Scaling IoT MUD Enforcement using Programmable Data Planes

Harish S A, *IIT Hyderabad, India*
Suvrima Datta, *IIIT Naya Raipur, India*
Hemanth Kothapalli, *IIT Hyderabad, India*
Praveen Tammana, *IIT Hyderabad, India*
Achmad Basuki, *Brawijaya University, Indonesia*
Kotaro Kataoka, *IIT Hyderabad, India*
Selvakumar Manickam, *Universiti Sains Malaysia, Malaysia*
Venkanna U., *IIIT Naya Raipur, India*
Yung-Wey Chong, *Universiti Sains Malaysia, Malaysia*

***Abstract*—IoT-based intrusions and network attacks are becoming ever more concerning. As a mitigatory measure, the IETF standardized Manufacturer Usage Description (MUD) which allows IoT device vendors to specify the legitimate communication patterns (as a MUD profile) of an IoT device. A MUD profile allows the validation of the actual communication pattern of an IoT device with the intended behavior at runtime. However, as the number of IoT devices increases, validation at runtime has scalability challenges in terms of the number of switch resources (*e.g.*, TCAM) required to maintain MUD profiles.**

In this work, we propose a scalable data plane primitive and a system on top of the primitive, which together enforce MUD profiles of thousands of IoT devices in a P4 programmable switch data plane. Our main idea is to avoid inefficiencies because of the repetition of header values while representing MUD profile-based ACL rules. Further, we exploit the characteristics of header values in ACL rules of real IoT devices and carefully partition the rules across multiple hash-based exact match-action tables in the switch data plane. Since hash-based data structures can be implemented using SRAM which is cheap and abundantly available (order of MBs) in commodity programmable switches, our approach scales well for a large IoT network.

I. Introduction

The Internet of Things (IoT) has become instrumental in enabling high-quality of service and automation in a wide range of application domains. Despite the advantages of IoT, lack of manufacturer attention to security has led to a flurry of vulnerabilities in IoT devices being exploited. Such IoT-based network attacks can cause damage to critical infrastructure (*e.g.*, DDoS attack [1], Mirai [2], VPNFilter [2]).

To address this problem, the Internet Engineering Task Force (IETF) recommends the use of Manufacturer Usage Description (MUD) to prevent malicious use of IoT devices [3]. MUD systematically captures the communication patterns of an IoT device in the form of a MUD profile. The profile is enforceable in a network (using firewalls, switches, etc) to ensure that the device is talking only to intended entities (*i.e.*, as per the specifications in the MUD profile).

Consider that an IoT device as shown in Figure 1 communicates with a relevant set of endpoints (*i.e.*, specific domains, DNS resolutions, etc). This traffic form identifiable patterns that can be captured as a MUD profile. If the IoT device tries to communicate with an attack server (*i.e.*, compromised), the MUD profile enforced on the path of traffic can apply mitigatory action on it (*e.g.*, block, mirror for further inspection, etc).

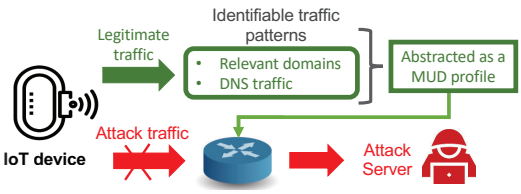


Fig. 1: **Generate MUD profile and enforce MUD profile in the network**

To realize this idea, few recent works [4]–[6] leverage Software-Defined Networks (SDN) and enforce MUD profiles of IoT devices. To be specific, whenever a new IoT device is introduced to the network, an SDN controller retrieves the device's MUD profile and generates whitelist access control rules (ACLs) based on the MUD profile. It is followed by the installation of these rules in either WiFi access points or generalized match-action tables of local switches that support OpenFlow protocol [7] and OpenFlow dataplane [8] having visibility over the IoT devices (*i.e.*, IoT MAC address)

However, this approach can prove to be problematic. Installing and managing MUD rules of IoT devices across multiple varieties of access points (*i.e.*, different vendors) is challenging. This problem is further compounded when IoT devices move between access points which requires rule migration.

Naturally, the solution is to enforce MUD at higher hierarchies that observe aggregated traffic. However, this approach would not scale well with an increase in the number of IoT devices in the network. This is because typically, whitelist ACLs are installed as match-action table entries in the switch data plane which internally uses Ternary Content Addressable Memory (TCAM) hardware. Since TCAM is power-hungry and expensive, it is a scarce resource and highly optimized for packet-header matching at high speeds. In general, TCAM table capacity in commodity switches is in the order of a few thousand entries [9], [10]. Given that the number of whitelist ACL rules for each IoT device could be anywhere from 20 to 50 [11], a single switch can enforce MUD ACLs of only a few

hundred devices (*e.g.*, 150-200). This is about a whole order of magnitude less than what is needed in a cluster of smart buildings with thousands of IoT devices or enterprise industrial establishments with IoT-based automation. Of course, one can add more switches or virtual network functions (VNFs) on servers to balance the load, but this will increase operational costs (*e.g.*, management overhead, handling consistency issues) and latency (when using VNFs).

In this paper, we propose an approach that scales MUD enforcement to thousands of IoT devices using P4-based programmable switches [12]–[14]. Further, we enforce MUD higher up in the network hierarchy on a P4 switch (as shown in Figure 4) but within the Local Area Network (LAN). This is specifically in the case of L2 routed networks where there is visibility of IoT MAC across the local network. This design by default reduces the management overhead and handles the mobility scenario of IoT devices across access points.

