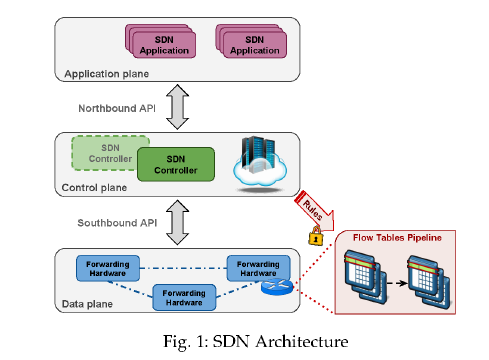
**Security Aspects of Software Defined Networks**

Software Defined Networking emerged as an attempt to introduce network innovations faster and to radically simplify and automate the management of large networks. In this emerging network paradigm, the control and management of the network is separated from the traffic forwarding primitives. The centralized control plane monitors the whole network and makes decisions on packet forwarding for the switches (data plane). The interface to the switches is OpenFlow which provides network administrators with a simple and uniform abstraction to the configuration of different, physical or virtual multi vendor’s network devices. Specifically, the SDN controller inserts and updates traffic rules for the current traffic flows into one or more flow tables inside each switch. The resulting decoupling simplifies network monitoring, fault tolerance, security policy enforcement, but it also introduces new security issues. In this report we discuss the security implications data plane programmability brings about.



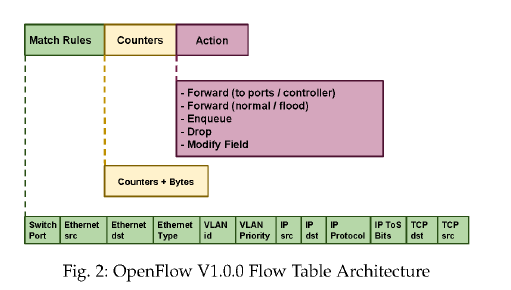
**Northbound API**: This interface is exposed to developers to develop simple-to-use high level abstractions devised to hide the complexity inherent in the underlying network topology and release the network administrator from the need to deal with low level network nodes configuration details. They basically allow the programmability of SDN. However, not only can they be used to develop security applications that use the network equipment as a check point for compliance, they can also constitute a privileged entering point for attackers to introduce malicious applications. Hence, the following preventive and corrective actions must be taken to minimize the risk of exploits:

* Locking the controller as much as possible and disabling unnecessary accounts and services.
* Maintaining the controller up to date
* Permanent monitoring of the activity of key resources such as CPU, memory and I/O.
* Log the activity of the controller
* Define as possible to profile normal behavior with reporting of deviations from the normal pattern.

**Southbound API**: Through this interface, network devices can be controlled, configured, managed and monitored by the controller so as to change their forwarding behavior and adapt to changing business requirements and to make the network more responsive to real time traffic demands. For securing the southbound access, authentication appears important. The recent OpenFlow versions use TLS as transport security protocol. It can also use TCP protocol without encryption. We must ensure the following for providing safe communication:

* The authentication of the controller.
* Authentication of network devices by certificates
* .Confidentiality of exchanged data (encrypted session).
* Integrity of data exchanged.

We now discuss an abstract model of a programmable flow table – referred to as **match/action abstraction** which is amenable to high performance and low cost implementations: capable of supporting a wide range of research and consistent with vendors need for closed platforms.



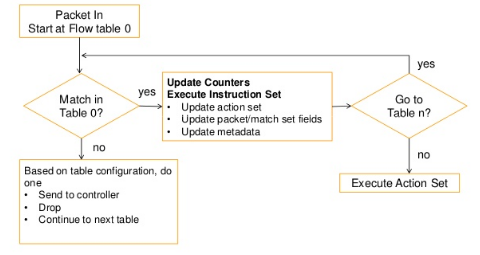
Match /Action abstraction:

1. Matching rule which permits the programmer to broadly specify a flow by matching any arbitrary combination of selected header fields from layer 2 to layer 4.
2. One or more forwarding/ processing actions freely associated by the programmer to the considered matching rule.
3. A field devised to collect statistics on the flow identified by the relevant matching rule, which allowed a meaningful level of programmability and it was immediately deployable on commodity hardware.

Messages exchanged between the control plane and data plane (OpenFlow):

1. Controller - to – switch: Initiated by the controller and used to directly manage or inspect the state of the switch – features, config, modify state, read state, packet\_out, barrier.
2. Asynchronous: They are sent without the controller soliciting them from a switch- packet in, flow removed/ expiration, port-status, error.
3. Symmetric: They are sent without solicitation in either direction- hello, echo, vendor.

The Open Flow instructions transmitted from the controller to the switch are structured as “flows”. Each flow contains packet match fields, flow priority, various counters, packet processing instructions, flow timeouts and cookies. These flows are organized in tables and an incoming packet may be processed by flows in multiple pipelined tables before exiting an egress port.



**Instructions and Action set:**

Each flow entry contains a set of instructions that are executed when a packet matches a flow entry. Instructions contain either a set of actions to add to the action set, contains a list of actions to apply immediately to the packet, or modifies pipeline processing. A flow entry modifies action set using write action or clear action instruction. Processing stops when the instruction does not contain a goto-table and the actions in the set are executed.

**Secure channel:**

It is the interface that connects the open flow switch to the controller. A controller configures and manages the switch, receives events from the switch, and sends the packets out the switch via this interface. It establishes and terminates this connection through connection setup and connection interruption procedures. The SC is a TLS connection which mutually authenticates by exchanging certificates signed by a specific private key.

**Stateful SDN data plane proposals:**

Motivated by the latency and signaling overhead that comes along with such a two tiered SDN programming model, innovative switch level programming abstractions capable to deploy some smartness directly inside the switches for localized stateful flow processing have been employed. The basic idea of stateful SDN can be summarized as:

1. Keeping the state information of the flows inside the switch and allow programmatically formalized packet level state transition. (state of the incoming/outgoing packets can be changed due to sending/receiving new packets)
2. Giving the capability of making forwarding state update decisions on the switch, based on the local state information of the flows without the need to contact the controller.

Even in stateful data plane, the same as basic SDN switches, routing decisions are based on a list of pre-defined action rules imposed by the controller. However, the stateful nature of the data plane gives the opportunity to choose an adequate action based on the historical action of the flows, independently. The underlying principle of flexible and efficient forwarding state re-configurability and data plane programmability is common amongst proposals.

**Platforms and enabling technologies:**

1. **OpenState** (special class of eXtended Finite State Machine, namely mealy machine to provide data plane programmability): XFSM is modeled by 4 tuples (S,I,O,T), where S is a finite set of states including a starting state S0 (default), I is the finite set of inputs (events), O is the finite set of outputs (actions) and T is the state transition function (rule) that maps <state, event> pairs into <state, action> pairs.

In open state, each switch stores 2 distinct tables:

1. State table that stores the current state of the flows based on the received packets related to that flow.
2. XFSM table which is used to define the rules (i.e. state transition) based on the received packet’s <state, event> information.

To handle an incoming packet, the open state switch performs:

* A state table lookup to retrieve the packet’s state; the state of the flow is determined through flow ID (eg. source IP address) which is the key for table lookup. If there is not match in the state table, the switch assigns a “default” state to the incoming packet. The switch then appends the retrieved state label to the packet as metadata field.
* The second step is the XFSM table lookup to find the matching rule with <state, event>, performing the associated action and updating the state field of the packet based on the predefined “next state” field of the XFSM table.
* State table update for the corresponding flow ID based on the retrieved “next state” field from the previous step.

It is evident that the switch does not need to contact the controller for each received packet. Instead it makes, routing decisions based on the received packet’s current state and the predefined rules for each <state, event> pair by the controller.

1. **FAST** (this design has pre-installed state machines inside each switch which are dedicated to special applications):

The control plane has 2 main components: the compiler and the switch agents. The switch agent takes care of the state machine functionality inside the switch. It also performs some computations such as flow rate locally by receiving updates from the switch. It can also manage memory restrictions for confined switches by implementing only a part of the state machine onside the switch. It basically handles the communication between the switch and the controller and updates the controller about the local states of the switch.

