



**POLITECNICO
DI TORINO**

POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Network attack detection in Software Defined Networks.

Supervisors

Prof. Paolo GIACCONE

Candidate

Alexey SHCHEDRIN

ACADEMIC YEAR 2017-2018

Abstract

Nowadays, the DDoS attack detection in traditional or Software Defined Network(SDN), is an area of active scientific research. In this work we evaluate amount of information that need to be transferred between controller and network devices in order to successfully detect a DDoS attack in OpenFlow SDN, and proposed a way to minimize this amount, using means of P4 programming language.

Contents

1	Overview	1
1.1	Networking paradigms	1
1.1.1	Distributed networking	1
1.1.2	SDN	2
1.1.3	Stateful SDN	6
1.2	P4	7
1.2.1	Overview	7
1.2.2	Table pipeline model	9
1.2.3	P4 Runtime	12
1.2.4	INband Telemetry	14
1.3	Open Network Operating System (ONOS)	16
1.3.1	Overview	16
1.3.2	Internal Structure	17
1.3.3	ONOS Applications	19
2	Distributed Denial of Service attacks	23
2.1	General description	23
2.2	General Classification	23
2.3	Detection methods	25
2.3.1	Anomaly detection methods	25
3	Experimental testbed	29
3.1	Tools	29
3.1.1	Mininet	29
3.1.2	BMV2	29
3.1.3	Open vSwitch	30
3.1.4	Traffic generators	30
3.2	Testing topology	31
3.3	General implementation components	33
3.4	P4Runtime implementation	35
3.5	ONOS internal application	37

4	Experimental evaluation	43
4.1	P4 Runtime implementation	43
4.2	ONOS internal application	48
4.3	Reference solution	53
4.4	Comparison	56
5	Conclusion	59
	Bibliography	61

Chapter 1

Overview

1.1 Networking paradigms

1.1.1 Distributed networking

The development of the Internet has created a digital society, where almost everything connected with everything and can be accessed from anywhere. The distributed network protocols that work inside network devices allow the information (in form of digital blocks, called packets), to reach any part of the world. But, the underlying technology, traditional IP networks, is very complex and hard to control. To create a network behavior in accordance with desired policies, network operators have to configure every individual switch or router separately, using low-level and mostly vendor-specific commands. In addition to this, network operators need to adapt their networks to faults and load changes, using almost non-existent in IP-networks mechanisms of automatic fault response and reconfiguration. To make situation even more complicated, traditional IP networks are vertically integrated. The control plane (that decides how to handle network traffic) and the data plane (that executes traffic forwarding according to the decisions made by the control plane) are bundled inside the traditional switches and routers, lowering flexibility and making very difficult any innovation and evolution of the networking infrastructure. As an example, we can remember the transition from IPv4 to IPv6, which started long time ago and still uncompleted, despite the fact, that this transition is just a protocol upgrade. Computer networks can be divided in three planes of functionality: the data, control and management planes (see Figure 1.1).

The data plane corresponds to the network devices, or components of devices, responsible for data forwarding. The control plane contains all the protocols used to manage forwarding tables of the data plane components. The management plane includes software services, for example SNMP-tools, responsible for remote control and monitoring of the control functionality. From the network-policy point of view, network policy is defined on the management plane, enforced by the control plane, and executed by the data plane. Control and data planes are coupled in traditional IP networks, usually they placed inside the same device, making network structure highly decentralized. This decentralization was considered as a very important design goal at the beginning of Internet - it guaranteed

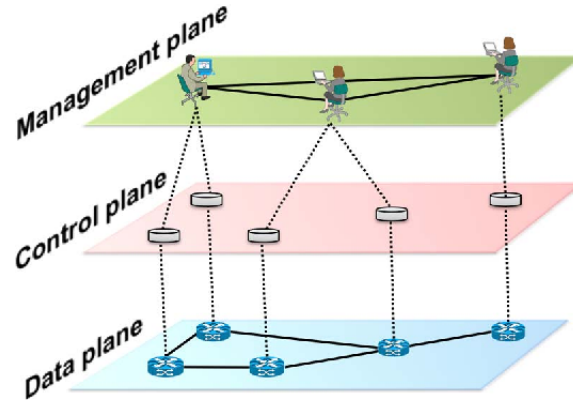


Figure 1.1. Layered view of networking functionality [1]

network resilience. And indeed, it worked very efficient, but as the outcome gave us very complex and almost static architecture, which is complex and hard to control. Network misconfigurations are very common in modern days, leading undesired network behavior, including packet losses, forwarding loops, unintended paths and other errors. In order to support network management, vendors offer proprietary management solutions of specialized hardware, operating systems and network applications. Because of proprietorships of these solutions, networks operators have to maintain separate solution for equipment of each vendor, and separate supporting teams, which increases capital and operational costs of the networks. Also, to solve problem of lacking functionality withing the network, network operators have to use middle-boxes, such as firewalls, intrusion detection systems, load-balancers and others, number of which is comparable, even if not larger, than number of basic network devices. Presence of these middle-boxes increases complexity and decreases flexibility of network design and operations.

1.1.2 SDN

Software-Defined Networking (SDN) is a relatively new paradigm aimed to solve the limitation of traditional IP networks. First, it solves the vertical integration problem by separating control logic (network control plane) form underlying hardware, responsible for data forwarding. Second, control logic is moved to a logically centralized controller (or network operating system), simplifying network configuration and policy enforcement. A simplified view of this architecture is shown in Figure 1.2. Important to mention, that logically centralized model do not signifies physically centralized system. Instead, industry SDN controllers has physically distributed control planes, keeping, nevertheless, centralized logical structure.

The separation of the control plane and the data plane requires well-defined API (Application Programming Interface) between switches and SDN controller, as depicted on the Figure 1.2. The most recognizable of these interfaces at the moment is OpenFlow. According to it OpenFlow switch can have one or several tables of packet-handling rules (flow

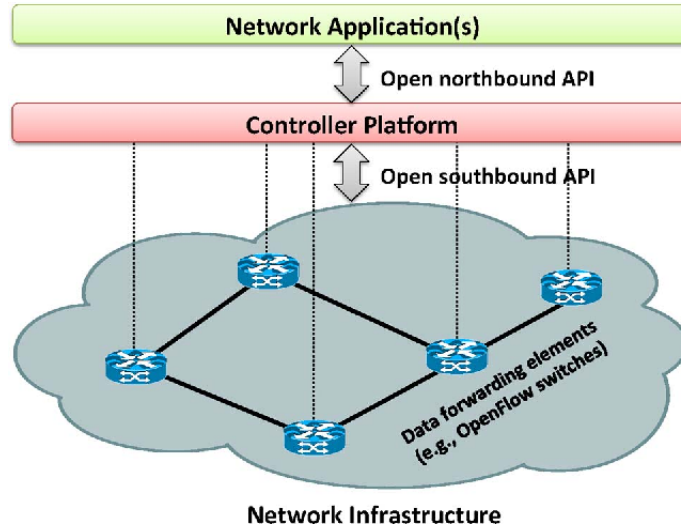


Figure 1.2. Simplified view of an SDN architecture. [1]

tables), each of them matches a subset of traffic and can perform actions like dropping, forwarding, modifying on the incoming packets.

With different sets of rules, installed by controller, an OpenFlow switch can behave like a router, switch, firewall, load balancer, traffic shaper, and any other middlebox. In result of application software-defined networking principles can be separated concerns of defining network policies, their implementation in hardware, and traffic forwarding. This separation is the key to desired flexibility, because network control is now broken into tractable pieces, which allows us to create new abstractions in networking, simplify network management and introduce innovations.

The term SDN (Software-Defined Networking) was originally coined to represent the ideas and work around OpenFlow at Stanford University [2]. Original definition of term SDN refers to a network architecture where the forwarding state of dataplane is managed by a remote control plane, separated from the former. The networking industry, in many cases, shifted from original view, calling SDN anything involves software. In this work we will refer SDN as described in [1], as a network architecture with four pillars:

1. The control and data planes are decoupled. Control functionality is removed from network devices that will become simple (packet) forwarding elements.
2. Forwarding decisions are flow-based, instead of destination-based. A flow is broadly defined by a set of packet field values acting as a match (filter) criterion and a set of actions (instructions). In the SDN/OpenFlow context, a flow is a sequence of packets between a source and a destination. All packets of a flow receive identical service policies at the forwarding devices. The flow abstraction allows unifying the behavior of different types of network devices, including routers, switches, firewalls, and middleboxes. Flow programming enables unprecedented flexibility, limited only to the capabilities of the implemented flow tables.

3. Control logic is moved to an external entity, the so-called SDN controller or Network Operating System (NOS). The NOS is a software platform that runs on commodity server technology and provides the essential resources and abstractions to facilitate the programming of forwarding devices based on a logically centralized, abstract network view. Its purpose is therefore similar to that of a traditional operating system.
4. The network is programmable through software applications running on top of the NOS that interacts with the underlying data plane devices. This is a fundamental characteristic of SDN, considered as its main value proposition.

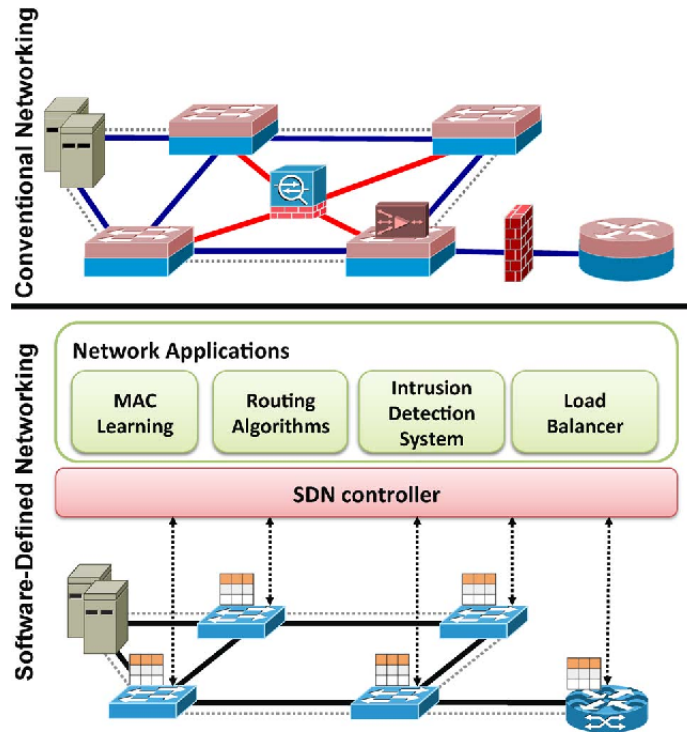


Figure 1.3. Traditional networking versus Software-Defined Networking (SDN). With SDN, management becomes simpler and middleboxes services can be delivered as SDN controller applications. [1]

Notably, the logical centralization of the control logic, has several additional benefits. First, it allows to modify network policies in simpler and less error-prone way by means of high-level languages and software components, comparing to low-level vendor specific device configurations. Second, a control program can automatically react to changes of the state of network and maintain the high-level policies in any circumstances, including device/link outages and spikes of the data traffic. Third, the centralization of the control logic in a controller gives global knowledge of the state of the network, and therefore simplifies the development of sophisticated networking functions, services and applications.

Following the SDN concept introduced in [3], an SDN can be defined by three fundamental abstractions: (i) forwarding, (ii) distribution, and (iii) specification.

- The forwarding abstraction should allow any forwarding behavior desired by the control program (network application), in the same time hiding details of the underlying hardware. OpenFlow is one realization of such abstraction, which can be considered as an equivalent to a "device driver" in an operating system.
- The distribution abstraction should protect SDN applications from the problems of distributed state, making the distributed control problem a logically centralized one. Its realization requires a common distribution layer, which in SDN resides in the NOS. This layer has two essential functions. First, it is responsible for installing the control commands on the forwarding devices. Second, it gathers information about the forwarding layer (network devices and links), to offer a global network view to network applications.
- The specification abstraction, allows a control program (network application) to express the desired network behavior without about implementation of that behavior. This goal can be reached with virtualization solutions, or network programming languages.

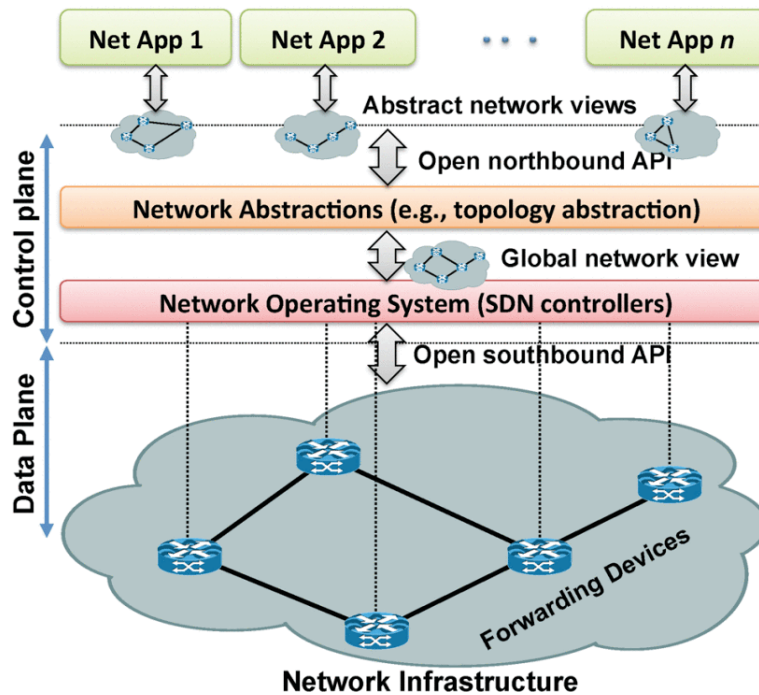


Figure 1.4. SDN Abstractions. [1]

These abstractions allows us to convert the abstract configurations that the applications express based on a simplified, abstract model of the network, into a physical configuration for the global network view exposed by the SDN controller. Figure 1.1.2 depicts the SDN architecture, concepts and building blocks. As was mentioned before, the tight binding between control and data planes makes the deployments of new networking features (for example, routing protocols) very hard, due to the fact, that we need to modify all networking devices. Therefore, as was mentioned before, new features are introduced mostly by means of new middleboxes, which are, in turn, vendor dependent and hard to configure. Resulting topology becomes more complex, and creates additional hurdles in implementing new functionality. In contrast, SDN separates the control plane from the network devices and becomes an external entity: the network operating system or SDN controller. This approach has several advantages:

- Provides a simple and cheap network testbed for developing SDN applications. It becomes easier to program these applications because the abstractions of the control platform and network programming languages can be moved and shared.
- All applications can use the same information (the global network view), making more effective policy decisions using control plane software modules.
- These applications can take actions (i.e., reconfigure forwarding devices) from any part of the network, eliminating problem of choosing the location for the new functionality.
- The integration of different applications becomes more straightforward. For example, load balancing and routing applications can be used simultaneously, with load balancing decisions having precedence over routing policies.

1.1.3 Stateful SDN

The basic innovation of SDN is the separation between control and data plane. With OpenFlow this separation is physically represented by "dumb" switches processing match/action rules (flow entries) installed by "smart" controllers. The controller is able to change dynamically these rules, rewriting them in order to react to changes of the state of the network. This approach allows simple network programming, and simplified view of the network as a "big switch", but introduces some disadvantages. In particular, there is a relatively long processing delay, caused by information exchange between switch and controller, which can be critical in cases of, for example, network outages or traffic management applications.

One of the possible ways of the advancement of the OpenFlow, according to some recent proposals [4] [5], is to introduce switch-driven adaptation of forwarding rules, based on the switch-local events. One of the proposals, OpenState [6], is the evolution of OpenFlow abstraction, which allows to introduce "stateful" dataplane with minimal changes of the existing OpenFlow standard. OpenState, is an OpenFlow extension that introduces the idea of delegating some control functions back to the devices, keeping the central SDN controller posted and fully in control of all delegated operations. The motivation of OpenState is to move simple control tasks, that require only switch-local knowledge, out of the

controller, nevertheless keeping controller responsible only for decisions requiring network-wide knowledge. With OpenState, custom states may be configured inside the switch, and may be programmed to evolve themselves, triggered by packet arrivals, measurements and timers.

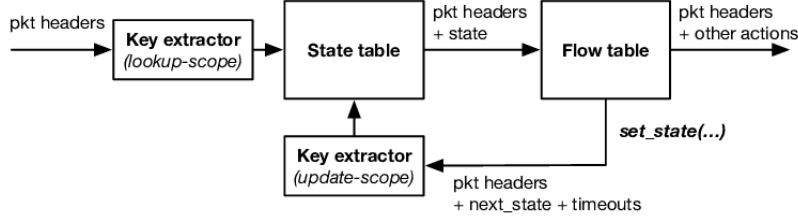


Figure 1.5. Simplified packet flow in OpenState. [7]

On the Figure 1.5 we can see proposed structure of the OpenState. State table is put first in the switch pipeline, and store "flow states". Every packet, arriving to the switch, is processed first by this table, by the user-programmed key (combination of values in protocol fields of the packet). After the processing, packet receives "state" metadata, which informs following packet processing stages about the "state" of the flow to which belongs this packet. Also, next processing stages sends back to the state table some information about processing results of the packet (for example, number of processed bytes in this flow), which can trigger user-programmed state changes.

In this work we use more radical solution to switch programmability limitations which was proposed in [5]: P4. P4 is a high-level language to program packet processors which focuses on protocol-independence.

