On the Analysis of MUD-Files' Interactions, Conflicts, and Configuration Requirements Before Deployment

Vafa Andalibi[1[0000−0003−1517−3185]], Eliot Lear[2[0000−0003−2724−0293]], DongInn Kim[1[0000−0001−6331−1410]], and L. Jean Camp[1[0000−0001−8731−7884]]

[1] Indiana University, Bloomington, IN 47408, USA
{vafandal,dikim,ljcamp}@indiana.edu
[2] Cisco Systems, Zurich, Switzerland
lear@cisco.com

Abstract. Manufacturer Usage Description (MUD) is an Internet Engineering Task Force (IETF) standard designed to protect IoT devices and networks by creating an out-of-the-box access control list for an IoT device. Access control list of each device is defined in its MUD-File and may contain possibly hundreds of access control rules. As a result, reading and validating these files is a challenge; and determining how multiple IoT devices interact is difficult for the developer and infeasible for the consumer. To address this we introduce the MUD-Visualizer to provide a visualization of any number of MUD-Files. MUD-Visualizer is designed to enable developers to produce correct MUD-Files by providing format correction, integrating them with other MUD-Files, and identifying conflicts through visualization. MUD-Visualizer is scalable and its core task is to merge and illustrate ACEs for multiple devices; both within and beyond the local area network. MUD-Visualizer is made publicly available and can be found in GitHub.

Keywords: Manufacturer Usage Description · Internet of Things · Network Security · MUD · IoT · Network Microsegmentation · IoT Protection, IoT Security

1 Introduction

The Internet of Things (IoT) has diffused across the globe and the estimates of IoT devices in the home range from billions to tens of billions. Yet, security has lagged [1]. The security of IoT devices is such that they are used to participate in DDoS attacks [18], are vulnerable to ransomware [46], and enable information exfiltration from within homes [7]. Media reports of abusive strangers engaging with families through IoT devices are not uncommon, e.g. [2]. Given the complexity of IoT devices, the lack of technical support, the level of technical expertise in the home, and the complexity of access control, how can these devices be managed?

Manufacturer Usage Description (MUD) is an Internet Engineering Task Force (IETF) standard created in response to the requirements for access control and device isolation for IoT devices [22]. It addresses multiple challenges regarding IoT security by relying on manufacturers providing an Access Control List (ACL) that identifies services and addresses for those services. The goal is to isolate devices; particularly those that cannot be relied upon to provide their own protection. Unlike more traditional verification approaches, MUD can work with devices that have highly limited processing power. In addition, rather than a single entity creating policy, each manufacturer creates the access control that defines the situation for their own devices. The second goal of the MUD standard is to provide an identifier so that updates to devices can be implemented only from authenticated and authorized sources. With such functionality, errors in device configurations can be mitigated.

We have chosen to focus on MUD because, in addition to being an IETF standard, MUD is also a core component of the *National Institute of Standards and Technology (NIST) security for IoT Initiatives*, particularly the thrust focused on stopping DDoS [6]. MUD can defend IoT devices in a home from other compromised ones in the household and on the network, with a specific goal of blocking the access of compromised devices to command and control channels.

One of the core components of the MUD is the MUD-File which is essentially an access control statement. The MUD-File enumerates the allowed (or specifically disallowed) services and sources for these services. In the MUD standard, it is defined as "a file containing YANG-based JSON that describes a Thing and associated suggested specific network behavior" [22]. MUD-Files could possibly be long and complex, making their reading, reviewing, and modification a laborious task if performed manually.

In this paper, we present MUD-Visualizer that addresses this issue. MUD-Visualizer provides 1) protocol checking to avoid formatting errors in the MUD-File, 2) optimization of the MUD-File which identifies internal inconsistencies and inefficiencies, and 3) visualization of the commands in MUD-Files. The first of these prevents coding errors. The second prevents logic errors. The third enables manufacturers and sysadmins to review, validate, and modify the MUD-Files prior deployment. The focus on coding, logic, and contextual errors aligns with the sources of most vulnerabilities [20].

2 Manufacturer Usage Description (MUD)

Understanding the importance of the MUD-Files requires some understanding of the MUD standard. For the readers not familiar with the workflow of MUD a brief summary of MUD workflow and its abstractions is presented in this section. Those familiar with MUD may want to continue to Section 3.

2.1 Components and Workflow

An implementation of MUD has six main components as presented in Fig. 1:

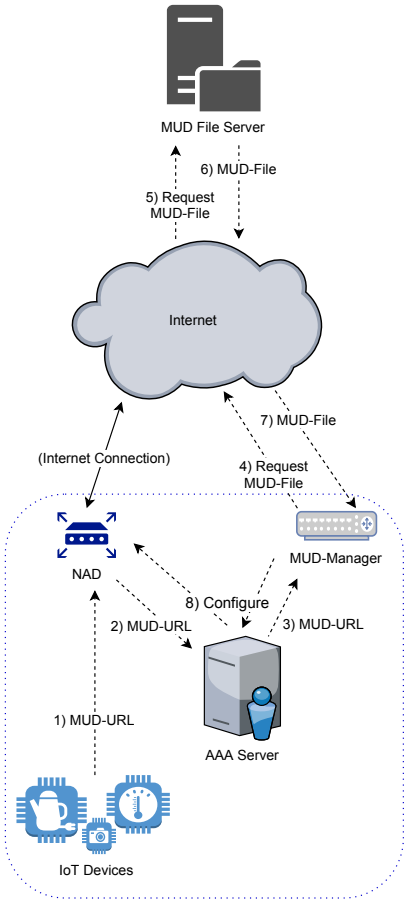

Fig. 1: MUD workflow in a LAN: the blue dotted line indicates the boundary of the LAN

1. *MUD-File:* is a YANG-based JSON file (RFC 7951) created, signed with a public key signature, and distributed by the manufacturer that describes the expected network behavior of the device.
2. *MUD file server:* on which MUD-File is hosted and the location of the file is embedded as a uniform resource locator called MUD-URI.
3. *MUD-URI*: This is used to locate and download the MUD-File to the local network.
4. *AAA Server*: The Authentication, Authorization, and Accounting (AAA) server enforces the traffic rules on the devices in the network. This server can be either an independent server or a built-in component in the *Network Access Device (NAD).*

5. *Network Access Device (NAD)*: acts as a router in the network and is usually equipped with an internal Firewall component which is used by the MUD-Manager via AAA server to control the traffic and enforce rules.
6. *MUD-Manager*: is the core of MUD architecture and is responsible for receiving the MUD-URI from the devices, retrieving the MUD-File from the MUD file server, and communicating the MUD-File rules to the AAA server.