The data plane consists of four tables: state machine filter, state table, state transition table and action table. The state machine filter table selects the corresponding state machine related to an incoming packet, while the state table stores the state information of the incoming flows. State table is a hash table which maps each packet header (eg. source IP or destination IP) to the corresponding flow’s state. State transition table is used to define the next state of the incoming packet based on its current state and packet fields. Finally, the action table specifies an action to be performed for an incoming packet, based on the packet header and its new state. However, , the controller has the control of network status and online debugging of the network, proposed switches send a copy of the packets to the controller per state transition.

3. **SDPA**

The SDPA platform consists of 3 tables: state table, state transition table, action table and a state processing module called the forwarding processor (FP). The match field of the state table can be any combination of the header fields of the packet. Depending on the match field and the state field, the state table has a state operating instruction or a packet processing instruction. The state transition table and action table define the next state and the corresponding action for the current flow. Each application in SDPA has dedicated instances of these 3 tables. The controller communicates with the FP module to initiate the state table upon receiving the first packet of the flow. The other packets from the same flow will be processed locally inside the switch and do not need to communicate with the controller. However, the controller still has full control and updated information of the state table as it receives updates from the FP. (periodic updates or event triggered updates is also decided by the controller).

**Compilers, programming languages and frameworks:**

1. **P4** (Programming Protocol independent Packet Processors):

This method helps the switch to parse new header fields, as the programs written in P4 can be mapped to several devices without knowing their underlying hardware technology. The programmers determine the packet processing procedure using P4. The program written in P4 is translated into a Table Dependency Graph (TDG) which defines dependency between header fields, corresponding actions and flow processes. The compiler subsequently maps the TDG to the target switch based on the available hardware resources.

P4 consists of 5 main components:

* Headers: Which describe header fields such as field name and field width.
* Parser: It defines the state machine that should traverse the packet header. The state machine is defined as a set of state transitions which is actually a transition from one header to another by matching header field values with match/action table rules.
* Tables: It is used to process the packets. The programmer defines the match conditions and corresponding action to each match.
* Actions: Defines some action such as add tag.
* Control Programs: Specify the order in which the packet should traverse the match/ action table.

1. **Domino** (data plane programming language):

Domino adopts “packet transaction” sequential code block through which programmer concerns about the operation on a single packet. To have a stateful data plane algorithm, the programmer provides an application written in Domino which is compiled by the Domino compiler, which is used in the Banzai machines through normalization, pipelining and code generation.

1. **SNAP** (Programming language and compiler for the stateful data plane):

Using SNAP, a programmer defines some global variables and arrays in order to store the state information of the flows. SNAP policies are defined for an abstract network topology, which means that the programmer does not need to know how and where to store the state information. The policies defined by SNAP and the automatic update of network variables and states are distributed through the network switches by the SNAP compiler automatically.

1. **Event driven programming** (moving from one configuration to another upon occurrence of an event):

To implement a stateful program, the Network Event Structure (NES) helps network designers to program switches so that they are able to store the incoming events, route the packets based on these events and announce other switches about the event.

**DoS (Denial of Service) Attacks**

The basic idea of DoS attacks is to overload network links or flood a victim with a massive number of flows until it fails to serve legitimate users. Generally, there are two modes in which rules can be installed at switches: proactive and reactive. For the proactive mode, the controller first breaks down network policies into flow rules, and installs them at switches when the network bootstraps. For the reactive mode, the controller computes and installs rules only when a switch explicitly requests them. Clearly, reactive mode enables switches to quickly adapt to network dynamics and does not require switches to have large flow tables. However, the reactive rule installation also makes SDN controllers and switches vulnerable to denial of service (DoS) threats.

**Background on Reactive rule installation:**

We consider a network that deploys OpenFlow, the de facto standard for SDN. An OpenFlow switch consists of application specific integrated circuit (ASIC) and an OpenFlow agent (OFA). The switching ASIC is a piece of hardware holding one or more pipelined flow tables, and each flow table consists of a set of flow rules that indicate how to process packets. The OFA is a software agent that talks with the controller using the OpenFlow protocol. The connection between the OFA and the controller is termed the control channel.

The process of reactive rule installation is described as follows: When a packet comes to a switch, the switch first looks up in its flow table using the packet header and input port. If there are no matching rules, the switch treats the packet as the first packet of a new flow and triggers a table\_miss event to the OFA. Then the OFA stores the packet in the packet buffer, encapsulates a fraction of the packet header into a flow request, and sends it to the controller via the control channel. When the packet buffer becomes full, the OFA sends the entire packet to the controller without buffering it.

Upon receiving a packet\_in message, the controller computes a set of rules and installs these rules at the switches along the path of the flow. Each rule is assigned a hard-timeout and a soft-timeout that jointly determine how long the rule can exist. On receiving the rule, a switch’s OFA checks whether its flow table is full. If not, the rule is inserted into the flow table; otherwise, the rule is dropped and an error message is sent to the controller. However, the latest versions of Open Flow have allowed the switches to evict the flow rules without consulting the controller. This means that switches can locally delete rules in order to make space for new flows.

**Control Channel Security and DoS**

DoS attacks against the SDN controller may be possible in the following three scenarios:

1. Unauthenticated Channel DoS: In the absence of an authentication mechanism, switches basically communicate with the controller through plain TCP connection. That is, only controller IP address and the port number are required for communication. Without authentication, an outsider can pretend to be a switch and flood the controller with a large number of packets.
2. Man-in-the-middle DoS: In this attack, the link between the controller and the individual switches is disrupted. First, an attacker (acting as a fake switch) which has an access to the network, obtains a target switch data path ID (DPID). Then the fake switch attempts to connect to the controller by using the obtained DPID and pretends to be the targeted switch. This will force the controller to terminate the legitimate switch-controller connection and diverts the target host traffic towards itself. The attacker basically exploits the Host Tracking System (HTS) of the controller which lacks an authentication mechanism for host mobility in SDN. As a result, network performance degrades as existing rules expire in the switch tables. This attack is known as process disruption DoS attack. Resource consumption is also possible by impersonating a switch or the controller.
3. Compromised Host DoS: In this case, an attacker utilizes the legitimate credentials of an authenticated switch to consume the controller resources. In cases of the link fabrication attack, the malicious OpenFlow switch relays the packets to another OpenFlow switch instead of forwarding packets to the controller. Subsequently, upon reception of LLDP packets by the new OpenFlow switch, the LLDP packets are sent to the controller in the form of a packet-in message. This tricks the controller into believing that there exists a link between the malicious switch and the legitimate switch. Such fake link injection attracts other attacks such as DoS attack, man-in-the-middle attack and more.

Controller Vulnerabilities

1. Host Tracking System: The vulnerability in HTS attracts attackers to hijack the location of the hosts. The host profile in the controller contains the DPID, ingress port ID, and other metadata information which exhibits the controller with the location of the host and the connected OF switch. The key exploited vulnerability includes the lack of authentication mechanisms that can be used to verify host updates received by the controller through Packet-In message. All updates related to host locations are considered genuine and accepted (due to the existing empty shell API). The attacker simply spoofs the packet with target host identity and forwards it to the connected switch which further sends it to the controller in the form of a Packet\_In message. The controller assumes a shift of position of the host and updates the host profile for the target host. The lack of authentication mechanism in HTS makes the controller update the topology with falsifies host information effecting routing and other services.
2. Link Discovery Procedure: Firstly, there is no authentication mechanism for the controller to ensure the origin of the LLDP packet. Secondly, the controller is unable to verify the traversed path used by LLDP. Addressing these issues is critical in preventing the insertion of fake links.