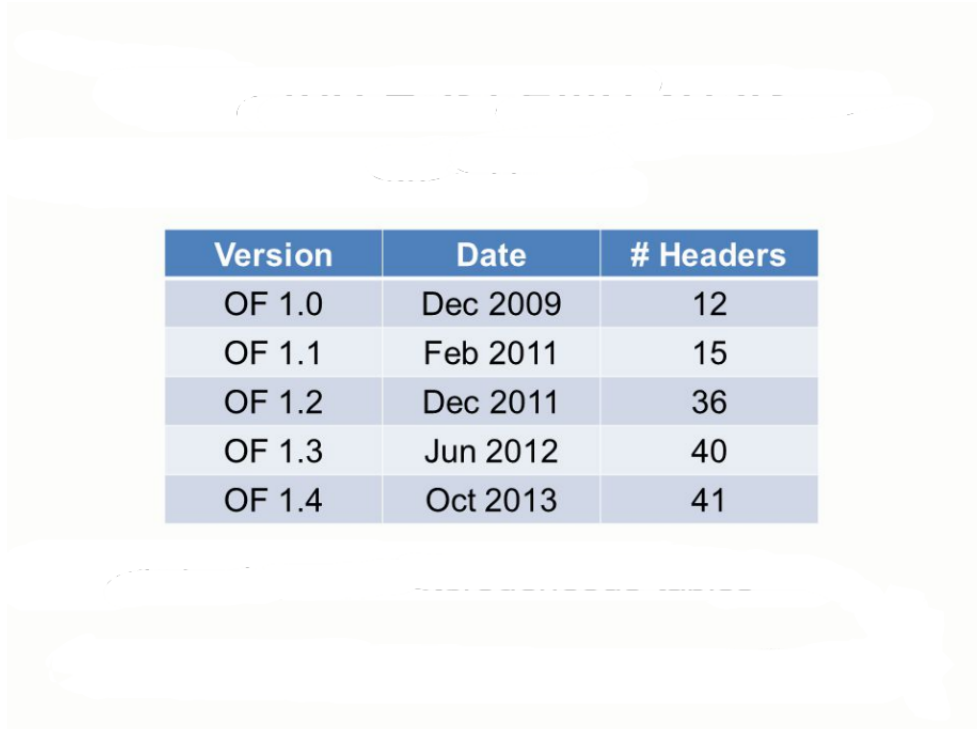
1.2 P4

1.2.1 Overview

Software-Defined Networking (SDN) has given network operators programmatic control over their networks. In SDN, the control plane is logically (and physically) separate from the forwarding plane, and one control plane controls multiple forwarding devices. The fact, that forwarding devices could be programmed in many ways, still having a common, open and vendor-independent interface (like OpenFlow) allows a control plane to control forwarding devices from different hardware and software vendors.

The OpenFlow interface started simple, having a single table of rules that could match packets on a small number of header fields (e.g., MAC addresses, IP addresses, protocol, TCP/UDP port numbers, etc.). Over the past five years, the specification has grown increasingly more complicated (see Figure 1.6), with much more header fields and multiple stages of rule tables, to allow switches to expose more advanced capabilities to the controller.

The growth of supported header fields doesn't show signs of slowdown. For example, data-center network operators need to add into OpenFlow protocol new forms of packet



Version	Date	# Headers
OF 1.0	Dec 2009	12
OF 1.1	Feb 2011	15
OF 1.2	Dec 2011	36
OF 1.3	Jun 2012	40
OF 1.4	Oct 2013	41

Figure 1.6. OpenFlow Development. [5]

encapsulation (e.g., NVGRE [8], VXLAN [9], and STT [10]), for which they resort to deploying more flexible software switches that are easier to extend with new functionality. Rather than extending the OpenFlow specification, it is better to make future switches to support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to use these capabilities through a common, open interface. Such a general, extensible interface is going to be simpler, more elegant, and more future-proof than today's OpenFlow 1.x standard.

Modern chip designs demonstrate that such flexibility can be reached in custom ASICs at terabit speeds. But programming this new generation of switch chips is far from easy. Each chip has its own low-level interface, vendor-specific, and often on the level of microcode programming. In an article [5] was proposed a sketch the design of a higher-level language for **Programming Protocol-independent Packet Processors (P4)**. Figure 1.7 shows the relationship between P4-used to configure a switch, telling it how packets are to be processed - and existing APIs (such as OpenFlow) that are designed to populate the forwarding tables in fixed function switches. According to the article [5] authors opinion P4 can raise the level of abstraction for programming the network, and can serve as a general interface between the controller and the switches. Also P4 allows SDN controller to program switch operations on the switch itself, making controller independent of the fixed function design of OpenFlow switches, and by this advances flexibility of operations

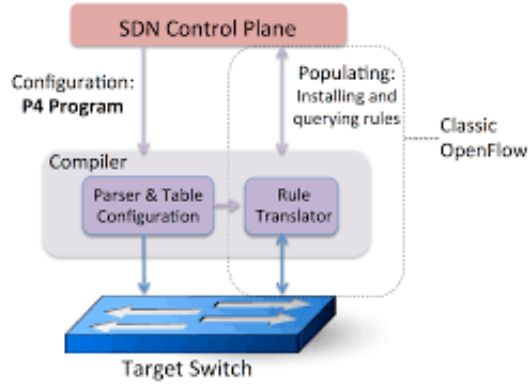


Figure 1.7. Proposed P4 position in OpenFlow framework [5]

of the network. In design of P4, authors put three goals:

- **Reconfigurability.** The controller should be able to redefine the packet parsing and processing of already deployed switches, right in the field.
- **Protocol independence.** The switch should not be tied to specific packet formats. Instead, the controller should be able to specify
 - a packet parser for extracting header fields with particular names and types
 - a collection of typed match + action tables that process these headers
- **Target independence.** As a C programmer does not need to know the specifics of the CPU, the controller programmer should not need to know the details of the switch. Instead, a vendor-supplied compiler should take care about the switch’s capabilities during compilation of a target-independent description, written in P4, into a target-dependent program (used to configure the switch).

1.2.2 Table pipeline model

P4 is a language which describes how packets are processed on the data plane of a programmable forwarding element such as a hardware or software switch, network interface card, router, or network appliance. P4 was initially designed for programming switches, but its scope has enlarged to cover a variety of devices, such as ASIC, FPGA and generally any type of device that implements both a control plane and a data plane functionality. But in this paper we will refer all possible P4-programmable devices as a switch.

P4 is designed to specify only the data plane functionality of the target. As a concrete example, Figure 1.8 illustrates the difference between a traditional fixed-function switch and a P4-programmable switch. In a traditional switch the manufacturer strictly defines the data-plane functions. The control-plane controls these functions by managing entries in tables (i.e routing or sitching tables), configuring specialized objects (e.g. meters),

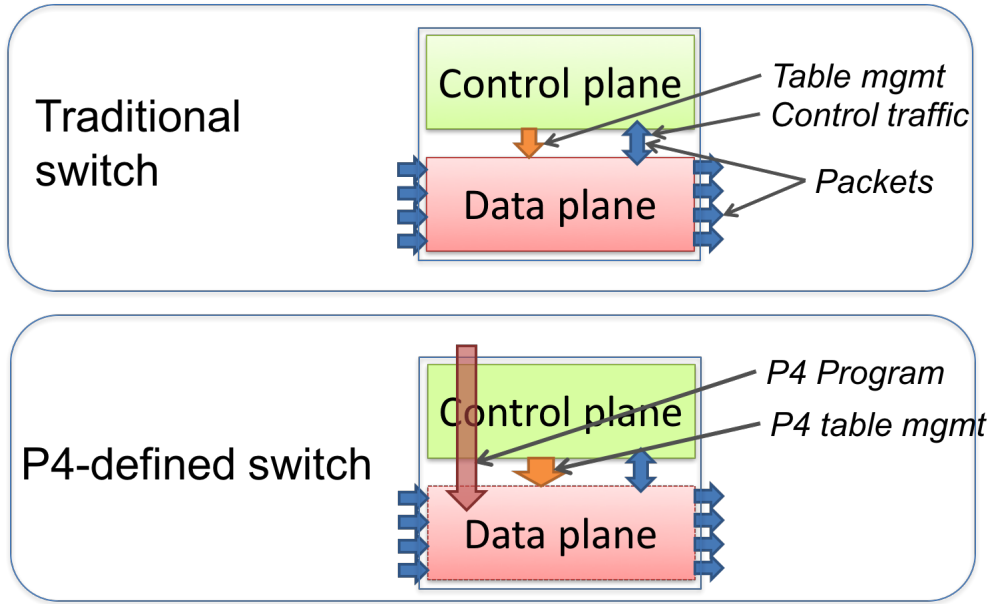


Figure 1.8. Difference between traditional and programmable switches [11]

and by processing control-packets (e.g. routing protocol packets) or other events, like link state changes or learning notifications. A P4-programmable switch differs from a traditional switch in two essential ways:

1. The data plane functions are not fixed in advance, now they are defined by the a P4 program. The data-plane is configured at device initialization to implement the functionality described by the P4 program (shown by the long red arrow) and has no built-in knowledge of existing network protocols.
2. The control plane communicates with the data plane using the same structure as in a fixed-function device, but the set of tables and other objects in the data plane are no longer fixed, since they are defined by a P4 program. The P4 compiler generates the API that the control plane uses to communicate with the data plane.

The core abstractions provided by the P4 language are: [11]

1. Header types describe the format (the set of fields and their sizes) of each header within a packet.
2. Parsers describe the permitted sequences of headers within received packets, how to identify those header sequences, and the headers and fields to extract from packets.
3. Tables associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multi-variable decisions.

4. Actions are code fragments that describe how packet header fields and metadata are manipulated. Actions can include data, which is supplied by the control-plane at runtime.
5. Match-action units perform the following sequence of operations:
 - (a) Construct lookup keys from packet fields or computed metadata,
 - (b) Perform table lookup using the constructed key, choosing an action (including the associated data) to execute, and
 - (c) Finally, execute the selected action.
6. Control flow expresses an imperative program that describes packet-processing on a target, including the data-dependent sequence of match-action unit invocations. Deparsing (packet reassembly) can also be performed using a control flow.
7. Extern objects are architecture-specific constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hard-wired (e.g., checksum units) and hence not programmable using P4.
8. User-defined metadata: user-defined data structures associated with each packet.
9. Intrinsic metadata: metadata provided by the architecture associated with each packet, for example, the input port where a packet has been received.

Figure 1.9 shows a typical set of tools used while programming a switch using P4. Switch manufacturer should provide the hardware or software implementation framework, an architecture definition, and a P4 compiler for that switch. P4 programmer writes programs for a specific architecture, which defines a set of P4-programmable components on the switch as well as their external data plane interfaces.

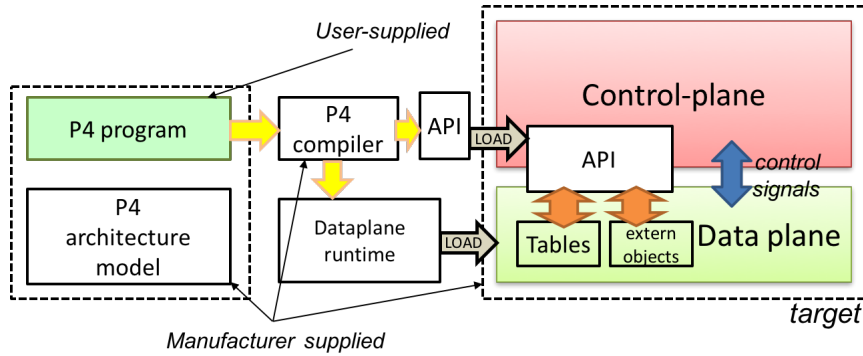


Figure 1.9. Programming a switch with P4 [11]

Compiling a set of P4 programs produces two artifacts:

1. device-specific data plane configuration to download into device, that implements the forwarding logic described in the input P4 program (example: JSON file for BMV2 software switch)

2. an API for managing the state of the data plane objects from the control plane (info file for the P4 Runtime)

Compared to state-of-the-art packet-processing systems (e.g., based on writing microcode on top of custom hardware), P4 has a number of significant advantages:

1. Flexibility: P4 makes expresses packet-forwarding policies as programs, in contrast to traditional switches, which expose fixed-function forwarding engines to their users.
2. Expressiveness: P4 can express complex, hardware-independent packet processing algorithms using only general-purpose operations and table look-ups. Such programs are portable across devices that implement the same architectures (assuming sufficient resources are available).
3. Resource mapping and management: P4 programs describe storage resources abstractly (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage low-level details such as allocation and scheduling.
4. Software engineering: P4 programs provide important benefits such as type checking, information hiding, and software reuse.
5. Component libraries: Component libraries supplied by manufacturers can be used to wrap hardware specific functions into portable high-level P4 constructs.
6. Decoupling hardware and software evolution: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing.
7. Debugging: Manufacturers can provide software models of an architecture to aid in the development and debugging of P4 programs.

1.2.3 P4 Runtime

P4 Runtime is an API, introduces new way by which control plane software is able to control the forwarding plane of a switch, router, firewall, load-balancer, and any other type of network device. Arguably, the most interesting aspect of P4 Runtime is that it allows to control any forwarding plane, independently of whether it is built from a fixed-function or programmable switch ASIC, an FPGA, NPU or a software switch running on an x86 server. The framework of P4 Runtime stays unchanged, being independent of forwarding plane capabilities, whatever protocols and features the forwarding plane supports. The same API can be used to control a huge variety of different switches. More than this - when new protocols and features are added to the forwarding plane, the P4 Runtime API is automatically updated by changing the control scheme to describe how a new feature is to be managed, without restarting or rebooting the control plane. P4 Runtime is independent of where the control plane is located; the control plane could be a protocol stack running on a local switch operating system (switch OS), or a remote

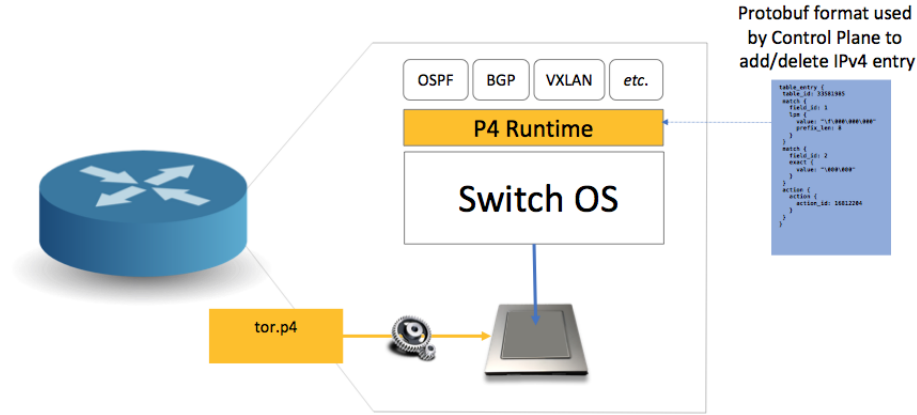


Figure 1.10. P4 Runtime for local Control Plane [12]

control plane running on x86 servers. On the Figure 1.10 is provided an example how a local control plane can use P4 Runtime as an API to control the switch ASIC directly.

At present, switches (and chips) are controlled by proprietary APIs, which are closed and mostly fixed to predefined set of functions. Such API is written to the target chip and covers the needs, and usually there is no need to extend the API over time. Furthermore, software distribution models, in particular NDAs and license agreements very often prohibit sharing the API with others, making it impossible for one API to be used to control switch ASICs from different chip vendors. In result, it is very difficult, oftentimes impossible, to add new protocols and features, and it is impossible for one network owner to leverage different features from another, stifling innovation.

The Figure 1.11 shows how a remote control plane (standard SDN controller, like ONOS) can use P4 Runtime API to control a switch (or router). In this example, the P4 program "tor.p4" specifies the switch pipeline. A P4 compiler generates the schema needed by the P4 Runtime API to add and delete entries into the forwarding table at runtime.

The P4 Runtime API can be used to control any switch, of any vendor, if behavior of this switch has been specified in the P4 language. When needed, P4 Runtime can be used to control existing fixed-function switches. In order to do this, a developer should write a P4 program to document the switch behavior using the P4 language. After this P4 compiler (e.g. p4c) will automatically identify elements that need to be controlled, such as lookup tables specified in the P4 program to which we need to add and delete entries.

P4 and P4 Runtime allow programming of network devices and raise flexibility of this process to the new level, but it's programming model is very far from simple imperative programming constructs, provided by OpenFlow. On the Figure 1.12 is shown the structure of the P4 Runtime objects.

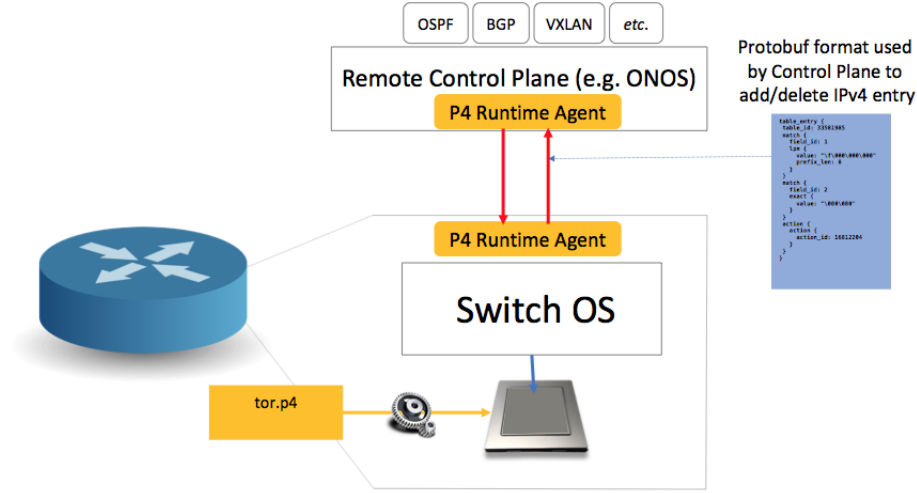
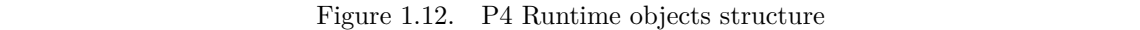


Figure 1.11. P4 Runtime for remote Control Plane [12]

As we may see from Figure 1.12, a programmer or network controller, who wishes, for example, to send IP packets with IP destination address 10.0.2.2 to switch port number 3, will not see here anything of help. In P4 and P4runtime structure this behavior must be described first in P4 program, where we need to create a match-action table to process IP packets and corresponding action to set destination port. After, we need an adapter program, which translates standard IP protocol abstractions of controller into P4Runtime specific action, i.e - to forward the packet mentioned above, controller must write particularly shaped entry into particular match-action table of P4 program, which calls a particular action with a particular parameter. Translating it to the objects from Figure 1.12 we need to create and send to switch p4.WriteRequest object, which contain p4.Update object of type INSERT, which contains p4.Entity object, which contains p4.TableEntry object, which contains objects p4.FieldMatch and p4.Action, each with corresponding types and parameters. Going on with programming paradigms, we may say that simple imperative programming model of OpenFlow became much more complicated, and now resembles Object-Oriented and Multithreaded programming.

