CS 8903: Real-time Darknet Detection

Dhruv Rauthan

May 1, 2024

1 Introduction

A network telescope passively monitors traffic destined for Internet address space that is not assigned to any hosts but is advertised to the global routing system. Most modern network telescopes monitor statically assigned address spaces, and MORP4 is a novel 'dynamic' network telescope, which allows real-time changes to the unused dark address space. This special problem sets out to answer 3 major questions with respect to MORP4:

1. Comparison with existing alternatives: Can other firewall-based alternatives replicate MORP4's behavior? Are there any limitations?
2. Incorporating monitoring of IPv6 addresses: Can IPv6 address monitoring be integrated into MORP4? What is the most efficient way to do so?
3. Performance enhancement of the controller: Can MORP4's performance be improved for controller operations?

2 IPFW

2.1 What is *ipfw*?

ipfw is FreeBSD's free open-source stateful firewall with additional functionalities such as traffic shaping, packet scheduling and in-kernel NAT [1]. More specifically, it achieves keeping state of IP flows by using dynamic rules, and offers the option of logging packets which match a rule. It can log packets up to a user-defined threshold. We should also note that this firewall checks rules sequentially starting from the rule with the lowest sequence number. *ipfw* is a potential alternative for MORP4 which, however, has limited capabilities and requires modifications. We now briefly describe how *ipfw* works, the changes that we introduced to the module and how we would configure it to serve the same purpose as MORP4.

2.2 How does it work?

For each monitored prefix, we add a stateful rule in the *ipfw* ruleset defined in *ipfw.rules*. When *ipfw* observes an outgoing packet from an address within a monitored prefix, it first checks if the packet has an existing state in the *ipfw* dynamic state table. If such a state does not exist, then it creates a unique one using the 5-tuple (protocol, source address, destination address, source port and destination port) of the outgoing packet. Note that this matches both incoming and outgoing packets as they will have the same 5-tuple for a persistent connection. If however, a match is found, it executes the action associated with the parent rule which generated this dynamic rule, otherwise it moves to the next rule.

For example, in Listing 1, which is the *ipfw.rules* file that we used, the monitored prefixes are 11.0.0.0/24 and 12.0.0.0/24. A lower rule number is assigned to the rule containing the 'check-state' action, allowing *ipfw* to first check if the packet belongs to an existing dynamic state. If it doesn't find a match, it moves on to the next sequential rule number, which allows an outgoing packet to go through and simultaneously stores its state in the dynamic state table using the 'keep-state' action. The next time *ipfw* encounters an outgoing packet, matching one of the dynamic states, it will execute the action of the parent rule, which in this case is allowing the packet to go through. This stateful rule thus creates a new dynamic rule permitting traffic towards that address. If the packet is incoming, i.e, towards a monitored prefix, *ipfw* drops and logs the packet. Finally, the last rule allows all packets which are not to or from a monitored prefix.

```
#!/bin/sh
ipfw -q -f flush

cmd="ipfw -q add"
pif="em0"

# check state for existing connections
$cmd 0002 check-state

# outgoing packet
$cmd 0004 allow ip from 11.0.0.0/24 to any keep-state
$cmd 0005 allow ip from 12.0.0.0/24 to any keep-state

# new incoming packet
$cmd 0006 deny log ip from any to 11.0.0.0/24
$cmd 0007 deny log ip from any to 12.0.0.0/24

$cmd 1000 allow ip from any to any via $pif
```

Listing 1: The ipfw.rules file

2.3 What changes did we make?

We modify this behavior by editing the *ipfw* kernel source files so that now, instead of matching packets with the 5-tuple, the dynamic rule now matches packets according to only the destination address (for an incoming packet) or source address (for an outgoing packet) instead. This required modifications to the dynamic state addition and lookup methods in *ip_fw_dynamic.c*. As *ipfw* calculates and stores the hash of the 5-tuple, changes were made so that now the hash is only calculated according to the single stored address.

