuration (a) only drop the traffic if and only if all attack stages have been reached (P6 in Figure 4). This results in dropping packets only when the most complex behavior is monitored. In case of $BF + IS$, a token is added in P2 and P3 and so a transition to P5 occurs. Assuming configuration (b), packets are not dropped in that case as confirmed in Figure 7. The detection probability can be directly derived from the detection threshold of the brute force which is inversely related to the number of malicious requests that are not filtered.

B. Switch processing time

Mininet and bmv2 switches are limited in providing an evaluation performance that will be representative of a deployment onto a real programmable switch. However, we can still compare different scenarios in our case.

In this experiment, we evaluate the switch processing time when increasing the number of the attack sub-goals artificially (3, 6, 9 and 15). 100 DNS packets are generated and Figure 8 reports the processing time per packet.

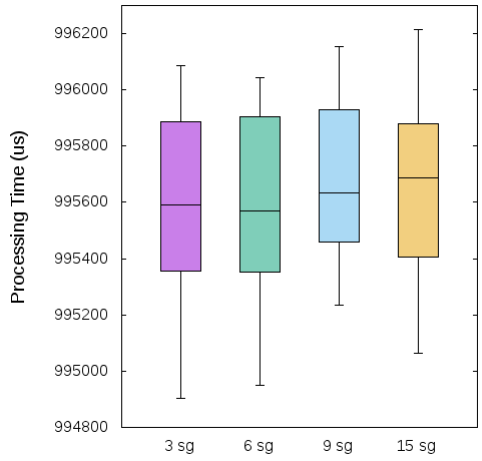

Fig. 8. Switch processing time vs. Number of attack sub goals

We can see that the the median value is approximately 995 ms without no significant increase between the scenarios. This shows that the proposed approach is scalable with respect to the number of an attack's sub-goals to be detected in parallel. As a reminder, one bit per goal is necessary to do the lookup in the MAT. Therefore, 15 bits are used at most in our case which thus can be matched efficiently.

In the next experiment, we evaluate the switch processing time when changing the position of a MAT entry to be matched (1, 10, 20 or 32). As highlighted in Figure 9, the switch average processing time is approximately 996 ms for all entries match positions. We can deduce that the position of an entry that is matched does not significantly impact the switch processing time. It is worth mentioning that the number of MAT entries is bounded by the number of Petri net places, $|P|$.

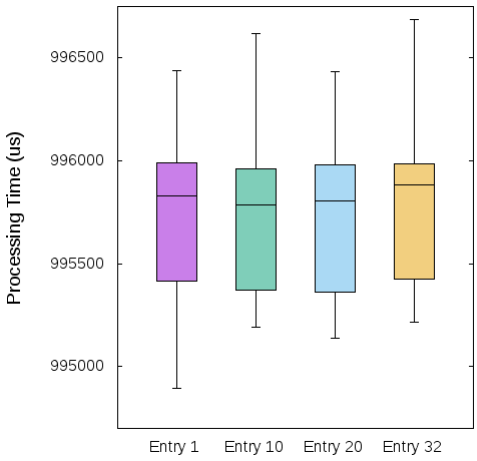

Fig. 9. Switch processing time vs. table entry position

VII. Discussion

A. DNS domain name parsing

Detecting DNS cache poisoning attack requires manipulating DNS domain names which are variable in length [22]. Parsing variable length fields in programmable data plane is challenging since the parser is designed to parse fixed length values. In the DNS header, the question field separates the domain name into a set of labels, each preceded by its length in terms of character in bytes, the last label is followed by a `0x00` byte which specifies the end of the domain name. This separation into labels makes practicable the parsing of domain names. Different parser states can be predefined based on different maximum label lengths so that fixed length values are parsed as proposed in [22]. Given this restriction, in our case, we suppose that each domain name is composed of 4 labels, each of 20 bytes length at most.