Next, to handle the scalability scenario, we base our approach on two key observations from MUD-based ACL rules of 28 IoT devices [15]. First, many header fields in ACL rules have few distinct values. This means placing ACL rules in a single large and wide TCAM table is inefficient because of the repetition of the same header values in multiple entries. Second, header values are only of two types, exact or don't care ('$*$'). Also, none of the headers have prefix-based values (*e.g.*, /24). This means fast parallel lookup supported by TCAM can be partially or fully replaced with SRAM-based exact match tables. Essentially, there is an opportunity to place a large fraction of MUD-based ACL rules in SRAM which is abundantly available in today's programmable switches [16].

Based on these two observations, we develop an SRAM-based packet classification algorithm that runs entirely in the switch data plane. To be specific, we leverage multi-stage programmable switch pipeline and carefully partition ACL rules based on multiple headers and map headers to stages. By doing so, we scale MUD rule enforcement to a large number of IoT devices.

The main challenge is that P4-based programmable data planes are resource-constrained supporting only a small set of per-packet operations (*e.g.*, hash, arithmetic, logical) with a small number of per-packet memory accesses (1-2 per-pipeline stage) and limited storage capabilities (tens of MBs of SRAM [17]). Moreover, MUD ACL rules contain don't care '$*$' values. Using only SRAM-based exact match-action tables to encode these MUD rules solicits the need for rule unfurling. We carefully design our system to work under these constraints. More specifically, to represent ACL rules, we find that the decision tree data structure suits well as it requires simple operations. Moreover, it is well known that decision trees are often used to represent functions on a wide input domain and save memory by avoiding common path repetition. We exploit this property and represent ACL rules in a decision tree followed by encoding the decision tree in exact match tables in the data plane.

The basic idea of representing ACL rules using a decision tree [18] and subsequently tilting the data structure sideways in the P4 data plane (to handle one header field per pipeline stage) is explored in recent work [19]. To realize this idea, we design and implement a system that has mainly three components. First, our system derives ACL rules of 28 different IoT devices from their MUD profiles. Second, it represents the derived ACL rules in a decision tree that is amenable to drive per-packet decision (*i.e.*, forward or deny) while allowing rule addition and rule deletion at run time. Finally, we encode the decision tree in the data plane using multiple match-action tables placed across multiple stages, which together guide the packet processing. We develop a prototype of the proposed approach on BMV2 software switch [20] and a Tofino-based hardware switch [16].

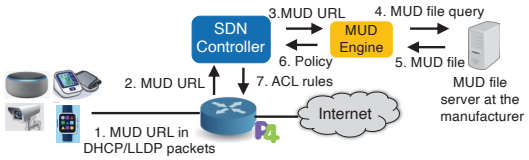
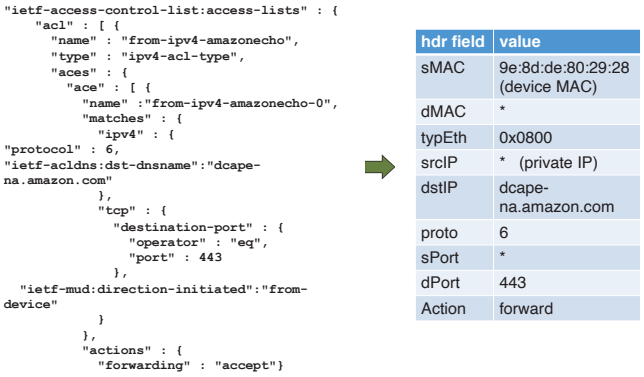


Fig. 2: **Enforcing MUD profiles using SDN framework**

II. Background

IoT device discovery and MUD profile retrieval. An IoT device can be uniquely identified using its MAC address (also called hardware address) on the local network. According to the MUD standard, DHCP requests from the device contain an associated MUD URL from which the MUD profile can be downloaded. As shown in Figure 2, a programmable switch can be configured to forward DHCP requests to an SDN controller, and the controller will download the MUD profile from the manufacturer's website. The SDN controller can also infer the type of the IoT device through it's MUD profile. The MUD profile has MUD access control entries (ACEs) which specify the expected behavior of the device. For example, the profile may state that the device can communicate with a specific set of domains (e.g., google, amazon, NTP servers, device vendors, etc) on a specific port, devices that belong to the same manufacturer, or devices connected to the same LAN. Figure 3 shows an example MUD profile and associated access control rule, where the rule is a tuple with 8 fields: sMAC, dMAC, typEth, srcIP, dstIP, protocol, sPort, and dPort.

TCAM-based lookup vs SRAM-based lookup. Consider that ACL rules of many IoT devices are installed in a switch table. On packet arrival, the switch reads header values that form a key, searches for the key in table entries (rules), and applies action (*e.g.*, allow or deny) associated with the matched entry. In general, the table key could be a combination of three match types: exact, longest prefix match (lpm), and ternary. If a table key contains lpm or ternary or both, to keep lookup time constant, TCAM hardware is commonly used because it supports parallel search across all table entries. On the other hand, when only the exact type (*i.e.*, without lpm or ternary) is present, entries can be stored in arrays of on-chip SRAM and do a hash-based lookup to find the matched entry in constant time. Between TCAM and SRAM, there is a fundamental tradeoff in terms of efficiency and cost. To be



```
"ietf-access-control-list:access-lists" : {
    "acl" : [ {
      "name" : "from-ipv4-amazonecho",
      "type" : "ipv4-acl-type",
      "aces" : {
        "ace" : [ {
          "name" :"from-ipv4-amazonecho-0",
          "matches" : {
            "ipv4" : {
"protocol" : 6,
"ietf-acldns:dst-dnsname":"dcape-
na.amazon.com"
            },
            "tcp" : {
              "destination-port" : {
                "operator" : "eq",
                "port" : 443
              },
  "ietf-mud:direction-initiated":"from-
device"
          }
        },
        "actions" : {
          "forwarding" : "accept"}
```

hdr field	value
sMAC	9e:8d:de:80:29:28 (device MAC)
dMAC	*
typEth	0x0800
srcIP	* (private IP)
dstIP	dcape-na.amazon.com
proto	6
sPort	*
dPort	443
Action	forward

Fig. 3: **MUD file to ACL rules**

more specific, compared to SRAM, TCAM requires $6\times$ more power and $6-7\times$ more area on a chip. As a consequence, many commodity switches with TCAM, support only a few thousands TCAM entries (*e.g.*, 2000-3000) [9], [10], [21], [22] with tens of MBs of SRAM. Since TCAM has to be shared with other core network functions such as routing, minimizing its usage is important.