The MUD workflow illustrated in Fig. 1 begins with the IoT device authenticating with X.509 certificate (although DHCP and LLDP are also available as means of authentication) transmitting the MUD-URI embedded in the device to the NAD. MUD-Manager will then receive the MUD-URI, validate the signature, retrieve the MUD-File from the manufacture's MUD file server and then enforce the access control rules of the MUD file to the network via AAA server.

2.2 MUD-File ACL Abstractions

The content of and ACLs in the MUD-Files are instantiated as firewall rules by the MUD-Manager. The ability to implement such rules enables a range of policies. Based on the MUD specifications, the contents of the MUD-File may include a range of default policies and defined types. There are seven approaches to define the behavior and constraints of a device in a MUD instance. These include constraints or identification of domain names, manufacturers, device class, device models, and the local context of the device. These extensions to the IETF-ACL were all addressed in our implementation. Each extension has the potential to simplify the use of MUD for the manufacturer and adopter. Yet the existence of all these options also drives the need for MUD-Visualizer.

A useful abstraction for cloud access is the **domain-name**. Of course, domain names are a global Internet namespace; which is not always well-suited for a specific, geographically located IoT device.

A second abstraction that is defined in the MUD specification is that of the **local-networks**. With local-networks abstraction, a node will be matched against the nodes in the same network. This is particularly useful for designs where there is an IoT hub in the house that is compliant with the local devices.

Another constraining abstraction is that of the **manufacturer**. In this case, the Hostname of the target node would match against the authority component, e.g., domain name, or MUD URL of another node. This constrains the reach of a device to other devices from particular manufacturers.

Beyond the basic domain name for every IoT device, MUD provides the indicator of **same-manufacturer**. In this abstraction, multiple devices, e.g., a house where its lightbulbs or outlets are from the same creator, can communicate only with each other. In operational terms, when devices use this extension the authoritative component will be checked against that of another node.

More than one extension can be used in the same file. Thus there is also an option that identifies a device requiring contextual configuration. The **controller** extension identifies the care where the network administrator needs to assign the target devices to a particular class. This may be particularly useful

when a single device embeds multiple services, and access control depends on these services. For example, a doorbell that offers wireless installation may also offer remote control for the owner, access to a home security firm, face recognition for local control, real-time video activity monitoring, social interactions (e.g., sharing moments), caregiving, and other services which impinge delegating access, network connections, data types, and port numbers.

With **my-controller** abstraction, the device will leverage its MUD-URL and will signal the MUD manager to use whatever mapping it has for this MUD URL to a particular group of hosts that would be used to manage or control this class of device. This mode requires a local decision-maker in the loop, whether that is a human or a process. With this option, the node initiates communication with the MUD-manager with a request that the MUD-Manager assign this node to a class.

Classes of devices and devices from the same manufacturer may be inadequately constraining. The seventh requirement is that model, manufacturer, and class must all match, defined as **model**. The potential for conflicts with this set of seven defined options argues for the importance of the visualizing tool.

3 Motivation

As described above, MUD's role in enforcing the Principle of Least Privilege (PoLP) is to limit the devices' reachability to a bare minimum by leveraging the manufacturers' knowledge about their devices. For a given configuration the ACLs should be validated (if not defined) manually to ensure PoLP, which makes this process prone to human error [27,42]. Errors in the ACLs defined by the manufacturer will result in unwanted access control privilege escalation which poses a security risk to the network.

For instance, consider a MUD-compliant smart slow cooker device that is only allowed to communicate with the manufacturer server and the manufacturer‘s site over the Internet. In addition, the slow cooker comes with a mobile application that allows controlling its functionality both online (via the Internet) and offline (locally). When used offline, the slow cooker will only use port 1300 for communication. Alice is a sysadmin in an enterprise network with more than 100 types of MUD-compliant devices. Bob decides to add the smart slow cooker to the kitchen of the office as a collegial act. A new MUD-File appears in Alice‘s MUD-Manager. To understand all the possible connections, Alice would be required to have an encyclopedic awareness of each MUD-File in the network, and be able to read these files while understanding the interactions.

Given the reality of the enterprise, it is almost impossible to prevent employees from bringing in personal fitness trackers, slow cookers, rice cookers, microwaves, or space heaters all of which can be Internet-connected devices. Because of the number of devices and corresponding ACLs in the network, it's arguably beyond human cognition for Alice to identify the possible unnecessary communication between the slow cooker and the smart bulb which puts the network at risk with only text files [42]. Although there exist other tools which have

the potential to identify the connection between the light bulbs and slow cookers or between the slow cooker and the attacker, MUD-Visualizer is one step ahead and tries to prevent these before they occur.

Imagine instead Alice has the MUD-Visualizer. When each device is added, MUD-Visualizer can help her to implement informed threat modeling and flag unwanted communications. Using MUD-Visualizer Alice could easily identify the connection and isolate the slow cooker from the internal system by adding an additional rule.

Beyond the aforementioned example, there are several points made in previous research studies indicating the importance of different aspects of a tool like MUD-Visualizer. We mention the most important of such works and corresponding points associated with MUD-Visualizer in the next section.

4 Related Work

In this section, we present related work that fits in one of the following categories: studies that focus on MUD research and tools, as well as the researches that focus on the importance of Human Computer Interaction (HCI) in mitigating human errors in regards to access control. Usable access control is a significant challenge, where even the relative responsibilities of the platform, the final user, and the developer are contested, e.g., [36,40].

The tool described here is developed as a complement to the MUD [22]. At the time of writing this paper, there are four main projects that implement the core of a MUD instantiation: the Cisco MUD-Manager [44], the Open Source MUD Manager [47], the MUD implementation of NIST[32], and CableLabs Micronets [31]. NIST has a special publication that thoroughly describes four different builds of MUD based on the above-mentioned implementations for mitigating network-based attacks [6].

Besides the MUD-Manager, Cisco also offers the *mudmaker*[3] which can be used for creating MUD-Files. We used *mudmaker* to generate a comprehensive MUD-File for the tests reported in section 7. MUD Pretty Printer [21] is another tool developed to summarize ACL information on the MUD files. However, it does not provide any User Interface (UI) or visualization. With regard to MUD-File modification, Cisco has a recent patent that discusses techniques for providing secure modification of MUD-Files based on the device applications [23].

