Towards Securing Duplicate Address Detection Using P4

Lin He, Peng Kuang, Ying Liu, Gang Ren, Jiahai Yang
Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*—Duplicate Address Detection (DAD) is one of the functions of the Neighbor Discovery Protocol (NDP), which determines whether the IPv6 address of a node conflicts with those of other nodes. However, due to the lack of verification of NDP messages, DAD is vulnerable to Denial of Service (DoS) attacks. Existing solutions suffer from high complexity and low security, need to modify the NDP, or have a single point of failure, which renders them infeasible to be deployed.**

To solve the above problems, we propose P4DAD, which is a secure DAD mechanism based on P4. By creating and maintaining a binding entry between an IPv6 address and a link-layer property of a host's network attachment, P4DAD can filter spoofed NDP messages in an in-network manner to prevent DoS attacks on DAD without modification to the NDP or host stack. We implement a prototype of P4DAD and evaluate it in terms of functionality, performance, and scalability. Evaluation results show that P4DAD can prevent DoS attacks on DAD successfully with negligible overhead and has satisfactory scalability.

***Index Terms*—Duplicate Address Detection, P4, DoS Attack, Neighbor Discovery Protocol, IPv6**

I. Introduction

Duplicate Address Detection (DAD) is a crucial procedure in the process of IPv6 address configuration [1]. It allows all nodes located on the same subnet to communicate and to join the network with unique IPv6 addresses. The DAD procedure is based on two Neighbor Discovery Protocol (NDP) [2] messages, namely, Neighbor Solicitation (NS) and Neighbor Advertisement (NA). In DAD, a node sends one or more NS messages, and it will configure the target address only if no NA messages from existing nodes are received. Original DAD assumes that all nodes on a subnet are trustworthy. However, this assumption does not hold in the real network environment. If a malicious node launches Denial of Service (DoS) attacks by replying to a victim's NS messages with spoofed NA messages continuously, the victim can not configure any IPv6 addresses.

The above problem is severe due to its denial of a node's access to the network. It will become more severe in the future because around 50 billion of the Internet of Things (IoT) devices will connect to the Internet by 2030 [3]. These IoT devices tend to use IPv6 addresses for communication [4], [5], and they still need to use DAD to determine the uniqueness of IPv6 addresses [6].

To address this issue, many solutions have been proposed, which can be classified into three categories: target address hiding, NDP message authenticating, and uniform NA message replying. The first category is to use hash encryption or other mechanisms to hide target addresses in NS messages from unintended recipients. Malicious hosts cannot spoof NA messages effectively without knowing target addresses in NS messages. The second category is to append a message authentication code calculated by a hash function to each NDP message, which makes it possible for a host to distinguish valid messages from fake ones. The third category is to introduce a central node to reply with NA messages uniformly when receiving NS messages. Any NA messages sent by other nodes (including attackers) will be neglected. However, the first two solutions require the update of the NDP, which results in a large-scale modification to host network stack across different manufacturers and operating systems. The third solution always suffers a single point of failure, not to mention the lack of the design of address state update.

In this paper, we ask the question: *can we solve DoS attacks on DAD in an in-network manner without any modification to the existing standardized protocols or network architecture?* We are attracted to this solution because it neither requires modification to the host network stack nor introduces a central controller to change the communication mode.

Recently, there has been a significant effort towards making network programmable, including data plane programming languages (e.g., P4 [7]) and the corresponding programmable targets (e.g., Barefoot Tofino [8]), which enables unprecedented network flexibility, promising an ever-evolving set of network functionalities at hardware speeds [9]–[11]. Inspired by this, we propose P4DAD, which can secure DAD in an in-network manner by leveraging P4. The core idea of P4DAD is to filter bogus NA messages of which the target addresses do not belong to the sender of NA messages.

P4DAD first uses P4 to monitor the address assignment procedure to create binding entries between an IPv6 address and a link-layer property of the host's network attachment (e.g., switch port). Then, P4DAD filters NA packets using binding entries to ensure the source address and target address of each NA packet are the same and belong to innocent hosts. P4DAD neither modifies the NDP nor requires a dependency on supportive functionality on hosts, which renders it widely deployable. Moreover, as P4DAD does not introduce cryptographic encryption computation in the host and brings small processing overhead in the devices, it is lightweight. Finally, P4DAD can be deployed on P4-enabled devices directly without the need for a central node, which makes it avoid single points of failure.

In summary, the main contributions of this paper are as follows:

- We conduct experiments on different operating systems and address assignment policies to show that DoS attacks on DAD are still very feasible and make hosts without network accessibility.
- We propose P4DAD, which is a lightweight, robust, and deployable DAD mechanism that can secure DAD in an in-network manner without any modification to the NDP or host network stack.
- We implement a P4DAD prototype, and its evaluation results show that P4DAD can prevent DoS attacks on DAD in a lightweight, robust, and deployable manner.

The remainder of the paper is organized as follows: The related work is reviewed in Section II. Section III describes the problem formulation of P4DAD. The details of P4DAD design are described in Section IV. Section V presents the implementation and evaluation of P4DAD. Security analysis is conducted in Section VI. The discussion is described in Section VII. Finally, Section VIII concludes the paper.

II. Related Work

In this section, we start with a brief introduction of DoS attacks on DAD. Then, we present experimental results of how mainstream operating systems tackle DoS attacks on DAD as well as the influence of address assignment mechanisms on them. Next, we compare existing solutions and analyze their limitations. Finally, we give a brief introduction of P4 and explain why P4 is used for securing DAD.

A. DoS Attacks on DAD

DAD is used by a node to determine whether or not an address it wishes to use is already in use by another node. DAD must be performed on all unicast addresses before assigning them to an interface, regardless of whether they are obtained through manual configuration, Stateless Address Autoconfiguration (SLAAC) [1], or Dynamic Host Configuration Protocol for IPv6 (DHCPv6) [12]. As shown in Table I, in DAD, the source address of an NS message is unspecified; the destination address is a solicited-node multicast address where the NS message will be multicast; the target address in an NS message is the tentative address which a node (A) wants to configure. When the target address in the NS message is a duplicate, another node (B) with this address will reply with an NA message. Different from the NS message, the source address in the NA message is the interface address of B; the destination address is an all-node multicast address, which means the NA message will be multicast to all nodes on the same link[1]; the target address in the NA message is the IPv6 address on which A performs duplicate detection.

During the procedure of DAD, a host first multicasts NS messages to verify the uniqueness of an IPv6 address, which it tries to configure. Generally, another host will reply with an NA message when it discovers the IPv6 address is a duplicate.

[1]In this paper, the term “link” refers to a topological area bounded by routers that decrement the IPv4 TTL or IPv6 Hop Limit when forwarding the packet, which is the same as the definition in RFC 4903 [13].

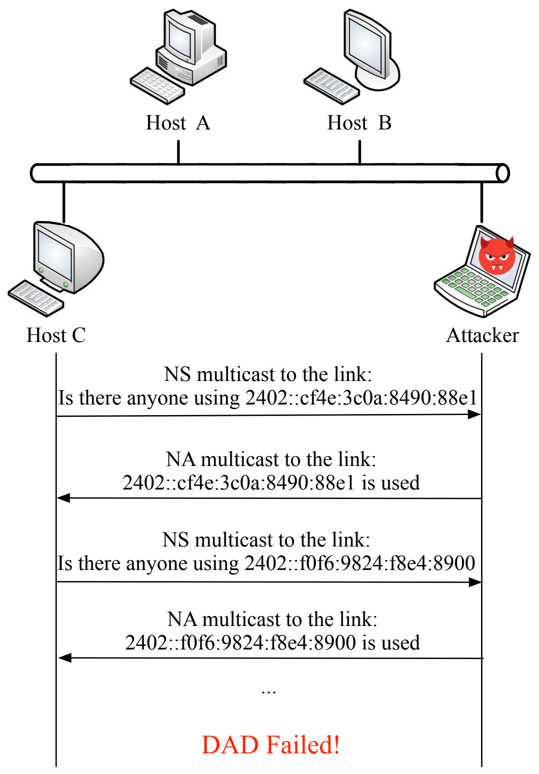

Host A
Host B
Host C
Attacker
NS multicast to the link:
Is there anyone using 2402::cf4e:3c0a:8490:88e1
NA multicast to the link:
2402::cf4e:3c0a:8490:88e1 is used
NS multicast to the link:
Is there anyone using 2402::f0f6:9824:f8e4:8900
NA multicast to the link:
2402::f0f6:9824:f8e4:8900 is used
...
DAD Failed!

Fig. 1. A DoS attack on DAD. Host C cannot configure any addresses to its interfaces when an attacker launches DoS attacks on DAD.