III. Problem Statement

One approach employed for representing ACL rules is to use TCAM-based match-action tables where each rule is encoded within one table entry. However, this approach is cost-inefficient as it requires more TCAM. The table would require a wide TCAM to cover all header fields (*e.g.*, 27 byte-wide for 8 fields as shown in Figure 3), but contain only a few unique entries per header field (see Figure 5). On the other hand, hash-based exact match lookups using SRAM are sufficient when a table key contains only an exact value, but not sufficient if either ternary ('$*$') or lpm ($/24$) is part of the key. As mentioned earlier, in MUD-based ACL rules for 28 IoT devices [4], we observe they have header values of only two types: exact or don't care ('$*$'). Based on these two observations, the key question we would like to investigate is: *Can we design an efficient SRAM-based classification algorithm that runs entirely in the P4 data plane so that we can scale MUD enforcement to thousands of IoT devices?*

IV. System Design

Overview. The key idea of our approach is that instead of placing ACL rules in a wide TCAM-based table, we partition them on a per-header basis and place associated header values in multiple SRAM-based exact match tables. However, while we do so, it is challenging to emulate wildcard ('$*$') behavior using purely SRAM-based operations (*i.e.*, exact matches) while considering all the 8 header fields shown in Figure 3. This is because '$*$' values need to be unfurled to cover all combinations of the exact match values which leads to the addition of a large number of rules.

To address this challenge, we reduce the impact of wildcard ('$*$') rules on the size of the ruleset. We first partition the MUD

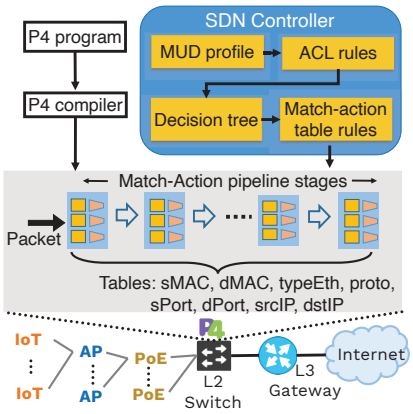


Fig. 4: **Possible deployment of proposed MUD system in a typical enterprise network hierarchy. AP: Access Point, PoE: Power over Ethernet**

ACL rules of an IoT device into: (1) forward traffic (from device) (2) backward traffic (to-device) and the remaining rules as (3) local rules (LCL). Subsequently, we represent the forward traffic (FT) and backward traffic (BT) rules as decision trees and then apply the decision tree on incoming packets by programming SRAM table entries on a P4 data plane. The local rules (LCL) are stored without modification in the 27 byte-wide TCAM table of the switch. Figure 4 shows our system workflow. In this section, we elaborate on five main components of our system which perform the following tasks: (a) preparation of MUD-based ACL rules; (b) handling wildcard ('$*$') entries; (c) representing ACL rules in a decision tree; (d) saving memory by avoiding repetition; and (e) implementing the decision tree in a switch data plane using exact match-action tables and associated table entries.

A. Preparation of MUD-based ACL rules

Rule partitioning. Our work builds on the MUD profiles of 28 real consumer IoT devices published at the UNSW MUD database [15]. We derive the corresponding ACL rules for every IoT device from these MUD profiles. Figure 5 depicts an example ruleset consisting of 14 rules. The rules in the example are chosen from the MUD profile of Belkin camera [23] which represent all 28 IoT devices in the dataset. Further, they are annotated with either forward traffic (FT), backward traffic (BT) or local (LCL) in Figure 5.

Classification into FT must meet three conditions: (1) the sMAC must be an exact value (2) the dMAC, sPort and srcIP must be '$*$' and (3) any combination of typEth, proto, dPort and dstIP can be either exact value or '$*$'. The conditions for classification into BT are the same but with their corresponding complementary header fields (sMAC-dMAC, sPort-dPort, srcIP-dstIP). The rules that remain are classified as local rules (LCL). This is based on the observation that the MUD entries (from their MUD profile) of the remaining rules are specifically annotated as 'local-networks', signifying that they determine local traffic (within the LAN).

Header partitioning. After rule partitioning, we observe that the FT and BT rules still contain many '$*$' values. Handling

Rule	sMAC	dMAC	typEth	Proto	sPort	dPort	srcIP	dstIP
FT 1	94:16:3e:3b:41:cf(A1)	*	0x0800(C1)	6 (D1)	*	8899(F1)	*	Devices.xbcs.net (52.23.33.88) (H1)
BT 2	*	94:16:3e:3b:41:cf(B1)	0x0800	6	8899(E1)	*	Devices.xbcs.net (52.23.33.88) (G1)	*
FT 3	94:16:3e:3b:41:cf	*	0x0800	17(D2)	*	67 (F2)	*	*
BT 4	*	94:16:3e:3b:41:cf	0x0800	17	67 (E2)	*	*	*
FT 5	94:16:3e:3b:41:cf	*	0x0800	17	*	53 (F3)	*	*
BT 6	*	94:16:3e:3b:41:cf	0x0800	17	53 (E3)	*	*	*
FT 7	94:16:3e:3b:41:cf	*	0x0800	6	*	9207(F4)	*	*
BT 8	*	94:16:3e:3b:41:cf	0x0800	6	9207(E4)	*	*	*
LCL 9	94:16:3e:3b:41:cf	*	0x0800	6	49153(E5)	*	*	*
LCL 10	*	94:16:3e:3b:41:cf	0x0800	6	*	49153(F5)	*	*
FT 11	94:16:3e:3b:41:cf	*	0x0800	1 (D3)	*	*	*	www.google.com (142.250.195.46)(H2)
BT 12	*	94:16:3e:3b:41:cf	0x0800	1	*	*	www.google.com (142.250.195.46)(G2)	*
FT 13	94:16:3e:3b:41:cf	*	*	2 (D4)	*	*	*	224.0.0.2(H3)
LCL 14	*	*	0x888e(C2)	*	*	*	*	*