Additionally, we reset the dynamic rule expiry timer to T_o when an outgoing packet is observed, by changing the variables handling the lifetime of dynamic states. When the timer for a state expires, that dynamic state is deleted from the state table, meaning that the address is considered inactive again. When inactive, any packets destined towards that address will be logged according to the lower priority rule set earlier in Listing 1.

Testing the correctness of this implementation was done using Scapy, by generating and sending packets for 4 cases, as listed in Table 1.

Flow of packets	**Expected behavior**
Outgoing packet from an IP address belonging to a monitored prefix Then an incoming packet to the same IP address.	Not logged
Outgoing packet from an IP address belonging to a monitored prefix Then an incoming packet to another IP address in the same prefix.	Not logged
Outgoing packet from an IP address belonging to a monitored prefix Then an incoming packet to an IP address in another monitored prefix.	Logged
Incoming packet to an IP address in a monitored prefix.	Logged

Table 1: Different scenarios with *ipfw*

These described rule expiry and logging mechanisms are similar to the ones implemented in MORP4.

2.4 Drawbacks

This implementation has a few drawbacks.

1. **IP Spoofing** This solution is not resilient to IP spoofing attacks. A malicious actor could send a packet with a spoofed source address within a monitored prefix to invoke the creation of a dynamic state, which would permit all inbound traffic towards that address.

2. **Limited dynamic states** *ipfw* stores a structure corresponding to each dynamic rule in a hash table. The number of buckets for each entry in the table is set to 8192 by default, and drastically increasing it would result in memory concerns due to under-utilization of entries in the table. Alternatively, we keep the number of buckets as the default, which would result in very long lists associated with each bucket, degrading performance during lookup. Both methods have clear drawbacks, since in the worst case we are looking at storing rules for all the addresses in each monitored prefix.

3. **Multiple ingress/egress paths** In case a network has multiple ingress or egress points, this implementation becomes even more complicated. According to MORP4's implementation, when a switch processes an outgoing packet that updates the state of an address to active for the current time bin, it creates a special control packet to send to all the other switches so that they apply the same modification. This operation is executed at line-rate using the cloning and multicast mechanisms of the programmable switches. However, *ipfw* does not offer such functionalities, as it is a straightforward firewall mechanism used to filter and log packets according to user defined rules. Hence, we would need to add another custom module for the notification procedure. We could build a controller script that regularly checks the dynamic rules of *ipfw* and upon detection of a new rule, it creates and sends a control packet to each other deployment of *ipfw* in the network to indicate that that address is active. This is not an optimal solution, since even if the controller retrieves the *ipfw*'s rules frequently, there would still be much greater delay in notifying the other *ipfw*'s compared to MORP4 which multicasts control packets at line rate. That could lead to unwarranted packet logging and consequently privacy violation of the network users.

2.5 Discussion

MORP4 definitely achieves higher accuracy and speed than an *ipfw*-based alternative telescope, especially with respect to user data privacy. Additionally, the *ipfw* solution requires significant changes to the kernel configuration files, making it a comparatively worse choice to dynamically detect dark addresses in a network.

3 IPv6

Currently, MORP4 can only store upto 2^{22} addresses in each register array (table), allowing monitoring of only 2^{22} addresses at a time. For a /10 prefix, this means that all addresses can be monitored (upto /32 granularity). However, these Tofino switch storage limitations means that for a /32 prefix, IPv6 addresses can only be stored for a maximum of /54 granularity. For Tofino-2 the maximum size of a stateful object decreases, allowing us to have multiple tables of smaller sizes. We increase the number of tables by 4, allowing monitoring at a maximum granularity of /56.

3.1 Bloom Filters

Bloom filters are space efficient probabilistic data structures used to check for set membership. A bloom filter will always return yes if an item is a set member. However, the bloom filter might still return yes although an item is not a member of the set (a false positive). Bloom filters only support addition and set membership testing, they do not allow deletion of elements. Ideally the false positive rate should be as low as possible as we do not want to log legitimate user packets in MORP4.