Regarding MUD deployment, there are some studies that focus on the effectiveness of MUD against DoS and DDoS attacks [14,29,4,38]. Afek et al. [3] proposed an ISP-level system architecture that enforces the ACL upstream at the provider network to protect the IoT at scale. Additionally, they extended their MUD-interoperable architecture to support peer-to-peer protocols.

With regard to combining Software Defined Networking (SDN) and MUD, the authors in [33] present a scalable implementation of the MUD standard

[3] https://www.mudmaker.org

on OpenFlow-enabled SDN switches. Hamza et al. in [15] attempted to create flow rules based on MUD policies so that they can be enforced using SDN. Matheu et al. [25] employed SDN technique to make MUD model more flexible to support additional aspects such as data privacy, channel protection, and resource authorization. In another work, an SDN-based architecture was proposed to make the process of obtaining and enforcement of MUD policies secure [12]. In a proposed expansion to MUD Matthíasso presented generating contracts and their local evaluation, [26]; this is complimentary to MUD-Visualizer as it assumes the existence of non-conflicting MUD files.

Some researchers focused on helping the manufacturers in the process of creating MUD-Files. In [17] Hamza et al. described *MUDgee* which uses the traffic of an IoT device to generate a MUD profile for that IoT device. Beyond the MUD-Files generated from *mudmaker*, we also performed some dry-run tests with the profiles provided by MUDgee project[4]. NIST also has an ongoing open-source project in their GitHub organization entitled *MUD-PD* [43] which is similarly targeted at profiling IoT devices in order to leverage the MUD architecture.

Feraudo et al. in their systematization of knowledge about MUD identified two challenges with MUD-Files that can be mitigated with the use of the MUD-Visualizer [9]. The first of these is inconsistent implementations; MUD-Visualizer can be used on the underlying files to easily determine developer intent, thus simplifying differences in results. The other is of course consistent generation of MUD-Files. In another survey paper, Mazhar et al. [28] detailed the role of the MUD in the IoT ecosystem, including the implementation, its role in IoT security, and its integration in different security frameworks. They also review the benefits of MUD to the industrial IoT, telecommunication networks, smart home, Fog and Edge computing, and mobile application. MUD-Visualizer can facilitate the deployment of MUD in all of these applications.

In a similar area of research related to configuration verification, Prabhu et al. [30] presented Plankton which is purposed for network configuration verification. In another study, Fogel et al. [10] proposed a high-fidelity declarative model of low-level network configurations and implemented it as a tool called Batfish. Fayaz et al. [8] implemented ERA which can be used for bug detection in reachability policies. With regard to routing, ARC [13] finds the possible impact of routing protocols on the network's data plan by abstracting their mechanics. Beckett et al. in [5] presented Minesweeper which can be used to ensure a wide range of intended properties, e.g., isolation among nodes, in the network. Unlike MUD-Visualizer none of these offer visualizations.

The earliest work in optimizing ACL interaction using HCI principles was by Maxion and Reeder [27]. They examine the Windows XP file-permissions and found that the visualizer tripled the rate of assigned task completion, and reduced errors in those completed tasks by up to 94%. This illustrates the importance of visualization and interaction design in access control.

Similar to MUD-Visualizer Vaniea et al. our work is grounded in the recognition of the difficulty of translating policy rules into access control rules [41].

[4] https://iotanalytics.unsw.edu.au/mudprofiles

We integrated their recommendations into our design, in particular, we integrate visual feedback while the developer is drafting the MUD-File. SPARKLE [41] followed the common visualization process of focusing on a presentation of data in a table, which is the visualization approach most commonly used. For example, Reeder et al. [34] developed an interactive matrix visualization, the Expandable Grid, to enable improved file permissions in Windows XP. This ACL visualization informed the design of MUD-Visualizer particularly in terms of understanding challenges to ease of cognition. In comparison, our approach provides a visualization based on flows more similar to the graph visualization approach by Kolomeets et al [19]; rather than asking developers to read rows or columns.

Salim et al. took a different view examining access control as a case of decision-making under uncertainty [37]. They provided a formal method to quantify how much uncertainty is inherent in the Role-Based Access Control (RBAC), one that illustrates the level of complexity required to provide reliably correct access in an organization.

Xu and colleagues investigated the role of uncertainty in access control decisions [45]. They implemented qualitative investigations into how systems administrators resolve access control conflicts, as these human errors are a known source of security vulnerabilities. Their fundamental finding was that a lack of feedback forced administrations into a trail and error mode. MUD-Visualizer provides real-time visual feedback about changes in access control. This verifies that a need for a high-level view providing information about access requirements and settings in a network. The complexity they documented may be far greater with the expansion of IoT.

Smetters et al. [39] in their study found that limitations in the UI would lead to the reluctance to change the access control settings. This finding applies to MUD deployment as well; it would be simply difficult and time-consuming for system administrators to manually evaluate the interaction between tens of types of MUD-File associated with their IoT device. We believe MUD-Visualizer's UI is significantly easier to use compared to manual analysis and we are going to thoroughly evaluate and show this in our future work.

Besides the lack of UI in the manual analysis of MUD-File, the other issue with manual analysis is processing errors. Liginlal et al. [24] focused on the importance of the analysis errors. They found that mistakes in the information processing stage constitute the most cases of human error-related privacy breach incidents, confirming the importance of MUD-Visualizer in MUD-File interaction analysis.

Another source of user errors is called goal errors, i.e. the failures of users to understand what to do. The main source of goal errors is found to be poor information representation in the interface. A study of highly skilled programmers found that even these participants struggled with access control [35]. This indicates the importance of information representation of MUD-Visualizer compared to other text-based tools like MUD Pretty Printer [21].

From another perspective, the issues with conflicts in MUD-Files are comparable to the challenges in the SDN flow information base (FLIB). None of the

aforementioned studies address the verification of MUD-Files. However, previous work on SDN verification and human subjects research on access control informed the design of the MUD-Visualizer.

The only work that attempts to help the IoT manufacturers and adopters of these devices in process of preparing or deploying MUD profiles is [16]. Their work focuses on different aspects compared to this study in two ways. First, similar to [17,43], they focus on automatic generation of MUD profile based on network traffic. Second, their tool validates the consistency and compatibility of the generated profiles with organizational policies. Because those project generate MUD-Files, code-checking is less of a challenge. Conversely since these products create MUD-Files automatically, logic errors can still be embedded. Their work does not have a visualization or usability component. MUD-Visualizer is a complement to the projects which automatically generate MUD-Files.