B. Memory requirement

The implementation of the proposed solution to detect DNS cache poisoning requires four registers to maintain and monitor per-packet information. We assume that 2-bits values are maintained in the two registers ($DNSRequestResponse$ and $ICMPQueryResponse$) and 32-bits values for count registers ($CountScan$ and $CountDNSResponse$). Therefore, tracking 10 000 DNS requests requires allocating 42.5 KB. The proposed approach also requires a single MAT to implement a Petri net. For DNS cache poisoning, 3 match fields corresponding to 3 sub-goals each with 1 bit are required. For 10000 entries, 3.75kB of memory is consumed. The detection of an attack composed of 6 sub-goals requires 7.5kB for 10000 entries.

VIII. Related work

A. DNS cache poisoning attack

Many attacks target the DNS protocol even with the recent proposed security techniques. Defence methods such as source port randomization, 0x20 encoding, and DNSSEC were proposed to prevent the attacks. DNSSEC and 0x20 encoding are still far from being widely deployed [23] due to compatibility issues and the necessity of DNS server modifications. Authors in [24] studied the injection of vulnerabilities in DNS resolution platforms and revealed significant security issues. Mainly, authors invoked that DNS caches can be poisoned with some efforts. On the other hand, RFC5452 [25] notes the effects of source port randomization is significantly reduced by NAT devices. Recently, Man *et al.* [2] have introduced a new attack capable of poisoning the DNS server cache. The authors have found weaknesses that allows an attacker to divide the attack by guessing the DNS server source port using a port scan at the first stage and then guessing the DNS transaction ID at a second stage. There are numerous techniques addressing the detection and characterization of port scanning. They are usually based on the ratio between successful and failed connections [26]. In [27] authors use predefined access probabilities for port scan detection. In [28], the authors have discovered a weakness in the pseudo random number generator used for generating UDP source ports in the linux/android kernel and they can speed up the DNS cache poisoning attack by a factor of 3000 to 6000. A new class of DNS poisoning attacks is described in [29]. It initiates the attack on the client cache by adopting a local port reservation forcing the server to pick one available port. These works demonstrate that the DNS cache poisoning attack can still be conducted and is a valid problem to be considered. In [30], the authors propose to duplicate DNS queries and send them to multiple resolvers for verification purposes but thus entail a major overhead. In [31] a randomly generated 4-bits with the random port and transaction ID is used. In contrast to our approach, these two solutions require the modification of the DNS protocol. Furthermore, authors in [2] recommend disabling outgoing ICMP replies or using rate-limiting methods which can have a negative impact on collocated traffic.

B. Attack detection in programmable data plane

SDN has introduced new possibilities for attack detection, especially with the emergence of data plane programmability. Harrison *et. al* [32] introduced a distributed system to detect heavy hitters. In [33], a real-time DDoS attack detection scheme is offloaded to a switch and monitors entropy changes. In the same direction, Poseidon [34] mitigated volumetric DDoS in the data plane. In [35], the authors introduced a link failure detection system. Actually, many research works have focused on a particular scenario or attack to detect and mitigate [10]–[12]. A few of them have addressed DNS attacks. In [22], a framework has been designed to associate traffic by domain name in order to let the operators applying rate limit traffic by domain names. P4DNS [36] implemented an in-network DNS server. Both works offered solutions for parsing variable domain name lengths by implementing pre-defined parsers. In our previous works, we introduced a general formalism based on an EFSM to describe a malicious behavior that can be mapped to a P4 pipeline. [13], [37].