**SPHINX**

Sphinx is an anomaly detection mechanism which can detect an unknown attack on network topology and forwarding devices. It provides a real-time and accurate verification solution of the network behavior by monitoring all OpenFlow messages, analyzing features set of the messages and providing a fast validation of the network updates. It then constructs a trust worthy topology graph, flow graph and uses it to detect anomalous messages from the switches. The basic idea is that messages from the controller to the switches are trustworthy, but the messages from the switches to the controller can be forced. Hence, the flow graph is constructed using only Flow\_Mod messages issued by the trusted controller. With the flow graph built, the SPHINX raises alerts when it detects untrusted entities triggering changes to existing flow behavior or if the flow violates any administrator specified security policy. SPHINX basically exploits predictability and pattern in topology and data plane forwarding to detect violations.

**Information Hiding Based Control Channel Authentication** (providing secrecy through obscurity)

Ever since TLS was made an optional connection mode, many vendors do not support TLS and utilize plain TCP due to complexity of configuration in generating certificates, lack of support from both controllers and switch vendors simultaneously, its CPU intensiveness and the rapid evolvement of OpenFlow. Information hiding exploits various types of carrier such as digital images, videos and more recently network protocols. Network information hiding in this case utilizes packet headers and protocol mechanism to hide information. Here, we utilize IP identification (IPID) field of IPv4 to verify the authenticity of SDN switches and controllers. The main purpose of IPID in IP protocols is to recover a packet after the IP fragmentation process. Fundamentally, the mechanism performs an XOR operation between a 16-bit unique ID (given to a switch/controller by the network operator) and a 16 bit random sequence. The 16 random bits are taken from the packets such as LSBs from 16 bytes or 2 bytes of OpenFlow transaction ID (xid, used to facilitate request reply pairing). The XOR operation creates a randomized table entry. Subsequently, a lookup table is utilized to determine the 16 bit decimal value as the output IPID to be used for the packet. Frequent lookup table permutations can be performed to increase resistivity against brute force attacks.

**Threat 1: Switch Software Overloading**

Threat: As discussed above, the OFA of the victim switch should generate a flow request and send it to the controller. Since OFA’s generally run on relatively low end CPUs, they can only generate a limited number of flow requests per unit time. Thus the OFA is indeed a bottleneck and flows from benign hosts may be delayed or dropped. However, this threat has a local impact, only those hosts connected to the victim switch are affected.

**Countermeasure**: To mitigate such a DoS threat, an overlay network of software switches is used as a complement to hardware switches since software switches can run on more powerful CPU’s and hence can generate many more flow requests compared to hardware switches. New flows received by hardware switches would be redirected to software switches, which are responsible for generating flow requests. Data plane traffic can still be forwarded by hardware switches for large throughput. Scotch just increases the number of new flows that a switch can handle in benign settings, while it may not be enough in adversarial settings.

**Threat 2: Control Channel Congestion/ exhausting the control plane bandwidth**

Threat: Even though switches individually maintain their control channels with the controller, these logically separate channels may share some common physical links. Thus flow requests flooded by the victim switch may overwhelm common bottleneck links, and normal flow requests using those links experience congestion. In addition, if the packet buffer of the victim switch sends entire packets instead of just packet headers to the controller, it results in even higher bandwidth consumption. Compared to the switch overloading threat, which only affects those hosts connected to the victim switch, control channel congestion affects all hosts with flow requests traversing the congested links.

**Countermeasure**:

**Floodlight** uses a simple port throttling method to prevent switches from requesting too fast. Specifically, the controller keeps monitoring the packet\_in rate for each switch, and once it exceeds a threshold, the controller begins to sample every packet\_in sent by that switch. If two packet\_in s from the same MAC address/switch port are sampled, the controller installs a rule at the switch to block flows from that MAC address/switch port for 5s. This simple throttling method has two problems. First, the threshold is a fixed value and cannot adapt to network dynamics. Second, blocking all flows from certain MAC addresses or ports may be too aggressive, since there may be multiple hosts connected to the same port, and blocking that port can punish benign hosts too.

However, the security enhanced floodlight introduces a security enforcement kernel (SEK) which includes a digitally authenticated northbound API. An administrator is required to pre-sign the OpenFlow applications class, which may be digitally verified by the SEK at run time. Once signed and validated, the application has permission to modify or query the network, or traffic on the network. The distinction between SE-Floodlight and PermOF and OperationCheckpoint is in the granularity of approach. PermOF and OperationCheckpoint allow a set of actions to be granted to an application rather than validating the complete application.

**Additive Increase Multiplicative Decrease (AIMD)**

AIMD is a better way to dynamically adjust the requesting rates of switches, instead of temporarily blocking. Once the controller detects that it is receiving more flow requests that it can handle, it can instruct the switches to reduce the requesting rates. A rate adjustment method was proposed to mitigate distributed DoS (DDoS) attacks in traditional IP networks. In this method, when the victim server notices that it is overloaded, it multicasts a rate-limit signal with parameter rs to routers. Each router adjusts their sending rates to be strictly less than rs. If the server is still overloaded, rs will be cut in half and multicasted again. On the other hand, if the server load is smaller than the threshold, the server increases rs additively.

**Resource Management of Switches and Controller during Saturation time to avoid DDoS**

When a new packet arrives for which there is no flow rule in the switch, the packet will be placed in the switch’s buffer and the packet header will be sent to the controller. The controller replies by installing a new flow rule in the switch flow table in response to the request. Hence, by default packet requests are addressed in the first come first serve basis due to which the legitimate user’s requests are dropped. The proposed system aims to keep the control plane in running state efficiently even when suffering from data-to-control plane saturation by assigning less timeout value for the flow rule in peak time. We also aim to differentiate between fake packet request and normal packet request by calculating the trust value for the host.

The process for achieving the same is described as follows:

When a packet arrives at the switch from the host and if the switch has a route for the packet, it will process the packet. However, if it does not have a route for the arriving packet, then it will check the trust value of the sending host. (if the source IP is not mentioned in the trust list index tehn TVS=1, else it is assigned the index value decrementing it if it has been suspended by earlier controller) According to the trust level, the priority is set for the packet and the packet will be placed in the switch buffer. Controller checks if it is peak time, if not controller replies to the switch by installing a new flow rule with normal timeout and the packet will be processes (mean value of packet transfer per session for every node and variance of every currently requesting node with respect to the mean value). If it is identified as peak time, then it will check for the number of requests from that particular host exceeds the threshold value or not. If the request received exceeds the threshold value, then it suspends the particular host for that peak time. If not, then the controller replies the switch with new flow rules, but with low time out value.

**Single Layer Fair Queuing (SLFQ, Mitigating the Controller Resource Saturation Attack)**

Different from previous approaches, our method works on the controller side and does not need any modifications of switches or any extra data plane services. In SLFQ, the controller maintains a queue for each switch. When a request comes, the controller assigns it to the queue of the corresponding switch. Then the controller uses weighted round robin to poll requests from these queues for processing, hence ensuring fairness among switches. However, this method incurs a large overhead as hosts under the same switch as the attacker may be punished unfairly since their flow requests share the same queue. Maintaining a queue for each port will solve the problem, but this will end up in even more queues taking up more memory.

**Multi-Layer Fair Queuing (MLFQ)**

To address the challenges with SLFQ, we propose MLFQ. The basic idea is to maintain a small number of queues at the controller when there are no attacks, and dynamically expand a queue into multiple sub-queues when its size exceeds a threshold (indicating the existence of flooded requests in this queue). If the sizes of these expanded sub-queues all drop below another threshold, they are aggregated into a single queue again. For example, initially each queue corresponds to a group of switches. When the size of a queue is beyond a threshold, it is expanded into per-switch queues. If the size of the per-switch queue is beyond the threshold, it is further expanded to per-port queues. Finally, the queues in the controller are organized in multiple layers, and the layer a queue is at depends on how many switches or hosts it aggregates.