To our knowledge, this is the only product targeted at the developers or sysadmins seeking to define or validate a MUD-File for a product to be deployed. MUD-Visualizer is also unique in that it validates interactions and identifies possible conflicts prior to deployment (for the manufacturer) or at the time of deployment (for the user).

5 Methods

Recall from Section 2 that the MUD-File of each IoT device typically contains access controls in the form of a white list. Each list entry contains information about one or more protocols and often a corresponding identifier, e.g., a domain name accessible only via SSH. The white list provides the identifier appropriate for the network layers of the protocol. Entries in this MUD-File list are called the Access Control Entries (ACEs).

The first task of MUD-Visualizer is to determine how the ACE information of different devices interact. We call this process *ACE Merging*. When we merge a set of ACEs of two devices, it is often possible that duplicates appear in the final list of merged protocol information. We address this issue by pruning the protocol stacks that are a subset of more generic protocol stacks. We call this process *ACE Pruning*. Both of these procedures are described in the following subsections.

5.1 ACE Merging

When the abstractions of two devices in the network allows them to communicate, e.g., two devices supporting local network connections, their ACEs should be inspected and merged accordingly. Of course, it is possible that even with matching specifications, two devices should only communicate if there is a common factor between their ACEs. Hence, one of the important tasks of the MUD-Visualizer (specifically *MUD-Network* module described in section 6) is to merge and validate the protocols of ACEs. This task is implemented by moving up the protocol stack and check whether the source protocol (sender) is a subset of the

Algorithm 1 Merging Two Protocol Stacks

```
1: initialize empty protocols stack PS_out
2: procedure MergeProtocolStacks(PS_src, PS_dst)
3:     for each layer l in protocol stack do
4:         for each protocol P_l in layer l do
5:             if P_{l_src} ⊆ P_{l_dst} then
6:                 PS_out ← PS_out + P_{l_src} ∩ P_{l_dst}
7:             end if
8:         end for
9:     end for
10:    if isFullStack(PS_out) then
11:        return PS_out
12:    else
13:        return
14:    end if
15: end procedure
```

destination protocol (receiver) in that layer. If so, the intersection of the protocols is added to the resulted protocol stack. This procedure is implemented in Algorithm 1.

We illustrate an example of this process in Table 1, where simple ACLs for two devices are presented (two ACEs per device). In this table, an intersection between the ACEs of these two devices exists. All possible pairs of ACEs from source and destination devices are checked against one another and the common factors are saved. The protocol stack in this example contains the Transports Layer and Network Layer. The first ACE of the first device is `[IPv4, UDP, any, any]`, representing the network, transport, source port, and destination port respectively. The first ACE of the second device is `[any, any, 5000, 400]`. By merging these two ACEs we find out that these two devices can only communicate if the network layer protocol is IPv4, the transport layer protocol is UDP, and source and destination ports are 5000 and 400, respectively. After a compre-

Table 1: Example of protocol merging between two devices

	Network	Transport	Src Port	Dst Port
Dev1	IPv4	UDP	any	any
	any	TCP	5000	any
Dev2	any	any	5000	400
	IPv6	any	any	8080
Merged	IPv4	UDP	5000	400
	IPv6	TCP	5000	8080
	any	TCP	5000	400

Algorithm 2 Building the ACE Tree

```
1: initialize node as a tree node
2: for each ACE in list of ACEs do
3:     initialize node to root
4:     Updateacetree(ACE, node, n)
5: end for
6: procedure UPDATEACETREE(ACE, node, n)
7:     if node is null then
8:         return
9:     end if
10:    protocols ← GetLayerProtocols(ACE, n)
11:    if Count(protocols) ¿ 1 then
12:        add a wild-card child W_n to node
13:        node ← W_n
14:    else
15:        add a child C_n to node
16:        node ← C_n
17:    end if
18:    ACE ← ACE[1:]
19:    Updateacetree(ACE, node, n-1)
20: end procedure
21: procedure GETLAYERPROTOCOLS(ACE, n)
22:    initialize protocols as an array
23:    for each protocol P in ACE do
24:        if P in layer then
25:            protocols ← protocols + P
26:        end if
27:    end for
28:    return protocols
29: end procedure
```

hensive matching of all possible combinations of the ACEs from both devices, the result is three merged protocols. This result is presented in the 3rd row of Table 1.

5.2 ACE Tree

When merging the protocols of the two MUD-Files that contain many ACEs, redundant protocols are a likely result. For example, suppose two ACEs are merged to `[IPv4, UDP, 400, 600]` and another two ACEs merged to `[IPv4, UDP, any, any]`. The result of the second merge in this example is a super-set of the first protocol, and therefore the first one should be pruned to prevent redundancy and further confusion.

To implement this efficiently for each IoT device, a tree structure was created and associated with each communication destination of that device. Each level of this tree contains information about a layer of the ACE protocol stack. Note that for each layer in TCP/IP model, one could add more than one level in the

Algorithm 3 ACE Pruning

```
1: initialize L, S, C, A_L, A_C to null
2: procedure PruneACETree(PT)
3:     for each leaf L in the Protocol Tree PT do
4:         for each L's sibling S in the Protocol Tree PT do
5:             if L ⊆ C then
6:                 Prune(L)
7:                 continue to the next leaf L
8:             end if
9:         end for
10:        for each L's cousin C in the Protocol Tree PT do
11:            if L ⊆ C then
12:                n ← 1
13:                A_L ← nthAncestor(L, n)
14:                A_C ← nthAncestor(C, n)
15:                while A_L is not null and A_L <> A_C do
16:                    if A_L ⊂ A_C then
17:                        Prune(L)
18:                        continue to the next leaf L
19:                    end if
20:                    n ← n + 1
21:                    A_L ← nthAncestor(L, n)
22:                    A_C ← nthAncestor(C, n)
23:                end while
24:            end if
25:        end for
26:    end for
27: end procedure
28: procedure nthAncestor(Node,n)
29:    initialize A_Node to null
30:    c ← 0
31:    while c < n do
32:        A_Node ← Parent(Node)
33:        c ← c + 1
34:    end while
35:    return A_Node
36: end procedure
```

tree as we will present in our next example. Moreover, at each level, we have added a wildcard node in case the communication is allowed through multiple protocols in that particular layer. The recursive implementation of this procedure is presented in Algorithm 2.