For sets with large number of members, such as IPv6 addresses, bloom filters have an extremely high false positive rate, as shown in Figure 1. If only 0.1% of 2^{32} addresses are active at any point in the network, the false positive rate is close to 30% which is undesirable.

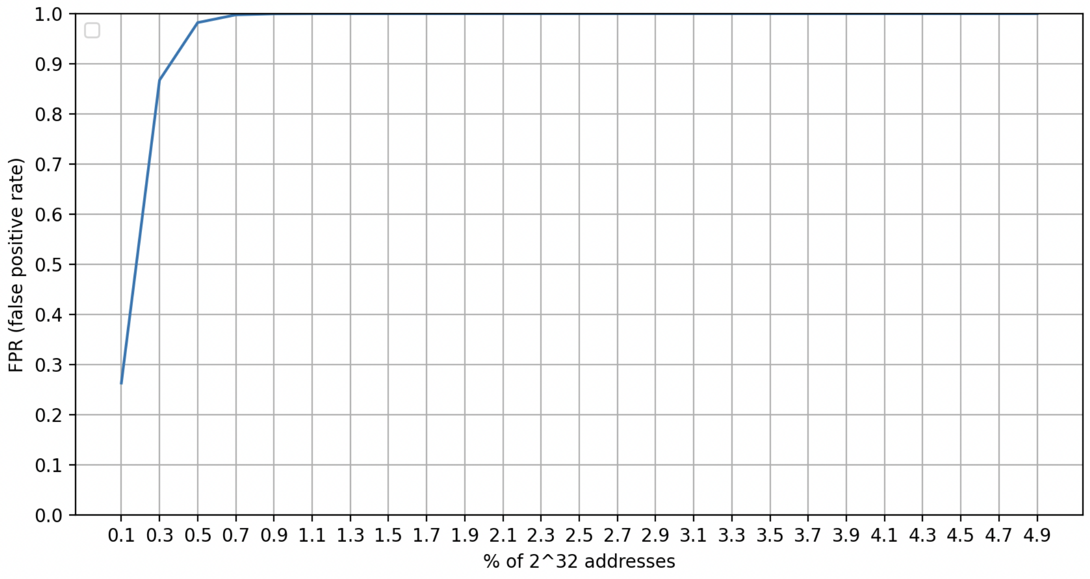

Figure 1: False positive rate for Bloom Filter storing IPv6 addresses

Furthermore, due to the lack of deletion of individual elements for bloom filters we cannot remove dark addresses without resetting the entire bloom filter. Again, this behaviour not optimal as we do not want to classify other dark addresses as active when the status of a particular address changes from dark to active.

3.2 Cuckoo filters

Cuckoo filters are another probabilistic data structure used to check whether an element is a part of a set or not. Similar to bloom filters, cuckoo filters have the possibility of false positives but do not allow any false negatives, i.e, they do not classify any active addresses as dark. Cuckoo filters have an advantage over bloom filters as they also support deletion of items from the set.

However, cuckoo filters only perform better than bloom filters for a few billion entries or fewer [2]. Aiming to store 2^{34} entries would result in a high false positive rate as shown in Section 3.1.

4 C++

The current MORP4 controller code is written in Python. Python, being an interpreted language, is significantly slower as compared to a compiled language like C++. Converting the existing controller code to C++ would lead to performance improvements when retrieving and updating the status of addresses in the Tofino switches. This was done using BareFoot Runtime APIs, which provided libraries for table operations.

4.1 What changes did we make?

In addition to the classes below, we add functionalities for parsing command-line arguments and setting the initial parameters of the controller accordingly. The **main** method initializes the switch context variables, BFRT session variables, and connects to the switch software via gRPC. When the program finishes running, it frees up the allocated memory and stops the session.