**Threat 3: Controller Resource Saturation**

Threat: If the flooded flow requests arrive at the controller, they will consume the controller’s resource (CPU, memory, bandwidth etc.) for rule computation and installation. Without any protection, the controller’s resource can be saturated by the flooded requests and legitimate requests may be dropped. Moreover, the packet\_in messages might get stuck in the controller’s queue itself leading to no more routing decisions for new incoming flows and flows with no forwarding entries will be stuck in the switches. Similar to the control channel congestion, control resource saturation has a network wide consequence as all switches connected to the controller will be affected

**Countermeasure:**

**AVANT-GUARD** is recommended to overcome the control plane saturation attacks by using a connection migration tool. This tool implements a SYN proxy (an application gateway between local network and larger scale network used for increased performance and security) mechanism to prevent SYN flooding. Every time an inbound TCP connection is received, the switch acts as a proxy and sends a flow request once only when a TCP connection finishes a handshake. However, this might also lead to a buffer saturation attack and limitations in the number of connections a switch can proxy. Moreover, Avant-Guard only works for SYN flooding and cannot defend against UDP or ICMP flooding and it also modifies switches which is undesirable for real deployment.

**LineSwitch (improvement over AVANT-GUARD):**

To overcome the control plane saturation attack while avoiding buffer saturation attacks, we needed to remove the possibility of SYN flooding with spoofed source IP addresses by discarding all the incoming packets through proxies. Hence, LineSwitch was proposed, which is an OpenFlow module deployed on edge switches: this method proxied all incoming TCP connections from a given IP until one was completed while subsequent connections were proxied with a very small probability. Once an attack was detected, the switch blacklists the IP address of the host with a time related to the probability of proxying tailored to the specific needs and history of the network for maximum efficacy.

Only considers the SYN flooding attacks, not designed for other attacks such as UDP or ICMP flooding.

**FloodGuard** (a scalable, efficient, lightweight and protocol independent defense framework which can defend against general flow request flooding attacks). It proposes two modules

1. Proactive flow rule analyzer: it consists of three components: symbolic execution engine, application tracker and proactive flow rule dispatcher. It can be activated anytime such as right after the detection of a saturation attack. Once activated, the analyzer module generates the proactive flow rules and directly installs these rules into the data plane switches. Consider, a packet\_in event handler for example, the function for this event maintains a MAC port mapping table, which can be learned from the source MAC address and incoming port of previous packet\_in messages. The function first checks if the destination MAC address of the packet is a broadcast address. If so, the function simply broadcasts the packet. If not, the function will search this MAC address in the MAC-port mapping table. If this MAC address has not been learned before, the function has no idea which port to forward it so just broadcast it. If the MAC address has been learned before, the function installs a relative flow rule and forwards this packet to the mapping port.
2. Packet Migration: This module consists of two components, migration agent and data plane cache. The migration agent detects the saturation attack using real time rate of packet\_in messages from the data plane and utilization of the infrastructure (buffer memory, controller memory and CPU) to calculate current usage percentage of our OpenFlow network. A potential flooding attack is identified on the basis of a certain anomaly threshold. When the saturation attack is detected, the migration component installs one entry of wildcard flow rule having the lowest priority and forwards all table-miss packets to data plane cache. By temporarily storing these packets, we do not overload the switch or flood the controller. Some packets are given priority to trigger packet\_in messages from the data plane cache. The migration agent also instructs the rate limit to the data plane cache.

The simulation performed considers one controller and only three clients which is not a practical scenario for real time applications. Even with three clients there is an overhead approximately equal to 20%, which can be reduced by deciding how many data plane caches are actually necessary for a large number of OpenFlow switches. Ideally, only one data plane cache has been deployed in the paper to serve all switches. However, to be more scalable, we could use a set of data plane caches each in charge of a subset of switches.

**Controller Protection Protocol**

Despite the robust designs for SDN controllers, these systems represent attractive targets for malicious attackers because the basic operation of SDN exhibits an imbalance between the small amount of work necessary to trigger the large amount of work which is performed by SDN controller (route computation for each packet and filling the flow tables). To level this imbalance, the Controller Protection protocol was introduced which requires systems wanting to connect through an SDN network to commit considerable resources before an SDN controller commits resources for route computation and setup. Hence, the connecting system needs to include a proof of work (POW) with the initial packet of a connection which can be verified easily and thus attack traffic with invalid POW’s can be discarded easily. When the first SDN switch encounters the packet from this new connection, it forwards the connection information, including the proof of work to the SDN controller, where its validity is checked. Later packets of the verified connection will include a proof of work, but are forwarded by the SDN switches as in conventional SDN. In this approach, an attacker needs to dedicate a large amount of computational resources in order to send large amounts of attack traffic, thus making an attack potentially prohibitively expensive.

**FlowRanger**: A request prioritizing algorithm for controller DoS attacks (implemented on the controller side, this guarantees that legitimate flows are served first in the controller)

1. The trust management component that calculates a trust value for each packet-in message based on its source.
2. The queuing management component that places the message in its priority queue corresponding to its trust value.
3. Message scheduling component that process messages according to a weighted Round Robin strategy.

Based on the IP filtering technique:

The proposed scheme analyzes user behavior and uses it to assign the timeouts for the flow entries. Short timeouts are assigned for malicious user flows and long timeouts for trusted ones. This forces the traffic to be quickly removed from the switches TCAM tables. However, this leads to new packet-in messages to be sent to the controller if the flow duration is higher than the set timeout.

Self-Management scheme which leverages centralized design and programmability:

In this proposal, the ISP and its customers cooperate to mitigate the DoS attacks. The ISP collects threat information provided by customers in order to use it to enforce security policies and update flow tables in the network. If a flow is considered legitimate by the customers, the ISP controller will tag it with a high priority value so that it takes a path with a higher quality. However, if there is a doubt about the legitimacy of the flow, the ISP controller will assign a low priority for the flow and direct it to the path assigned for malicious flow. This reduces the impact of the DoS attack on the network performance by balancing the load across different paths.

**Threat 4: Flow Table Overflow/ Switch TCAM memory saturation attack:**

Threat: DoS attacks purposely create a large number of new flows that continuously extend and update the forwarding tables of the switches. While in general, there is no need to keep the state of all the flows traversing a switch, an adversary can “smartly” force this behavior in specific applications and exhaust the memory of the switch. Similar to switch software overloading, this threat has a local impact as it only reduces the throughput of victim switches.

**Countermeasures:**

To provide functionality even when resources or communication bandwidth are over consumed, several methods were proposed. Some versions already implement queues to support the slicing feature which slices the whole network into multiple logical sub-networks. Another aspect is rate limiting, this process can be employed in various ways: by limiting the amount of traffic that a single port can send to the switch, limiting the amount of packets sent to the controller, or limiting the number of rules that the controller can insert into the switch in a short period of time. Flow timeouts can be adjusted, with smaller time-to-live properties, thus making flow tables difficult to be overflowed. An eviction algorithm was also proposed which is a dynamic event-driven method triggering actions such as evicting rules from flow tables and vacating some flow entries for accepting new rules. This was done on the basis of a flow checking module, which analyzes every packet by validating the source IP address by querying log and adding it to black list if it does not pass the validation and removing flows that exceed packet per second threshold and packet bytes threshold.

**Defending against flow table overloading attacks in SDN**

When a flow table overloading attack occurs, the attacker sends a lot of “spoofed” new flows to the targeted switch, installing new rules in the flow table to process the flow. However, these rules are set with a finite timeout and they will disappear from the flow table. However, under attack the rate of installing new rules is much higher than the rate of deleting rules and we cannot differentiate attacking flows from legitimate flows in real time. Soon enough the limited flow table space is consumed and the network service is destroyed. To solve this problem, we propose the QoS aware mitigation strategy, peer support strategy, under which the switches treat each other as peers. When a peer switch runs out of memory space, other switches support the targeted switch by offering their idle flow table spaces to install the new rules and help in mitigating the attack. The status collection module enables the controller to get an early warning on the flow table space based on the capacity threshold chosen by the controller. When a packet arrives at the switch, if it matches an existing rule, it will be processed according to the matched rule. If not, the switch will recognize the new flow and report to the controller Then we check whether the switch is full or not, if it is full, the traffic guiding module helps distribute the traffic across the network rather than being aggregated at the targeted switch and the idle flow table resources of the entire network are leveraged to mitigate the attack.