As an example for Algorithm 2, we present a ACE Tree built from the set of ACEs presented in the *Original* row in Table 2. In this example, the protocol stack has simply two layers: Network Layer and Transport layer. However, as you can see, we have more than one level in the tree associated with the Transport layer, i.e. transmission protocols (TCP/UDP) and ports.

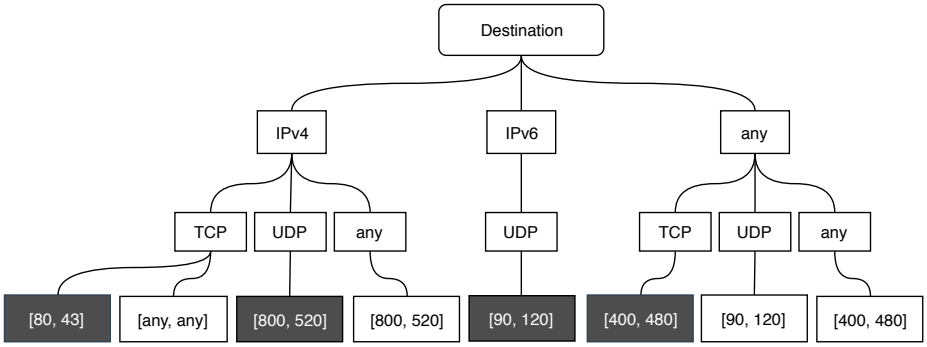

Fig. 2: The tree structure containing the protocol information for a particular destination. The first children are the network protocols, the second children are the transport protocols and the leaves are pairs of source and destination ports. The nodes colored as dark gray could be removed as a super-set of them already exists in the tree.

In simple words, for each ACE, the algorithm does as follows: it starts from the lowest layer (in this case Transport layer), gets the protocols associated with that layer, and if they are more than one, it adds a wild card to that level of the tree. The ACE Tree containing the information of ACEs presented in the *Original* row of Table 2 is presented in Fig. 2.

Table 2: Result of the root to leaf tree traversal from the trees shown in Fig. 2 (Original) and Fig. 3 (Pruned)

	Network	Transport	Src Port	Dst Port
Original	IPv4	TCP	80	43
	IPv4	TCP	any	any
	IPv4	UDP	800	520
	IPv4	any	800	520
	IPv6	UDP	90	120
	any	TCP	400	480
	any	UDP	90	120
	any	any	400	480
Pruned	IPv4	TCP	any	any
	IPv4	any	800	520
	any	UDP	90	120
	any	any	400	480

Pruning ACE Tree In this section, we describe how the ACE Tree is pruned. An example is provided for each of the scenarios where all the examples are based on the ACLs provided in *Original* row in the Table 2 which are depicted as a ACE Tree in Fig. 2. Consider the following notations:

- L: denoting one of the leaves of the ACE Tree
- S_i: denoting ith sibling of the leaf L
- C_i: denoting ith cousin of the leaf L
- $A_n(L)$: denoting nth ancestor of the leaf L
- $A_n(C_i)$: denoting nth ancestor of cousin node C_i

Each leaf node L in the ACE Tree can be pruned if it satisfies one of the following conditions:

- Consider L has a sibling S_i and $L \subset S_i$. In this case, L can be pruned with no futher conditions
  Example: Consider the leaf `[80,43]` and its sibling `[any,any]`. As can be seen, `[80,43]`$\subset$ `[any,any]`, hence `[80,43]` can be pruned. Note that the upwards tree traversal stops without any outcome when either we reach to the `root` or when $A_n(L) = A_n(C_i)$.
- Consider L has a cousin C_i where $L \subseteq C_i$ and we traverse the tree starting from both L and C_i simultaneously up towards the root of the tree. L then could be pruned if at any point during traversal $A_n(L)$ becomes a sibling of $A_n(C_i)$ and $A_n(L) \subset A_n(C_i)$.
  Example: Consider the fifth leaf of the tree (counting from left to right) `[90, 120]` and its sixth cousin C_6 = `[90, 120]` which is the seventh leaf of the tree. As can be seen, `[90, 120]`$\subseteq$`[90, 120]` indicates that the first condition holds, i.e. $L \subseteq C_i$. As we traverse the tree towards the root by visiting the ancestors of each of the two target nodes, we see that $A_2(L)$, i.e. `IPv6` node, is a sibling of $A_2(C_i)$, i.e. `any` node, and `IPv6`$\subset$ `any` indicating that $A_2(L) \subset A_2(C_i)$. Hence, the fifth leaf `[90, 120]` could be pruned.
  Another example in this case would be the third leaf `[800, 520]` and its third cousin C_3, i.e. the fourth leaf, `[800, 520]`. Since `[800, 520]`$\subseteq$`[800, 520]` and their first ancestors, i.e. parents, are siblings and $A_1(L) \subset A_1(C_3)$, therefore the third leaf `[800, 520]` can be pruned. We illustrate the pruned version of ACE Tree of Fig. 2 in Fig. 3.

6 Implementation

We implemented MUD-Visualizer in JavaScript for two main reasons: the prevalent visualization libraries and enormous visualization capabilities in JavaScript, and the possibility of creating both a stand-alone application and a web application with minimum changes to the codebase.

The UML diagram of the MUD-Visualizer is presented in Fig. 4. As shown, the D3 library[5] is a vital component to visualizing the MUD-Files in MUD-Visualizer. The main function of MUD-Visualizer is to provide the appropriate

[5] https://d3js.org/

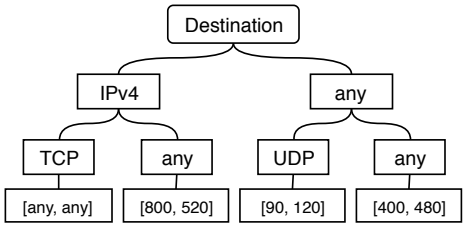

Fig. 3: Result of the protocol pruning

data to the D3 library. There are four main internal components of the MUD-Visualizer: MUD-File processor, visualization data generator, rendering engine, and standalone extension.