However, it is easy for a malicious host to spoof NA messages because there exist no secure mechanisms to verify whether the sender of an NA message owns the IPv6 address or not. For example, Figure 1 sheds light on a DoS attack on DAD. When host C wants to configure a global unicast address $\mathtt{IP}_1$ (e.g., 2402::cf4e:3c0a:8490:88e1), it first multicasts an NS message to the link to verify the uniqueness of $\mathtt{IP}_1$. However, the attacker on the same link can reply with a spoofed NA message to indicate that $\mathtt{IP}_1$ is already used by another node. When host C tries to configure another address $\mathtt{IP}_2$ (e.g., 2402::f0f6:9824:f8e4:8900) and sends another NS message to verify the uniqueness of $\mathtt{IP}_2$, the attacker can also launch the attack by sending another NA message to indicate $\mathtt{IP}_2$ is already used. The result is that host C cannot configure any IPv6 addresses to its interfaces.

B. Target Address Spoofing

We conduct two experiments on different operating systems and address assignment mechanisms to evaluate whether they influence the detection results of DAD when target addresses are spoofed in NA messages.

- **Different Operating Systems:** We build a simple topology where two hosts (a victim and an attacker) are connected to a switch. The victim sends an NS message to verify the uniqueness of a link-local address. At the same time, the attacker will reply with a spoofed NA message when receiving the NS message. We test two cases: 1) when the source address in the NA message is the same as the target address, and 2) when the source address in

TABLE I
COMPARISON OF NS AND NA MESSAGES IN DAD.

Message	Source Address	Destination Address	Target Address
NS	Unspecified	Solicited-node multicast address	Tentative address
NA	Interface address	All-node multicast address	Interface address

TABLE II
TEST RESULTS OF SPOOFED TARGET ADDRESSES ON LINK-LOCAL ADDRESS AUTOCONFIGURATION.

Operating System	Source Address = Target Address	Source Address $\neq$ Target Address
Ubuntu 16.04	Three tentative link-local addresses	Three tentative link-local addresses
CentOS 7.6	Three DAD-failed link-local addresses	Three DAD-failed link-local addresses
Windows 10	No link-local addresses	No link-local addresses
Windows 7	No link-local addresses	No link-local addresses
macOS 10.15	No link-local addresses	No link-local addresses

TABLE III
TEST RESULTS OF SPOOFED NA MESSAGES IN DIFFERENT ADDRESS ASSIGNMENT MECHANISMS.

Address Assignment Mechanism	Source Address = Target Address	Source Address $\neq$ Target Address
SLAAC	No global unicast addresses	No global unicast addresses
DHCPv6	No global unicast addresses	No global unicast addresses
Manual Configuration	No global unicast addresses	No global unicast addresses

the NA message is different from the target address. The results are shown in Table II. As we can see, the victim can not configure any link-local addresses successfully no matter whether the source address is equal to the target address or not on different operating systems.

- **Different Address Assignment Mechanisms:** We also explore the influence of different address assignment mechanisms on the possibility of the attacker's launching DoS attacks on DAD when the target address is spoofed. Similarly, we also test two cases: 1) when the source address equals the target address in the NA message, 2) when the source address is not equal to the target address in the NA message. The results are shown in Table III. The victim can not configure any global addresses successfully regardless of whether they are obtained through SLAAC, DHCPv6, or manual configuration when the target addresses in NA messages are spoofed.

To conclude, mainstream operating systems do not check and filter spoofed NA messages no matter whether the tentative addresses are obtained through SLAAC, DHCPv6, or manual configuration, even when the source address is not the same as the target address in an NA message. Therefore, the target address in an NA message is the decisive factor which affects the detection result of DAD.

C. Existing Solutions

Currently, existing solutions to preventing DoS attacks on DAD can be classified into three categories: 1) hiding target addresses, 2) authenticating NDP messages, and 3) replying with NA messages uniformly.

- **Hiding Target Addresses:** The core idea of this kind of solution is to hide target addresses in NS messages during the procedure of DAD. DAD-match [14] and Pull Model [15] embed the hash value of the target address instead of putting it directly in an NS message. Other hosts compute the hash values of their addresses and compare the results with the value in the NS message to determine whether they need to reply with NA messages. Wang et al. [16] proposed a mechanism that a host masks a segment of the target address, and other hosts compare the unmasked part of the target address and reply with standard NA messages if the unmasked part of the target address is a duplicate. The host collects all NA messages to determine whether the address is unique or not. Although this kind of solution can mitigate DoS attacks on DAD, they require modification to the NDP and host stack, which makes it challenging to deploy in the current networks.
- **Authenticating NDP Messages:** This kind of solution enhances DAD security by appending a message authentication code to each NDP message. Trust-ND [17] and Secure-DAD [18] are representatives of the methods of this type, which calculate the message authentication code by using SHA-1 and UMAC hash function, respectively. However, Trust-ND is not secure because it is vulnerable to collision attacks due to its design. Moreover, an attacker can also use SHA-1 to compute correct message authentication codes and append them to spoofed NA messages to launch DoS attacks on DAD. Secure-DAD does not address how secret keys are shared between a new host and existing hosts. If all the hosts share the same key, an attacker on the same link can also launch DoS attacks. If a new host is required to establish secret keys with all existing hosts, an attacker can still use the secret key shared with the new host to compute message authentication codes and launch DoS attacks, not to mention the scalability of key exchange between hosts. Besides, they both introduce new options to NDP messages, which also requires modification to the NDP and host stack.
- **Replying with NA Messages Uniformly:** The last kind

of solution is to utilize a central reliable node to reply with NA messages uniformly, and any other NA messages sent by other nodes will be neglected. Nelle et al. [19] used a Software-Defined Networking (SDN) switch as an NDP proxy that intercepts all NDP messages and forwards them to the SDN controller. As the controller stores all the assigned IPv6 addresses, it replies an NA message if a tentative address is a duplicate. Rule-based mechanism [20] uses a central node to store all configured IPv6 addresses. Other nodes must receive its confirmation messages to verify whether a tentative address is a duplicate or not. However, SDN suffers from a lot of security issues [21]–[23], e.g., DoS attacks. Both of them need a central node to store all IPv6 addresses, which thus becomes a single point of failure and renders the whole network down in case of a security compromise. Moreover, Rule-based mechanism also requires modification to the NDP.

Table IV summarizes the comparison and limitations of existing solutions. We aim to design P4DAD to overcome these limitations.

D. Programmable Devices and P4

P4 [7] is a domain-specific language for programming switching ASICs (Application-Specific Integrated Circuits) based on the reconfigurable match tables (RMT) model [24]. As shown in Figure 2, RMT consists of five components, i.e., the parser, ingress pipeline, queue, egress pipeline, and deparser. The parser is used to extract user-defined header fields. The ingress and egress pipelines contain a sequence of logical match-action stages, which can match specific header fields and perform corresponding actions. The queuing system between ingress and egress pipeline is associated with common data buffers to store packets. The deparser is used to reconstruct a packet after the egress pipeline. Moreover, P4 supports stateful components, e.g., registers, counters, to store states in the data plane directly. A P4 program only defines data plane functions and requires flow rules from the control plane to decide which action to take for each packet.

Given the flexibility, we briefly highlight the new opportunities that programmable devices bring for preventing DoS attacks on DAD. In the DAD scenario, the programmable parser in P4 can be defined to extract target addresses in NS and NA messages. Besides, P4 can utilize stateful registers to preserve binding entries in the data plane directly. Therefore, P4DAD can monitor DAD procedures to create binding entries between legal addresses of each host and switch port, and then use them to filter spoofed NS and NA messages.

Compared with the previous version of this paper [25], we have made substantive enhancements in this manuscript. Firstly, we conduct experiments to prove that mainstream operating systems do not check and filter spoofed NA messages no matter how the tentative addresses are obtained, which indicates an attacker can launch DoS attacks on DAD easily. Secondly, we have thoroughly optimized the binding entry update algorithm (including the design of binding entry expiration mechanism) and provided the target address validation algorithm for P4DAD. Finally, we have updated the evaluation and discussed the deployment strategies of P4DAD to demonstrate that P4DAD can enhance DAD security in a lightweight, robust, and deployable manner.

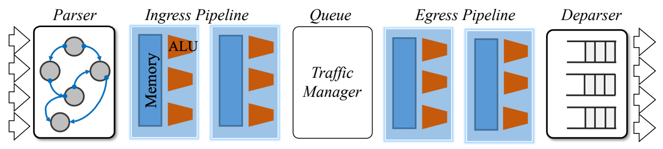


Fig. 2. Switch chip architecture.