**UDP flooding attacks**

Identification of an UDP flooding attack was done by distinguishing the users who send an unexpected number of UDP packets. They considered several variables:

1. A counter Cudph for the number of UDP packets originated by host H (i.e. source IP of each packet).
2. A variable UDP flooder [H] to keep the state of the client [H], true if H is a UDP flooder and false otherwise.
3. A threshold T for the legitimate number of UDP packets that could be sourced from an IP address.

The state of H is investigated with each packet, to check if it has already been recognized as a malicious user: if UDP flooder [H] is true, then the algorithm will drop the packet. Otherwise, the algorithm increments the value of the counter Cudph and checks if Cudph=T, it drops the packet and sets the H to true. In order to mitigate such an attack, one may think of considering a time-out for the counter and the state variable [H]. Meaning that, if the switch does not receive any UDP packet from a client H during a pre-defined amount of time, it can free the memory space allocated to the client H. However, a smart attacker can send the UDP packets periodically in order to force the switch to keep the state of the corresponding host. An attacker floods UDP connections through a large number of spoofed source IP addresses. The attacker simply opens up a new UDP connection per source IP address, and this way forces the switch to consider a new state for each UDP connection inside the state table leading to a memory saturation attack.

In the beginning, network is initialized. Then it runs for some time and attains stability. Then we start conducting the analysis mechanism where we sacrifice some time for all ARP request packets, so that we can determine if we are under attack (in case of a large number of ARP request packets). After this duration, we use traffic monitoring where the controller periodically sends the requirements to the switch so as to obtain all the statistics received by/sent to(the number of packets) from specific ports on the switch. The analysis model has two specific conditions:

1. If the received packet > send packet: this means the destination of the sending packet does not exist in the current network; resulting in the controller constantly broadcasting. This may be exploited to attack: the countermeasure is to start dropping packets. The number of packets in the port received by / send to statistics is set to zero. This is an iterative process, where the system is set back to its initial state.

2. If the packet sent > receive packet: this implies that the destination of the sending packet exists in the current network. Hence, the network is in its normal state and it can handle the packet-in message and broadcast packets.

**Actuating triggers to deal with UDP flood attacks**

1. Traffic percentage trigger: The outgoing UDP traffic should be small compared to the rest of the network traffic, so if a host in a network starts sending too much traffic to one destination, this may be an indication that the network has a compromised host and it could be a part of DoS attack. Hence, a threshold was proposed for the percentage of UDP packets to a destination relative to the overall traffic within a predefined period of time. If this threshold is exceeded, a trigger will be fired and the switch runs a predefined action sent by the controller to deal with the situation.
2. Amplification rate trigger: In some cases, the UDP response to some commands is larger than the corresponding request. So, if the network receives a great number of UDP traffic in a very short time, this could be considered as an indication of UDP amplification attack. Eg. DNS amplification attack: in this attack a query is sent for as much information as possible. The DNS server generates a large amount of response messages and since it is coming from valid servers, it is hard to prevent this attack. In the NTP amplification attack: A “monlist” request sent to the NTP server can result in the retrieval of a list of addresses of clients that queried the server before the request. This list generates a large number of IP’s causing a DDoS attack. To avoid these attacks, we can monitor the UDP traffic coming in through the ports where amplification usually happens (eg. DNS port 53, NTP port 123) and if the traffic reaches a predefined threshold within a predefined time window, the switch would take action.

**FLIP (Fast Lightweight Policy preserving SDN updates):**

Combining rule replacements and additions opens additional degrees of freedom in the policy preserving update problem and hugely reduces the number of added rules. FLIP takes as input an update problem, which is defined by the pair of initial and final states and the properties that have to be preserved during the update.

For the same flow, we can have multiple forwarding paths that exist between a source and the destination (equal cost multipath), but through FLIP we have to guarantee forwarding correctness and preservation of input policies. For forwarding correctness, we have to solve two problems: a blackhole which occurs when a forwarding path terminates in a switch different from the destination and without a rule to forward the packet further, an evil loop occurs when packets of a given flow are bounced back and forth indefinitely, among a finite number of switches. Policy preservation means that a set of input policies, satisfied in both the initial and final states are not violated in any intermediate stage generated during the update. FLIP supports policies such as traversal of single nodes or links, service chaining or QoS based traffic engineering.

FLIP returns a partial order between operations. This partial order represents an operational sequence, including rule replacement, tagging and matching operations. At a high level, FLIP adopts a divide and conquer approach. It divides the input update problem into sub-problems, one per impacted flow. For every sub problem, FLIP independently computes a sequence. Per-flow sequences are finally merged into the output operational sequence. However, this can be optimized merging a set of dependencies and using a scheduling algorithm to optimize update speed. Computation of policy preserving per flow sequences is the most novel part of FLIP. The constraint extraction procedure takes as input a per flow problem and performs two tasks. First, for each possible forwarding incorrectness or policy violation, the procedure identifies the constraints that ensure a safe update and then we distinguish between replacement and tag-and-match constraints. A replacement constraint imposes a certain ordering between rule replacements, a tag-and-match replacement imposes that some switches have to tag packets consistently with the applied rule and another switch has to match those tags during the update. Secondly, the constraint extraction procedure infers relationship between constraints, namely it pinpoints alternative and dependent constraints.

After having extracted constraints, FLIP selects all rule replacement constraints and marks them active. FLIP tries to compute a solution that satisfies all active constraints by translating the set of active constraints into a linear program where the objective function is to minimize the number of update steps. FLIP then tries to solve this LP with standard optimization algorithms. If a solution can be found, FLIP outputs the corresponding operational sequence. Otherwise, FLIP applies the constraint swapping procedure to replace some active constraints with alternative ones (and their dependencies). Since a matching constraint is always satisfiable, FLIP eventually reaches a combination of active matching and replacement constraints for which a solution exists.

**SDN Security Plane: An architecture for resilient security services**

A third plane is introduced in SDN relative to the data and control planes, this security plane provides a connection between the switches and the controller. The security module at the controller side is responsible for collecting and analyzing all data traffic coming from all agents to detect abnormal events and to trigger alarms to the controller to take necessary preventive action. The proposed technique aims at detecting DDoS attacks from malicious hosts or bots aimed at other hosts or at the controller. First, the collected packets targeting a specific IP address are analyzed and their packet count/sec is tracked and compared to a threshold. A database is kept for mapping MAC-IP-Switch-Port binding for the purpose of detecting spoofed IP’s and packets. Upon detection, an alert is sent along with the source IP of the attacker in order to be blocked. A trace back procedure is used to identify the origin of the attack Switch-Port so it can be blocked from its origin in case of IP spoofing.

**SDN-Guard which can simultaneously reduce the controller overload, the switch to controller bandwidth and also avoid the CAM table flooding of the OpenFlow switches:**

1. Flow Management Module: This selects the routing paths for each of the flows and decides the hard timeout for the corresponding TCAM entries based on the threat probability of the flow. The SDN guard constantly communicates with the Intrusion Detection System (IDS) that analyzes the packet-in messages and decides the threat probability of each flow.
2. Rule aggregation module: It aggregates the flow entries of malicious traffic in order to reduce the number of entries used in the switches TCAM’s.
3. Monitoring module: It collects multiple statistics about flows, switches and links (flow throughput, switch TCAM usage and link bandwidth usage) which helps in redirecting malicious traffic through the path having least-utilized links. Hence, the malicious traffic will reach the destination even if it is not through the shortest path, so that it can be further analyzed giving chance for the false positives but with higher delays. However, legitimate flows are always routed through the shortest paths so as to ensure minimal round trip time delays.