1.2.4 INband Telemetry

Inband Network Telemetry ("INT") [13] is a framework designed to allow the collection and reporting of network state data, directly by the data plane, without requiring intervention or work by the control plane (controller). Data exchange in INT model is implemented by adding special headers, containing state information, to the forwarded packets. In the INT architectural model, packets contain mentioned header fields that are interpreted as "telemetry instructions" by network devices. These instructions tell an INT-capable device



15

the packets "observed" while being forwarded. Some examples of traffic sink behavior are described below: (Citation from [13])

- OAM - the traffic sink might simply collect the encoded network state, then export that state to an external controller. This export could be in a raw format, or could be combined with basic processing (such as compression, deduplication, truncation).
- Real-time control or feedback loops - traffic sinks might use the encoded data plane information to feed back control information to traffic sources, which could in turn use this information to make changes to traffic engineering or packet forwarding. (Explicit congestion notification schemes are an example of these types of feedback loops).
- Network Event Detection - If the collected path state indicates a condition that requires immediate attention or resolution (such as severe congestion or violation of certain data-plane invariances), the traffic sinks could generate immediate actions to respond to the network events, forming a feedback control loop either in a centralized or a fully decentralized fashion (a la TCP).

The INT architectural model is intended to be generic and enables a number of interesting high level applications, such as: (Citation from [13])

- Network troubleshooting
- Traceroute, micro-burst detection, packet history (a.k.a. postcards)
- Advanced congestion control
- Advanced routing
- Utilization-aware routing (For example, HULA 1 , CLOVE 2)
- Network data plane verification

Theoretically, using INT we can define and collect any switch-internal information. In practice, however, developers of the protocol consider useful to define a small baseline set of metadata that can be useful for a wide variety of devices. With the evolution of the protocol this set can be extended to encompass more specific metadata, useful for special cases, in particular for the scope of this work: Network attack detection. At present we don't use this protocol, but consider it worthy for the future research work.

1.3 Open Network Operating System (ONOS)

1.3.1 Overview

ONOS [14] abbreviation represents **Open Network Operating System**. ONOS acts as a remote control plane for a software-defined network (SDN), manages network devices, like switches and links between them, and runs special applications or modules to provide

desired network behavior to clients, which can be end hosts or other networks. Structure and capabilities of ONOS are better explained with following comparisons:

- Comparing with server operating systems, ONOS provides some analogous types of functionality, including APIs and abstractions, resource allocation, permissions, user-facing software such as a CLI, a GUI, and set of system applications.
- Comparing with traditional "inside the box" switch operating systems, ONOS performs similar functions, but controls entire network rather than a single device, and doing this radically simplifies management, configuration, and deployment of new network software, hardware and services.
- Comparing with SDN controllers, ONOS platform and applications act as an extensible, modular, distributed SDN controller.

The most important advantage of a network operating system is that it provides a platform for software components designed for a particular network behavior or standard scenario like corporate network, provider network or another use case. ONOS applications mostly consist of customized communication routing, management, or monitoring services for software-defined networks, which can separately or simultaneously, providing desired network behavior.

ONOS can run as a distributed system across multiple servers, allowing it to use the CPU and memory resources of multiple servers, and providing fault tolerance in case of server failure. Also it supports live/rolling upgrades of hardware and software of controlled devices, that oftentimes can be deployed even without interruption of network traffic.

All ONOS parts, including ONOS kernel and core services, and all ONOS applications, are written in Java as bundles that are loaded into the Apache Karaf OSGi [15] [16] containers. OSGi [16] is a component system for Java that allows modules to be installed and run dynamically in a single JVM. The fact that ONOS runs in the JVM, makes it platform independent, it can run on all OS platforms for which exists JVM.

ONOS is an open source project with wide community of developers and users, and everybody is able to take part in discussions, development, documentation, and improvement of the ONOS systems.

1.3.2 Internal Structure

As was mentioned before, ONOS is a multi-module project whose modules are managed as OSGi bundles. This structure was chosen to achieve the following goals, according to [14]:

- Code Modularity. The structure should be extensible, and allow to add new functionalities as a self-contained units, which can work together with existing functionalities.
- Configurability. The structure should allow to load and unload various features, whether at startup or at runtime, to provide the desired network behavior.
- Separation of Concern. There should be clear boundaries between subsystems to facilitate modularity. Each module should have its own area of responsibility

- Protocol agnosticism. ONOS, and all its applications, should not be bound to specific protocol libraries or implementations.

In the following paragraphs we will consider all there goals in more detailed way:

Code Modularity The project consists of a set of sub-projects, each of them has its own source tree and can be built independently. To achieve this, the ONOS source tree is organized in a hierarchical fashion that takes advantage of Maven’s notion of a hierarchical POM file organization. Each sub-project inside ONOS has its own pom.xml file (build instructions). Intermediate directories have parent aggregate pom.xml files, which contain shared dependencies and configurations for those sub-projects, enabling them to be built independently of unrelated sub-projects. The ONOS root contains the top-level pom file used to build the full project and all of its modules.

Configurability ONOS is written to leverage Apache Karaf as its OSGi framework. In addition to dependency resolution at startup and dynamic module loading at runtime, Karaf provides the following, according to [15]:

- Enable use of standard JAX-RS API to develop our REST APIs and make them secure
- The notion of features as a set of bundles allowing assembly of custom setups
- Strict semantic versioning of code bundles, including third-party dependencies
- Local and remote ssh console with easily extensible CLI
- The notion of run-time log levels

Separation of Concern In order to achieve this goal, ONOS structure is partitioned into following three layers :

- Protocol-aware network-facing modules (southbound interfaces) that interact with the network
- Protocol-agnostic system core that gathers and provides information about network state, and
- Applications that consume information from the core and use this information in order to provide desired network behavior

Each of the above are tiers in a layered architecture (see Figure 1.13), where network-facing modules interact with the core via a southbound (provider) API, and the core with the applications via the northbound (consumer) API. The southbound API defines protocol-neutral means to relay network state information to the core, and for the core to interact with the network via the network-facing modules. The northbound API provides applications with abstractions that describe network components and properties, so that they may define their desired actions in terms of policy instead of mechanism.

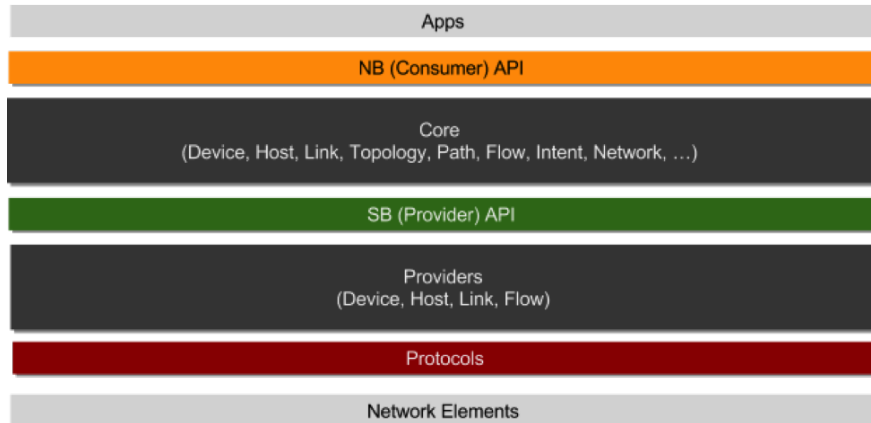


Figure 1.13. ONOS system components [14]

Protocol Agnosticism In cases when ONOS needs to support a new protocol, it should be possible to build a new network-facing module against the southbound API as a plugin that may be loaded into the system.

1.3.3 ONOS Applications

ONOS architecture is build with tiers of functionality. The Figure 1.13 represents a summary of ONOS components discussed in the this section.

Services and Subsystems A service [14] is a unit of functionality that is comprised of multiple components that create a vertical slice through the tiers as a software stack. We refer to the collection of components making up the service as a subsystem. The terms 'service' and 'subsystem' used interchangeably in this text.

ONOS defines several primary services:

- Device Subsystem - Manages the inventory of infrastructure devices.
- Link Subsystem - Manages the inventory of infrastructure links.
- Host Subsystem - Manages the inventory of end-station hosts and their locations on the network.
- Topology Subsystem - Manages time-ordered snapshots of network graph views.
- PathService - Computes/finds paths between infrastructure devices or between end-station hosts using the most recent topology graph snapshot.
- FlowRule Subsystem - Manages inventory of the match/action flow rules installed on infrastructure devices and provides flow metrics.

- Packet Subsystem - Allows applications to listen for data packets received from network devices and to emit data packets out onto the network via one or more network devices.

The Figure 1.14 figure illustrates the various subsystems that are part of ONOS today and a few that are planned in the near future:

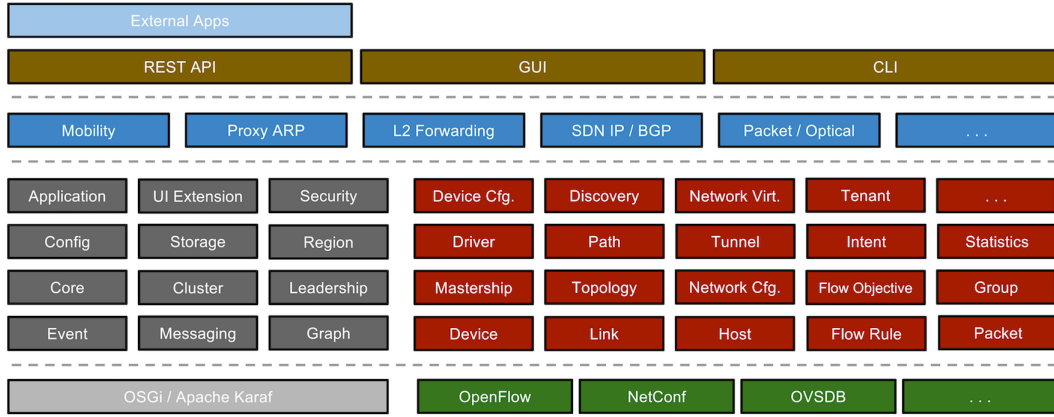


Figure 1.14. ONOS subsystems [14]

Each of a subsystem's components resides in one of the three main tiers, and can be identified by one or more Java Interfaces that they implement.

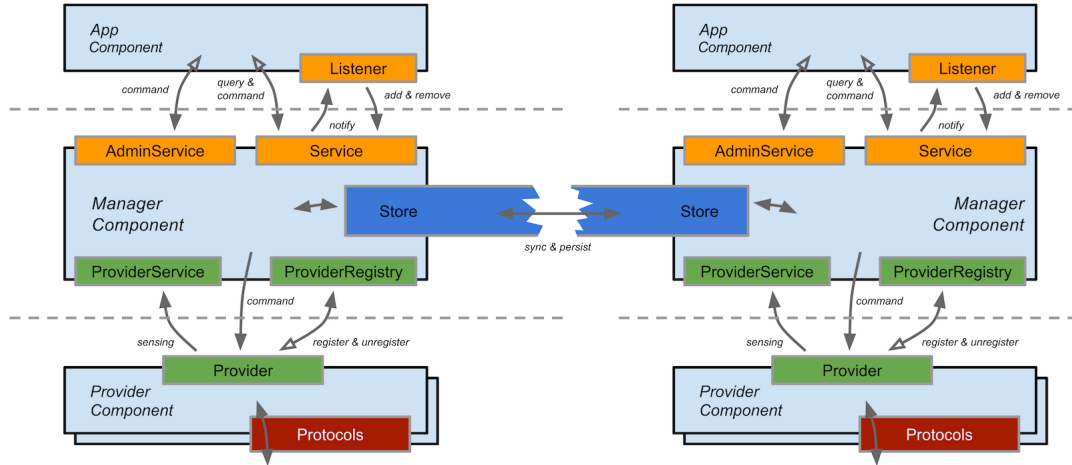


Figure 1.15. ONOS subsystems relationship [14]

The Figure 1.15 summarizes the relationships of the subsystem components. The top and bottom dotted lines in the figure represent the inter-tier boundaries created by

the northbound and southbound APIs, respectively. For the purpose of this work are important that application component can control manager component through its AdminService interface, can query it through service interface, and can receive asynchronous messages from it through listener interface. Also important, that Manager components keeps gathered information in persistent and synchronized store, which guarantees us consistency of the information, even in cases when application component and controller devices are placed on the different servers of the ONOS cluster.

Chapter 2

Distributed Denial of Service attacks

2.1 General description

At present, services like banking, electronic commerce, social networking (chat rooms) and many others are accessed through the Internet. Denial of Service (DoS) attacks can create many issues, influencing functioning and growth of these Internet-based applications. Attacks destroy or impede access to the network services, through depletion of the network bandwidth or network devices processing power, or through depletion of victim resources, such as disk space, file descriptors, buffers, sockets, CPU cycles, memory, in order to prevent legitimate users to access a specific Internet service. DoS attacks exhaust resources of the victim, so it cannot respond to legitimate user, and cannot provide requested services. Distributed Denial of Service (DDoS) attacks are the next step in techniques of DoS attacks, and now have become a severe problem of today's Internet. DDoS attacks are much more complicated to resist, despite the fact, that they use the same techniques as regular DoS attacks, because they are performed on a much larger scale using botnets as shown in Figure 2.1. A botnet is a large group of hosts, which consists of hundreds or thousands of remotely controlled computers (called zombies, bots or slave agents) under the control of attackers, attacking a particular server, service or organization. Almost every computer, connected to the Internet, is under a risk to become a zombie or a bot, through the infection by worms, backdoors or Trojan horses, which are distributed through e-mail content, a captivating Internet link, or a trust-inspiring sender address to the vulnerable machines.

2.2 General Classification

Basically, DDoS attacks are of two types namely flooding attacks and vulnerability attacks as described in following Figure 2.2 In flooding attacks, the attacker commands his army of zombies to send junk or attack packets to the target server or service, in order to raise the amount of traffic to a level that a victim cannot handle, which causes victim system crash

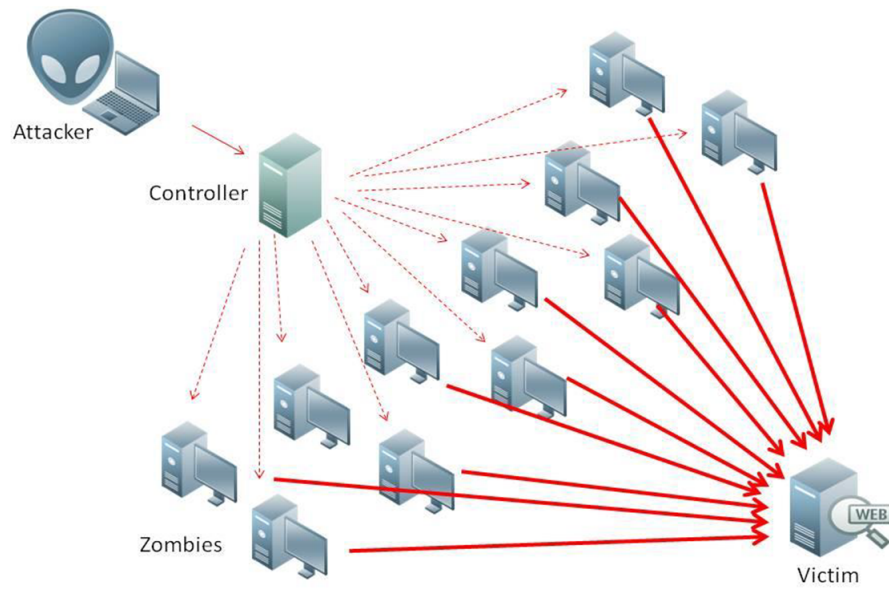


Figure 2.1. Modus operandi of DDoS attacks

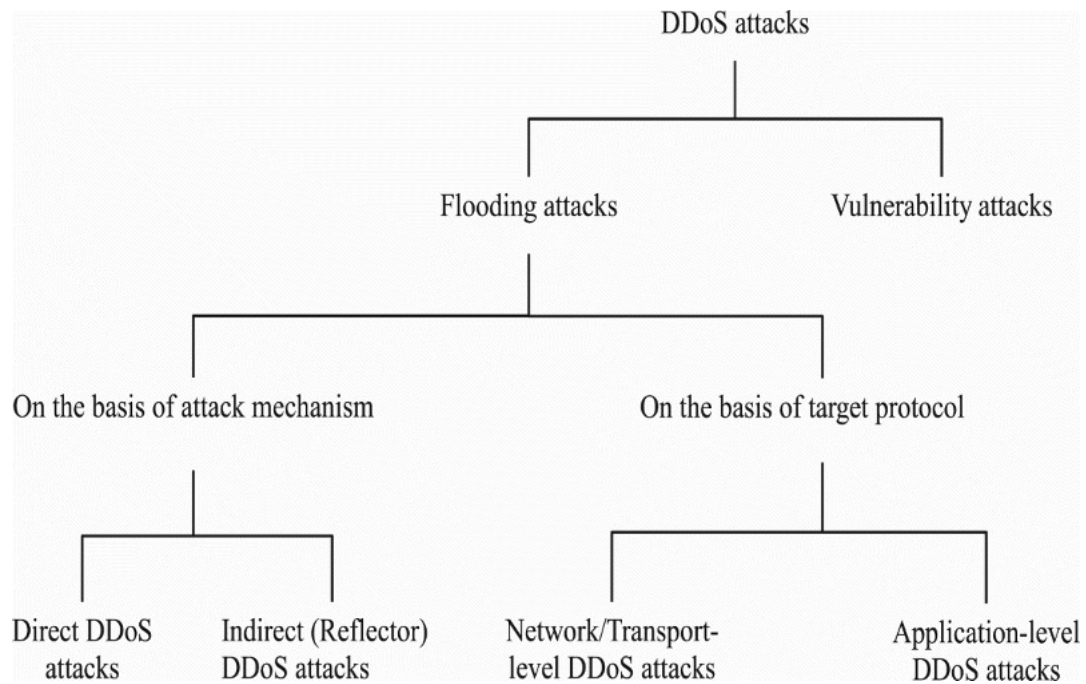


Figure 2.2. Types of DDoS attacks [17]

or overload. Flooding attacks can be further categorized basing on attack mechanisms, into direct and indirect (with usage of reflectors) DDoS attacks. Following categorization is possible on the basis of protocol level that is targeted, here flooding attacks are grouped into Network/Transport level and Application level DDoS flooding attacks. The attacks like TCP, UDP, and ICMP flooding fall into the category of Network/Transport DDoS flooding attacks, while HTTP flooding comes under Application DDoS flooding attacks. In recent years, Application attacks grow rapidly, they are harder to detect and may cause more serious problems for a particular on-line service (or web server) comparing to the Network DDoS attacks. During vulnerability attacks, the attacker looks for flaws in the software of the service and exploits them to against the system to cause system crash or to create new zombie for following attacks. Not differently from Networks DDoS attack, these attacks exploit the particularities in performance of different protocols (such as TCP and HTTP) to exhaust the resources of the victim server and prevent it from providing services for the authorized users.