III. Problem Formulation

A. Design goals

P4DAD aims to enhance DAD security without introducing any modification to the NDP and host stack in a lightweight and robust manner, which implies three goals:

- **Lightweight**: P4DAD should not introduce hash encryption computation to hosts or network devices, which will increase computation complexity. Besides, P4DAD should only bring a small performance overhead to switches.
- **Robust**: P4DAD should not introduce a central node, which requires that P4DAD be able to prevent DoS attacks on DAD in a distributed manner, which ensures that P4DAD can avoid a single point of failure problem.
- **Deployable**: P4DAD should not modify the NDP or host stack. It means P4DAD should be designed to be purely network-based, which enables network operators to deploy it without a dependency on supportive functionality on hosts.

B. Threat Model

We assume that hosts on a link connect to P4DAD-enabled devices (e.g., P4 switches) directly. We relax this assumption in Section VII-B2 and discuss the deployment strategy when first-hop devices do not upgrade to support P4DAD. It is also assumed that hosts may be malicious and can send arbitrary spoofed NDP messages, which may contain spoofed source addresses or target addresses. Besides, we assume that P4DAD-enabled devices are trustworthy and can process received NDP messages correctly.

C. Challenges

To design P4DAD, we mainly face three challenges:

1) **Multiple IPv6 Addresses Per Host**: In IPv6 networks, a host usually configures more than one IPv6 address. One is a link-local address, and the others may be assigned by SLAAC, DHCPv6, or manual configuration. How to deal with them in P4DAD is challenging.
2) **Auto-Configuration of Flow Rules without a Central Controller**: P4 is a data plane programming language that can be used to describe how to parse and handle packets. P4 programs often use flow rules from the control plane to decide which action to take for every packet. Although P4Runtime [26] provides a control

TABLE IV
COMPARISON OF EXISTING SOLUTIONS.

Type	Mechanism	Secure	Lightweight	Robust	Deployable
Hiding Target Addresses	DAD-match [14]	✓		✓	
	Pull Model [15]	✓		✓	
	Wang et al. [16]	✓		✓	
Authenticating NDP messages	Secure-DAD [18]			✓	
	Trust-ND [17]			✓	
Replying with NA Messages Uniformly	Nelle et al. [19]		✓		✓
	Rule-based [20]	✓	✓		
Monitoring & Filtering	**P4DAD**	✓	✓	✓	✓

plane programming interface to configure and populate flow rules, it needs to maintain complete network topology information, which may be a single point of failure. How to configure and populate flow rules in switches locally is a challenge.

3) **Address State Change without Timer**: During the procedure of P4DAD, there are two updates of the address state. Firstly, when a binding entry is established, P4DAD will set the state of an IPv6 address as "tentative". If there are no NA messages received within a certain period of time (e.g., 1 second [1]), the address will pass the DAD procedure, and its state will be updated as "preferred". Secondly, each preferred address has a lifetime. For example, in DHCPv6, the preferred lifetime will be carried in Reply messages [12] when assigning an IPv6 address to a client. In SLAAC, the preferred lifetime will be carried in the Prefix Information option in Router Advertisement messages. A preferred address becomes deprecated when its preferred lifetime expires. P4DAD should update or clear the expired binding entries in time. However, in the current version of P4 [7], there is no related timer mechanism. Thus, how to change the address state without a timer is a challenge.

IV. P4DAD DESIGN

This section presents the details of P4DAD design. Generally, P4DAD utilizes two key features of P4 to secure DAD:

- it uses the programmable parser to extract target addresses in NS and NA messages;
- it uses stateful memory registers to preserve bindings between link-layer properties of nodes' network attachment (e.g., ports) and IPv6 addresses in the data plane directly.

To solve the problem of multiple IPv6 addresses assigned to a single host, P4DAD allows a switch port[2] to correspond to multiple IPv6 addresses. To realize the auto-configuration of flow rules, P4DAD uploads the digest of required information in packets to the control plane and notifies it of populating appropriate flow rules in real-time. As for the address state change, we utilize a "sketchy timer" to solve it (See more details in Section IV-D).

[2]We take P4 switches as an example to demonstrate the design of P4DAD. In wired networks, we use switch ports to create bindings. In wireless networks, we can create bindings between MAC addresses and IPv6 addresses. The security of MAC address is assured by 802.11i or other mechanisms.

A. Overview of P4DAD

On the whole, P4DAD is composed of a data plane and a control plane. As shown in Figure 3, the data plane contains a parser, pipeline, deparser, and registers. The parser is responsible for extracting the header of each NS or NA message for processing by pipeline. The pipeline is the core component of the data plane composed of if-else structures and match-action tables (MATs). As shown in Table V, the pipeline performs different operations on NS and NA messages for different functions. First, the pipeline verifies the source and target addresses in the header to drop spoofed packets (Section IV-B). Then, it sends MAC addresses and ingress ports to the control plane for MAC address learning (Section IV-C). Next, it queries the target address to determine whether to update the binding entry of IPv6 address and port in the register (Section IV-D). Finally, it forwards packets based on destination MAC addresses. The control plane is a local controller of the switch, which receives information from the data plane and populates appropriate flow rules promptly.

B. Source and Target Validation

To verify the authenticity of the source and target addresses in NS and NA messages, P4DAD utilizes bindings between IPv6 addresses and their corresponding switch ports. More details of creating and updating binding entries are provided in Section IV-D. When a P4 switch receives a packet from a host, it first checks whether the packet is an NDP message. If the packet is an NS message, the switch first determines whether it is used for DAD. If the NS message is used for DAD, the switch just forwards the packet; otherwise, the switch queries the source address of the NS packet in $ipv6\_reg$ to determine whether to forward the packet. If the packet is an NA message, the switch first validates whether the source address equals the target address or not. Spoofed NA messages will be dropped. Then, it queries the target address in $ipv6\_reg$ to determine whether to forward the packet. If the above check fails, the switch drops the packet; otherwise, it forwards the packet. The above steps are summarized in Algorithm 1.

Register Design: P4DAD employs on-switch registers to store the bindings. Then, a key question is how to determine the entry in a register corresponding to a specific switch port, i.e., the method of calculating the register index. Intuitively, as a host is connected to a port in a switch, a simple solution is to regard a switch port as the register index and set the

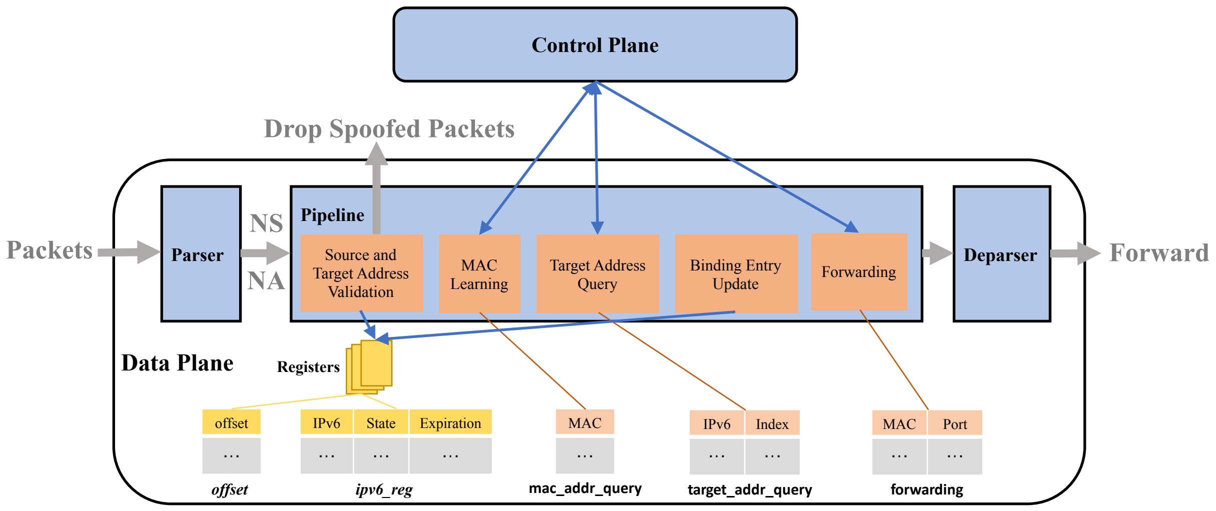


Fig. 3. Design overview of P4DAD.

TABLE V
DIFFERENT OPERATIONS ON NS AND NA MESSAGES FOR DIFFERENT FUNCTIONS.