Fig. 5: **Example set of 14 ACL rules from real IoT device: Belkin camera [23]; FT: Forward traffic (from-device), BT: Backward traffic (to-device), LCL: local rules**

'$*$' values require rule unfurling and the addition of a large number of extra rules (more on this in §IV-B). In order to avoid this, we further partition the FT and BT based on their header fields. Since the (dMAC, sPort, srcIP) header fields of FT rules contain only '$*$' values, we omit these header fields and create the FT ruleset as shown in Figure 6. The omission does not affect the semantic meaning of the FT ruleset. Similarly, for BT, header fields (sMAC, dPort, dstIP) are omitted. It is observed that the number of '$*$' values encountered in the resulting rulesets reduces significantly as seen in Figure 6).

Domain name resolution. The 'srcIP' and 'dstIP' field values based on MUD profiles contain domain names. Therefore, prior to enforcing rules in the data plane, the domain names have to be resolved to their corresponding IP addresses. During resolution, some DNS replies may contain multiple 'A' records or may give a temporal difference in the IP address for the same domain. Thus, there arises a need for consistency between the resolved IP addresses to be used in the switch and the one being used by the end IoT device. We maintain consistency through a domain name to IP address mapping in the control plane which initially generates ACL rules for all possible destination IPs followed by pruning of rules with unused destination IPs (based on flow activity). However, this is not the only approach to solving the inconsistency. Possible enhancements, challenges, and methods will be addressed in future work.

B. Wildcard entries

Rule unfurling. We encode the already partitioned rules (FT and BT) using decision trees (explained in §IV-C). For this, we map each level of a decision tree to a particular header field as shown in Figure 8. The decision tree is then converted to

Rule	sMAC	typEth	proto	dPort	dstIP
1	94:16:3e:3b:41:cf (A1)	0x0800 (C1)	6 (D1)	8899 (F1)	devices.xbcs.net (52.23.33.88) (H1)
3	94:16:3e:3b:41:cf	0x0800	17 (D2)	67 (F2)	*
5	94:16:3e:3b:41:cf	0x0800	17	53 (F3)	*
7	94:16:3e:3b:41:cf	0x0800	6	9207 (F4)	*
11	94:16:3e:3b:41:cf	0x0800	1 (D3)	*	www.google.com (142.250.195.46) (H2)
13	94:16:3e:3b:41:cf	*	2 (D4)	*	224.0.0.2 (H3)

Fig. 6: **Forward traffic (FT) ruleset after header partitioning**

SRAM-based match-action table entries as described in §IV-E. Here, we assign separate SRAM memory stages to each header field as shown in Figure 9. An incoming packet will traverse through the match-action pipeline stage by stage (*e.g.*, sMAC-typEth-proto-dPort-dstIP). That is, for an incoming packet, we can match on a stage only once (*i.e.*, already matched stages cannot be revisited).

This design leads to an inherent inability to handle '$*$' values properly. For example, a packet containing the header field values (sMAC - 94:16:3e:3b:41:cf, typEth - 0x0800, proto - 2 and dstIP - 224.0.0.2) is expected to be matched and allowed according to rule 13 Figure 6. In the decision tree implementation(Figure 8), tracing through the nodes A1: '94:16:3e:3b:41:cf' and C1: '0x0800', we see that there is no node with value '2' (D4) in the level 'proto' in the initial device subtree of Figure 8. That is, the packet takes the path of matching with either of the other rules (1,3,5,7,11) having a common ancestor '94:16:3e:3b:41:cf' and does not find a match for the header field 'proto' with value 2 in the path.

Therefore, the addition of the following rule:

Forward Traffic (FT)

sMAC	typEth	proto	dPort	dstIP	Action
E	E	17	53	*	No
E	E	17	67	*	No
E	E	1	*	*	No
E	E	1	*	E	No
E	*✓	2	*	E	Yes
E	E	6	*✓	*	Yes
E	E	17	*✓	*	Yes
E	E	17	E	*✓	Yes

Backward Traffic (BT)

dMAC	typEth	proto	sPort	srcIP	Action
E	E	17	53	*	No
E	E	17	67	*	No
E	E	1	*	*	No
E	E	1	*	E	No
E	E	6	*✓	*	Yes
E	E	17	*✓	*	Yes

Fig. 7: **Wildcard handling; E: Exact values**

(94:16:3e:3b:41:cf, 0x0800, 2, '∗', 224.0.0.2) is necessary as shown in Figure 8 in dotted red lines labeled 'unfurled rule'. That is, the wildcard '∗' value has been unfurled to include exact values (only '0x0800' in this case) from other rules following the same ancestor (94:16:3e:3b:41:cf). Please note that the height of the decision tree remains constant. Essentially, the implication is that for every such exact value (satisfying the ancestor condition), a new rule has to be added to handle wildcard '∗' values. However, this will lead to an explosion of the number of rules being added to the ruleset. But through careful observation of the resulting FT and BT rulesets of all IoT devices after partitioning, only the following cases of wildcards ('∗') need to be handled for either of them:

- A '∗' occurring in a rule with protocol '6' (TCP) or '17' (UDP): Handle only the earliest occurrence of a '∗'.
- A '∗' occurring in a rule containing protocol value 2: Handle '∗' only in the typEth field since IGMP does not use port numbers.