2.3 Detection methods

One of the most common sight of the presence of the DDoS attack is the fluctuations in the characteristics of network traffic. In the detection phase, these fluctuations are noticed, DDoS attacks are recognized, and legitimate packets are distinguished from attack packets. Detection methods can relatively easily recognize DDoS attacks with the known (or familiar) attack patterns. Methods, based on the discovery of irregularities in network behavior are more complicated. In these cases, without clear DDoS attack profile or signature, the detection schemes recognize unexpected shifts in IP packet characteristics or traffic volume to catch attacks. Attack detection methods build a model or profile by observing the regular behavior of the interface, evaluate the incoming flow against the built model and discover irregularities, distinguishing them from the regular changes in the network behavior. The detection approaches can be implemented locally, to protect a particular part of the network, or remotely, to expose propagating attacks in the core network. Detection time and detection accuracy of DDoS attacks have become the very important measures for any system of defense. Considering this, every detection technique must describe normal network traffic in precise and accurate way, and recognize irregularities with high probability of survival of regular packet, low false positive and false negative ratios and it must be cost effective in terms of consumption of resources and computations per packet. At present, developed many detection methods. They can be classified on the basis of analysis methods and detection approaches, and according to [17] can fall into categories of Signature-based, Anomaly-based and Hybrid detection, as described in Figure 2.3. In this work we are going to use only Anomaly-based detection methods, in particular Point anomaly-based detection.

2.3.1 Anomaly detection methods

Anomaly-based detection approach (also known as novelty detection, outlier detection, behaviour based or one-class learning scheme [17]) is able to detect new, unknown and

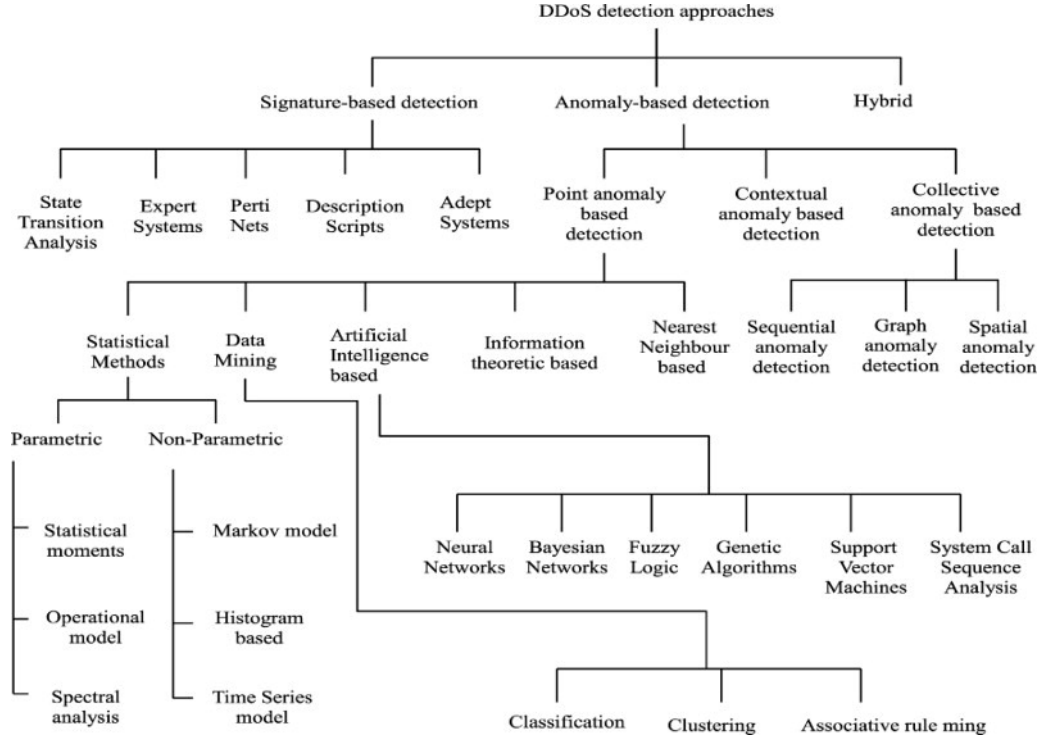


Figure 2.3. Classification of detection approaches [17]

modified attacks, for which doesn't exist well described attack pattern. This approach creates model of the standard network behavior and constantly compares it with the current network behavior. When the difference between an observed and expected behavioral patterns crosses a predefined threshold, the detection system gives an anomaly alarm; this way an attack is discovered. Unfortunately, anomaly-based detection schemes produce a lot of false signals because of the unstable nature of a system or network behavior and uncertainties present in the gathered data. The incoming data to a detection mechanism can be shaped in the form of individual data instances, such as an data object, data vector, some measurement point, data from observation period as a collection of data instances. Data instances may be or may not be connected to each other. Each incoming instance can have a set of attributes and every attribute can be discrete, categorical or continuous. Mostly the detection mechanisms are dealing with the individual incoming data instances in which there are no relationships between the different instances. On the basis of nature of anomalies, detection approaches are sub-categorized into Point anomaly, Contextual anomaly, and Collective anomaly-based detection [17].

Point anomaly-based detection. If a single data instance is compared with the remaining dataset and on the base of this comparison considered as an anomaly, then the approach is known as Point anomaly-based detection. At present times, anomaly-based

detection is one the most significant and interesting field of the research on the field of anomaly-based detection. Many various approaches have been invented to recognize point anomalies in the network traffic, in particular Statistical Methods, Data Mining, Artificial Intelligence (AI) Based, Information Theoretic Based, Nearest Neighbour Based Detection. Statistical methods: Statistical methods used in anomaly detection systems prepare a model or profile to represent the normal behavior of a system (or network) and constantly monitors the traffic, which flows between the potential victim network and the rest of the Internet. These methods can be applied in on-line as well as off-line detection modes. This is done through measuring statistical properties (i.e. means and variances) of parameters of normal traffic (like activity measures i.e. session duration for each session, traffic rates, CPU load, packet rate for each protocol and the number of different clients (number of different IP addresses). Statistical methods, in turn, are categorized into Parametric and Non-parametric detection methods. Parametric detection methods assume that the system has previously collected data about parameters distribution and use the statistical specifications from the this data. The techniques like Statistical Moments, Operational (or Threshold Based) Model, Gaussian Model, Regression Model and Spectral Analysis comes into the category of parametric detection. In this scheme, is set a predefined confidence interval or range, based on statistical properties (correlations or moments) like statistical mean, standard deviation. If any network event (which can be a specific packet, or change of traffic flow, of appearance of new clients) falls outside the predefined interval, i.e. above or below the moment, it is considered as an anomaly. This scheme has more flexibility in comparison with operational model, because the confidence interval is derived from the observations, that can vary from system to system. Because of this, method gives higher weights to the recent activities. In Operational (or Threshold based) model, we compare the given observation (or an event) with a predefined limit (upper limit is n and the lower limit is m or 0). When the count of observed events that occur during a particular period exceeds ' n ' or falls below than ' m ' then the detection system considers the anomaly detected and generates an alarm. As an example, when the count of failed log-in attempts exceeds the threshold, the system results in failed authentication. Moreover, the threshold is based on the mean of various parameters or metrics. This scheme is effective only if there are not any intermittent variations in standard network behaviour and the level of tolerance of an event needs to be set in advance. If the malicious activities have more than one event or the threshold limits are not significant, the scheme cannot detect anomalies efficiently.

EWMA

In statistics, a moving average (rolling average or running average) is a calculation to analyze data points by creating series of averages of different subsets of the full data set. It is also called a moving mean or rolling mean. Variations include: simple, and cumulative, or weighted forms. An exponential moving average (EMA), also known as an exponentially weighted moving average (EWMA), is a first-order infinite impulse response filter that applies weighting factors which decrease exponentially. The weighting for each older datum decreases exponentially, never reaching zero. For the purposes of this work

we use classical formula for EMWA computing, shown on (2.3.1)

$$S_t = \begin{cases} Y_1 & \text{if } t=1 \\ \alpha \times Y_t + (1 - \alpha) \times S_{t-1} & \text{if } t>1 \end{cases}$$

Where:

- The coefficient α represents the degree of weighting decrease, a constant smoothing factor between 0 and 1.
- Y_t is the value at a time period t .
- S_t is the value of the EMA at any time period t .

Simple moving average (SMA) with N samples can be approximated by EMWA with parameter α choosed according to Formula 2.1

$$\alpha = \frac{2}{(N + 1)} \quad (2.1)$$

Chapter 3

Experimental testbed

In this chapter we will describe our simulation methodology, tools we used to emulate network environment, hosts, switches, DDoS attack traffic pattern and other components of our solution. We will show general structure of solution we propose, and two implementations of this solution. Our simulation is of continuous type, all component of our simulator are virtual, all of them are placed within laptop we used to conduct our experiments.

3.1 Tools

This section is dedicated to tools we used to emulate our testing networks, network equipment to execute our algorithm and program products we used to emulate DDoS attack traffic.

3.1.1 Mininet

As our main instrument we use Mininet [18] network emulator, which creates a virtual network of hosts, switches and controllers. Mininet hosts can run standard and non-standard Linux software, and its switches support OpenFlow/P4 for very flexible custom routing and Software-Defined Networking. With Mininet we can prototype and test almost any SDN solution, having an experimental network inside our laptop.

For our project especially important, that Mininet provides powerful and flexible Python API, which allow us to automatize processes of creating testing network, programming hosts and switches, running applications and many others.

3.1.2 BMV2

In our experiments with P4 protocol in our Mininet network we use Behavioral model two (BMV2) [19] software switch that emulates a P4 datapath. It implements the full P4 specification and created to be architectural independent.

3.1.3 Open vSwitch

In experiments with OpenFlow protocol we use standard Mininet Open vSwitch [20], multilayer virtual switch licensed under the open source Apache 2.0 license.

3.1.4 Traffic generators

In order to emulate DDoS attack traffic in our testing network we evaluated several tools:

Scapy [21] is a powerful packet manipulation Python library. It is capable perform many classical networking tasks, including network discovery, testing, trace-routing and many others. This library allows to create packets for large number of protocols and send them into the network according to rules, set by programmer. In our experiments we use widely all power of this library, but it was not used for our main experiments. In our testing environment, working inside the emulated Mininet host, this library cannot send packets faster than 18 packets per second even with the simplest protocol stack and pregenerated packet load. To emulate DDoS attack scenario we wanted to be able to increase packet rate considerably higher, for hundreds of thousand times, like in real DDoS attacks.

Iperf is a widely used tool for network performance measurement and tuning. It is significant as a cross-platform tool that can produce standardized performance measurements for any network. Iperf has client and server functionality, and can create data streams to measure the throughput between the two ends in one or both directions. In our experiment we used this tool to emulate heavy packet load on the entrance of our network. To our regret, this tool is not intended to be traffic generator, and doesn't support any traffic pattern, except constant bit rate.

D-ITG [22] Distributed Internet Traffic Generator (D-ITG) is a platform capable to produce traffic that accurately adheres to patterns defined by the inter departure time between packets (IDT) and the packet size (PS) stochastic processes. Such processes are implemented as an i.i.d. sequence of random variables. A rich variety of probability distributions is available: constant, uniform, exponential, Pareto, Cauchy, normal, Poisson and gamma. This tool was widely used for debugging our programs, but in the DDoS attack emulation scenario, with many generators and many listeners, shows itself unstable. In the final experiments this tool was substituted by Socket generator.

Socket Generator and Listener In order to simplify traffic generation we write our own traffic generator, using standard Python Socket library. It generates Poisson traffic pattern in a very simple way of operation:

- Generator opens standard UDP socket
- In infinite cycle, until the end of the experiment, the generator:
 - sends packet from the socket to destination address
 - generates random time interval according to exponential distribution
 - waits for generated time interval

Size of the packet set to be 1250 bytes, which corresponds to 10 kilobits on the wire. In order to emulate periods of normal and attack traffic patterns, our generator changes exponential distribution parameters to obtain different traffic rate. Resulting traffic pattern

has two repeating phases : For the first interval generator emulates normal traffic, for the next interval generated "attack" traffic, for next again normal traffic and so on. Duration of these intervals goes as an incoming parameter of our generator.

Following example illustrates work of the generator. If it started with command:
`gen3_sock.py 10.0.2.2 0 30 10 17`

it will be generating packets, directed to address 10.0.2.2 (first parameter), without starting delay (second parameter), changing traffic rate once every thirty seconds (third parameter), from 10 packets per second (forth parameter) to 170 packets per second (forth and fifth parameters multiplied). Resulting traffic pattern presented on the Figure 3.1

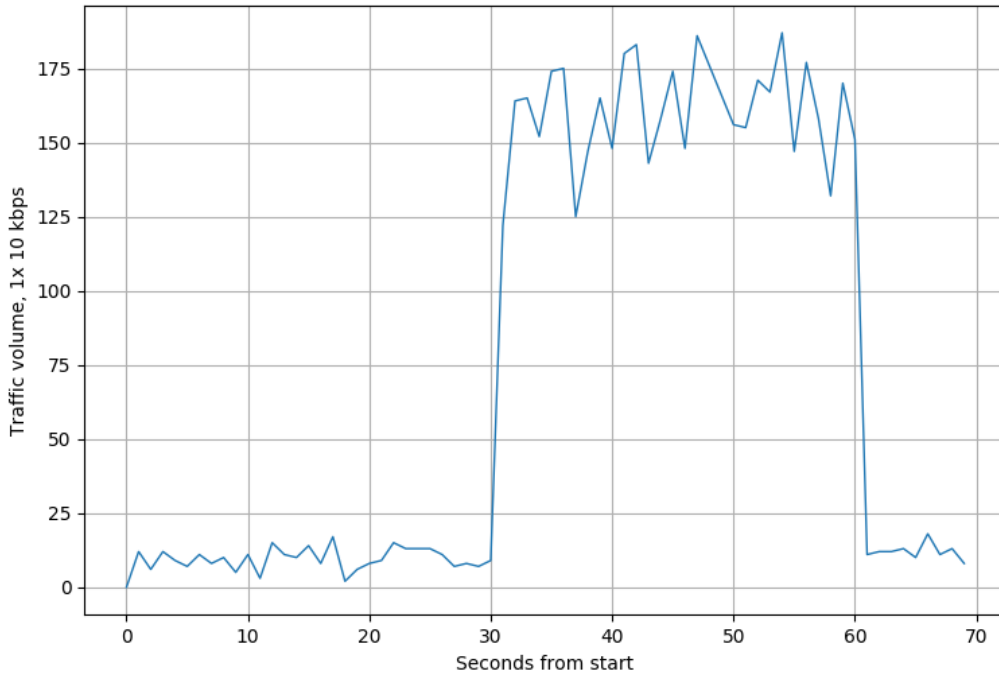


Figure 3.1. Socket generator traffic pattern

In pair with socket generator we use Socket listener. This program receives packets from the generators, counts them and then discards.