C. Attack models

Attack Trees and Petri Net have been widely used to describe an attack process [38]. The attack tree approach adopts a tree representation of the dependencies among the actions performed by attackers. The Petri net approach provides more flexibility and expressiveness in capturing multiple and simultaneous actions [39]. The utility of using Petri nets for attacks modeling was first proposed by McDermott [14] as an alternative to attack trees. Colored Petri Nets bring more expressiveness by distinguishing tokens with colors [40]. In the same direction, the authors in [41] proposed a Petri based net for threat modeling. Such models are widely used in the field of cyber-physical and industrial systems [42], authors in [39] focused on smart grid attacks. Although more advanced technique exists, in particular using machine learning [43], such solutions must be deployed at end-hosts or on specialized middleboxes. Our proposed in-network approach can be considered as complementary to filter traffic as earlier as possible and at line rate.

IX. Conclusion

This paper proposed a layered approach based on a Petri net model to represent multi-step attacks with a systematic method to offload it to a programmable data plane. Our approach relies on a set of individual attack detection functions. Those functions can be abstracted as an EFSM but could also use another formalism or be directly implemented in the switch. Our main objective is to capture the dependencies between these functions as a Petri net and represent the Petri net transitions as a single MAT which can be reconfigured at runtime. As a result, it is possible to compose and recompose multi-step attack mitigation procedures in a dynamic manner. It thus provides an effective way to adapt reaction actions based on the attacks progression. Our implementation demonstrates that the approach is effective in detecting the DNS multi-step cache poisoning attack in the data plane. In the case of deploying the Petri net model in a distributed manner, the user must consider the trade-off between communication overhead with state distribution management with regard to the scalability of a centralized deployment, which is part of our future work.

Acknowledgement

This work was partly supported by the FrenchPIA project Lorraine Université d'Excellence, reference ANR-15-IDEX-04-LUE and by the AI@EDGE project, funded by the European Union's Horizon 2020 research and innovation programme under grant agreement no. 101015922.

REFERENCES

[1] J. Navarro, A. Deruyver, and P. Parrend, "A systematic survey on multi-step attack detection," *Computers & Security*, 2018.

[2] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "Dns cache poisoning attack reloaded: Revolutions with side channels," in *SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. ACM, 2020.

[3] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, "Stateful intrusion detection for high-speed network's," in *Proceedings 2002 IEEE Symposium on Security and Privacy*, 2002.

[4] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," in *Information Security*, S. K. Katsikas, J. López, M. Backes, S. Gritzalis, and B. Preneel, Eds. Springer Berlin Heidelberg, 2006.

[5] P. Chaignon, K. Lazri, J. François, T. Delmas, and O. Festor, "Oko: Extending open vswitch with stateful filters," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018.

[6] "Zeek: An open source network security monitoring tool," accessed: May 29, 2020. [Online]. Available: https://zeek.org/

[7] "Fortigate 7000e series ips," Fortinet, accessed: May 29, 2020. [Online]. Available: https://www.fortinet.com/content/dam/fortinet/assets/data-sheets/FortiGate_7000_Series_Bundle.pdf

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, 2014.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of ACM SIGCOMM*, 2013.

[10] M. Dimolianis, A. Pavlidis, and V. Maglaris, "A multi-feature ddos detection schema on p4 network hardware," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020.

[11] T.-Y. Lin, J.-P. Wu, P.-H. Hung, C.-H. Shao, Y.-T. Wang, Y.-Z. Cai, and M.-H. Tsai, "Mitigating syn flooding attack and arp spoofing in sdn data plane," in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2020.

[12] L. A. Q. González, L. Castanheira, J. A. Marques, A. Schaeffer-Filho, and L. P. Gaspary, "Bungee: An adaptive pushback mechanism for ddos detection and mitigation in p4 data planes," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021.

[13] A. Laraba, J. François, S. R. Chowdhury, I. Chrisment, and R. Boutaba, "Mitigating tcp protocol misuse with programmable data planes," *IEEE Transactions on Network and Service Management*, 2021.

[14] J. P. McDermott, "Attack net penetration testing," 2001.

[15] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *ACM SIGCOMM*, 2014.

[16] M. Jose, K. Lazri, J. François, and O. Festor, "Inrec: In-network real number computation," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021.