The rest of the rules with ∗ values need not be handled. This is due to the absence of other rules with common ancestors (*i.e.*, they occur only once in a ruleset). The observations are summarised in Figure 7. Only the columns marked as 'yes' and ticked are required to be substituted with exact values from other rules.

C. Representing ACL rules in decision tree

The partitioning approach described in §IV-A results in two separate rulesets for an IoT device: forward traffic (FT) and backward traffic (BT). They contain 5 header fields each. Further, we unfurled the '∗' values as described in §IV-B and added rules to the existing FT and BT rulesets. Now we represent them as decision trees in order to exploit the repetition of values to save memory.

Tree structure. The tree is built from an explicit 'root' node. An input rule contains a total of 5 header fields. Therefore the absolute height of the tree remains constant at 6 during all operations. While traversing the tree, a node is identified by three parameters: the incoming edge label (also called state),

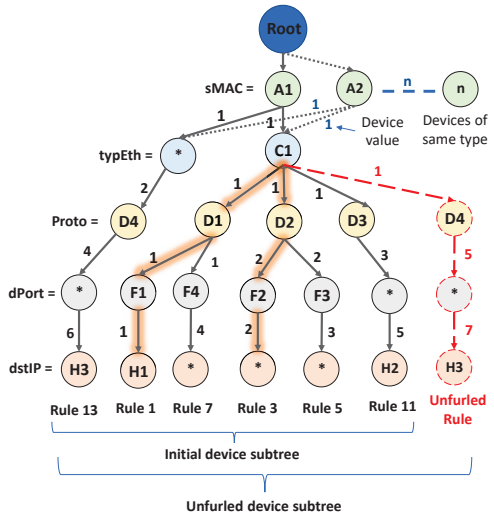


Fig. 8: **Completely constructed decision tree**

header field, and the field value carried in the packet. In essence (state, header field=value) are combined to form a node. We use 3 bytes to store a state value. For example, a value of '94:16:3e:3b:41:cf' under the header field 'sMAC' with state say 1 is represented as (1, sMAC=94:16:3e:3b:41:cf). Also, don't care values ('∗') are treated as individual values (as they are already unfurled). To summarize, the decision tree is constructed by stringing together individual nodes with appropriate edge labels using state values.

Tree construction. Nodes are added to each level by checking if the value of the node to be added is not already present in the path. For example, consider adding rule 1 in Figure 6 to an empty tree. It starts with (sMAC=A1) and ends with the dstIP (1, dstIP=H1) which is straightforward. When the second rule (rule 3) is added, there is a bifurcation from (1, typEth=C1) to (2, dstIP='∗'). It can be inferred that the path till (1, typEth=C1) remains the same for the second rule. Both the rules have been highlighted in orange in Figure 8.

However, the question arises as to which number to assign as edge label (*i.e.*, state). We treat edge labels as unique identifiers (when combined with header field and its value) at a particular level such that these edge labels (or states) drive the per-packet decision (*i.e.*, forward or deny) in the switch data plane. More specifically, the system maintains one state variable for each level to keep track of the current state values. Based on the following two conditions, the value of the per-level state variable is assigned as a label to the edges connecting the rule's nodes.

- If a node addition occurs under a pre-existing path at that level, the existing state value of that level is assigned to the edge. In the running example, while adding the second rule, we add a node (1, proto=D2) indicating that we use the existing state value 1 as the label for the edge connecting C1 and D2.
- If a node addition leads to a new path to be followed at that level, the level's state value is incremented and assigned as a label to the new edge. In the running

example, while adding the second rule (rule 3), we create a node (2, dPort=F2) where state value 2 is assigned to the edge after incrementing the current per-level state value by 1 (when adding rule 1). The same can be observed across the addition of other rules in Figure 6.

D. Memory savings

Header field value repetition. As a direct consequence of decision tree representation, we can observe that the longest common contiguous sub-sequence (taken from the first level) between rules of the same device type is not repeated. After partitioning the ACL rules of 28 IoT devices, we observe that majority of ACL rules pertaining to a single device differ by only a few values towards the leaves. For example, the path from (sMAC=A1) to (1, proto=D2) is common for both rule 3 and 5, thus storing the values only once. We save memory equivalent to the size of fields in the common sequence of every repetition. Even though the addition of state values introduces an overhead of 3 bytes (per-state value), it is more than compensated for by the compression.

Repeating device types. Each IoT device MAC address serves as the root of a device subtree as shown in Figure 8. When a new device of an already added device type (*i.e.*, same MUD profile) joins the network, the existing unfurled device subtree is reused. In Figure 8 the new device A2 is simply connected to the next level. We exploit this property to save memory across repeated device types in the network. That is, the MUD rules of a particular device type is encoded only once in the decision tree and subsequently in the match-action table. More details of memory savings are presented in the evaluation section §V.

E. Decision tree to match-action table entries

The decision tree (*e.g.*, Figure 8) is translated to match-action table entries in the switch pipeline (*e.g.*, Figure 9). State values in the decision tree are addressed by two different names 'currentState' and 'nextState'. In the figures, they are shortened to 'cs' and 'ns' respectively for ease of representation. The former refers to the state value of the current node and the latter to the state value of the node to which the pipeline is driven towards. In the decision tree, the edge label between two nodes is essentially the state value contained in the 'ns' and 'cs' of two adjacent levels. It can be observed that every level contains (currentState, nextState) values except the first and last level. The first level will always contain only the 'nextState' value. Similarly, the last level will only contain the 'currentState' value. The last level's 'nextState' contains an action which in the case of MUD rules (whitelists) is always forward (f).