The **MUD-File Processor** initially parses the MUD-Files and extracts the data needed for the identification of possible flows. This information includes the MUD-URL, manufacturer, incoming and outgoing ACE, existence of my-controller or controller nodes, and associated data defined for the seven optional fields in Section 2.2. Using this data the connection between the instances of the MUD-Files are analyzed. This includes whether or not two nodes should connect and if so, what are the assumptions about the protocols allowed between them. Note that the rules for each extension in Section 2.2 might be different and are kept separately for further analysis in the MUD-File Processor component. In the case of a my-controller node, the required *promises*[11] are also saved so that they can be fulfilled by the user later on. Finally, the protocols are stored

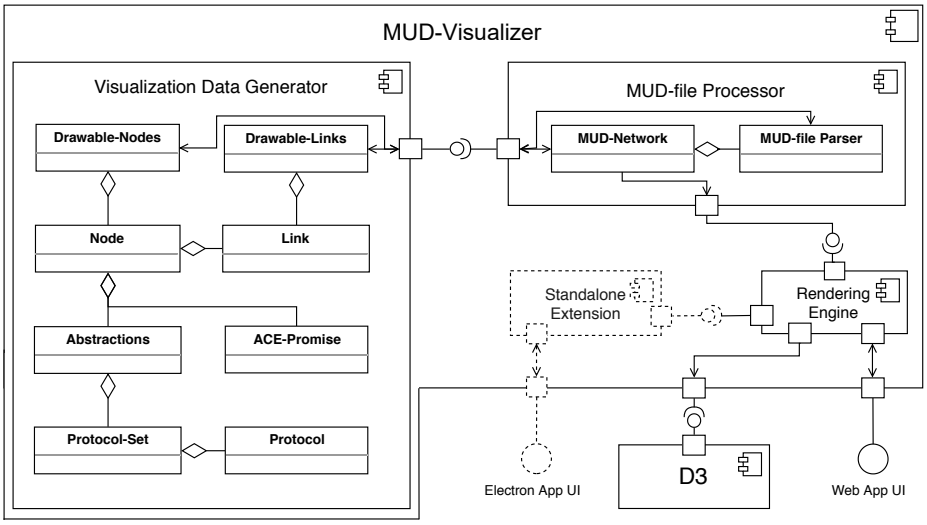

Fig. 4: UML Diagram of the MUD-Visualizer. The dotted line used for the Electron UI is merely for the two UIs, i.e. Web and Electron, to be distinguishable.

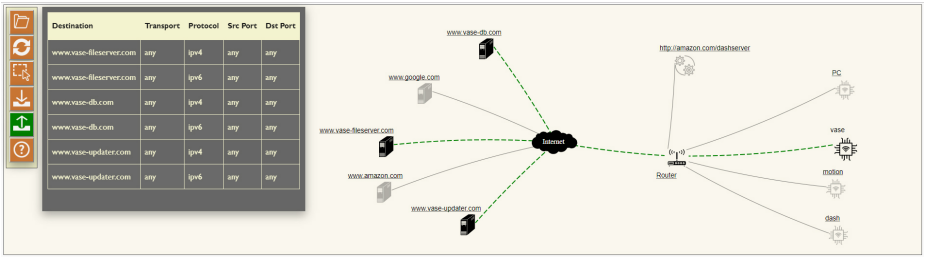

Fig. 5: Screenshot of the UI of the MUD-Visualizer

and merged in a way to minimize the redundant information, as was previously described in the sub-section entitled ACE Merging. Moreover, further information is requested from the user if needed, e.g., the configuration of my-controller nodes. This step of MUD-Visualizer requires that the developer explicitly recognize their assumptions about the user or contextual knowledge required for configuration.

The **visualization data generator** converts the data extracted from MUD-Files into structures for the following visualization. These structures include nodes, links, and direction of the links as well as the corresponding protocols and rules that are assigned to the MUD-Files. For instance a simple MUD-File describing an IoT device with a domain-name abstraction that is only allowed to communicate with a remote server would generate several nodes and links including the IoT device and the remote server. In contrast, a device that should only communicate internally might seek to connect to a device that is constrained to only connect to a manufacturer. The concepts previously described in Methods section including building the ACE Tree in 5.2 and ACE Tree pruning in 5.2 are both implemented in the *Abstractions* module of this component.

The **rendering engine** is the component that combines all data generated by the other components and creates the final visualization. This component has several responsibilities, including interfacing with MUD-File Processor, the visualization libraries (i.e., D3), and also the Web App UI. If the application is running as a stand-alone app, it also communicates with the following component.

The extension, called **Standalone Extension** is not used for the MUD-Visualizer web app. It enables a stand-alone version of the MUD-Visualizer. It consists of the components above, calls for those components, and the main script for the Electron UI application.

7 Results

The UI of the MUD-Visualizer is shown in Fig. 5. This screenshot is consistent for both the stand-alone version and web app version of the tool.

Given that the Electron framework also supports the DevTools, we used the Chrome Performance Analysis tool available as part of the Devtools for

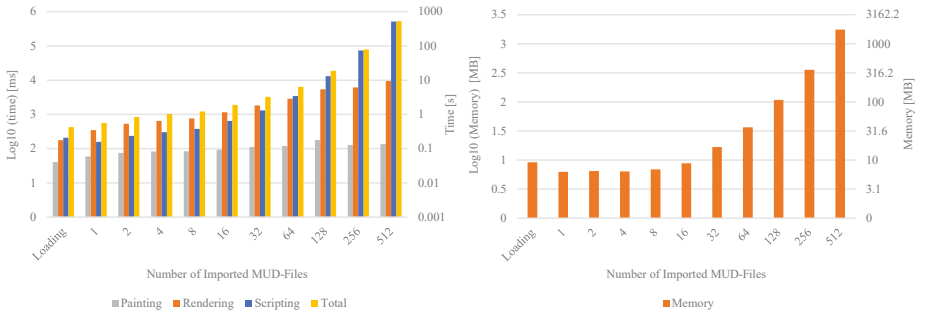

Fig. 6: Performance evaluation of MUD-Visualizer for first-time loading as well as importing 1 to 512 MUD-Files. Left: individual and total runtime for Painting, Rendering, and Scripting. Right: peak Javascript heap memory usage. Both charts are presented in logarithmic scale with the primary axis indicating the logarithmic values and secondary axis indicating actual values in *seconds* and *MegaBytes* respectively. The maximum runtime is for loading 512 MUD-File which is equal to 526 seconds and the corresponding memory usage is 1754 MB.

benchmarking the web application version of the MUD-Visualizer. We considered the time spent for *Scripting*, *Rendering*, and *Painting* as well as peak Javascript heap memory usage. Our experiments were run on a Macbook Pro late 2013 computer with 2.6GHz Quad-Core Intel Core i7, 16GB 1600 MHz DDR3 RAM, and $2880x1800$ pixels of screen space.