[17] N. Alexiou, S. Basagiannis, P. Katsaros, T. Dashpande, and S. A. Smolka, "Formal analysis of the kaminsky dns cache-poisoning attack using probabilistic model checking," in *IEEE 12th International Symposium on High Assurance Systems Engineering*, 2010.

[18] *P4 Language Consortium. P4-16 Language Specification*, 2018. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html

[19] *P4 Language Consortium. 2018. Behavioral Model (BMv2)*, 2018. [Online]. Available: https://github.com/p4lang/behavioral-model

[20] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. ACM CoNEXT '12, 2012.

[21] "p4app," https://github.com/p4lang/p4app, accessed: 2020-01-09.

[22] J. Kim, "Meta4: Analyzing Internet Traffic by Domain Name in the Data Plane," Princeton University, Tech. Rep., 2021.

[23] L. Yuan, K. Kant, P. Mohapatra, and C.-n. Chuah, "Dox: A peer-to-peer antidote for dns cache poisoning attacks," in *2006 IEEE International Conference on Communications*, 2006.

[24] A. Klein, H. Shulman, and M. Waidner, "Internet-wide study of dns cache injections," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*.

[25] R. v. M. A. Hubert, "Measures for making dns more resilient against forged answers," Internet Requests for Comments, IETF, RFC, 2009.

[26] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 2004.

[27] S. Staniford, J. Hoagland, and J. McAlerney, "Practical automated detection of stealthy portscans." *Journal of Computer Security*, 01 2002.

[28] A. Klein, "Cross layer attacks and how to use them (for DNS cache poisoning, device tracking and more)," *CoRR*, vol. abs/2012.07432, 2020. [Online]. Available: https://arxiv.org/abs/2012.07432

[29] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh, "Collaborative client-side dns cache poisoning attack," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019.

[30] H.-M. Sun, W.-H. Chang, S.-Y. Chang, and Y.-H. Lin, "Dependns: Dependable mechanism against dns cache poisoning," in *Cryptology and Network Security*, J. A. Garay, A. Miyaji, and A. Otsuka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[31] J. Mohan, S. Puranik, and K. Chandrasekaran, "Reducing dns cache poisoning attacks," in *2015 International Conference on Advanced Computing and Communication Systems*, 2015.

[32] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches." Association for Computing Machinery, 2018.

[33] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019.

[34] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of NDSS*, 2020.

[35] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019.

[36] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4dns: In-network dns," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[37] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, "Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification," in *2020 IFIP Networking Conference (Networking)*, Paris, France, Jun. 2020. [Online]. Available: https://hal.inria.fr/hal-02993199

[38] I. Nai Fovino, M. Masera, and A. De Cian, "Integrating cyber attacks within fault trees," *Reliability Engineering & System Safety*, 2009, eSREL 2007, the 18th European Safety and Reliability Conference.

[39] T. M. Chen, J. Sánchez-Aarnoutse, and J. Buford, "Petri net modeling of cyber-physical attacks on smart grid," *IEEE Transactions on Smart Grid*, 2011.

[40] R. Wu, W. Li, and H. Huang, "An attack modeling based on hierarchical colored petri nets," in *2008 International Conference on Computer and Electrical Engineering*, 2008.

[41] D. P. Mirembe and M. Muyeba, "Threat modeling revisited: Improving expressiveness of attack," in *2008 Second UKSIM European Symposium on Computer Modeling and Simulation*, 2008.

[42] K. N. Junejo and J. Goh, "Behaviour-based attack detection and classification in cyber physical systems using machine learning," ser. CPSS '16. New York, NY, USA: Association for Computing Machinery, 2016.

[43] A. Hemmer, M. Abderrahim, R. Badonnel, J. François, and I. Chrisment, "Comparative Assessment of Process Mining for Supporting IoT Predictive Security," *IEEE Transactions on Network and Service Management*, Dec. 2020.