Each path in the decision tree is strung together with the help of (currentState, nextState) values in the switch match-action pipeline table. All the tables corresponding to each level (except sMAC and proto) in the match-action pipeline are subdivided into 'exact' and 'default' shown in Figure 9. This is because, in the partitioned ruleset, sMAC and proto only contain exact values. The 'exact' table contains values

Rule no.	sMAC_ exact		typEth_ exact			typEth_ default		proto_ exact			dPort_ exact			dPort_ default		dstIP_ exact			dstIP_ default	
	val	ns	cs	val	ns	cs	ns	cs	val	ns	cs	val	ns	cs	ns	cs	val	act	cs	act
1	A1	1	1	C1	1			1	D1	1	1	F1	1			1	H1	f		
3								1	D2	2	2	F2	2						2	f
5											2	F3	3						3	f
7											1	F4	4						4	f
11								1	D3	3				3	5	5	H2	f		
13						1	2	2	D4	4				4	6	6	H3	f		
Unf								1	D4	5				5	7	7	H3	f		
New	A2	1																		

Fig. 9: **Match-action tables split into 'exact' and 'default'**

for exact matches on the pair (currentState, value) whose nextState is followed as the next local action in the pipeline. The 'default' table contains just the (currentState, nextState) essentially directing the pipeline towards the 'nextState' signifying wildcard entry '$*$'. The ingress logic in the switch data plane searches for possible match combinations in the 'exact' table before moving on to the 'default' where only the 'currentState' is matched. If at any stage, a match is not found in either of the tables, the packet is either dropped or sent for further analysis. This completes the one-to-one correspondence between the decision tree and data plane.

Example. Let us consider an ingress packet with header values ['A1', 'C1', 'D4', '$*$', 'H1'] and trace the actions taken in the pipeline. 'A1' is matched in the initial table (sMAC_exact) which leads to 'nextState' (ns) value of 1. The combined values of (1, 'C1') is matched in the ('typEth_exact') table which further drives the packet to the next table ('proto_exact'). This header value (*i.e.*, proto) being a '$*$' can be any value. Thus no matches are found in the ('dPort_exact') table. Thus, the 'nextState' value of '5' is matched in the level's (dPort_default) which leads to the 'nextState' value of 7. This process repeats until the packet either reaches the end action or gets dropped because of no matches in either of the tables.

F. Dynamic properties

A network is a highly dynamic environment with frequent addition and disconnection of devices. Essentially, the proposed approach should accommodate high-frequency additions and deletions of ACL rules from the switch. Since we do not encode the rules of a particular device type more than once, the addition and deletion of new devices of the same type are as trivial as removing a single connecting node at the first level (node A2 in Figure 8). However, adding individual new rules and deleting existing rules require further operations.

Rule addition. When a new IoT device is added to the network, the control plane retrieves its MUD profile and the corresponding ACL rules are added to the decision tree which in-turn updates the match-action table entries in the data plane. However, some scenarios may require the installation of a new rule at runtime (*e.g.*, discovering a new IP address for a particular domain mapping). Algorithm 1 handles the addition of a rule level by level. As soon as a rule is added, its corresponding switch commands to install the equivalent table entries at the switch are also executed.

Rule deletion. Deletion of rules are performed by traversing the tree to the leaf. We then store the path taken and start deleting nodes from the leaf only if there are no other outgoing edges (no children). This flow is suggested in algorithm 2. Simultaneous with the deletion of a node, a switch command is sent to the switch to remove the entry from the table.

Algorithm 1: ACL rule insertion into decision tree

```
Input: ACL Ruleset
Output: (1) decision tree (2) corresponding per node switch command
for each rule r in ACL Ruleset do
    initialize tree_iterator to root_node;
    for each header field 'hval' of r do
        children = child_list(tree_iterator);
        if 'hval' present in 'children' then
            set tree_iterator to matched node;
        else
            edge_label = get_current_level_state_value();
            insert node with hval, edge_label;
            add node to new_nodes[];
        end
    end
    send_switch_commands_from(new_nodes[]);
end
```

Algorithm 2: ACL rule deletion from decision tree

```
Input: (1) decision tree (2) ACL rule to be deleted r
Output: (1) pruned tree (2) corresponding table_delete switch commands
initialize tree_iterator to root_node;
for each header field 'hval' of r do
    children = child_list(tree_iterator);
    if 'hval' present in 'children' then
        set tree_iterator to matched node;
        add matched node to traversed_nodes[];
    end
end
for each 'node' in 'traversed_nodes[]' in reverse order do
    if 'node' has no children then
        delete node;
        send_switch_command(node, edge_label);
    else
        exit the loop
    end
end
```

V. Evaluation

Our approach utilizes a combination of both TCAM (only for local rules) and SRAM to store IoT device MUD ACL rules. Therefore, our main goal for evaluation is: (1) to study the impact of rule unfurling; (2) to study TCAM and SRAM usage as the number of IoT devices in the network increases. (3) to study the impact on switch packet processing latency.

Experiment setup. The experiments were carried out in both the BMV2 switch environment [20] and on Tofino-based switch hardware [16]. A topology with two hosts 'H1', 'H2', with a switch in between is used where 'H1' is the packet generator that simulates IoT devices. The switch decides based on the encoded rules whether to forward the packet to 'H2' or drop.

A. Effect of rule unfurling

In order to emulate the functionality of wildcard '$*$' based MUD ACL rules on SRAM memory, we unfurl the rules. That is, we add extra rules to the existing device ruleset. Figure 10 tabulates the number of MUD ACL rules added after unfurling. We observe that the highest percentage rule added is 62.5% for the device 'samsung smart camera'. A total of 15 rules are added to the original 24. The cause for the mild explosion is the presence of UDP flows with '$*$' values in dPort/dstIP. However, we observe that only 3 devices (Samsung smart cam, TP link camera, and Wemotion) out of 28 exhibit a high percentage (above 45%) of rule addition. Using our approach of rule unfurling, we see that a total of 16 device types do not have any additions in spite of containing rules with '$*$' values.