3.2 Testing topology

For our experiments we use Mininet emulated topology, presented on the Figure 3.2

Also, we made next assumptions

- Our network is SDN domain, under control of single SDN controller

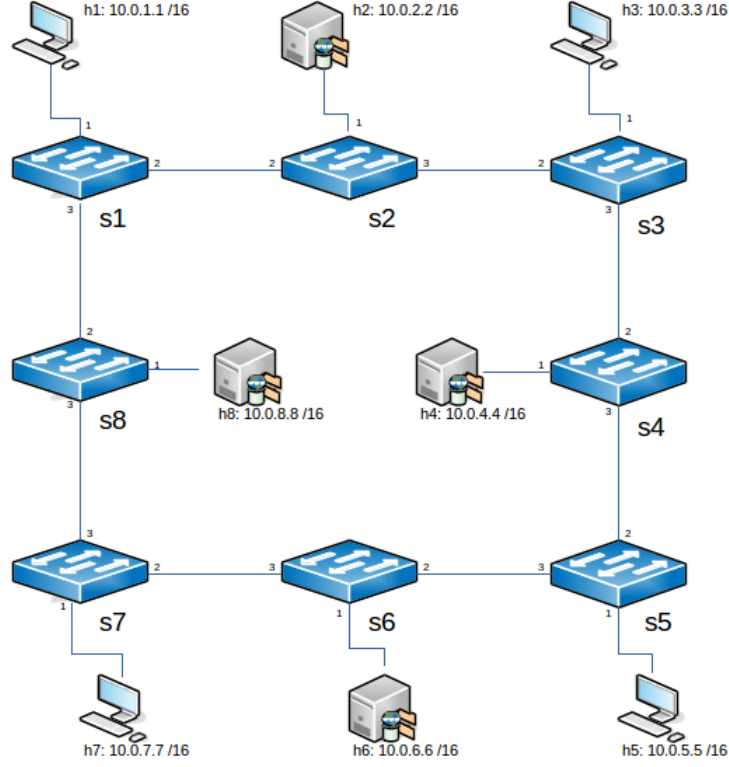


Figure 3.2. Testing topology

- Our network is included into larger network, structure of which is unknown to us
- In order to simplify experiments, we don't count return traffic from our network

In our topology, hosts with odd numbers (i.e. h1, h3, h5, h7) emulate external network, packets from these hosts have arbitrary source addresses. Hosts with even numbers (i.e. h2, h4, h6, h8) emulate internal networks of our SDN domain. Sequentially, switches with odd numbers (s1, s3, s5, s7) imitate entrances in our network and may be referred as "edge" or "border" switches. Switches with even numbers (s2, s4, s6, s8) are internal. Routing (switching) configuration is static in all experiments, and written into switches by means of controller, used in the experiment. In order to simplify experiments, each host has set of static ARP entries reflecting all other hosts in the topology. Control (or controller) traffic exchange is emulated by means of Mininet and has place outside the topology, traffic of this type does not appear on the interswitch links. Traffic measurements is done by internal means of applications (traffic generators, traffic receivers and controller application), by means of internal counters in Mininet switches (BMV2) and, in some cases, by analysis of traffic captured into PCAP file format.

3.3 General implementation components

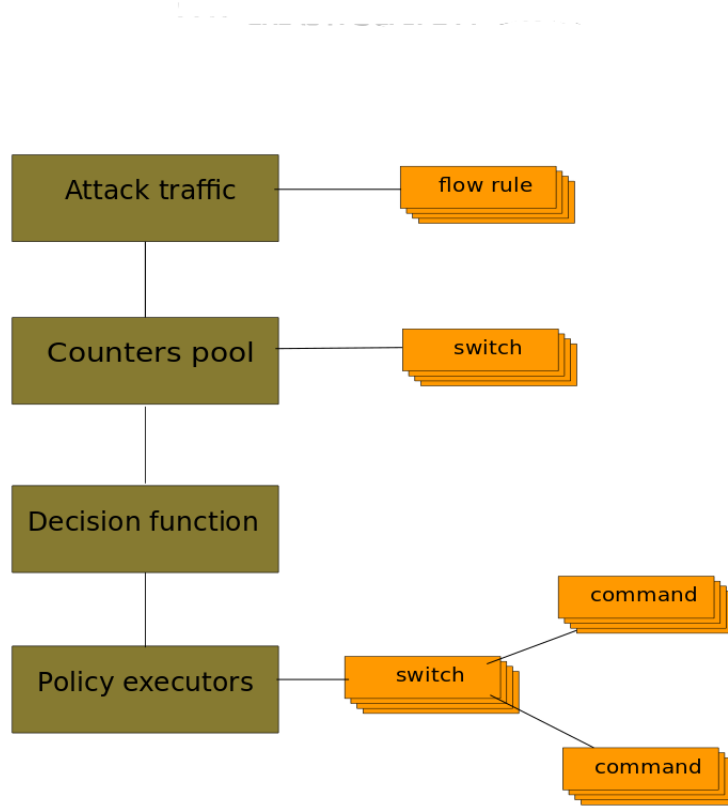


Figure 3.3. General structure of the proposed solution

Generally, to solve DDoS attack discovery and mitigation problem, we should answer some questions about traffic in our network, define what type of traffic is dangerous, define possible sources of this traffic, define dangerous levels of the traffic and many others. In our solution we put answers to all these questions into the components, presented on the Figure 3.3. Let us consider them:

Traffic pattern As we know from previous chapters, all DDoS attacks are different. Generally, they are targeted to specific targets, exploiting particular vulnerabilities of each server platform. Every network has its own pattern of traffic, which is dangerous for its servers and applications. This traffic pattern is the first component of our solution. In order to be able to detect the attack on the network devices, attack traffic must be described in terms, understandable for network devices, i.e. as a bit pattern. P4, because of its protocol independence, provides to us ability to describe traffic in very detailed way. For example, we can discover only HTTP GET requests directed only to particular

set of servers. Generally, description of traffic pattern could consist of arbitrary number of rules, each of them could have arbitrary complex bit pattern. For the simplicity, in our implementations we consider the most simple case - IP packets, that have particular destination address.

Counters placement Next step after description of attack traffic pattern is a decision where we count this traffic pattern. In case of vanilla SDN, traffic counting executed by controller, which is dangerous in case of DDoS attack - controller receives part of attack traffic and found itself under attack, even if attack is not directed to it. In our scenario, traffic counting delegated to network devices. This creates next problem illustrated by the Figure 3.4:

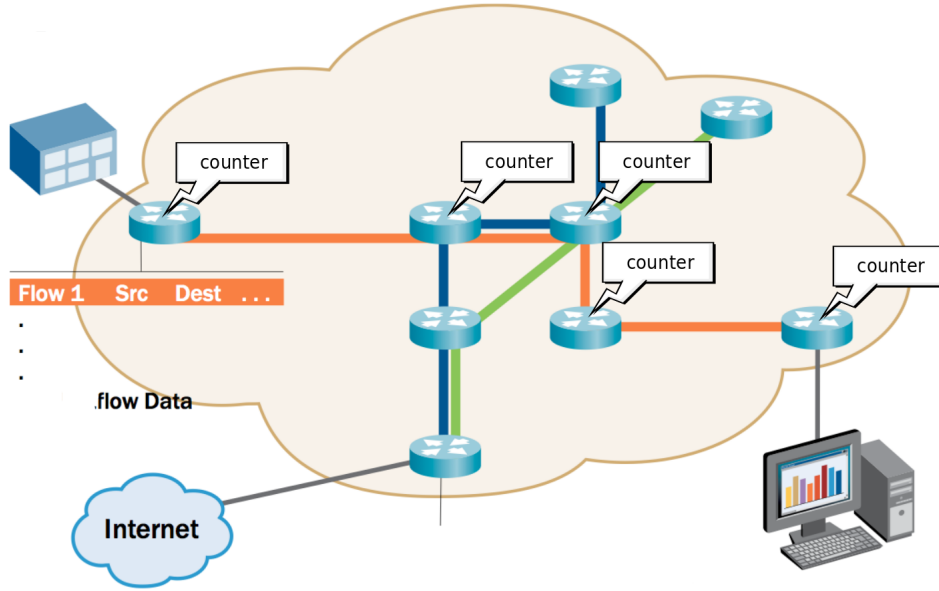


Figure 3.4. Sequential counters

In the example on the Figure 3.4 Flow1 is counted on each network device which it crosses on the way to its target, and simple summation of all counters could give to network controller incorrect information about total amount of traffic in the network.

There are several strategies to avoid this scenario, from which we chose one with creating the set of counting switches by solving network graph in order to any traffic flow, from any attack source, to any attack target, must cross one and only one network device from the set. This strategy allows us not to count traffic on all switches, but use minimum number of counting devices, needed for our purpose. Also, this approach will decrease operational cost of the solution. In our implementations we are going to count traffic only on switches imitating entrances in our network.

Decision function is the component, which will process information about total attack traffic in our network, gathered by our controller from the counting switches. On the exit of the function our controller will get simple decision - true, if there is the attack, and false otherwise. In our solution we use Exponential Moving Average (EWMA) as a decision function, with pair of static thresholds - one for discovering the start of attack and second to discover the end of attack. Decision function will be called with interval of one second.

Policy executors are the set of network devices, responsible for the counter-attack actions. For each device we keep two sets of commands: first, policy enforcement, for the case of attack detection, and second, policy canceling, for the case of the end of attack. This structure allows us to use relatively complex counter-attack measurements - for example, rerouting traffic to the special device in case of attack detection, and restoring the normal routing scheme after the end of attack. Placement of policy executors is chosen in a way, which allows us to keep our counting ability even in case when counter-attack policy is active, i.e. each policy executor is placed "downstream" of each counter relatively to possible attack traffic flow. In our solution we consider traffic dropping action, executed by one or several network devices.

3.4 P4Runtime implementation

Our first solution consists of simple P4 program and mini-controller written in Python, which uses means of P4Runtime library to control our switches.

Structure of P4 program On the Figure 3.5 represented the structure of our P4 program.

From the Figure 3.5 we can see, that this pipeline consists from two match-action tables `cnt_lpm` and `cnt_ipv4`. Each packet going through switch will be processed first in table `cnt_lpm` and after by table `cnt_ipv4`. Let's consider purposes of these tables:

Table `cnt_lpm` designed for attack traffic counting. It contains traffic pattern rules paired with number of internal switch counter. This table placed first in the P4 pipeline in order to count attack traffic independently of countermeasures status. Work of this table illustrated by following example:

```
table_add cnt_lpm count_it 10.0.2.2/32 => 1
```

This table entry means, that packet matches the pattern 10.0.2.2/32 must be processed with action `count_it` with parameter one, which just increases corresponding counter (number one) by the size of this packet.

Table `ipv4_lpm` executes routing of the traffic. It contains addresses of internal networks of our SDN domain paired with switch port, on which should be directed traffic. Also, this table works as "policy executor". In case of attack detection our controller will change routing rules in this table.

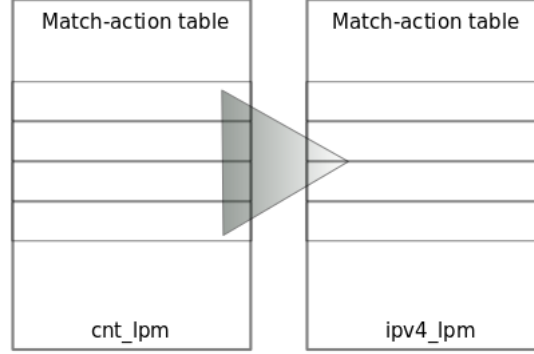


Figure 3.5. Structure of P4 program

Mini-controller structure As the incoming data our mini-controller receives P4 program, forwarding rules for each switch, set of the "counters" switches, set of the "policy executors" switches, in which each element goes with two sets of commands for use in case of policy enforce/cancel. At the start of the experiment no one of the switches in our network has commands about special actions applied to attack traffic, we consider it as passive counter-attack policy status. This status becomes active, if special commands are executed by "policy executor" switches General mini-controller sequence of operations is following:

- Install P4 program into BMV2 switches
- Install routing rules into ipv4_lpm tables
- Installs attack pattern rules into cnt_lpm tables of the switches from "counters" list
- Repeats following actions in infinite cycle, until the end of the experiment
 - Collect counter information from "counters" switches
 - Process collected information with decision function
 - In case of positive decision function answer and passive policy status, send "policy executors" switches commands to enforce policy, and change policy status to active
 - In case of negative decision function answer and active policy status, send "policy executors" switches commands to cancel policy, and change policy status to passive

- Wait for predefined period of time (in our solution - 1 second) ¹

As an advantage of this solution should be mentioned its simplicity - mini-controller program is less than one hundred lines of code. As a disadvantages we consider limited scalability of the solution - "counter" switches are polled sequentially, and with large number of switches and high network RTT, decision delay can become unacceptably long.

3.5 ONOS internal application

As our next step we insert our application into real SDN controller - ONOS, described in the section 1.3.

In the Figure 3.6 is shown the place of our application in the structure of ONOS.

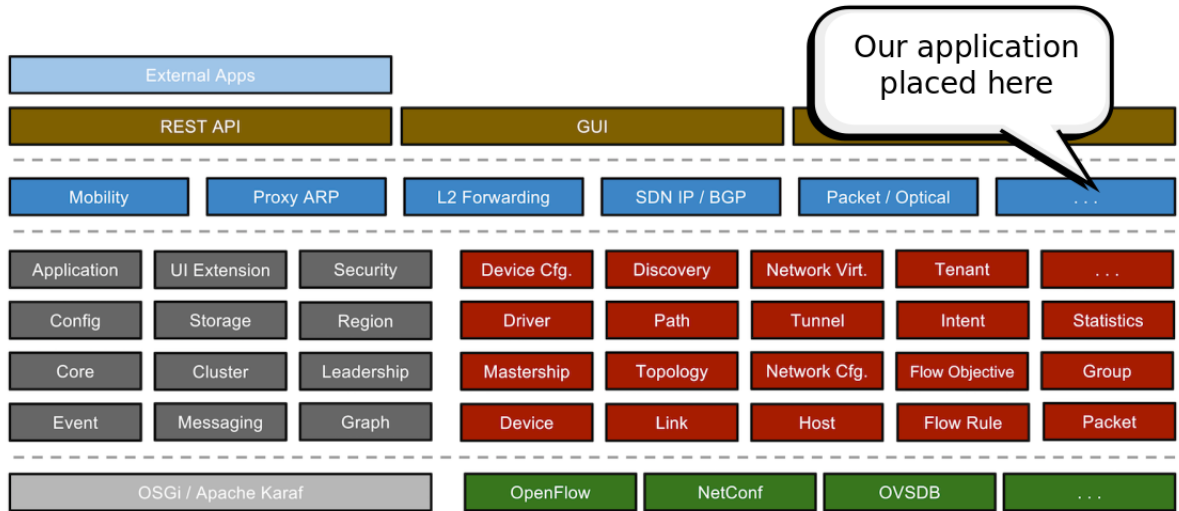


Figure 3.6. Place of our application inside ONOS architecture

Placed this way our application has direct access to all ONOS services, including statistics collection, which we can use for our purpose. For our experiment we use static network configuration, and shut down all ONOS applications, except our application and BMV2 device driver.

Structure of P4 program for this implementations is different from P4Runtime implementation, described in Section 3.4. We consider, that our application could work with other ONOS applications, and forwarding table may not be fully under our control. To avoid possible conflicts, now we do not use forwarding table for policy enforcing, this task

¹Mini-controller synchronized in a way to start polling in the beginning of the second. In reality waiting time equal one second minus time spent on polling.

is transferred to separate match-action table. On the Figure 3.7 presented our new P4 pipeline:

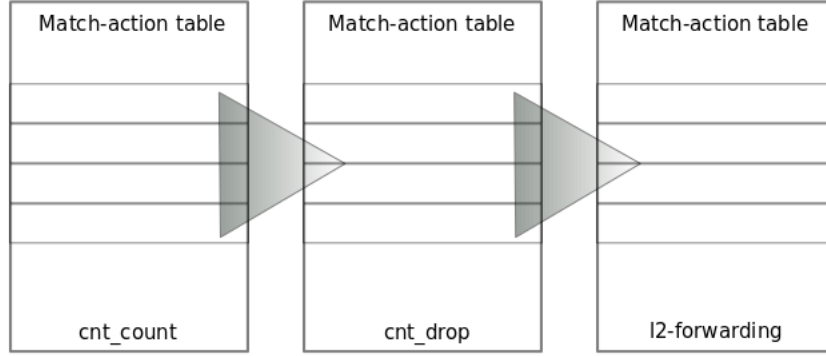


Figure 3.7. Structure of P4 pipeline for our application

As we can see on the Figure 3.7, now we have three match-action tables in our pipeline. Let us explain the purpose of each:

- Table `cnt_count`. As in our previous solution, counting table placed first in P4 pipeline. The purpose of the table stays the same - it holds the records of attack traffic pattern, and number of switch counter on which counted attack traffic.
- Table `cnt_drop`. This table is our new policy enforcer - in case of attack records from attack traffic pattern are copied to this table and switch starts dropping attack traffic. If traffic "hits" any record in this table, following tables in the pipeline are skipped, they are useless in case of activated policy. But, our counting ability stays untouched, because counting table placed before this.
- Table `l2_forwarding`. This table is responsible for packet forwarding in our network. Our application does not touch directly this table.²

Work of this pipeline is better explained by following examples:

- If incoming packet does not belong to attack traffic pattern, it doesn't hit table `cnt_count`, nor the `cnt_drop` table. It goes straight to the `l2_forwarding` table, receives from there its exit port, and goes by its way.

²Not completely correct. Our application creates static forwarding rules by means of ONOS FlowObjectives service, and eventually these rules are written in this table.