4.1.1 Register

This class defines a register data structure to store register arrays defined in the switch. Both *global_table* and *flag_table* are defined as an instance of this class. The class methods are described as:

- **read_from_sw**: Enables reading from Tofino software.
- **start_sync**: Starts register synchronization.
- **end_sync**: Ends register synchronization.
- **reg_sync_cb**: Synchronizes register callback.
- **read_data(start index, end index)**: Reads data from the register array between 2 indices and returns a 2-dimensional vector array.
- **write_data(list of keys, value)**: Sets a particular value for all the passed keys (register indices).

4.1.2 MonitoredTable

It defines the *monitored* table which contains the list of monitored prefixes defined by the network operator. It has only 1 method, which is used to add the monitored prefixes to the switch table:

- **add_entry(prefix, length, base index, mask, dark base index)**: Converts the prefix and length strings to 32 bit integers, sets them as the key for the table and adds the other values as the data value to the table.

4.1.3 PortsTable

This class is used to define an instance of the *ports* table, which sets the incoming and outgoing ports of the switch. The class has 2 methods:

- **set_action_id(direction)**: Sets the table's action ID as incoming or outgoing depending on the argument passed.

- **add_entry(port, direction)**: Adds the port as the key and the empty action ID according to the direction to the ports table.

4.1.4 Meter

Meters are used for measuring and controlling the rate of incoming traffic, and are helpful in rate limiting logging in Tofino. This class is used for defining 2 instances, *dark meter* and *dark global meter*, and has the following method:

- **add_entry(average packet rate, maximum packet rate, index)**: Sets the key value as the index, and the average packet rate, maximum rate, average allowed as 100 and maximum allowed as 100 as the data for the meter table entry.

4.1.5 LocalClient

This is largest and most important class in the program. It is used to define instances of all the classes mentioned previously, and executes methods associated with each class accordingly. It handles the entire logic of the controller. A few major functions are described as:

- **setup**: Initializes instances of all the classes, and performs basic operations such as populating the monitored table and setting the packet rates.

- **parse_monitored(file path)**: Takes the file path where the network operator has defined the prefixes they want to monitor, opens the file and returns the list of monitored prefixes as strings.

- **populate_monitored(list of prefixes)**: Iterates all entries in the monitored prefixes. It extracts the mask, length and the monitored prefix string, and converts them into integers. These values along with the current base index are added to the *monitored* table. All IP addresses in this prefix are then added to a list which maintains the index to prefix mapping. The base index is incremented according to the number of IP addresses in that prefix.

- **add_ports(ports)**: Adds the incoming and outgoing ports to the *ports* table accordingly per the map passed to it.

- **set_rates**: Simply sets the dark global meter rates.

- **update_rates(inactive prefixes, inactive addresses)**: Calculates the average and maximum packet rates and adds an entry for that inactive prefix/address in the dark meter.

- **run**: The 'brain' of the program, this method runs as a loop every T_o period and performs logical operations relating to the updation of *global table* and *flag table* in the switch. To improve performance, registers are updated in batches of 10^5 at a time.

4.2 Discussion

Porting the codebase from Python to C++ increased performance significantly and reduced request times. With Python, it took 5 minutes for an iteration to complete. This was reduced to 40 seconds with C++, an **86%** improvement. MORP4 operators can now read and write switch tables and register entries much faster than before, allowing faster updates to the status of addresses as 'dark' or 'active'.

5 Conclusion

Throughout the course of the special problem, we worked on enhancing MORP4's features and performance. Potential alternatives to MORP4, such as *ipfw* were implemented and compared to the existing solution. We identified difficulties in the implementations as well as potential drawbacks of the alternative. For incorporating IPv6 addresses, we showed that probabilistic data structures, bloom and cuckoo filters, would be impossible to implement while maintaining a low false positive rate for identification of dark addresses. Finally, we programmed the controller in C++, which significantly improved MORP4's performance overall.

References

[1] Ipfw description. `https://docs.freebsd.org/en/books/handbook/firewalls/#firewalls-ipfw`. Accessed: 2024-04-29.

[2] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.