Device	Original	Added	Total	%
Awair Air Quality	13	0	13	0
Amazon Echo	50	1	51	2
August Doorbell	53	0	53	0
Withings Baby monitor	19	0	19	0
Hellobarbie	11	0	11	0
Belkin Cam	31	9	40	29
BlipcareBP meter	7	1	8	14.3
Canary Cam	25	0	25	0
Chromecast Ultra	157	4	161	2.54
Dropcam	15	0	15	0
Hpprinter	24	2	26	8.33
Hue Bulb	32	1	33	3.12
ihome smart plug	11	1	12	9.09
lifx bulb	12	0	12	0
Nest smoke	99	0	99	0
Netatmo Camera	41	0	41	0
Netatmo Weather Station	7	0	7	0
Pixstar photo frame	7	0	7	0
Ring doorbell	15	0	15	0
Samsung Smart camera	24	15	39	62.5
Withings sleep sensor	10	1	11	10
Smart Things	11	0	11	0
Tplink Cam	22	11	33	50
Tplink switch	27	0	27	0
Triby speaker	37	0	37	0
WeMo switch	25	8	33	32
WeMo motion	22	10	32	45.4
Withings cardio	7	0	7	0

Fig. 10: **Effect of rule unfurling. %: percentage of increase in number of rules**

B. Resource overhead

Scaling ACL rules. To test the scalability, we simulate an environment with a large number of IoT devices. To do so, we use the distribution of actual ACL rules for 28 IoT devices as a base and generate ACL rules for more IoT devices. More specifically, we change the MAC address of each new instance of a device to simulate a new device. However, the device type is decided based on random mapping with one of the 28 IoT devices. That is, when we add a new device, the type is chosen at random from one of the 28 IoT devices. We then assign a random MAC address and add their MUD ACL rules to the decision tree. This way, we generate MUD ACL rules for up to 4480 IoT devices to study the memory overheads.

TCAM overhead. Instead of storing all the MUD ACLs in TCAM, our approach partitions the rules such that only a small subset of rules remain in the TCAM. Therefore, we

No. of Devices	TCAM usage without using decision tree (in KB)	TCAM usage using decision tree (in KB)	SRAM usage for forward rules (in KB)	SRAM usage for backward rules (in KB)	SRAM total (in KB)
28	19.14	0.19	4.41	2.02	6.43
140	89.91	0.84	7.96	4.28	12.24
280	178.37	1.68	9.63	5.93	15.56
560	355.29	3.10	12.91	9.21	22.12
1120	709.14	6.12	19.47	15.78	35.25
1680	1062.99	9.51	26.03	22.34	48.37
2240	1416.84	11.16	32.60	28.9	61.5
2800	1770.68	18.85	39.16	35.46	74.62
3360	2124.53	22.34	45.72	42.03	87.75
4480	2832.23	25.52	58.85	55.15	114

Fig. 11: **TCAM and SRAM overhead**

observe significant savings in the TCAM usage. Each TCAM entry consumes 27 bytes. As seen from columns 2 and 3 of Figure 11, our partition-based decision tree approach saves 99% TCAM memory in comparison to storing all rules in the TCAM (without decision tree). We observe the savings in TCAM primarily due to rule partitioning. We partition the rules with the intent of maximizing the number of rules that are offloaded to SRAM memory. However, we also do so under the constraint of not causing an explosion of rules during the unfurling process (*i.e.*, handling '$*$' values).

SRAM overhead. Our approach uses a decision tree-based encoding which occupies the SRAM memory in the switch data plane. We store the rules for forward traffic and backward traffic in two different switch pipelines governed by separate switch ports. Therefore we partition the MUD ACL rules to either forward traffic (from-device) or backward traffic (to-device) and store them in their respective pipelines using a decision tree approach ('forward pipe' or 'backward pipe').

Additionally, we add an overhead of state variables for each SRAM table entry (*i.e.*, currentState and nextState). We allocate 3 bytes for a state (*i.e.*, 24-bits). Columns 4 and 5 of Figure 11 show the SRAM utilized by the 'forward pipe' and 'backward pipe', respectively. We observe that, in order to store rules of 4480 devices in our system, we only consume 114KB of memory (column 6). For further context, we report that a naive decision tree approach (without partitioning and device type optimization) requires 2.8MB of SRAM memory to store rules of the same 4480 devices. Comparatively, our approach saves 96% of SRAM memory.

C. Switch per-packet latency

We investigate per-packet switch latency on Tofino [16] to answer two key questions: (1) What is the effect of scaling IoT devices?; and (2) How does our approach compare to a purely TCAM-based approach?

First, we consider a single pipe and scale the number of IoT devices from 28 to 0.7 million (only SRAM-based rules). The minimum, average and maximum per-packet latency observed are 364ns, 365ns, and 366ns respectively. Thus, scaling seems to have no impact on per-packet switch latency. Second, a purely TCAM-based approach consumes 290ns per packet, differing from our approach by 75ns.

VI. Related work

Decision tree based ACL rules. [18] elaborates on algorithmic solutions for packet classification based on multiple header fields without relying on TCAM. [19] provides a blueprint for realizing binary decision trees on match-action tables of a P4 switch. This work combines the two approaches and extends them to implement a non-binary decision tree on the switch data plane to subsequently optimize the memory occupied by MUD ACL rules.

IoT MUD profiling. MUD maker [24] and MUDgee [4] can be used to generate MUD files. Such tools are very useful due to the lack of universal vendor support for MUD profiles. Our work focuses on a systematic and scalable design for enforcing MUD profiles in the switch data plane. Our system uses MUD profiles generated by one of these tools.