- If incoming packet belong to attack pattern, but policy is not active, it hits `cnt_count` table, corresponding internal counter of the switch will be increased by size of this packet, and packet goes to the next table. Because policy is not active, table `cnt_drop` is empty, packet cannot hit it, it follows to the next table, receives its exit port and leaves the switch.
- If incoming packet belongs to attack pattern and policy is active, it will be counted as usual by the `cnt_count` table, but this time it hits `cnt_drop` table, because when policy is active, our attack pattern rules written there. All following processing steps for this packet are skipped, and packet is discarded by switch.

Technical details of the application Despite the fact, that our ONOS application uses the same decision structure as our previous solution (mini-controller, described in Section 3.4), technical particularities is very different. We cannot command switches directly, and have to use ONOS services to do the same tasks. Here we provide short description of used ONOS services and ways we using them.

ONOS DeviceService is responsible for device management, which includes statistics gathering. This service intended to be universal to any switch model, and because of this has some restrictions:

1. It does not support P4 indirect counters we used before, it supports only switch port counters. To solve this problem we create virtual ports on our counting switches and binds our P4 counters with these ports.³ This way we gather information from out attack pattern counters.
2. It gathers statistics according to its own schedule. We can set frequency of statistics gathering, but not the precise moment of switch polling. To determine if we have updated statistics we use **ONOS Event Subsystem** illustrated by Figure 3.8. Our application registers itself as a listener for DeviceService (which is Manager, according to ONOS service classification), and now able to receive device-binded events, in particular `PORT_STATS_UPDATED` event, which means that DeviceService updated port statistics for one of the switches.
3. Next problem connected to previous. DeviceService gathers statistics according to its own schedule and does not keep precise order of switch polling. The same switch can be polled last in one polling cycle, and the first in the next. In order to not count statistics from one switch twice, we introduce notation of **statistics gathering interval**, binding it to ONOS server time. It is better to explain it with the example: When we receive `PORT_STATS_UPDATED` event we check its timestamp, for example it is 10:15:55.1034, and compare it with the time of previous event, for example 10:15:54.8945. If both events happened within the same second, they considered to be in the same interval.

³To simplify the solution we use one of existing ports on the switch, namely number one.

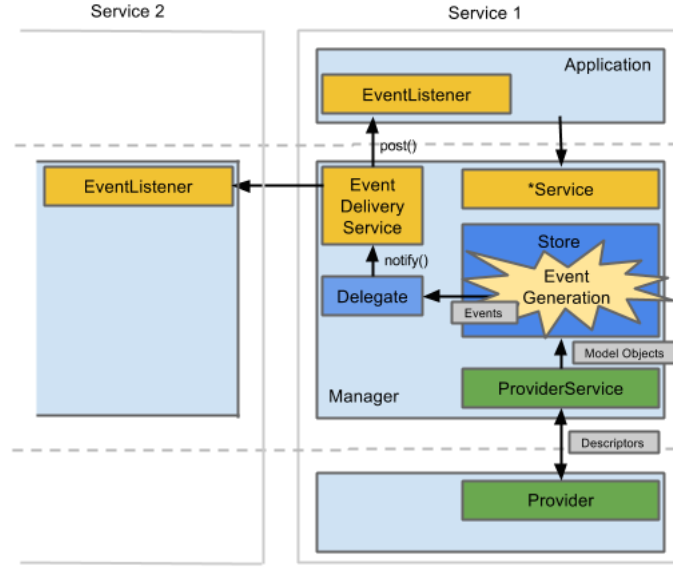


Figure 3.8. ONOS events subsystem [14]

If it is not so, as in our example, we consider than previous gathering period is completed and started new one.

FlowObjectiveService intended to be universal service to manage flow rules in any switch, of any type and any model, connected to ONOS. Because of this it support only basic operations, for example sending particular packet to particular port. We use this service to install forwarding rules to the switches.

ONOS FlowRuleService is much more specific than FlowObjectiveService, it knows particularities of the switch, and in our case can send to the switch P4-specific commands. We use it to implement counting and dropping rules.

Main flow of the application

- On the first step our application set statistics gathering parameters. In order to set statistics polling frequency to once per second we set parameter deviceStatsPollFrequency of ONOS DeviceService to 1 second.⁴
- On the second step we write to our switches predefined forwarding rules by means of ONOS FlowObjectiveService. All rules set to be permanent.
- On the third step our application, by means of FlowRuleService, writes counting rules in the tables cnt_count of each switch form the counters list.

⁴Using ONOS CLI.

- On the forth step our application register itself as a Listener of the ONOS Device-Service. We listen only in `PORT_STATS_UPDATED` events, which means, that statistics for the one of the switches is updated.
- On the fifth step we enter infinite cycle, which does main operations of our application. This cycle continues until the end of the experiment or until deactivation of our application. Inside cycle application waits for the `PORT_STATS_UPDATED` event, in response of which does following operations
 - Check, if source of event is one of the counting switches.
 - If no, ignore the event and skip all following steps.
 - Checks, if current statistics gathering period is completed
 - If so, calls decision function
 - Asks source of the event for gathered statistics by means of `getDeltaStatisticsForPort` method of `DeviceService`, writes result into corresponding data structure.

If called, **desicion function** does following operations:

- Summarizes gathered statistics for all switches from counters list
- Processes this sum with EWMA
- If policy is passive:
 - Compares result with high threshold
 - If result is above:
 - * By the means of ONOS `FlowRuleService`, writes attack traffic pattern to `cnt_drop` tables of all switches from "policy enforcers" list
 - * Changes policy status to active
- If policy is active:
 - Compares result with lower threshold
 - If result is below:
 - * By the means of ONOS `FlowRuleService`, deletes attack traffic pattern from `cnt_drop` tables of all switches from "policy enforcers" list
 - * Changes policy status to passive

In order to simplify previous paragraph, and as an alternative explanation of our algorithm, we provide description in pseudocode:

```
if policy status is passive then
  if  $EWMA\_result \geq high\_threshold$  then
    write counter – attack rules into switches
    change policy status to active
```

```
    end if
else
    if  $EWMA\_result \leq low\_threshold$  then
        delete counter – attack rules from switches
        change policy status to passive
    end if
end if
```

Chapter 4

Experimental evaluation

4.1 P4 Runtime implementation

For our first experiment we use components placement according to following picture:

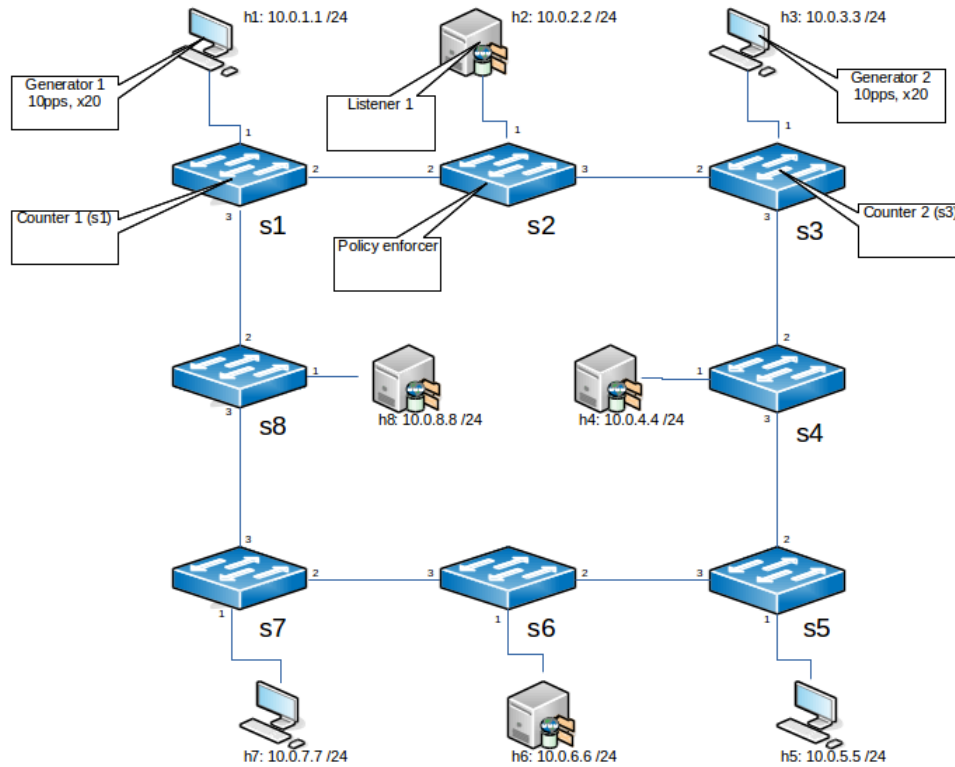


Figure 4.1. Experiment topology

Generators are Python Socket Generators, described in Section 3.1.4. They positioned on the hosts h1 and h3 and generate Poisson traffic according to the following pattern: For 30 seconds traffic is generated with parameter equal 10 packets per second, for the next 30 seconds parameter changes to 200 packets per second, then pattern repeats. Target for the both generators is address 10.0.2.2 (host h2).

Listener is Python Socket Listener, and placed on the host h2. In this experiment it performs only logging of received traffic.

Counters In the counters list our mini-controller has two switches: s1 and s3. It corresponds to the counting strategy near the source.

Policy executor in this experiment placed on the switch s2.

In this experiment Decision function analyses packet counters of the switches¹. High(attack) threshold set to 350 packets per second, low threshold set to 300 packets per second. Resulting behavior of our system presented on the Figures 4.2 and 4.3:

Our analysis starts with the graph on the Figure 4.2. Here we can see with all particularities working algorithm of our testing environment:

At the time 1s of the experiment started our mini-controller. On the graph we see corresponding spike of the control traffic. At this moment was happening setup of our network: P4 program was written into the switches, forwarding and counting rules were written into corresponding tables and packets started moving through our network.

During the time from 2nd to 25th second we see normal functioning of our network. Should be noticed, that controller and receiver statistics gathering periods are synchronized almost perfectly - their graphs almost coincide.

At the time 26s our generators go to the attack mode. Total data traffic and receiver graphs go up synchronously, after them goes up our decision function graph.

At the time 33s our decision function registers the attack. Policy executor switch receives command to enforce policy, and switch started dropping traffic. On the control traffic graph we can see little spike, corresponding to the command. Should be noticed, that at the time 34s, receiver counter is zero. It means, that processes of statistics gathering, processing and policy execution were completed before the arrival of the first packet of 34th second (there were 387 packets during 34th second of the experiment)

During the time from 34th to 55th seconds we see behavior of our system under attack. As we planned, our counting ability is kept, we still see amount of the attack traffic in our system, despite the fact, that this traffic does not reach its target.

At the time 56s our generators return to normal traffic pattern. At the same time our decision function came below low threshold, and policy executor switch received the

¹All packets in our experiment have the same size, bitrate strictly proportional to the packet rate

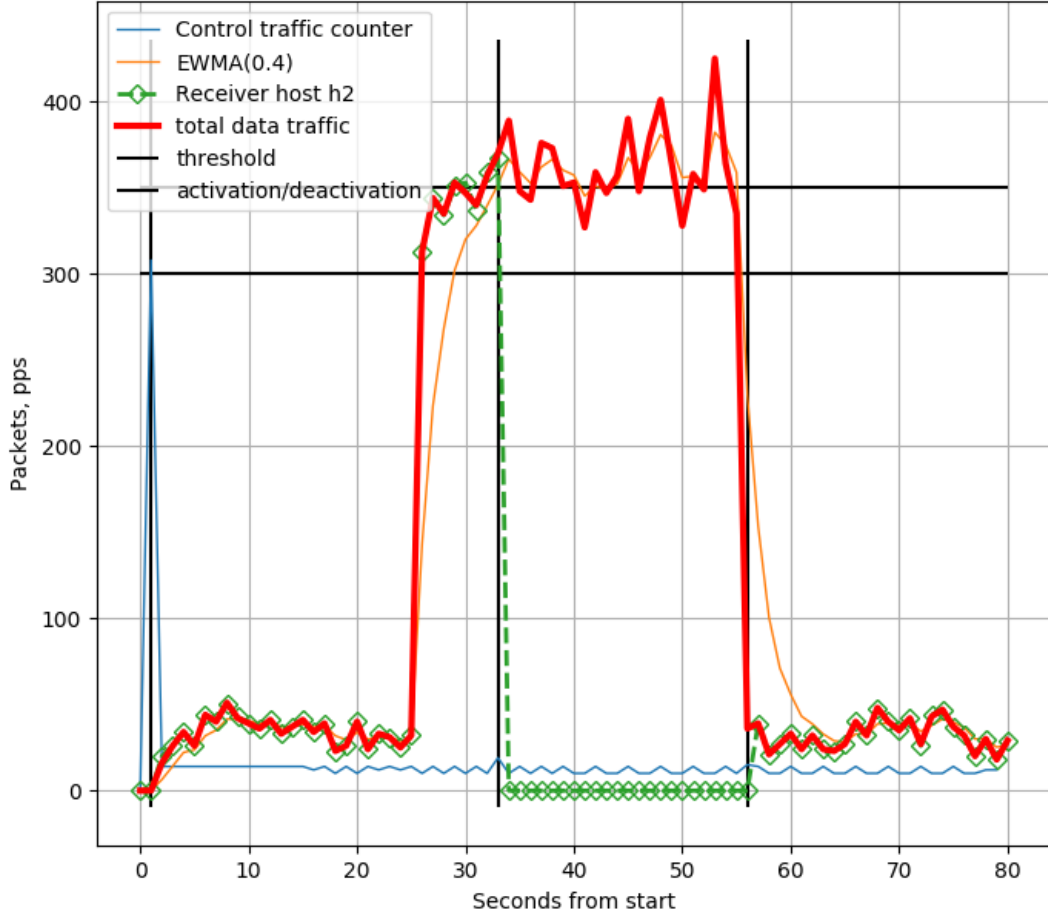


Figure 4.2. P4Runtime Experiment traffic in pps

command to cancel the policy. As in the case of policy enforcing, all these commands were executed before arrival of the first packet of the 57th second.

From the time 57s to the end of the experiment we again see normal system behavior.

From the second graph on the Figure 4.3 we can evaluate amount of data, exchanged by network components in order to provide this behavior. All packets on our dataplane has the same size, therefore corresponding graphs are strictly proportional to graphs on the previous picture (Figure 4.2), and don't merit separate discussion. The only exception is the graph control traffic graph from which we can see, that in our experiment amount of

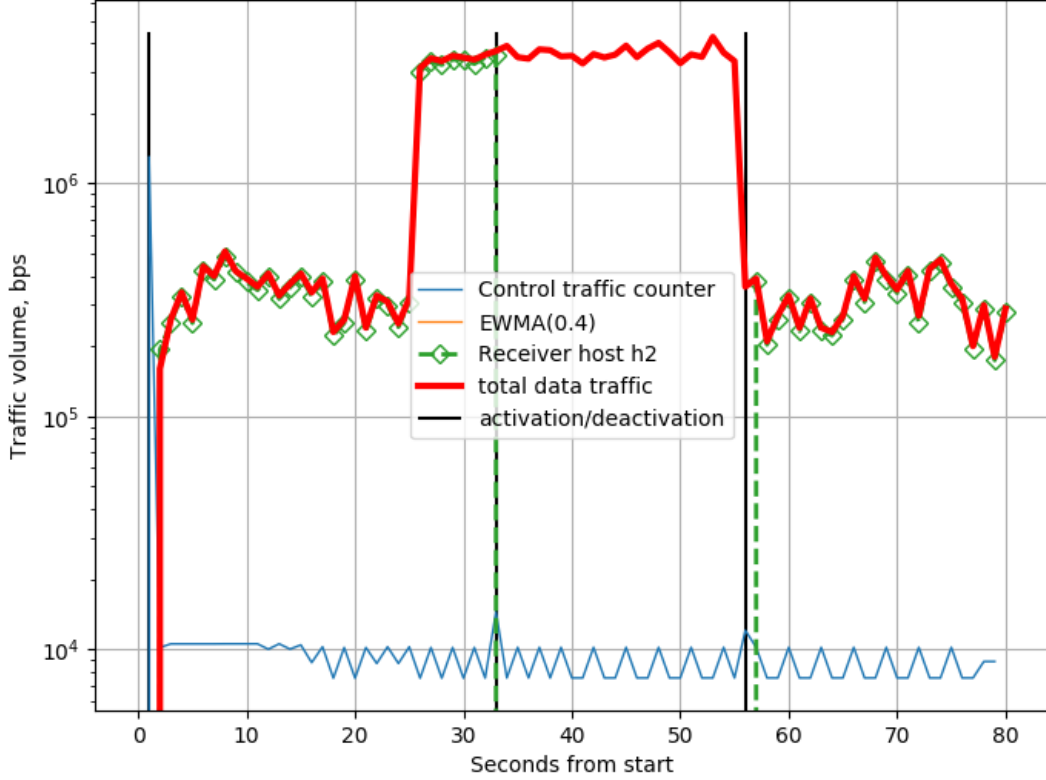


Figure 4.3. P4Runtime Experiment traffic in bps

control data, exchanged by our devices, is not dependent on rate of data traffic, and stays stable independently of the presence of the attack and of status of the policy. Content of the control traffic was evaluated separately, in the following paragraph we provide a description of its structure. Almost any request from controller to the switch, independently of the type of request (information request or command execute), requires following packet exchange:

1. TCP packet from the controller to the switch, with request
2. TCP packet from the switch to the controller, with answer
3. TCP ACK from the controller
4. TCP packet from the switch
5. TCP ACK from the controller

6. Very often, but not always, switch sends one more packet to the controller
7. Controller confirms receiving of this packet (TCP ACK)

In all cases information transmitted in first two packets, and in our opinion, packets from number 4 to number 7 are related to the internal mechanics of gRPC (Google Remote Procedure Call) on which based P4Runtime protocol. Arbitrary appearance and disappearance of messages number 6 and 7 form "saw" shape of the control traffic graph.