Message Type	Function	Source Validation	Target Validation	MAC Learning	Target Query	Binding Entry Update	Forwarding
NS	DAD			✓	✓	✓	✓
	Address Resolution	✓					✓
	Unreachability Detection	✓					✓
NA	DAD	✓	✓		✓	✓	✓
	Address Resolution	✓			✓		✓
	Unreachability Detection	✓			✓		✓
	Unsolicited	✓			✓		✓

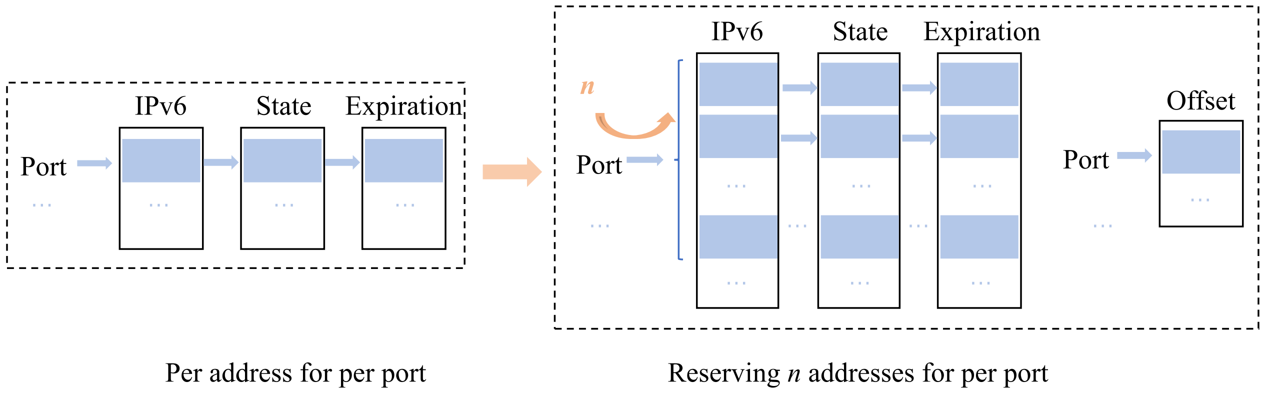


Fig. 4. Register design for P4DAD.

host's IPv6 address as the corresponding value. However, the following issue is that a host often has multiple IPv6 addresses. Therefore, we have to reserve multiple IPv6 addresses for every switch port in registers. It is assumed that a host has n IPv6 addresses ($n \geq 2$)[3]. As P4$_{16}$ [27] only supports multiplication by a power of two currently, n must be a power of two. The register index based on the switch port can be calculated as follows:

$$index = n \cdot port + offset, \tag{1}$$

where $port$ is the switch port number and $offset$ is the number of IPv6 addresses currently stored in the register corresponding to the port.

To distinguish different IPv6 address states (e.g., tentative, preferred) [1], registers also need to store the state of each IPv6 address. As DAD is performed on the same link, registers can store 64-bit IPv6 interface identifiers without the common prefix. Figure 4 sheds light on the design of two registers P4DAD uses. More specifically, one register is used to store

[3]Generally, one is a link-local address, and the others are global unicast addresses, which are configured through manual configuration, SLAAC, or DHCPv6. Moreover, it is common practice for most mainstream operating systems to configure two SLAAC addresses for hosts.

Algorithm 1 Source and Target Address Validation in P4DAD.

```
Input: $pkt$, $meta$, $ipv6\_reg$, $offset$, $n$
 1: ▷ $pkt$ refers to an ICMPv6 packet.
 2: ▷ $meta$ refers to the metadata associated with a packet,
    e.g., ingress port.
 3: ▷ $ipv6\_reg$ refers to a register used to store IPv6 addresses
    corresponding to each port.
 4: ▷ $offset$ refers to a register used to store the offset
    corresponding to each port.
 5: ▷ $n$ refers to the maximum number of addresses that the
    register can store for each host.
 6: $ip_{src} \leftarrow pkt[source\_address]$
 7: $ip_{dst} \leftarrow pkt[destination\_address]$
 8: if $pkt$ is NS then
 9:     $ip_{tgt} \leftarrow pkt[target\_address]$
10:     if $ip_{src}$ is not unspecified || $ip_{dst}$ is not a solicited-
        node multicast address then
11:         $addrs$ = READADDRS($meta$, $offset$, $n$, $ipv6\_reg$)
12:         if $ip_{src}$ is not in $addrs$ then
13:             drop $pkt$
14:         end if
15:     end if
16: else if $pkt$ is NA then
17:     $ip_{tgt} \leftarrow pkt[target\_address]$
18:     if $ip_{tgt} != ip_{src}$ then
19:         drop $pkt$
20:     end if
21:     $addrs$ = READADDRS($meta$, $offset$, $n$, $ipv6\_reg$)
22:     if $ip_{tgt}$ is not in $addrs$ then
23:         drop $pkt$
24:     end if
25: end if
26: forward $pkt$
27:
28: function READADDRS($meta$, $offset$, $n$, $ipv6\_reg$)
29:     $port \leftarrow meta[ingress\_port]$
30:     $os \leftarrow offset[port]$
31:     $index \leftarrow n \cdot port$
32:     Delete the expired entries corresponding to $port$ and
        move the subsequent entries forward
33:     Read $os$ addresses into $addrs$ from $index$ in $ipv6\_reg$
34:     return $addrs$
35: end function
```

the IPv6 addresses and their corresponding states, and the other is used to store the offset corresponding to each port.

C. Self-learning Table Building

To avoid using a remote central controller to configure and populate flow rules, P4DAD implements MAC address and IPv6 address self-learning in switches locally. Specifically, P4DAD employs three MATs, i.e., mac_addr_query, target_addr_query, and forwarding, to query MAC addresses, query target addresses, and forward packets, respectively.

The key issue of self-learning is how to notify the control plane of populating specified flow rules. Fortunately, $P4_{16}$ provides two methods to solve this problem: packet cloning and packet digest. The former is used to copy the entire content of packets to the control plane. The latter is used to send the digest of some fields in the packet to the control plane. As we only need to inform the control plane of the source MAC address and ingress port queried from the mac_addr_query and forwarding, and the IPv6 address and register index from the target_addr_query, we prefer to use packet digest in P4DAD.

D. Binding Entry Update

P4DAD creates bindings between IPv6 addresses and ports by listening to the NS and NA messages exchanged in the procedure of DAD. For NS and NA messages, P4DAD adopts different processing logic, which is summarized in Algorithm 2. In P4DAD, there are two states for each address, namely $tentative$ and $preferred$. The $tentative$ state of an address means its uniqueness on a link is being verified, prior to its assignment to an interface. The $preferred$ state of an address assigned to an interface means its use by upper-layer protocols is unrestricted.

First, P4DAD extracts the source and destination addresses from a packet. Then, P4DAD determines whether the packet is an NDP message. If the packet is an NS message, P4DAD extracts the target address in the packet. If the NS message is used for DAD, P4DAD queries whether the target address is in the target_addr_query or not. If the query misses, P4DAD first deletes the expired entries corresponding to the ingress port and moves the subsequent entries forward. Then, it calculates the register index based on the ingress port as shown in Equation (1), and wraps the IPv6 address, its $tentative$ state, and expiration time in the corresponding register, which implies this is a new DAD procedure. The reason that we set 1 second after the current time as the expiration time is because the $tentative$ state of an IPv6 address will be expired in 1 second no matter what the result of the DAD procedure is. If the address is a duplicate, this entry will be deleted; otherwise, P4DAD will change the state of the address to $preferred$. 1 second is a default value as specified by IETF in RFC 4861 [2]. If the NS message is not used for DAD, P4DAD changes the expired $tentative$ state of the target address to the $preferred$ state. In this case, when P4DAD receives an NS message not used for DAD, this means that the host directly connected to the P4 device starts to discover other nodes' presence, which implies the address passed the DAD procedure. Note that we solve challenge 3) by using a "sketchy timer". When the tentative state of an address expires, P4DAD will change its state into preferred after receiving NS messages for other functions rather than DAD. Because a tentative address of a host passes the DAD procedure after 1 second, the host will send NS messages for address resolution and unreachability detection, which triggers the change of address state in P4DAD. Similarly, if the preferred state of an address expires, P4DAD will delete this entry when receiving NS messages for DAD or filtering spoofed NS or NA messages corresponding to the same port.