By dynamically rerouting potential malicious traffic, adjusting flow timeouts and aggregating flow rules associated with malicious traffic, the impact of DoS is reduced by 32% and switch memory usage is cut down by 26%. The method also reduces packet loss and packet round trip time in the network during DoS attacks .However, performance has to be evaluated for more realistic and larger-scale deployments. Another avenue can be to investigate the accuracy of intrusion detection systems in estimating flow threat probability and to study the impact of malicious flow accuracy.

**PATMOS (Protocol for DDoS Attack miTigation in Multi contrOllers SDN networkS):**

This method employs three phases: Finding bottlenecks, Election and Composition. The first phase identifies overloaded controllers by exchanging control messages, eliminating the dependence of detections based on network traffic analysis. The election phase chooses a leader to coordinate the clustering procedures, based on the controller’s performance level (candidacy, leader, vice-leader and elite roles). The composition phase minimizes the DDoS attacks effects by choosing the best cluster configuration to mitigate the attack effects, setting up that cluster and restoring the network configuration. (The clusters are chosen by employing the genetic algorithms which use the capacity and population size to find a solution).

**OpenFlow SIA (optimized protection scheme that can handle flooding attacks and resource exhaustion of SDN components)**

The OpenFlow SIA consists of five major modules:

* Flow Collector: This module runs in the SDN controller in which it sends and receives statistical messages from OpenFlow switches in a predetermined period of time.
* Feature Extractor: Features such as packet number i.e. number of packets transmitted via the flow and duration i.e. how long the flow has existed in the flow table are extracted from the former module outputs.
* SVM-I module: This module is a classifier for each type of network protocol such as TCP, UDP, ICMP or ARP. If I is TCP then the TCP flows will be forwarded to SVM-TCP, the SVM-I uses a dataset for training procedure and then it produces a data distribution and draws the hyperplane which has one axis as duration and the other as packet number.
* Policy enforcement module: There are two types of attacks addressed in this module: Firstly, attackers send a large number of flows and each flow consists of an absolutely high packet number which is tackled with a drop action for the attack flow, thus the victim or networks are not affected by these abnormal flows. Secondly, flooding attackers generate a vast number of flows using fake source IP addresses and each flow transmits only from 1 to 3 packets, so a message is sent to the OpenFlow switches to delete these flows from the flow tables.
* Idle-timeout adjustment algorithm: It adjusts the flow-idle timeout based on the SVM-i outputs: when a new entry is installed in the flow table, it is set to IniVal which is less than MaxVal, the default value of the SDN controller. If it has been classified as an attack flow, the timeout is either SetToZero to drop the packet or it becomes a DeleteFlowEntry function. In case the flow is believed to be normal, the idle flow timeout value is defined by SetVal , this compares the CurVal i.e. the current idle timeout to IniVal, if these values are equal this implies that the flow has just been added in the flow tables and later the SetIdletimeout function is called to adjust the flow-idle timeout value mostly setting them to MaxVal so that normal flows exist in the flow table longer than anomaly flows and this process is repeated as a loop each time when the SDN controller receives flow statistics.

**For secure multi party SDN updates (History Based Approach):**

An attacker can influence the information available to the controllers and may trigger reactions such as link failures or changes in network traffic. A benign controller may be in a variety of states: up-to-date (updated snapshot of the network state), delayed (not yet received the latest events from the network elements), gapped (gaps in the receiving of events from switches, it might have received a recent event but missed an earlier one), joined (just joined and has not yet learnt about the current network state), left (controller has permanently left the control plane). Hence, the proposed approach suggests:

1. Linearization: critical events must be linearized i.e. ordered uniquely by the network element and on a per network element basis. (TCP connection is not enough as orders may differ across controllers).
2. History digest: based on the linearization, a history digest is computed i.e. a secure (collision-free) hash over the event history. This history digest is stored at the switches and the controllers.
3. Distributed verification: a network element accepts an update if and only if the request and the history digest is signed by a majority of controllers.

Whenever a controller wants to issue an update command, it will compile a signed message containing the update as well as the current history and state hash. Moreover, each controller implements a service which allows other controllers to ask for the current state. A network element only accepts updates if they are signed by a majority of controllers and include the correct history and state hash. Network elements always inform controllers about new events and also allow any controller to request the current state hash. However, if an attacker is able to delay or prevent the delivery of information to the controller, it can be ticked to agreeing to certain configuration changes. Due to this delay, there is a change in the sequence of attempts and hence the history digest of different controllers may not match. An attacker can even replay old signed agreements from other controllers to effect a configuration change in a new context.

**Selectively Distributed Firewall Control in SDN**

A firewall prevents unauthorized access to or from a private network and all of the existing solutions follow either the centralized firewall or the distributed firewall concept. In the centralized firewall, the controller maintains firewall rules and filters out the traffic as all the packets are redirected to the controller. Hence, the controller acts as a firewall and no configuration on the network devices is required but this method also raises concerns about controller overload and scalability. In the distributed firewall approach, the controller installs all the forwarding rules in the flow table of every switch and hence the workload is totally migrated off the controller and the system also becomes less sensitive to topology changes. However, configuring and maintaining the rules in each switch is complicated.

In the distributed firewall system (considering it on layer 2), each flow entry consists of source MAC address, destination MAC address and action is set to DROP. After the rules are installed on each of the switches, the packets will be matched against flow table entries and if the packet matches one of the firewall rule, the switch will drop the packet immediately. In this approach, the firewall-violated traffic will be dropped right away at the switch connecting directly to the host and hence it can never go more than one hop and come through an intermediate switch. Hence, installing similar solutions in every hardware device could be redundant and to solve this problem a selective firewall installing method is proposed. The idea here is to install a firewall rule in the switch that directly connects to the host which has the MAC address in source MAC address of the firewall rule. This reduces redundancy as it cuts down the firewall setup commands from the controller to all switches, it also means lesser flow entries and one less matching process every time a packet goes through these switches. A reverse firewall rule must also be installed in at the receiver’s side so that the packets are dropped at the nearest forwarding device even when the sender and receiver roles are interchanged.

**Port Knocking as a basis for firewall applications:**

The port knocking process allows a stateful firewall to grant access to one or multiple hosts on a specific port only if they are able to successfully conclude a pre-defined (ordered) sequence of connection attempts to different ports. Such ordered sequence represents a “shared secret” between the firewall and the hosts.

Port Hopping is a typical kind of MTD technology, which borrows idea from frequency hopping communication and dynamically maps service ports to unused and/or random ports. Port hopping technologies can confuse attackers and increase their attack cost, as they can’t locate opened ports used by network services.

When a host H (identified by the source IP address) tries to establish a SSH session (on default port 22) passing through a firewall F implemented on the Open State switch: Let {3306, 1810, 450} be the sequence of ports to “knock”. If H sends the exact three ordered requests as connection attempts, F will authenticate the request and open port 22, otherwise the request will be dropped. Each open state switch keeps an extensible finite state machine and for port knocking the rules in these state tables would be defined for five events (i.e. packets received on ports 3306, 1810, 450, 22 or any other port), four states ( default, State 1, State 2, open) and two actions (drop and forward). Upon receiving a packet from H, the switch performs the state lookup procedure to retrieve the current state of the packet. Starting from the default state, a state transition is triggered if and only if H knocks the expected port of F in the sequence, this leads to a packet drop action and a state transition to state 1. The new state is written back in the state table for the corresponding host ID (IP source). This procedure continues until H knocks all the expected ports, and then its status will be updated to open. In this state, upon receiving a request for port 22, the firewall open this port for user H. In any state, if the host sends a request to an incorrect port, its state rolls back to default state. It also has a timeout field in the state table, for a transition to the next state (i.e. next knocked port), if the user does not knock the correct ports within a pre-defined timeout, its state rolls back to the default state.