As we can see from the graphs, Reactivity of our system, i.e. period of time between start of the attack, and the detection of the attack by our system, is considerably large. From the parameters, influencing the reactivity, namely sampling period, decision function parameters, and time needed to information exchange, our experiment allows evaluate the latter. In our experiment, amount of time spent to information exchange between switches and controller is so small, that could be not taken to consideration with small number of switches.

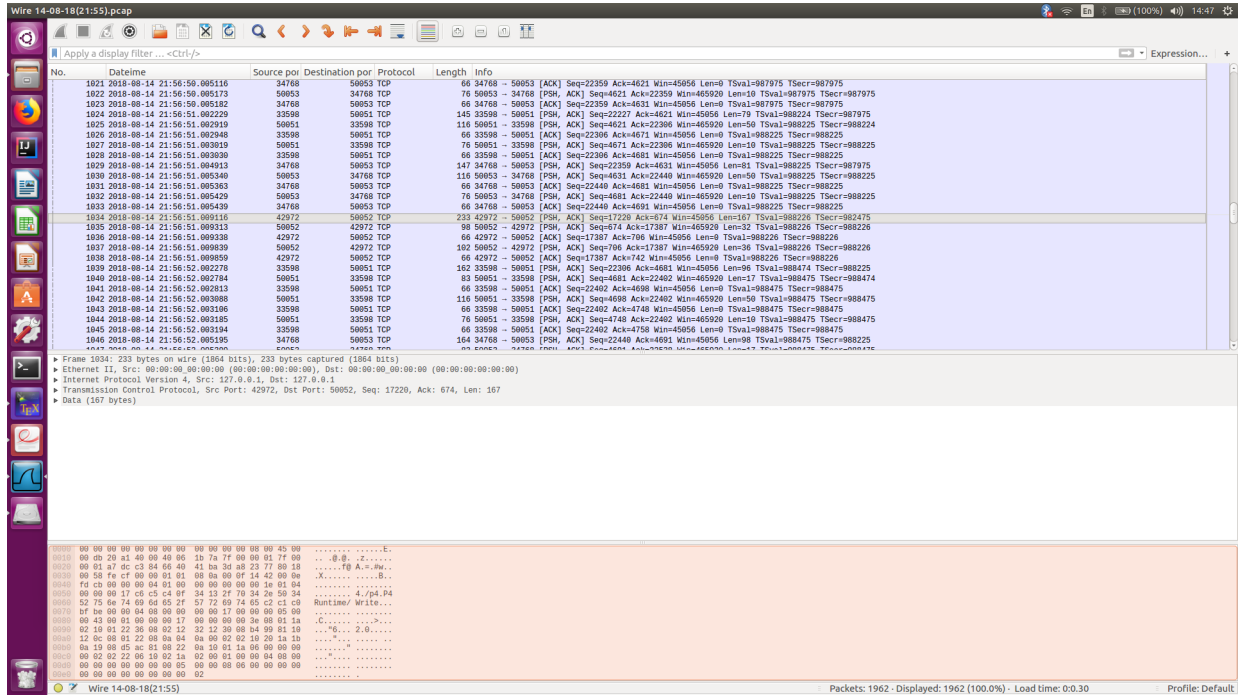


Figure 4.4. P4Runtime Experiment traffic screenshot

Figure 4.4 illustrates this conclusion. On the screenshot we can see traffic exchange, that had place during 33rd second of the experiment (Figure 4.2), which corresponds to the absolute time 21:56:51, packets 1024-1038. In order to facilitate reading, we extract these packets:

```
1024 2018-08-14 21:56:51.002229 TCP 145 33598 - 50051 [PSH, ACK]
1025 2018-08-14 21:56:51.002919 TCP 116 50051 - 33598 [PSH, ACK]
```

1026	2018-08-14	21:56:51.002948	TCP	66	33598 - 50051	[ACK]
1027	2018-08-14	21:56:51.003019	TCP	76	50051 - 33598	[PSH, ACK]
1028	2018-08-14	21:56:51.003030	TCP	66	33598 - 50051	[ACK]
1029	2018-08-14	21:56:51.004913	TCP	147	34768 - 50053	[PSH, ACK]
1030	2018-08-14	21:56:51.005340	TCP	116	50053 - 34768	[PSH, ACK]
1031	2018-08-14	21:56:51.005363	TCP	66	34768 - 50053	[ACK]
1032	2018-08-14	21:56:51.005429	TCP	76	50053 - 34768	[PSH, ACK]
1033	2018-08-14	21:56:51.005439	TCP	66	34768 - 50053	[ACK]
1034	2018-08-14	21:56:51.009116	TCP	233	42972 - 50052	[PSH, ACK]
1035	2018-08-14	21:56:51.009313	TCP	98	50052 - 42972	[PSH, ACK]
1036	2018-08-14	21:56:51.009338	TCP	66	42972 - 50052	[ACK]
1037	2018-08-14	21:56:51.009839	TCP	102	50052 - 42972	[PSH, ACK]
1038	2018-08-14	21:56:51.009859	TCP	66	42972 - 50052	[ACK]

As we can see controller synchronizer put the polling in the very beginning of the second, namely at the 51.002229, 2229 microseconds from the second start. At time 51.003830, packet 1028 concluded information exchange with switch s1. At time 51.004913 was concluded exchange with switch s3 (packet 1033), data was processed by controller, and decision of policy enforcing was taken. At time 51.009116 command was sent to the switch s2, and at time 51.009859 (packet 1038) information exchange was concluded completely. All exchange took 7630 microseconds, which corresponds to 0.007630 second and is one thousand times shorter than seven seconds needed for our decision function to discover the attack.

Nevertheless, exchange time must be taken into consideration, because for big number of switches it can become considerably large. For example we can say, that scalability of our testing environment, with polling time equal one second, is limited by five hundred switches.

4.2 ONOS internal application

Structure of testing environment in this experiment is exactly the same, as in Experiment described in Section 4.1, with the same types, parameters and positions of generators, listeners, counters and policy enforcers. The only difference - now our testing environment is under control of real SDN controller - ONOS, inside which works our internal application. This solution is much more "heavyweight", and our decision function will work with different parameters. Parameter alpha stays 0.4, but both thresholds are shifted down by 100 packets each, and now we have high threshold on 250 packet per second, and low threshold on 200 packets per second.

Also, because of the "weight" of the ONOS, in this experiment appeared starting order of the testing network components:

- First, starts ONOS. It takes about three minutes even with empty configuration. This time is not included in the experiment time.

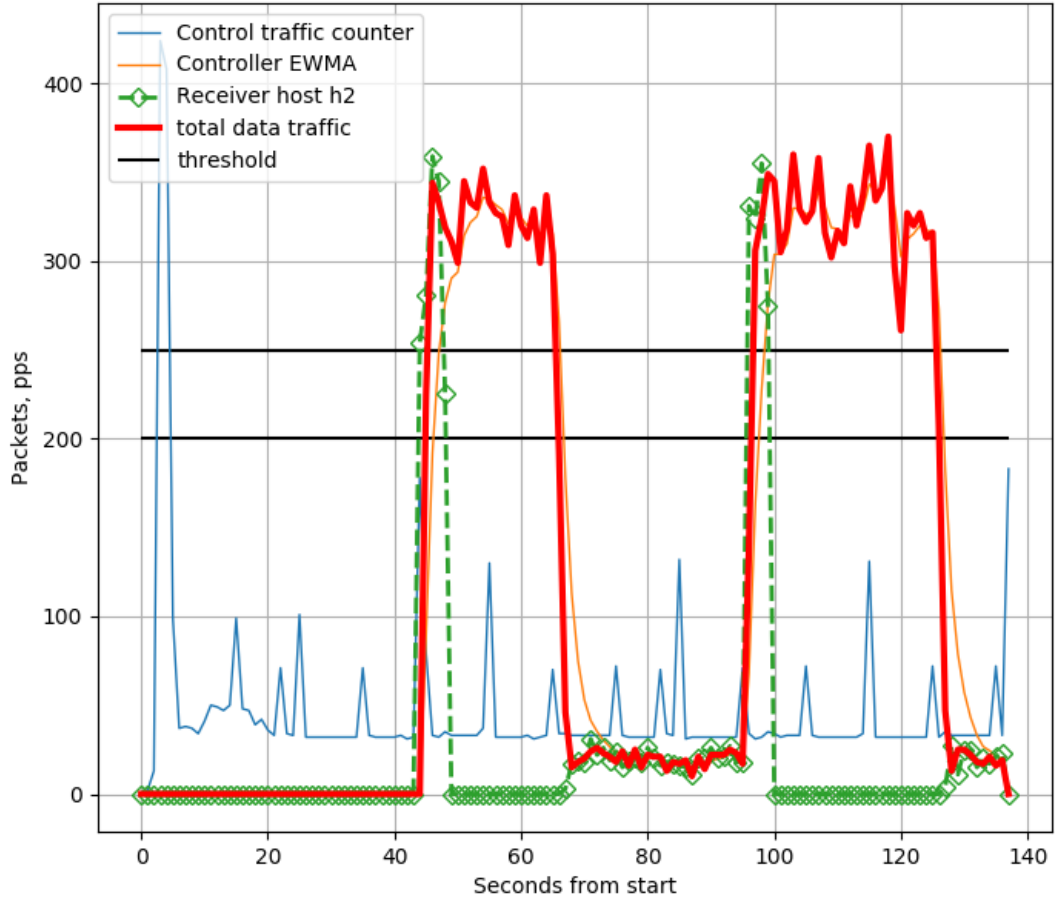


Figure 4.5. ONOS Experiment traffic in PPS

- Second, starts Mininet network emulator together with listeners and generators. This moment considered as start of our experiment.
- Third, starts our ONOS application.

Full duration of our experiment can be seen on the Figure 4.5, but in order to simplify our analysis we are splitting it by two smaller figures, Figure 4.6 and Figure 4.7

On the Figure 4.6 we can see the start of our experiment, from the starting moment of the Mininet simulator, until second 80. Let's look at it:

At time 1s our testing network is started. By this moment ONOS is already running, it receives "hello" from our switches, recognizes BMV2 switch type, and writes P4

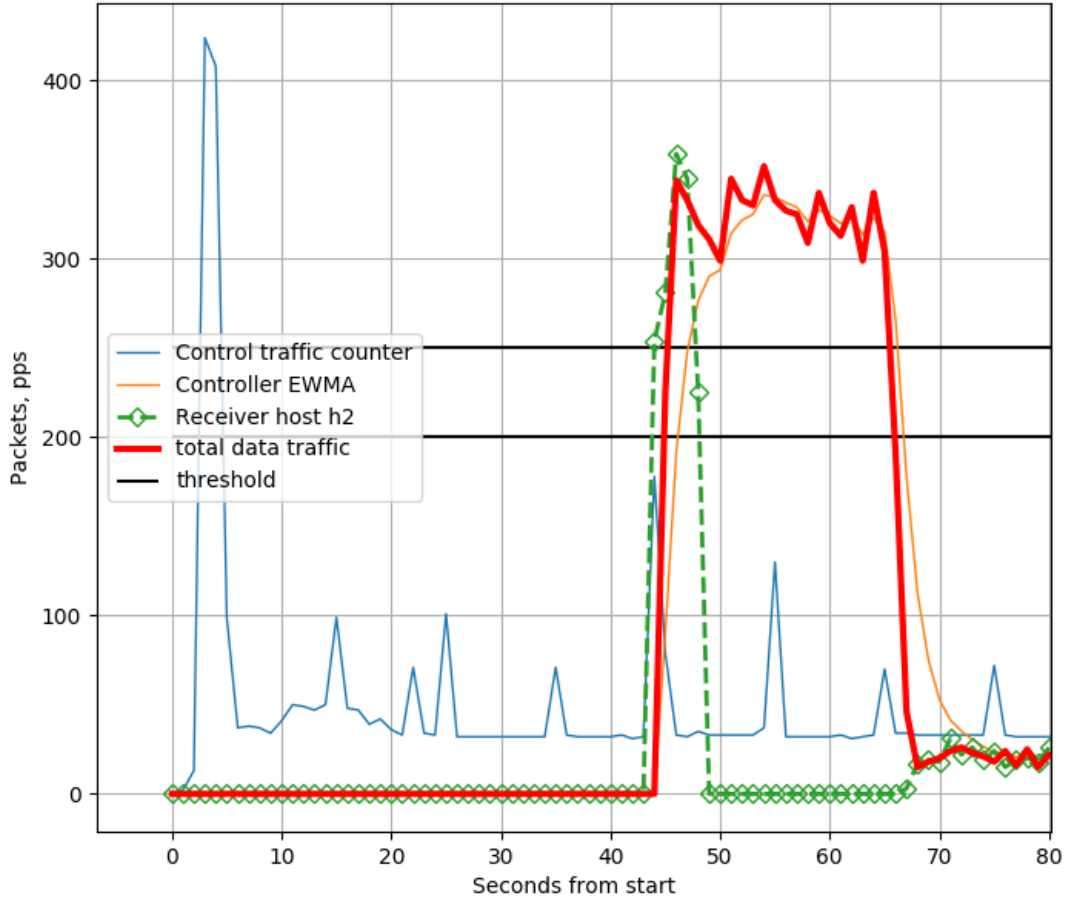


Figure 4.6. ONOS Experiment traffic in PPS

program into each switch. On the graph we can see corresponding peak of the control traffic.

From time 2s until time 43s we can see ONOS transient period and then stable ONOS behaviour. Should be noticed, that during this period our network has no configuration - we hadn't write any rules into switches yet.

At time 43s our application is started. It writes forwarding rules into switches and traffic starts moving through our network. On the graph we can see corresponding spike of control traffic, lower than previous. By this time our generators are already in "attack mode", and the very first data received by our application is already higher

than attack threshold.

At time 47s decision function discovers the attack. Policy enforcer switch receives corresponding command and receiver stops receiving traffic. Should be noticed, that in this experiment policy executed with some delay, and our receiver received some packets from next second. Reasons of this behavior will be explained later.

At time 67s our generators returns to normal traffic, on the same second decision function registers the end of attack, and commands to restore normal network behavior. On the graph we can see, that receiver counter graph moved up.

In the first part of the Experiment (Figure 4.6) we described interesting, but non-standard situation, our application was started right during the attack. Second part of the Experiment, on the Figure 4.7, describes more "normal" situation, where our application is already activated, and attack starts after period of normal network activity, like in our previous experiment, described in Section 4.1.

Here we can see:

From time 68s until time 95s we can see normal network behavior under normal network load. Should be noticed, that receiver graph does not correspond precisely with the total data traffic graph, produced by our application. This behavior will be explained later.

At time 95s both generators switch to attack traffic.

At time 99s decision function discovers the attack, and enforces policy. Should be noticed, that in reality decision was taken and executed between 98th and 99th second, receiver got some traffic from 98th second. On the graph we can see, that during 98th second, total data traffic graph is still increasing, but receiver graph became decreasing.

From time 99s until time 127s we can see behavior of our system under attack.

At time 127s our generators return to normal traffic, on the same second decision function discovers the end of attack and returns the system to normal behavior.

It's time to explain the reason, why total data traffic and receiver graphs do not correspond precisely. As was explained in Section 3.5, ONOS polls switches according to its own schedule, keeping consistent only polling interval (in all our experiments - one second). In result we have desynchronized counters. For example: Receiver counter binded to the absolute time, in starts counting period at the beginning of the second, and finishes at the beginning of the next. In the same time, polling for the switch in ONOS may have place at 400 milliseconds from the beginning of the second, and repeats once per second, making effective counting period from 0.4 to 1.4 second. Another switch may have another polling shift, which makes situation even more complicated. Our ONOS application considers all polling results, gathered during statistics gathering interval, belonging to this interval, but in reality these results include some data from previous interval and not include some data



Figure 4.7. ONOS Experiment traffic in PPS

from current. In the worst case it makes worse reactivity of our system, adding worst case delay equal to duration of statistics gathering interval.

Due to the same reason we have a delay in policy enforcing/cancelling. Our Internal ONOS application is binded to the ONOS events, which may not correspond to the beginning of the next statistics interval (second). Processing starts with first polling event of the new statistics interval, and therefore, policy execution shifted from the beginning of the second to the middle.

This better be explained by Figure 4.8. Receiver interval is binded to the absolute time. ONOS polling intervals have the same length, but binded to the other moments in time, chosen by ONOS, and therefore shows different values in their counters. Due to the

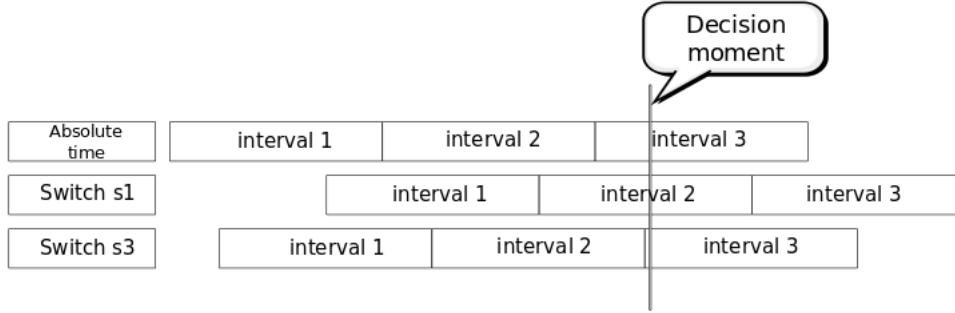


Figure 4.8. Statistics gathering intervals shift

same reason we have a delay at decision function. For example, let's consider interval 2 on the Figure 4.8. During this interval, our application received and stored data from switch s3, gathered in interval 1, and data from the switch s1, also gathered during interval 1. Event, which triggers our decision function, is the first event after the end of interval 2, in this case - end of polling interval 2 of switch s3. This way, decision moment appears in the middle of the interval 3, and at this moment function processes the results from the interval 1.