If the packet is an NA message used for DAD, P4DAD extracts the target address from the packet and queries it in

Algorithm 2 Binding Entry Update for P4DAD.

```
Input: $pkt$, $meta$, target_addr_query, $ipv6\_reg$, $offset$, $n$, "T", "P"
1:  ▷ $pkt$ refers to an ICMPv6 packet.
2:  ▷ $meta$ refers to the metadata associated with a packet, e.g., ingress port.
3:  ▷ target_addr_query refers to a match-action table used to query target addresses.
4:  ▷ $ipv6\_reg$ refers to a register used to store IPv6 addresses corresponding to each port.
5:  ▷ $offset$ refers to a register used to store the offset corresponding to each port.
6:  ▷ $n$ refers to the maximum number of addresses that the register can store for each host.
7:  ▷ $L$ refers to the lifetime of a $preferred$ address.
8:  ▷ "T" and "P" refer to the $tentative$ and $preferred$ states of an address, respectively.
9:  $ip_{src} \leftarrow pkt[source\_address]$
10: $ip_{dst} \leftarrow pkt[destination\_address]$
11: $port \leftarrow meta[ingress\_port]$
12: if $pkt$ is NS && $pkt$ comes from a host-connected port then
13:     $ip_{tgt} \leftarrow pkt[target\_address]$
14:     Get the current timestamp $cur\_time$
15:     if $ip_{src}$ is unspecified && $ip_{dst}$ is a solicited-node multicast address then
16:         if $ip_{tgt}$ misses target_addr_query then
17:             Delete the expired entries corresponding to $port$ and move the subsequent entries forward
18:             $index \leftarrow n \cdot port + offset[port]$
19:             $ipv6\_reg[index] \leftarrow ip_{tgt}$,"T", $cur\_time + 1$
20:             target_addr_query[$ip_{tgt}$] $\leftarrow index$
21:             $offset[port] \leftarrow offset[port] + 1$
22:         end if
23:     else
24:         $index \leftarrow$ target_addr_query[$ip_{tgt}$]
25:         $ipv6$, $state, exp\_time \leftarrow ipv6\_reg[index]$
26:         if $state$ == "T" && $exp\_time < cur\_time$ then
27:             $ipv6\_reg[index] \leftarrow ip_{tgt}$,"P", $exp\_time + L - 1$
28:         end if
29:     end if
30: else if $pkt$ is NA then
31:     if $ip_{dst}$ is an all-node multicast address then
32:         $ip_{tgt} \leftarrow pkt[target\_address]$
33:         if $ip_{tgt}$ hits target_addr_query then
34:             $index \leftarrow$ target_addr_query[$ip_{tgt}$]
35:             $ipv6$, $state, exp\_time \leftarrow ipv6\_reg[index]$
36:             if $state$ == "T" then
37:                 reset $ipv6\_reg[index]$
38:                 reset target_addr_query[$ip_{tgt}$]
39:                 $offset[port] \leftarrow offset[port] - 1$
40:             end if
41:         end if
42:     end if
43: end if
```

the target_addr_query. If the query hits, P4DAD acquires the index from the hitting entry and obtains IPv6 address and its state from the $ipv6\_reg$ using the index. If the state of IPv6 address is tentative, P4DAD resets the corresponding entry in $ipv6\_reg$ and target_addr_query. This means the IPv6 address is used by another node on the link. After that, the addresses corresponding to a port should be adjusted to take up continuous space in a register.

E. P4DAD Workflow

We illustrate the P4DAD workflow in a simple network topology, which contains two P4 switches S_1 and S_2, as shown in Figure 5. Host H_1 and H_2 configured IPv6 addresses IP_1 and IP_2, respectively. Note that H_2 is a malicious host, and it will launch DoS attacks on DAD. So the binding entries of IP_1 and IP_2 have been established in S_1. H_3 and H_4 are performing DAD on addresses they wish to configure, respectively. H_3 tries to configure the same IPv6 address as that (i.e., IP_1) of H_1. H_4 wishes to configure a new IPv6 address IP_3 to its interface. When receiving NS packets sent by H_3 and H_4, S_2 will establish the binding entries of IP_1 and IP_3 and set the corresponding states of IP_1 and IP_3 to "Tentative". H_1 will reply with an NA packet with IP_1 as the source and target addresses. However, the malicious H_2 will send two spoofed NA packets with IP_1 and IP_3 as the target address, respectively. When spoofed NA packets are received by S_1, they will be filtered directly, which means the DoS attacks on DAD are prevented effectively. When S_2 receives the NA packet sent by H_1, the binding entry of IP_1 will be cleared. When no NA packets with IP_3 as the target address are received by S_2 at a certain time, the corresponding state of IP_3 will be set to "Preferred".

V. Implementation and Evaluation

In this section, we present the implementation and performance evaluation of P4DAD.

A. Implementation

We implement a P4DAD prototype which is composed of a data plane program and a control plane program. The data plane program is implemented using $P4_{16}$ [27], which can be compiled directly on bmv2 [28]. The control plane program is written in Python, which communicates with the data plane by notifications-addr [28].

Our experiments are conducted on a Dell OptiPlex 7050 server with Intel Core i7-7700 CPU (one core, 3.4 GHz) and 8 GB of memory.

B. Evaluation

1) Functionality: We build a simple topology where 10 hosts (5 innocent hosts and 5 attackers) installed with the parasite6 tool are connected to a P4 software switch implemented by bmv2. Both innocent hosts and attackers send NS and NA messages randomly. The difference is that innocent hosts send valid messages, while attackers send spoofed messages. We count the number of packets received and filtered by P4DAD,

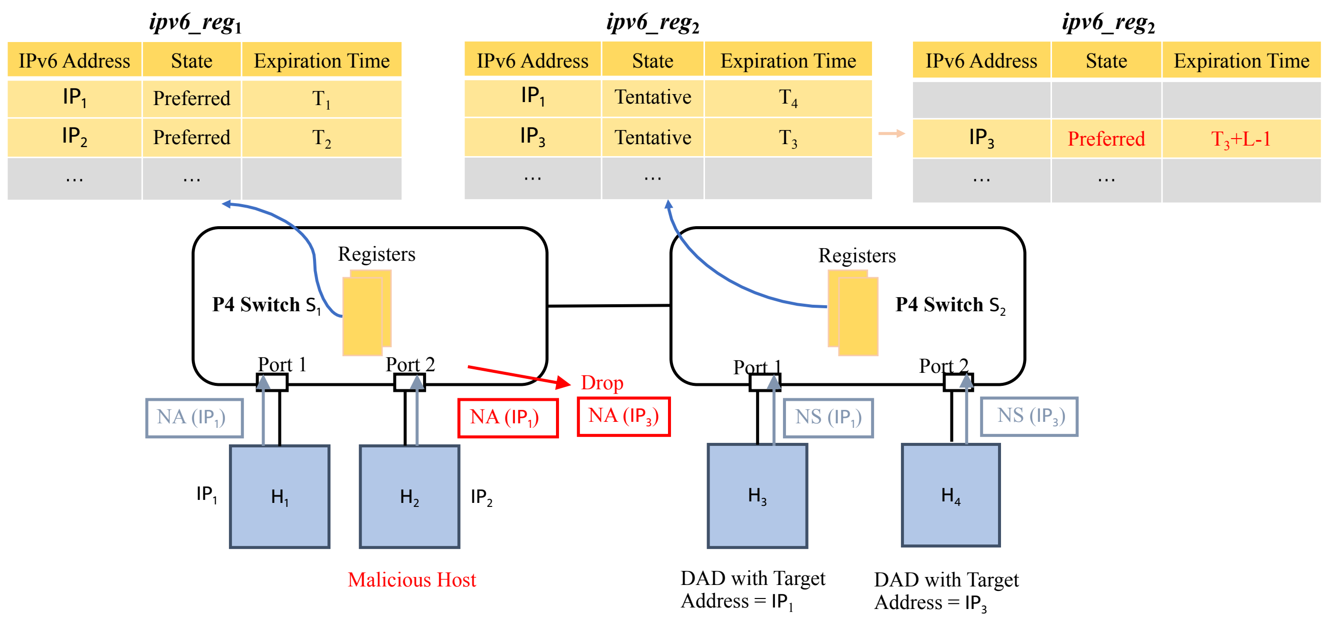


Fig. 5. An example of workflow for P4DAD.

sent by the attacker and the victim, respectively. The results are shown in Figure 6. The number of NS and NA packets filtered by P4DAD is equal to that sent by the attacker, which means P4DAD can recognize and filter spoofed NS and NA messages effectively.

2) Performance: **Message Processing Overhead.** We use the performance of original NDP (no modification to switches) as the baseline and compare the processing time of NS and NA messages between P4DAD and the baseline. As mentioned in Table V, NS and NA messages have different usages in different scenarios. Therefore, for simplicity of notation, we denote NS-for-DAD and NS-for-Others as NS messages used to handle DAD and other NDP functions, respectively. Similarly, we denote NA-for-DAD and NA-for-Others as NA messages used to handle DAD and other NDP functions, respectively. As we expected, P4DAD increases processing time for each type of message by 27%, 6.8%, 36%, and 14% as shown in Figure 7. This is because P4DAD introduces new operations and validates NDP messages to secure the security of DAD at the cost of slight performance degradation. More specifically, NA-for-DAD has the biggest increase in processing time because P4DAD utilizes several operations (i.e., source validation, target validation, target query, and binding entry update as shown in Table V) to handle this type of messages. Conversely, NS-for-Others has the smallest increase in processing time because P4DAD only validates the source addresses for this type of message. In an IPv6 network, only when a host configures IPv6 addresses to its interfaces, NS-for-DAD and NA-for-DAD messages are used. Therefore, the message processing overhead introduced by P4DAD is lightweight.