Problems in existing port knocking authentication techniques (An attacker can discover the actual sequence of packets in port knocking process and launch attacks in the following ways):

1. A sequence replay attack in which a particular set of packets can be sent to the victim again and again.
2. Getting the sequence number by sniffing the packets.
3. Observing the pattern of knock sequences in promiscuous mode and running port scans.
4. Packets are delivered and received out of order due to network latency and other factors, thereby exposing the knock sequence in most of the cases.
5. Single packet authentication can lead to disclosure of the complete data if the packet is captured by the attacker in between the client and the server machine.
6. Knock sequence being influenced by the use of spoofed packets.

Port knocking attacker models:

1. Informed attacker model: This is an attacker who knows the correct sequence of ports to knock. This information could be gained through several ways such as sniffing the traffic. The attacker sends a large number of connection attempts to the first port in a predefined sequence, from a set of spoofed IP’s towards a switch F. On receiving the packet from the source IP, F checks its state table to retrieve the state of the incoming flow. If there is no record for this IP address, F assigns the default state for this corresponding IP. Now, since all the incoming packets are destined to the correct port, their state will be updated to the next state (i.e. 3306 knocked). Therefore we will have a large number of entries inside the state table of the switch. This way a conscious attackers is able to force the generation of thousands of records in the state table, and thus exhaust the switch memory.
2. Oblivious attacker scenario: this attacker does not know the exact port sequence to knock. The attacker, A generates a large number of packets from a set of spoofed IP’s (eg.D: 128.0.0.1 – 128.0.255.254) towards all the ports of F. Assuming none of the IP’s are used before, F assigns the default state to all the corresponding IP’s and performs a XFSM table lookup. If a programmer due to an incorrect implementation, stores one record in the table for each of the incoming flow, then all ports can be easily flooded completing the memory saturation attack. However, in a correct implementation there should be only one record in the default state.

Once the switch memory is exploited by the adversary, it is not able to process new incoming packets which could be issued by a legitimate user. Then, the switch can decide to perform the following actions:

1. Send the packet to the controller: in this case, its behavior is same as normal SDN.
2. Drop the packet: this leads to denial of service to a legitimate user.
3. Overwrite on one of the existing records in the state table: the challenge is how to chose the record to overwrite.

To tackle the random port scanning attack, authors proposed a new state-attack state. A metric M, was considered to count the SYN rate. If the switch detects a port scan by flow, the status is updated as attack state, the host is blocked for 2 minutes and an alert is triggered.

**Advanced Port Knocking Authentication Scheme with QRC using AES**

The user sends an SMS to the SMS server connected to the pseudo random number generator requesting for the 256-bit One-Time key which will act as the key for AES encryption and an 8-bit random number R which should be relatively prime to P and N. The numbers P and Q are prime numbers unique for each other and already stored in the database server connected to the SMS server for each user. The SMS and the reply sent by the server is sent through a dedicated channel which is out of band for general communication. The user’s mobile number should be registered with the SMS server and the message should be sent in a predefined format. The OTP is generated with validity time period so that there is no duplicate OTP generated before the use of original OTP or the expiration of time period. The 8-bit R is used to spoof the IP address every time the request packet is sent to the application server through any firewall. Timestamp is used to verify authenticity of the OTP and will be required in the prevention of spoofing of the mobile numbers and DoS attacks. The obtained OTP and the last 8 bits of the client’s IP address is passed to the client to work as the key for the AES which will be used to encrypt the data to be sent i.e. the knock sequence which is our authentication data and R will be used in the Quadratic Residue Cipher. Multiple packets are sent to the firewall server of the target to provide authentication with the help of multiple packet authentication (MPA). The server is constantly monitoring the incoming packets silently. The received packets are stored in a queue to be decrypted (with the help of One Time Key and AES cipher) until it has received the required number of packets. If the obtained sequence is matched with the generated sequence at the server, access is granted to the client by opening the port and allowing the connection to be initiated, otherwise the request is denied at the firewall itself.

**Covert Communication using Port Knocking**

Encrypted channels keep the confidentiality of data and communications, but steganography takes data confidentiality to a whole different level by hiding secret data in ordinary looking files, making the very existence of secret data practically undetectable. The existence of encryption makes the communication suspicious for undesirable observers, however covert channels conceal the fact that communication paths presence, making it strenuous to be detected. Covert channels take advantage of existing protocols to transfer the information like hiding data within TCP/IP packets by modifying protocol headers and contents which might be detected by using anomaly detection techniques. Hence, recent techniques are using steganography which is the science of hiding information into a media file such as image, text, audio, video etc. This method not only results in stealthier channels but also provides channels with a larger capacity to hide data. Hence, a Tariq Port Knocking mechanism was proposed using covert channels. In the first initialization stage, the sender and the receiver will agree upon specific ports to be used for sending the knocks and the server has to import the sender’s GnuPG public key. The encrypted message will be hidden using the Least Significant Bit (LSB) steganography in an image selected by the sender. This image will be divided into a number of packets and will be sent as the payload of the packets used for knocking the ports. In the second stage, the sender will knock the server by sending packets which hold an image as a payload on the specified port numbers in an attempt to send a secret message. Thirdly, the server will reply back to the sender, by sending an authentication packet. Finally, when the client’s identity is confirmed by the server, the server will reassemble the received packets and extract the secret message by using the private key.

**Port knocking method against TCP replay and Port Scanning**

The primary aim of port knocking is to provide an extra layer of security by providing authentication with the additional benefit of concealment. However, port knocking itself suffers from vulnerabilities like TCP Replay attack, Port Scan, security obscurity and packet delivery out of order due to network latency. An attacker uses the port scan technique to sniff on promiscuous mode and observe a knocking sequence attempt by unauthorized users. Once, a true sequence is complete, they will use a TCP replay attack to knock on a server, but the server has no clue whether it is a genuine user. In addition, a port knocking server cannot limit how much connection will come from a user to the server concurrently. Improvements such as Single Packet Authorization (SPA), Internet Protocol Security (IPSec) or a hybrid of cryptography, steganography and mutual authentication have been proposed.

The new port knocking method:

First, the client makes an attempt to establish a connection to the server to use certain services such as SSH with the defined ports. Here, the client sends a packet containing detailed information such as the source port sequence and the destination port is pre-assigned by the server and that listens to the specific port attempts from the port knocking client. The source port sequence (also pre-assigned by the server) is used as an authentication method to validate its client. The second step is to determine whether it is a valid client, since only they know what port sequence should be used to establish a connection. The server monitors any connection attempts from the client and a simple logic programming is used either to accept or drop a connection whenever it matches the rules. Once validated the service is started and the client is notified by sending an execute message, otherwise the connection is dropped and the service is not started. In the final step, the client connects again to the server with the predefined port assigned for the service thus establishing a connection. (eg. SSH can be pre-configured to use the port 8080 instead of default port 22. In future, this port will be configured to use other TCP ports when necessary).

**Port Sequence Selection is suggested as future work**

**Secure Port Knock Tunneling (enhanced port security authentication mechanism)**

This method overcomes the DoS knocking and NAT-knocking attacks. In one of these situations, the attackers send random packets to the server repeatedly and the server should allocate a buffer for remaining log of each client and hence the DoS knocking attack consumes a significant amount of memory. The other situation arises when the monitoring system cannot distinguish trusted users from others. This scenario arises when Network Address Translate (NAT) is used in the network and hence all the users have the same address outside the local network. Hence, when one user completes the port knocking process and gets permission for accessing the server, all the clients which are located behind the similar NAT can use the service.

The Secure Port Knock Tunneling (SPKT) is the method proposed to counter NAT-knocking and DoS knocking attacks and it can also increase the protection of the authentication process. If a client wants to establish a connection to SSH server, it starts the SPKT process as a port knocker via sending a UDP packet to the server. The UDP packet contains Ethernet header, IP header, UDP header, data text passphrase and Ethernet trailer. This mechanism uses UDP because it does not require ACK from server, moreover without the responding packet the network is less vulnerable. In the previous port knock processes, when a client sends a valid sequence, connection establishment is done but in SPKT it should send the legal sequence with valid text passphrase. (Server checks the passphrase only if the knocked port numbers are valid) After this step is finished successfully, then the firewall opens one port for the client and triggers the VPN connection on it. If the passphrase has been verified by the server and is similar to a pre-configured passphrase, server buffers the information for 10 seconds in a list. Otherwise it means the malicious user sends packet and server does not allocate memory space for it and drops it. Therefore, the DoS knocking problem does not occur anymore. For the next knock, besides checking the secret text, the server must check whether the IP address exists in the temporary list or not. This process continues till the information of all the knocks is stored in the buffer. The whole process should take about 40 seconds and if each packet cannot arrive to server side before 10 seconds, buffer will flush automatically and the process should be started once again. But if the port knocking process is successful, then the second phase will start.