Technically, we can call decision function after each ONOS event, related to statistics gathering. It can shorten the decision interval and better the reactivity of the system, but with large number of switches it will create considerable computation load to the ONOS servers. Also, we can notice, that with large number of switches will solve itself - large number of events will be nearly uniformly distributed during the interval, it will shorten the interval between end of the previous interval and the first event of the next.

On the Figure 4.9 is shown results of our experiment in order to evaluate amount of control traffic, needed for this solution.

As we can see from this graph, amount of control traffic is relatively stable, and, as in our previous solution from Section 4.1, it does not depend neither on the presence of the attack, nor on the policy status, nor even of the status of our application. Baseline of control traffic graph is statistics gathering mechanism, which polls all switches with frequency once per second. Spikes of the control traffic graph are formed by switch availability check ONOS mechanism, working once in 10 second in base mode, and once per 30 seconds in extended mode (BMV2 "hello")

4.3 Reference solution

This solution exists in order to get reference value of control traffic in SDN networks without special abilities of P4, used by our application. Testing environment for this solutions is exactly the same as described in Section 4.1 and was used in all our Experiments. In

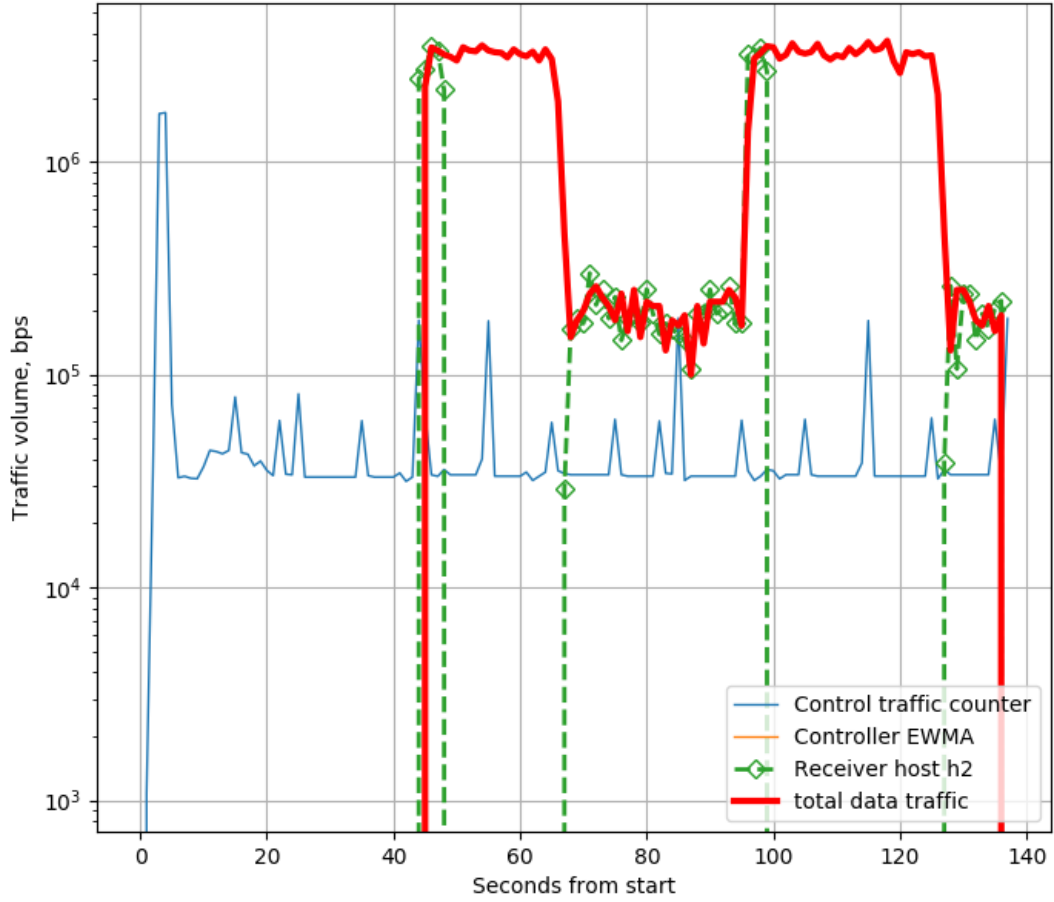


Figure 4.9. ONOS Experiment traffic in bits per second

this experiment our test network uses switches of Open vSwitch type, instead of BMV2 and in ONOS configuration BMV2 driver is changed to Openflow-base application, which gives as classical OpenFlow testbed. Figure 4.10 presents results of the experiment.

Our ONOS internal application is used in this experiment, but its scope is very limited. It is able to setup forwarding rules, using ONOS FlowObjective Service, but it attempt to setup counting tables fails, due to the fact, that Open VSwitch does not supports P4 specific commands. In result ONOS does not setup traffic counting scheme, provided by our application, and falls back to the standard OpenFlow statistics gathering, by means of FlowStatistics mechanism. It gives us possibility to evaluate amount of control traffic in the standard OpenFlow configuration. Several words about the graph presented on the

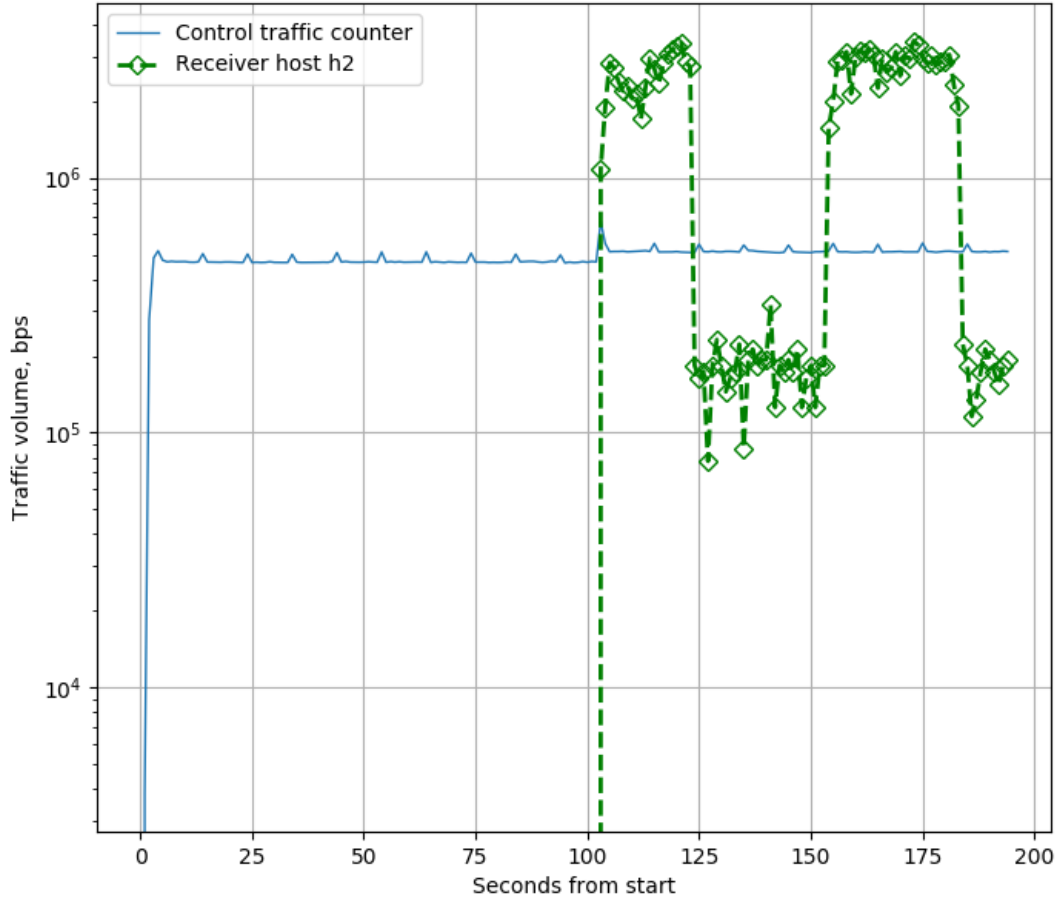


Figure 4.10. ONOS OpenFlow Experiment traffic in bits per second

Figure 4.10

From the start of the Experiment until 100th second we see behavior of the network, which has no rules. All switches are up, but traffic is not going through network. Statistics gathering mechanisms are working idle.

From the 101th second to the end of the experiment we see network behavior under load. Our application started, wrote forwarding rules into switches, failed to create counting tables and worked idle during this period. Amount of control traffic changed for this period, but not much. And, as in our previous experiments, amount of control traffic does not depend on the presence of the attack.

Should be noticed, that in case of OpenFlow switches we don't see usual spike of the traffic in the beginning of the experiment. This one more time underlies the difference between the structures of OpenFlow and P4. OpenFlow based on predefined set of fields and actions, known to both controller and switch, this structure doesn't need any initialization, switch is ready to action immediately after turned on.

4.4 Comparison

In order to compare amount of control traffic dependency in our experiments we put control traffic graphs on the same picture, presented on Figure 4.11.

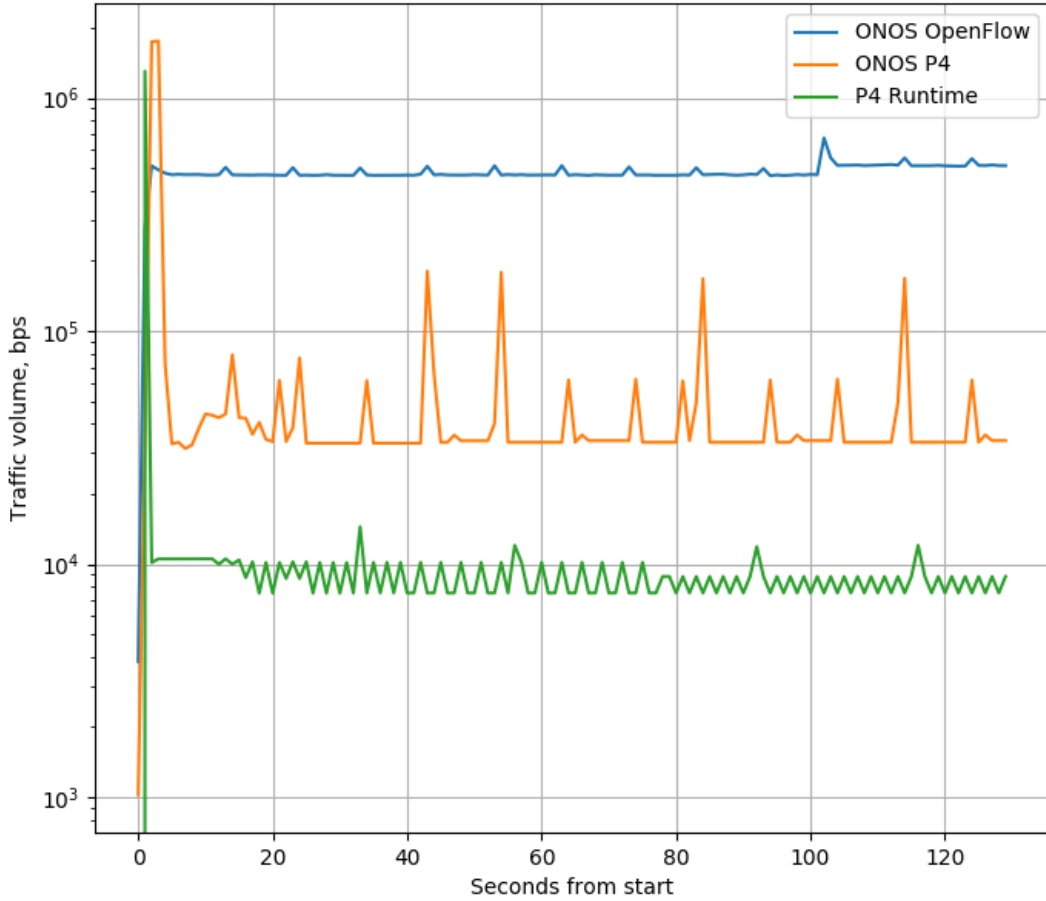


Figure 4.11. Control traffic comparison in bits per second

As we can see on the graphs, our P4Runtime solution, described in section 4.1 has the lowest amount of control traffic. This solution is kind of baseline - we collect minimum information, only for our purposes.

Next from it goes our internal ONOS application. ONOS collects data from all data from all switches, but our application managed to present needed data in compact form.

Above all, we see standard OpenFlow statistics gathering through flowcounters. It is ten times higher than our ONOS P4 solution.

Chapter 5

Conclusion

In this work we have shown one of the ways to control the expenses, needed for the purpose of discovering network DDoS attacks. Proposed solution allows to make significantly lower the amount of control information, transmitted between controller and network devices. Also, proposed solution has the structure, which allows to use much more advanced methods of processing gathered information, without radical changes of the solution structure.

In the first part of this work we evaluated protocol P4. In our opinion, this solution solves the problem of network protocol numeracy, allowing to "teach" switch chip to understand protocol fields and corresponding set of action for any existing protocol. This also solves the problem of a new protocols, because any emerging protocol can be described by means of P4 as a number of information fields and possible actions. Description, made this way, can be uploaded into any P4-compliant switch, even if mentioned switch already works as a part of a network, making the switch, and therefore the whole network able to understand new protocol. Capabilities of P4 can significantly speed up the implementation of new protocols, in the same time allowing to create custom solutions, which support only those protocols which needed for particular network.

We evaluated protocol P4Runtime, which allows to control and adjust operations of any P4-compliant switch at runtime. This protocol allows full spectra of control operations on the switches, including interchange of switch P4 programs almost without interruption of operations. Also, combination of P4 program and P4 Runtime control interface made possible very flexible ways of statistics gathering, and therefore allow to lower significantly amount of control traffic, needed to perform special operations, like DDoS attack detection. As we mentioned before in this work, structure of P4 Runtime is very different from its predecessors, like OpenFlow, and from conventional ways of network control, based on network protocols. This structure more resembles objects interdependence, typical for programming, than any set of abstractions, typical for network engineers. In our opinion, adoption of P4 Runtime protocol in networking community is going to be slow.

We evaluated capabilities of both P4 and P4Runtime in virtual testing environment, on the scenario of DDoS network attack detection. For our first solution we wrote our own version of SDN controller on Python, and evaluated our ability to lower control traffic in SDN network. Our solution, basing on abilities of P4 and P4Runtime, allows us to successfully detect DDoS, while transmitting more than ten times lower amount

of control traffic. Also, this solution allows us better evaluate way of operations of P4 and P4 Runtime, better understand the structure of control traffic, and evaluate, in first approximation, parameters influencing reactivity of our DDoS detection system and its scalability.

Also, we evaluate ability of our solution to work inside real SDN controller - ONOS. We rewrite our solution on JAVA, according to principles of OSGi, on which is built ONOS, and include it into ONOS structure as one of the ONOS application. Our experiments have shown us, that our solution can work successfully inside real multithreaded industrial SDN network controller, and still allows us to lower amount of control traffic needed for DDoS attack detection, this time less than ten times, as in our previous experiment, but still significantly. Also, these experiments shows us, that set of parameters influencing reactivity and scalability of our solution is different in case of industrial network controller, in particular, that network RTT in this case has much less influence because of multithreading way of work of the controller.

Bibliography

- [1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," in Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, Jan. 2015.
- [2] K. Greene, MIT Tech Review 10 Breakthrough Technologies: Software-defined Networking, <http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/>, 2009.
- [3] S. Schenker, The Future of Networking, and the Past of Protocols, October 2011. [Online]. Available: <http://www.youtube.com/watch?v=YHeyuD89n1Y>
- [4] M. Moshref, A. Bhargava, A. Gupta, M. Yu, R. Govindan, "Flow-level state transition as a new switch primitive for SDN", Proceedings of the Third Workshop on Hot Topics in Software Defined Networking ser. HotSDN '14, pp. 61-66, 2014.
- [5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, "P4: Programming protocol-independent packet processors", SIGCOMM Comput. Commun. Rev., vol. 44, no. 3, pp. 87-95, Jul. 2014.
- [6] G. Bianchi, M. Bonola, A. Capone, C. Cascone, "OpenState: programming platform-independent stateful OpenFlow applications inside the switch", SIGCOMM Comput. Commun. Rev., vol. 44, no. 2, pp. 44-51, Apr. 2014.
- [7] C. Cascone, L. Pollini, D. Sanvito and A. Capone, "Traffic Management Applications for Stateful SDN Data Plane," 2015 Fourth European Workshop on Software Defined Networks, Bilbao, 2015, pp. 85-90.
- [8] <https://datatracker.ietf.org/doc/rfc7637/>
- [9] <https://datatracker.ietf.org/doc/rfc7348/>
- [10] <https://tools.ietf.org/html/draft-davie-stt-01>
- [11] <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>
- [12] P4 Runtime - Putting the Control Plane in Charge of the Forwarding Plane <https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html>
- [13] P4 Data-plane telemetry <https://p4.org/specs/>
- [14] <https://wiki.onosproject.org/display/ONOS/Wiki+Home>
- [15] <https://karaf.apache.org/>
- [16] <https://www.osgi.org/>
- [17] Parneet Kaur, Manish Kumar & Abhinav Bhandari (2017) A review of detection

approaches for distributed denial of service attacks, *Systems Science & Control Engineering*, 5:1, 301-320

[18] www.mininet.org

[19] <https://github.com/p4lang/behavioral-model>

[20] www.openvswitch.org

[21] scapy.net

[22] Avallone Stefano, Guadagno S, Emma Donato, Antonio Ventre. (2004). D-ITG Distributed Internet Traffic Generator.. 316-317. 10.1109/QEST.2004.1348045.