Digest Overhead. As the self-learning of P4DAD notifies the control plane of populating flow rules, we measure its overhead. Specifically, we measure the time from the beginning of the data plane's digesting packet information to the end of the control plane's populating flow rules. We conduct 1,000 tests, and its average time is 4.126 ms. Generally, the communication between the control plane and the data plane only occurs in the DAD procedure, and DAD only appears during the process of address configuration. Therefore, the processing overhead brought by packet digest is acceptable.

3) Scalability: In this paper, the scalability of P4DAD means the capability to cope and perform well when more and more hosts are connected to a P4 switch. More specifically, as the number of hosts connected to the P4 switch increases, the P4 switch will consume more memory to store the states of IPv6 addresses. The memory consumed by P4DAD consists of two folds: register memory and MAT memory. We measure both of them and then calculate their sum. As shown in Figure 8, the memory consumption of P4DAD increases linearly with the number of hosts when n, as shown in Equation (1), is equal to 2 or 4. When there are 1,000 hosts connected to P4 switches and $n = 2$, $\sim$30 KB memory is consumed to store the states of IPv6 addresses, and $\sim$33 KB memory is used to store mac_addr_query, target_addr_query, and forwarding.

Before further analysis, we present the notation we use throughout the description, which is summarized in Table VI. We theoretically calculate the register memory m_r (i.e., the memory consumed by $ipv6\_reg$ and $offset$) and MAT memory m_t (i.e., the memory consumed by mac_addr_query, target_addr_query, and forwarding) as follows:

$$\begin{aligned} m_r &= [(b_i + b_s + b_e) \cdot n + b_o] \cdot h,\ \ h \le N, \\ m_t &= [(b_i + b_x) \cdot n + 2b_m + b_p] \cdot h,\ \ h \le N, \end{aligned} \tag{2}$$

where h is the number of the hosts connected to a P4 switch and N is the number of the ports that the P4 switch has. In our implementation, for m_r, each entry of $ipv6\_reg$ consumes 15 B of memory (8 B for an IPv6 address ($b_i = 8$), 1 B for its state ($b_s = 1$), and 6 B for expiration time ($b_e = 6$)), and each entry of $offset$ consumes 1 B of memory ($b_o = 1$). For m_t, each entry of mac_addr_query, target_addr_query, and

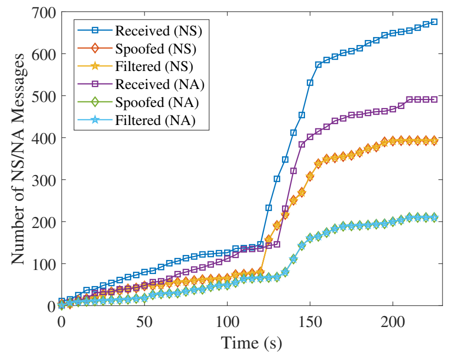


Fig. 6. Filtered messages using P4DAD.

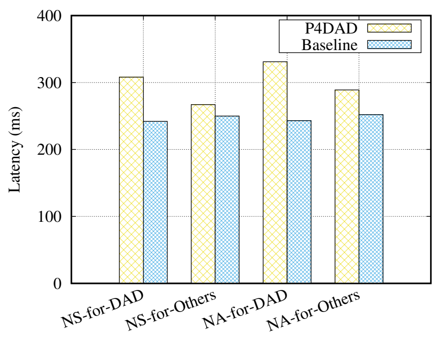


Fig. 7. Processing time of NS and NA messages in P4DAD.

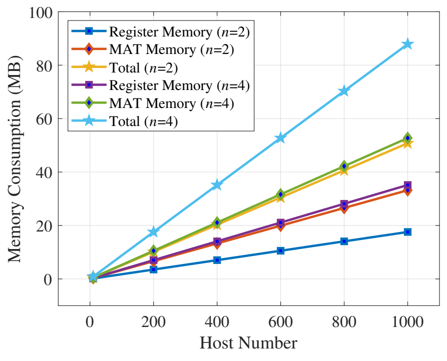


Fig. 8. Memory consumption of P4 switches.

TABLE VI
NOTATION.

Symbol	Remarks
b_i	Memory consumed by an IPv6 address
b_s	Memory consumed by the state of an IPv6 address
b_e	Memory consumed by expiration time of an IPv6 address
b_o	Memory consumed by storing an offset in a register
b_m	Memory consumed by storing a MAC address
b_x	Memory consumed by storing the index of an IPv6 address in target_addr_query
b_p	Memory consumed by storing the port of an MAC address
m_r	Memory consumed by all the registers in P4DAD
m_t	Memory consumed by all the MATs
m_i	Memory consumed by *ipv6_reg*
h	The number of the hosts connected to a P4 switch
N	The number of the ports that the P4 switch has
i	The number of IPv6 addresses that a host configures to its interfaces
u	Register utilization of *ipv6_reg*

forwarding consumes 6 ($b_m = 6$), 10 (2 B for $index, b_x = 2$), 8 B (2 B for $port, b_p = 2$) of memory, respectively.

As shown in Equation (2), at a particular value of n, both register memory and MAT memory are linear with respect to the number of hosts, which is consistent with the previous measurements. Nowadays, mainstream programmable switches like Tofino [8] can support 260 ports. Even if n is set as 4, only ∼29 KB memory is consumed by registers and MATs in P4DAD. The latest generation of switching ASICs have more than 50-100 MB SRAM [29]. Moreover, external memory can be used for programmable switch data planes [30]. In conclusion, both experimental measurements and theoretical analysis show that P4DAD has satisfactory scalability.

VI. Security Analysis

P4DAD is designed to mitigate DoS attacks on DAD using P4. However, P4DAD may face security threats. In this section, we demonstrate how P4DAD prevents attacks that undermine its effectiveness and availability.

A. Attacking P4DAD-enabled Devices

An attacker launch attacks against the resources of P4DAD-enabled devices. For example, the attacker may attempt to send NS packets with different source addresses for duplicate detection, making the P4DAD-enabled devices create binding entries and waste memory. However, P4DAD only reserves n IPv6 addresses for every port in registers. n is often set as 2 or 4 in a network. According to the analysis in Section V-B3, it is infeasible for the attacker to make the P4DAD-enabled devices run out of memory by launching such attacks.

B. Sharing the Same Port

If an attacker and legitimate users share a port of a P4DAD-enabled device by connecting a legacy switch to that port. P4DAD will create binding entries for the addresses of the attacker and legitimate users connected to the legacy switch. In such a case, the attacker can still not launch DAD DoS attacks on hosts that connect to other ports of the P4DAD-enabled device because spoofing the NDP packet will be filtered by P4DAD. However, the attacker can launch DoS attacks on DAD when the hosts connected to the same legacy switch perform duplicate address detection. Therefore, sharing the same port of a P4DAD-enabled device is not allowed, which is also claimed in Section III-B.

C. Privacy Considerations

A P4DAD-enabled device should not log information about hosts that never spoof. Besides, a P4DAD-enabled device

TABLE VII
IMPACT OF n ON REGISTER UTILIZATION.

n	SLAAC-Only	DHCPv6-Only	SLAAC+DHCPv6
2	✗	100%	✗
4	75%	50%	100%
8	37.5%	25%	50%

should delete binding entries as soon as possible when the addresses are released. If the binding entries are likely to be involved in the detection, prevention, or tracing of actual target address spoofing, that information may be stored for a while, the details of which depend on the specific policy of a network.

VII. DISCUSSION

In this section, we discuss register utilization and deployment issues of P4DAD.

A. Register Utilization

We explore the impact of n on the register utilization of $ipv6\_reg$ and then evaluate the relationship between register memory m_i and register utilization u. The definition of the register utilization u of $ipv6\_reg$ is as follows:

$$u = \frac{i}{n} \quad (i \leq n), \tag{3}$$

where i is the number of IPv6 addresses that a host configures to its interfaces. We consider three scenarios according to different address assignment mechanisms. In each scenario, a host configures a link-local address first. The details of each scenario are as follows: 1)

1) **SLAAC-Only**: Only SLAAC is adopted. The access routers in the network periodically multicast the Router Advertisement (RA) messages [2] to all on-link nodes with the Managed and Otherconfig flags off. RA messages contain a Prefix Information Option (PIO). When receiving RA messages, a host uses SLAAC and the prefix in PIO to configure two global unicast IPv6 addresses;
2) **DHCPv6-Only**: Only DHCPv6 is adopted. The access routers in the network periodically multicast the RA messages to all on-link nodes with the Managed and Otherconfig flags on. When receiving this kind of RA messages, a host requests IPv6 addresses from the DHCPv6 server. The DHCPv6 server usually assigns a global unicast IPv6 address to each host;
3) **SLAAC+DHCPv6**: Both SLAAC and DHCPv6 are adopted, and a host can configure two global unicast IPv6 addresses through SLAAC and a global unicast IPv6 address through DHCPv6.