The benchmark results are presented in Fig. 6. We evaluated the performance of MUD-Visualizer when loaded for the first time, i.e. indicated as *Loading* in the figure, as well as when we import MUD-Files. The number of MUD-Files that was used in the benchmark ranged from 1 to 512. To evaluate the scalability of MUD-Visualizer, we used copies of a relatively heavy MUD-File created by *mudmaker* which includes five out of seven implemented abstractions, i.e. all abstractions except *my-controller* and *model*. Please recall that *my-controller* requires end-user to manually enter data about their selected point of control. Had we included *my-controller* the results would have been dominated by user response time. Examining user interaction is a component of our future work. Also, the *model* abstraction would result in all copies of the sample MUD-File to communicate with each other in a local network. Therefore, we decided not to include that to make the benchmarking more rigorous by letting the MUD-Visualizer process all other abstractions.

Note that in a real-world setting, although the enterprises might have thousands of IoT devices in their networks, the *type* of devices in their network is significantly lower than that. For instance, a hospital that has 2k MUD-compliant smart bulbs in its network will have bulbs of a few brands and types. In this case, if a hospital system administrator tries to use MUD-Visualizer, they will not need to load 2k MUD-files but rather a handful of them. In other words,

our benchmark shows the scalability of MUD-Visualizer with regard to the *type* of MUD-Files not the number of MUD-compliant devices in the network. The threat model and risk posture of enterprise will determine if new devices should be manually added, if MUD-Visualizer should interact with automated detection and identification, or if there should be some combination of these strategies.

The runtime benchmark data in Fig. 6 indicates that the total time gets very close to scripting time as the number of MUD-Files increases. This is because for each MUD-File that is newly imported, its relation and interaction with other MUD-Files with respect to all MUD abstractions should be analyzed and processed. The maximum loading time is for 512 MUD-Files which is slightly longer than 10 minutes and the memory that the application needs exceeds slightly higher than 1754 MB. Be advised that this benchmark is performed rigorously and even enterprise networks barely have 512 *types* of MUD-Files in their network each being as heavy as the one we used in this benchmark. Moreover, verification of MUD-File using the MUD-Visualizer is not performed on short intervals and is done only when, for instance, a new device is introduced.

8 Conclusions and Future Work

In this work, we described a tool for visualizing the interaction of MUD-Files which also explicitly identifies any information required by the use of the controller options. The challenge in visualizing the MUD-files is the way they interact with each other and how the ACL of the devices affects the communication between them. We presented methods for addressing this challenge and implemented MUD-Visualizer and made it open-source and publicly available on GitHub. The main purpose of MUD-Visualizer is to facilitate the review and validation phase of MUD deployment for developers, engineers, and system administrators. We also performed a benchmark for runtime and memory consumption of the tool and showed that it can be used on a personal computer to load hundreds of MUD-File for evaluation.

Our future work includes several phases. First, a user study in which we examine the practicality of the MUD-Visualizer and the extent to which it can help the target audience. Second, there are a few points in the tool that we are particularly interested to improve, including visualizing the controllers' MUD-File, support for including network configuration in the visualization, e.g., the IP address of IoT devices and the corresponding controllers, DHCP configuration, etc. Third, we want to test the functionality that facilitates the local modification of the MUD-Files without creating the opportunity for an attacker to move around these. Essentially our goal is to allow decreased but not increased connectivity. Finally, we want to implement the abstraction analysis of MUD-Visualizer in parallel to improve the performance and scalability even more.

Availability

MUD-Visualizer is made publicly available in GitHub at https://github.com/iot-onboarding/mud-visualizer and can be used both as a stand-alone tool and as a tool integrated into web apps.

Acknowledgments

This research was supported in part by the National Science Foundation awards CNS 1565375 and CNS 1814518, as well as the grant #H8230-19-1-0310, Cisco Research Support, Google Research, and the Comcast Innovation Fund. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Cisco, Comcast, Google, nor Indiana University.

References

1. State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating. [Online]. Available on: `https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b` (Aug 2018)
2. Ring security camera hacks see homeowners subjected to racial abuse, ransom demands. [Online]. Available on: `https://abcnews.go.com/US/ring-security-camera-hacks-homeowners-subjected-racial-abuse/story?id=67679790` (Dec 2019)
3. Afek, Y., Bremler-Barr, A., Hay, D., Goldschmidt, R., Shafir, L., Abraham, G., Shalev, A.: NFV-based IoT Security for Home Networks using MUD. arXiv preprint arXiv:1911.00253 (2019)
4. Andalibi, V., Kim, D., Camp, L.J.: Throwing MUD into the FOG: Defending IoT and Fog by expanding MUD to Fog network. In: 2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19) (2019)
5. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. pp. 155–168 (2017)
6. Dodson, D., Polk, W., Souppaya, M., Barker, W., Lear, E., Weis, B., Fashina, Y., Grayeli, P., Klosterman, J., Mulugeta, B., et al.: Securing Small Business and Home Internet of Things (IoT) Devices: Mitigating Network-Based Attacks Using Manufacturer Usage Description (MUD). Tech. rep., National Institute of Standards and Technology (2019)
7. D'Orazio, C.J., Choo, K.K.R., Yang, L.T.: Data exfiltration from Internet of Things devices: iOS devices as case studies. IEEE Internet of Things Journal **4**(2), 524–535 (2016)
8. Fayaz, S.K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., Varghese, G.: Efficient network reachability analysis using a succinct control plane representation. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). pp. 217–232 (2016)
9. Feraudo, A., Yadav, P., Mortier, R., Bellavista, P., Crowcroft, J.: Sok: Beyond iot mud deployments–challenges and future directions. arXiv preprint arXiv:2004.08003 (2020)

10. Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., Millstein, T.: A general approach to network configuration analysis. In: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15). pp. 469–483 (2015)
11. Friedman, D.P., Wise, D.S.: The impact of applicative programming on multiprocessing. Indiana University, Computer Science Department (1976)
12. García, S.N.M., Molina Zarca, A., Hernández-Ramos, J.L., Bernabé, J.B., Gómez, A.S.: Enforcing Behavioral Profiles through Software-Defined Networks in the Industrial Internet of Things. Applied Sciences **9**(21), 4576 (2019)
13. Gember-Jacobson, A., Viswanathan, R., Akella, A., Mahajan, R.: Fast control plane analysis using an abstract representation. In: Proceedings of the 2016 ACM SIGCOMM Conference. pp. 300–313 (2016)
14. Hamza, A., Gharakheili, H.H., Benson, T.A., Sivaraman, V.: Detecting volumetric attacks on IoT devices via SDN-based monitoring of MUD activity. In: Proceedings of the 2019 ACM Symposium on SDN Research. pp. 36–48 (2019)
15. Hamza, A., Gharakheili, H.H., Sivaraman, V.: Combining MUD policies with SDN for IoT intrusion detection. In: Proceedings of the 2018 Workshop on IoT Security and Privacy. pp. 1–7 (2018)
16. Hamza, A., Ranathunga, D., Gharakheili, H.H., Benson, T.A., Roughan, M., Sivaraman, V.: Verifying and monitoring IoTs network behavior using MUD profiles. arXiv preprint arXiv:1902.02484 (2019)
17. Hamza, A., Ranathunga, D., Gharakheili, H.H., Roughan, M., Sivaraman, V.: Clear as MUD: generating, validating and applying IoT behavioral profiles. In: Proceedings of the 2018 Workshop on IoT Security and Privacy. pp. 8–14. ACM (2018)
18. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: DDoS in the IoT: Mirai and other botnets. Computer **50**(7), 80–84 (2017)
19. Kolomeets, M., Chechulin, A., Kotenko, I., Saenko, I.: Access control visualization using triangular matrices. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). pp. 348–355 (2019). https://doi.org/10.1109/EMPDP.2019.8671578
20. Landwehr, C.E., Bull, A.R., McDermott, J.P., Choi, W.S.: A taxonomy of computer program security flaws. ACM Computing Surveys (CSUR) **26**(3), 211–254 (1994)
21. Lear, E.: MUD Pretty Printer. [Online]. Available on: `https://github.com/iot-onboarding/mudpp` (2020)
22. Lear, E., Droms, R., Romascanu, D.: Manufacturer Usage Description Specification. RFC 8520 (Mar 2019). https://doi.org/10.17487/RFC8520, `https://rfc-editor.org/rfc/rfc8520.txt`
23. Lear, E., Steck, C.S., Weis, B.: Secure modification of manufacturer usage description files based on device applications (Oct 17 2019), uS Patent App. 15/954,875
24. Liginlal, D., Sim, I., Khansa, L.: How significant is human error as a cause of privacy breaches? an empirical study and a framework for error management. computers & security **28**(3-4), 215–228 (2009)
25. Matheu, S.N., Robles Enciso, A., Molina Zarca, A., Garcia-Carrillo, D., Hernández-Ramos, J.L., Bernal Bernabe, J., Skarmeta, A.F.: Security Architecture for Defining and Enforcing Security Profiles in DLT/SDN-Based IoT Systems. Sensors **20**(7), 1882 (2020)
26. Matthíasson, G., Giaretta, A., Dragoni, N.: Iot device profiling: From mud files to s× c contracts. Open Identity Summit 2020 (2020)

27. Maxion, R.A., Reeder, R.W.: Improving user-interface dependability through mitigation of human error. International Journal of human-computer studies **63**(1-2), 25–50 (2005)
28. Mazhar, N., Salleh, R., Zeeshan, M., Hameed, M.M.: Role of device identification and manufacturer usage description in iot security: A survey. IEEE Access **9**, 41757–41786 (2021)
29. Polk, W., Souppaya, M., Haag, W., Barker, W.: [Project Description] Mitigating IoT-based Distributed Denial of Service (DDOS). Tech. rep., National Institute of Standards and Technology (2017)
30. Prabhu, S., Chou, K.Y., Kheradmand, A., Godfrey, B., Caesar, M.: Plankton: Scalable network configuration verification through model checking pp. 953–967 (2020)
31. Pratt, C.: micronets Manufacturer Usage Description (MUD) tools. [Online]. Available on: `https://github.com/cablelabs/micronets-mud-tools` (2019)
32. Ranganathan, M.: Openflow SDN Manufacturer Usage Description (MUD) Server implementation on OpenDaylight Nitrogen Release. [Online]. Available on: `https://github.com/usnistgov/nist-mud` (2018)
33. Ranganathan, M., Montgomery, D., El Mimouni, O.: Implementing Manufacturer Usage Descriptions on OpenFlow SDN Switches
34. Reeder, R.W., Bauer, L., Cranor, L.F., Reiter, M.K., Bacon, K., How, K., Strong, H.: Expandable grids for visualizing and authoring computer security policies. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 1473–1482 (2008)
35. Reeder, R.W., Maxion, R.A.: User interface dependability through goal-error prevention. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). pp. 60–69. IEEE (2005)
36. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C.: User-driven access control: Rethinking permission granting in modern operating systems. In: 2012 IEEE Symposium on Security and Privacy. pp. 224–238. IEEE (2012)
37. Salim, F., Reid, J., Dawson, E., Dulleck, U.: An approach to access control under uncertainty. In: 2011 Sixth International Conference on Availability, Reliability and Security. pp. 1–8. IEEE (2011)
38. Schutijser, C.: Towards automated DDoS abuse protection using MUD device profiles. Master's thesis, University of Twente (2018)
39. Smetters, D.K., Good, N.: How users use access control. In: Proceedings of the 5th Symposium on Usable Privacy and Security. pp. 1–12 (2009)
40. Tahaei, M., Vaniea, K.: "developers are responsible": What ad networks tell developers about privacy. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI'21 Extended Abstracts). pp. 1–12 (2021)
41. Vaniea, K., Karat, C.M., Gross, J.B., Karat, J., Brodie, C.: Evaluating assistance of natural language policy authoring. In: Proceedings of the 4th symposium on Usable privacy and security. pp. 65–73 (2008)
42. Wang, M.: Accessible access control: A visualization system for access control policy management (2019)
43. Watrobski, P.: A tool for characterizing the network behavior of IoT devices. [Online]. Available on: `https://github.com/usnistgov/MUD-PD` (2019)
44. Weis, B.: MUD-Manager Version 3.0. [Online]. Available on: `https://github.com/CiscoDevNet/MUD-Manager` (2018)

45. Xu, T., Naing, H.M., Lu, L., Zhou, Y.: How do system administrators resolve access-denied issues in the real world? In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems. pp. 348–361 (2017)
46. Yaqoob, I., Ahmed, E., ur Rehman, M.H., Ahmed, A.I.A., Al-garadi, M.A., Imran, M., Guizani, M.: The rise of ransomware and emerging security challenges in the Internet of Things. Computer Networks **129**, 444–458 (2017)
47. Yeich, K.: osMUD- Open Source MUD Manager. [Online]. Available on: `https://github.com/osmud/osmud` (2019)