MUD enforcement using SDN. Recent work [5], [6], [11], [25] proposes techniques to enforce and realize MUD profiles on OpenFlow-based SDN switches. This work assumes MUD-based ACL rules are installed in match-action tables that internally use TCAM, thus do not scale well when there are a large number of IoT devices in the network. In contrast, we focus on realizing MUD in the P4 context. To be specific, we program a P4-based switch data plane with tables and associated exact match entries which can be realized using SRAM, thus scales well for a large network.

MUD enforcement at ISP level. [6] and [26] propose end-to-end system designs for MUD enforcement in home and ISP networks. In comparison, this work focuses on building a scalable data plane primitive that enables such end-to-end designs in practice.

VII. Conclusion

We develop a data plane primitive that scales MUD enforcement for a large number of IoT devices. For this, we exploit the fact that many header fields in MUD ACL rules have few unique but repeating values. We propose a systematic way to partition the MUD ACL ruleset and store them in SRAM memory such that their TCAM utilization is significantly reduced. We subsequently propose a procedural way to represent the rules (bound for SRAM) using a decision tree followed by encoding the decision tree efficiently using multiple SRAM-based exact match-action tables in two separate pipes of the switch data plane. We prototype our system and evaluate the system based on MUD profiles of 28 real IoT devices. Our solution scales to 0.7 million IoT devices per switch using SRAM-based MUD ACL rules.

Acknowledgement

We are very thankful for the technical support extended by Sankalp Mittal, Divya Pathak and Yuvraj Makkena for the hardware-based experiments. The work is supported by the NM-ICPS TiHAN sanctioned by DST, India and SERB ASEAN (CRD/2020/000347) initiative.

REFERENCES

[1] C. Xenofontos, I. Zografopoulos, C. Konstantinou, A. Jolfaei, M. K. Khan, and K.-K. R. Choo, "Consumer, commercial and industrial iot (in) security: Attack taxonomy and case studies," *IEEE IoT*, 2021.

[2] H. Griffioen and C. Doerr, "Examining mirai's battle over the internet of things," in *ACM CCS*, 2020.

[3] (2019) RFC 8520. Accessed: 2021-02-06. [Online]. Available: https://rfc-editor.org/rfc/rfc8520.txt

[4] A. Hamza, D. Ranathunga, H. H. Gharakheili, M. Roughan, and V. Sivaraman, "Clear as mud: Generating, validating and applying iot behavioral profiles," in *Workshop on IoT Security and Privacy*, 2018.

[5] M. Ranganathan *et al.*, "Soft mud: Implementing manufacturer usage descriptions on openflow sdn switches," 2019.

[6] Y. Afek, A. Bremler-Barr, D. Hay, R. Goldschmidt, L. Shafir, G. Avraham, and A. Shalev, "Nfv-based iot security for home networks using mud," in *NOMS IEEE/IFIP*, 2020.

[7] (2015) OpenFlow protocol specification. Accessed: 2021-02-05. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[8] (2018) OpenFlow DataPlane specification. Accessed: 2021-07-05. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[9] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev *et al.*, "B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan," in *ACM SIGCOMM*, 2018.

[10] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *ACM CoNEXT*, 2015.

[11] A. Hamza, H. H. Gharakheili, T. A. Benson, and V. Sivaraman, "Detecting volumetric attacks on lot devices via sdn-based monitoring of mud activity," in *ACM SOSR*, 2019.

[12] A. Agrawal and C. Kim, "Intel tofino2 – a 12.9 tbps p4-programmable ethernet switch," in *IEEE HCS*, 2020.

[13] Y. Yan, A. F. Beldachi, R. Nejabati, and D. Simeonidou, "P4-enabled smart nic: Enabling sliceable and service-driven optical data centres," *Lightwave Technology*, vol. 38, no. 9, pp. 2688–2694, 2020.

[14] (2020) Smart NIC. Accessed: 2021-09-04. [Online]. Available: https://www.netronome.com/products/agilio-cx/

[15] (2018) iotanalytics UNSW 2018 MUD profile. Accessed: 2021-02-04. [Online]. Available: https://iotanalytics.unsw.edu.au/mudprofiles

[16] P4_16 Intel Tofino Native Architecture - Public Version. Accessed: October 2022. [Online]. Available: https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf

[17] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," 2017.

[18] G. Varghese, *Network Algorithmics: an interdisciplinary approach to designing fast networked devices*. Morgan Kaufmann, 2005.

[19] T. Jepsen, A. Fattaholmanan, M. Moshref, N. Foster, A. Carzaniga, and R. Soulé, "Forwarding and routing with packet subscriptions," in *ACM CoNEXT*, 2020.

[20] (2020) BMV2 software switch. Accessed: 2021-07-05. [Online]. Available: https://github.com/p4lang/behavioral-model

[21] 300 Series Managed Switches. Accessed: 2021-05-05. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/

[22] Sx500 Series Stackable Switches. Accessed: 2021-05-05. [Online]. Available: shorturl.at/eflIO

[23] (2018) Belkin camera MUD profile. Accessed: 2021-02-04. [Online]. Available: https://iotanalytics.unsw.edu.au/mud/belkincameraMud.json

[24] MUD Maker tool. Accessed: 2021-04-06. [Online]. Available: https://www.mudmaker.org/

[25] A. Hamza, H. H. Gharakheili, and V. Sivaraman, "Combining mud policies with sdn for iot intrusion detection," in *Workshop on IoT Security and Privacy*, 2018.

[26] H. S. A, H. Kothapalli, S. Lahoti, K. Kataoka, and P. Tammana, "Iot mud enforcement in the edge cloud using programmable switch," in *Proceedings of the ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures*, ser. FFSPIN '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–7. [Online]. Available: https://doi.org/10.1145/3528082.3544832