We calculate the register utilization in the above three scenarios, and the results are shown in Table VII. When n equals 2, a network can only adopt DHCPv6. In such a case, the register utilization reaches 100%. When n is equal to 4, a network can adopt either SLACC or DHCPv6. The register utilization reaches 100% if hosts configure both SLAAC and DHCPv6 addresses. However, when n equals 8, the register utilization decreases by half.

The register memory m_i consumed by $ipv6\_reg$ is defined as follows:

$$m_i = \frac{h \cdot i}{u} \cdot unit, \tag{4}$$

where h is the number of hosts, $unit$ is the register memory size that an IPv6 address occupies, and u and i are shown in Equation (3). In our example, the size of $unit$ is 15 B, with 8 B for storing an IPv6 address, 1 B for storing its state, and 6 B for storing its expiration time. As shown in Equation (4), under the same number of hosts and a certain number of IPv6 addresses assigned to each host, the register memory m_i consumed by the switch decreases with the increase of register utilization u.

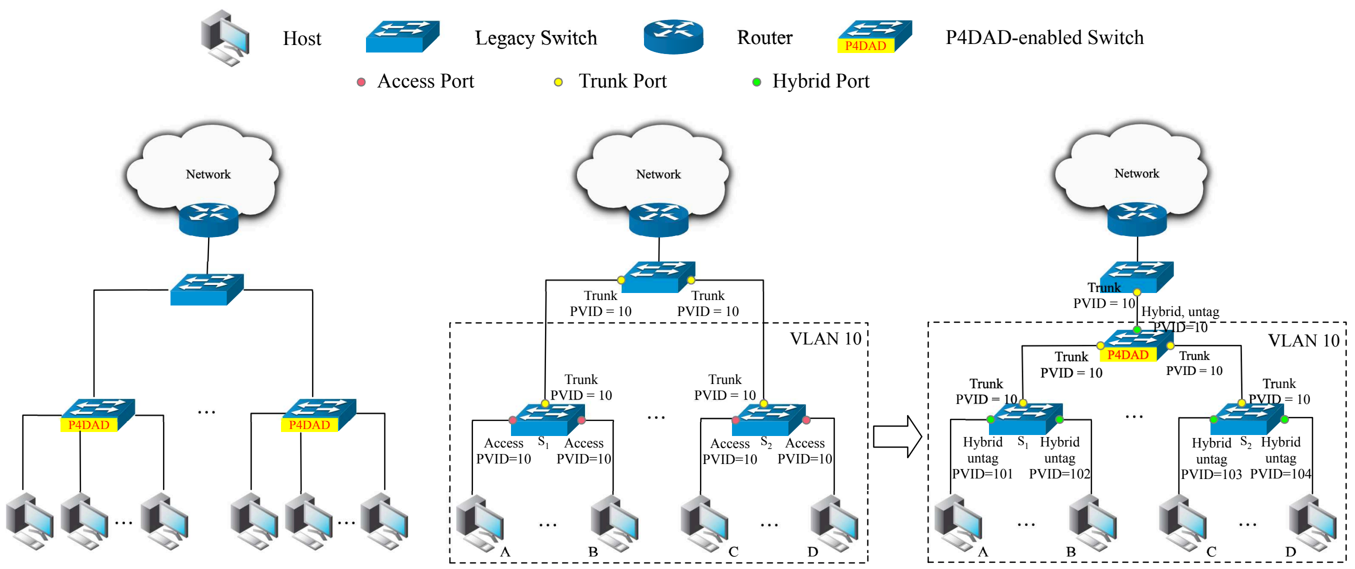
B. Deployment Issues

As P4-enabled devices may be expensive, we consider two deployment strategies for different scenarios:

1) First-Hop Level: This is an ideal deployment strategy. As shown in Fig. 9 (a), in this case, all the first-hop switches in a network are P4-enabled switches with the functionality of P4DAD, which can filter spoofed NS and NA messages. As for the aggregation level, a network can deploy legacy switches. This does not have an influence on the functionality of P4DAD.

2) Aggregation Level: In this scenario, we deploy a P4DAD-enabled switch for each subnet at aggregation level. The aggregation and first-hop switches do not have to be updated. A P4DAD-enabled switch is deployed between the legacy aggregation and first-hop switches. The network operator should update the configurations of each first-hop switch to help support P4DAD through virtual local area network (VLAN) classification. A deployment example is shown in Fig. 9 (b). The left sub-figure is a typical wired network topology. Generally, the number of aggregation switches are much less than that of first-hop switches. Only deploying a P4DAD-enabled switch for each subnet can save a lot of cost. In this example, there is a VLAN 10. If the network operator only deploys a P4DAD-enabled switch between the aggregation switch and first-hop switches to support P4DAD, the configurations of legacy switches should be updated as follows:

- **VLAN Classification**: All ports of the first-hop switches are configured with different VLANs (i.e., each with a unique PVID). We call them first-hop VLANs. It is assumed that the set of all the first-hop VLANs and the subnet VLANs is P.
- **Downlink Ports of First-Hop Switches**: A downlink port of a first-hop switch should be configured with a unique PVID in the subnet. The port is configured as an untagged port, and its port type is configured as *hybrid*. The port allows all the VLANs from P through.
- **Uplink Ports of First-Hop Switches**: An uplink port of a first-hop switch should be configured with the same VLAN as that of the subnet. Its port type is configured as *trunk*. The port allows all the VLANs from P through.
- **Downlink Ports of P4DAD-Enabled Switches**: A downlink port of a P4DAD-enabled switch should be configured with the same VLAN as that of the subnet. Its



(a) First-hop level deployment

(b) Aggregation level deployment

Fig. 9. Deployment Scenarios.

port type is configured as *trunk*. The port allows all the VLANs from P through.

- **Uplink Ports of P4DAD-Enabled Switches**: An uplink port of a P4DAD-enabled switch should be configured with the same VLAN as that of the subnet. Its port type is configured as *hybrid*. The port allows all the VLANs from P through.
- **Downlink Ports of Aggregation Switches**: An downlink port of an aggregation switch should be configured with the same VLAN as that of the subnet. Its port type is configured as *trunk*. The port only allows subnet VLAN through.

We use a case to demonstrate the effectiveness of this strategy. Take host A as an example. We discuss the VLAN tag of its upstream and downstream traffic.

- **Upstream Traffic**: When receiving a packet from host A, switch S_1 will add a VLAN tag (VLAN ID = 10) in the packet according to the policy of hybrid ports. As the uplink port of switch S_1 allows VLAN 101 through and the VLAN tag of the packet is different from VLAN 10, S_1 will forward the packet according to the policy of trunk port. Similarly, the P4 switch does not change the VLAN tag when the packet arrives at its downstream port. However, when the packet leaves the uplink port of the P4 switch, the VLAN tag will be removed due to its untagged attribute. Finally, the packet will be tagged with VLAN 10 at the downlink port of the legacy aggregation switch.
- **Downstream Traffic**: When the legacy aggregation switch forwards a packet, it first strips the VLAN information in the packet at its downlink trunk port. The P4DAD-enabled switch adds a VLAN tag (VLAN ID = 10) in the packet at its uplink port. Similarly, when the packet is forwarded to S_1, it is first untagged at the downlink port of the P4DAD-enabled switch and tagged with VLAN 10 at the uplink port of S_1. Finally, the packet is untagged at the downlink port of S_1 due to its untagged attribute and forwarded to host A.

As we can see, a packet sent from host A will be tagged with VLAN 101 at the P4DAD-enabled switch, but with VLAN 10 at the legacy aggregation switch. VLAN 101 is transparent to the outside subnet. Under this circumstance, the P4DAD-enabled switch can use the VLAN ID as the key to store the bindings between a VLAN and IPv6 addresses and filter spoofed NS and NA messages.

VIII. Conclusion

In this paper, we propose P4DAD which is a lightweight, deployable, robust, and secure DAD mechanism. By introducing P4, P4DAD secures DAD in the network without the modification of the NDP or host network stack. Also, no extra calculations are introduced to the host. By storing IPv6 addresses assigned to hosts locally and utilizing packet digests, P4DAD does not rely on a central controller and thus avoids single points of failure. To the best of our knowledge, P4DAD is the first secure DAD mechanism using P4. We implement a P4DAD prototype, and our evaluation results show that P4DAD can prevent DoS attacks on DAD effectively with negligible overhead, and has satisfactory scalability.

Acknowledgment

This work is supported by the National Key R&D Program of China (Grant No. 2018YFB1800405, 2018YFB1800402, and 2017YFB0801701), and National Natural Science Foundation of China (Grant No. 61772307).

References

[1] D. T. Narten, T. Jinmei, and D. S. Thomson, “IPv6 Stateless Address Autoconfiguration,” RFC 4862, Sep. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4862.txt

[2] W. A. Simpson, D. T. Narten, E. Nordmark, and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” RFC 4861, Sep. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4861.txt

[3] Statista, “Number of Internet of Things (IoT) Connected Devices Worldwide in 2018, 2025 and 2030,” https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/.

[4] H. Lamaazi, N. Benamar, A. J. Jara, L. Ladid, and D. El Ouadghiri, “Challenges of the Internet of Things: IPv6 and Network Management,” in *Proceedings of the Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE, 2014, pp. 328–333.

[5] A. J. Jara, D. Fernández, P. Lopez, M. A. Zamora, and A. F. Skarmeta, “Lightweight Mobile IPv6: A Mobility Protocol for Enabling Transparent IPv6 Mobility in the Internet of Things,” in *Proceedings of 2013 IEEE Global Communications Conference, GLOBECOM 2013*, Atlanta, GA, USA, 2013, pp. 2791–2797. [Online]. Available: https://doi.org/10.1109/GLOCOM.2013.6831497

[6] T. Savolainen, J. Soininen, and B. Silverajan, “IPv6 Addressing Strategies For IoT,” *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3511–3519, 2013.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming Protocol-Independent Packet Processors,” *Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[8] Barefoot Networks, “Barefoot Tofino,” Website, https://www.barefootnetworks.com/products/brief-tofino/.

[9] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, “NetLock: Fast, Centralized Lock Management Using Programmable Switches,” in *Proceedings of the 2020 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2020*. Virtual Event, USA: ACM, 2020, pp. 126–138. [Online]. Available: https://doi.org/10.1145/3387514.3405857

[10] K. Zhang, D. Zhuo, and A. Krishnamurthy, “Gallium: Automated Software Middlebox Offloading to Programmable Switches,” in *Proceedings of the 2020 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2020*. Virtual Event, USA: ACM, 2020, pp. 283–295. [Online]. Available: https://doi.org/10.1145/3387514.3405869

[11] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, “TEA: Enabling State-Intensive Network Functions on Programmable Switches,” in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 90–106.

[12] T. Mrugalski, M. Siodelski, B. Volz, A. Yourtchenko, M. Richardson, S. Jiang, T. Lemon, and T. Winters, “Dynamic Host Configuration Protocol for IPv6 (DHCPv6),” RFC 8415, Nov. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8415.txt

[13] D. Thaler, “Multi-Link Subnet Issues,” RFC 4903, Jun. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4903.txt

[14] A. K. Al-Ani, M. Anbar, S. Manickam, and A. Al-Ani, “DAD-Match: Technique to Prevent DoS Attack on Duplicate Address Detection Process in IPv6 Link-local Network,” *J. Commun.*, vol. 13, no. 6, pp. 317–324, 2018. [Online]. Available: https://doi.org/10.12720/jcm.13.6.317-324

[15] G. Yao, J. Bi, S. Wang, Y. Zhang, and Y. Li, “A Pull Model IPv6 Duplicate Address Detection,” in *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks, LCN 2010*, Denver, Colorado, USA, 2010, pp. 372–375. [Online]. Available: https://doi.org/10.1109/LCN.2010.5735746

[16] X. Wang, H. Cheng, and Y. Yao, “Addressing With an Improved DAD for 6LoWPAN,” *IEEE Communications Letters*, vol. 20, no. 1, pp. 73–76, 2016. [Online]. Available: https://doi.org/10.1109/LCOMM.2015.2499250

[17] S. Praptodiyono, I. H. Hasbullah, M. M. Kadhum, C. Y. Wey, R. K. Murugesan, and A. Osman, “Securing Duplicate Address Detection on IPv6 Using Distributed Trust Mechanism,” *Int J Simulation—Systems, Sci Technol*, vol. 17, no. 26, 2016.

[18] S. U. Rehman and S. Manickam, “Improved Mechanism to Prevent Denial of Service Attack in IPv6 Duplicate Address Detection Process,” *Int. J. Adv. Comput. Sci. Appl*, vol. 8, no. 2, pp. 63–70, 2017.

[19] D. Nelle and T. Scheffler, “Securing IPv6 Neighbor Discovery and SLAAC in Access Networks Through SDN,” in *Proceedings of the Applied Networking Research Workshop, ANRW 2019*, Montreal, Quebec, Canada, 2019, pp. 23–29. [Online]. Available: https://doi.org/10.1145/3340301.3341132

[20] S. U. Rehman and S. Manickam, “Rule-based Mechanism to Detect Denial of Service (DoS) Attacks on Duplicate Address Detection Process in IPv6 Link Local Communication,” in *Proceedings of 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*. IEEE, 2015, pp. 1–6.

[21] S. Scott-Hayward, G. O’Callaghan, and S. Sezer, “SDN Security: A Survey,” in *Proceedings of 2013 IEEE SDN for Future Networks and Services, SDN4FNS 2013*. Trento, Italy: IEEE, 2013, pp. 1–7. [Online]. Available: https://doi.org/10.1109/SDN4FNS.2013.6702553

[22] S. Scott-Hayward, S. Natarajan, and S. Sezer, “A Survey of Security in Software Defined Networks,” *IEEE Commun. Surv. Tutorials*, vol. 18, no. 1, pp. 623–654, 2016. [Online]. Available: https://doi.org/10.1109/COMST.2015.2453114

[23] I. Ahmad, S. Namal, M. Ylianttila, and A. V. Gurtov, “Security in Software Defined Networks: A Survey,” *IEEE Commun. Surv. Tutorials*, vol. 17, no. 4, pp. 2317–2346, 2015. [Online]. Available: https://doi.org/10.1109/COMST.2015.2474118

[24] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. A. Mujica, and M. Horowitz, “Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN,” in *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2013*, D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds. Hong Kong, China: ACM, 2013, pp. 99–110. [Online]. Available: https://doi.org/10.1145/2486001.2486011

[25] P. Kuang, Y. Liu, and L. He, “P4DAD: Securing Duplicate Address Detection Using P4,” in *Proceedings of 2020 IEEE International Conference on Communications, ICC 2020*. Dublin, Ireland: IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/ICC40277.2020.9149310

[26] A. Bas, “P4Runtime,” https://github.com/p4lang/p4runtime.

[27] M. Budiu and C. Dodd, “The P4$_{16}$ Programming Language,” *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, pp. 5–14, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3139645.3139648

[28] A. Bas, “The Second Version of the P4 Software Switch, BMv2,” https://github.com/p4lang/behavioral-model.

[29] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs,” in *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. Los Angeles, CA, USA: ACM, 2017, pp. 15–28. [Online]. Available: https://doi.org/10.1145/3098822.3098824

[30] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, “Generic External Memory for Switch Data Planes,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018*. Redmond, WA, USA: ACM, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3286062.3286063

Lin He is currently a postdoctoral researcher at Institute for Network Sciences and Cyberspace of Tsinghua University. He received his Ph.D. from Tsinghua University in 2019. His major research interests include network architecture and protocol design, Internet accountability and privacy.

Peng Kuang is currently pursuing his master degree at Institute for Network Sciences and Cyberspace, Tsinghua University, China. His major research interests include network architecture and protocol design.

Ying Liu received the MS degree in computer science and the Ph.D. degree in applied mathematics from Xidian University, China in 1998 and 2001 respectively. She is currently an associate professor at the Tsinghua University, China. Her major research interests include network architecture design, next generation Internet architecture, routing algorithm and protocol.

Gang Ren received the Ph.D. degree in computer system architecture from Tsinghua University, China in 2009. He is currently an associate professor at the Tsinghua University, China. His major research interests include network architecture design, next generation Internet architecture, and network security.

Jiahai Yang received the B.S. degree from Beijing Technology and Business University, the M.S. degree and Ph.D. degree from Tsinghua University, Beijing, China; all in Computer Science. He is a professor of the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China. His research interests include network management, network measurement, Internet routing and applications, cloud computing and big data applications. He is a member of IEEE & ACM.