**Port Hopping DoS mitigation (PH-DM):**

According to SDN network’s logically centralized control and network programmable features, port hopping module on the server side is forwarded to the SDN controller, which reduces large load caused by port hopping on the server. SDN controller (cooperating with OpenFlow switches) acts as a DoS attack filtering gateway, and checks the network connection launched by the clients, which mainly checks whether the destination port of the connection suits the opened port used in the current timeslot. If the two ports are the same, SDN controller will install specific flow entries to the corresponding OpenFlow switches along the transmission path, and the packets of this connection will be transmitted to the target server if they match these installed flow entries, by this way, the trusted and authorized clients can communicate with the target server normally. If the destination port differs from the opened port, this connection will be considered to be malicious, which will be discarded directly and can’t be forwarded, then illegal and malicious network flows cant access the SDN network. As the port hopping module of the server moves forward to SDN controller, thus the server does not need to make any changes of configuration, it just listens to the fixed ports to provide services. SDN controller and the client need to realize the port hopping function respectively, including port hopping synchronization, random port generation and other functions.

Port hopping synchronization is the key to ensure the continuity of communication between the two sides: otherwise it is extremely easy to cause the interruption of communication, which affects the communication efficiency of both sides. Current hopping synchronization methods include strict time synchronization, ACK (acknowledge) based synchronization and timestamp based synchronization. Strict time synchronization can be easily affected by network delay and network congestion, and ACK based synchronization puts synchronization information in ACK message, which can be intercepted and analyzed by attackers. The timestamp feedback based hopping synchronization method combines timestamp based synchronization method with ACK synchronization method.

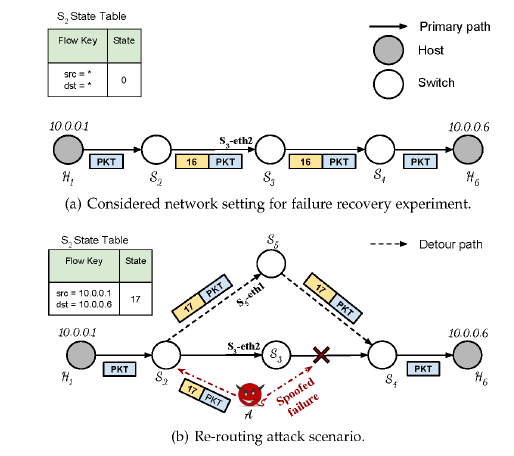
The server S always opens two listening ports waiting for receiving the data from the client C, and the client C opens one port for receiving the timestamp feedback from the server S. Both sides maintain a port counter P respectively, which is initialized to a number of seeds (such as 1). Both the client and the server use the counter P to determine which port should be open and which port should the message be sent to. The server S opens port pdd (using the P-1th elements in the pseudo random sequence), and opens port pnew(using the Pth elements in the pseudo random sequence). The client sends the message to the Pth port, and opens the Pth port pack in the second pseudo random sequence (used to receive the feedback message). When the server S received the message sent from the client C in port pdd, feedback message is sent to old feedback port (containing the timestamp information). When it receives the message from the port pnew, feedback information is sent to the Pth feedback port, and P increases. This method is easy to deploy and we only need to transfer timestamp information during the synchronization process. If the attacker does not know the hopping algorithm, he can’t perform effective tasks.

**Stateful Failure Recovery (controller independent stateful link/node failure detection and recovery scheme based on Open State).**

Traditionally, restoration and protection are the most common failure recovery strategies employed. In the restoration strategy, alternate paths are installed in the flow tables after failure detection and hence network resources are dynamically allocated which delays the recovery process. However, the protection strategy has alternate backup paths preconfigured in the network. Eg. One implementation of this solution (1:1 path protection scheme) tags packets with labels, upon detecting a node or link failure, in order to communicate to previous nodes to use a detour path for subsequent packets. A source host, H is sending a packet pt to a destination host I, through a network of stateful SDN switches. Upon recognizing a link or node failure, the nearest node to the failure tags the same data packet pt with a label. (eg. MPLS label) containing information about the failure event, instead of sending a notification of the failure. This tagged packet is routed back on the original path to a convenient reroute node (N). Upon receiving a tagged packet, this node N performs a state transition for the corresponding flow in its state table and reroutes (through a predefined route) all the subsequent packets of the flow afterwards. When reaching to a merge node in the route, the tag will be removed from the packet and the packet will be forwarded to the destination I. The stateful processing feature of this scheme allows the switch to reroute the packets without the need to report link failure to the controller, since each node maintains the state of every flow, it can decide the forwarding path anonymously.

In the proactive recovery scheme, we use the fast failover feature of OpenFlow without contacting the controller. The flow rules point to a group table which provides multiple ways of forwarding. The flow rules uniquely identifies a group using a group’s identifier. A group entry consists of a group identifier, counter and action buckets. The counter field counts the packets processed by the group. An action bucket contains a set of actions to execute and its associated parameters. When a switch/link fails, the fast failover group executes the next available action bucket (without any intervention from the controller) and outputs the packet to an intermediate switch of the backup path. But in this case, flow rules for every disrupted flow must be preconfigured on the alternate path and this places a burden on the switch memory. To eliminate this burden, the recovery approach aggregates the disrupted flows having the same immediate destination switch using the flow tagging mechanism. These detoured paths are logically identified as a single unified flow until they arrive at the immediate destination switch. Therefore, instead of installing flow rules for every detoured flow, the controller just installs a single flow rule matching the header tag of the aggregated flows.

However, an attacker can eavesdrop the exchanged traffic between two nodes, capture it, tag the packet by appending a label “broken link”, spoof a link failure event and send back the tagged packet to N. Upon receiving a counterfeit tagged packet, N performs a state transition for the corresponding flow and detours all the pursuant packets. Such tags (MPLS) are a progressive sequence of numbers starting from 1 and whose state values are shifted by 16, as the values between 0 and 15 are reserved. Therefore, identifier 16 denotes no failure, while failure events are numbered starting from 17. Hence, if an attacker can forge a packet having an MPLS tag of 17, she will be able to pretend that there is a failure in the forwarding path which will require re-routing of traffic. Hence, inconsistency and latency is easily imposed on a network, as well as decreased performance in the absence of an encryption scheme and no authentication protocols between the switches in SDN architecture.



**Backup-Resource Based Failure Recovery Approach in SDN Data Plane**

Due to TCAM (Ternary Content Addressable Memory) being an expensive and limited hardware with high energy consumption) we want to reduce the number of flow entries stored for failure recovery scenarios. Failover Group Tables have been proposed by open flow specification to decrease the failure recovery time. Some methods set priority for traffic or backup paths. A failure recovery approach based on backup resources was proposed to minimize the consumption of storage resources of switches, while meeting the required delay when link fails. This method proposed two metrics to determine the importance level of a link. Three kinds of strategies for different graded links are provided, based on which the algorithm has been designed to solve the problem. The high importance links have a requirement of lower latency and higher transmission quality. Hence, a double path backup strategy is proposed from the detectable switch to the next switch and if this fails, through the next next switch belonging to the working path. For middle importance level links, a single backup path is proposed to recover the required failover delay. If this fails, we trigger the reactive recovery strategy. For low importance level links, the backup path is dynamically allocated, but resources required by recovery paths are not allocated until a failure occurs. Thus, when a failure occurs, additional signaling is required to establish the backup path.

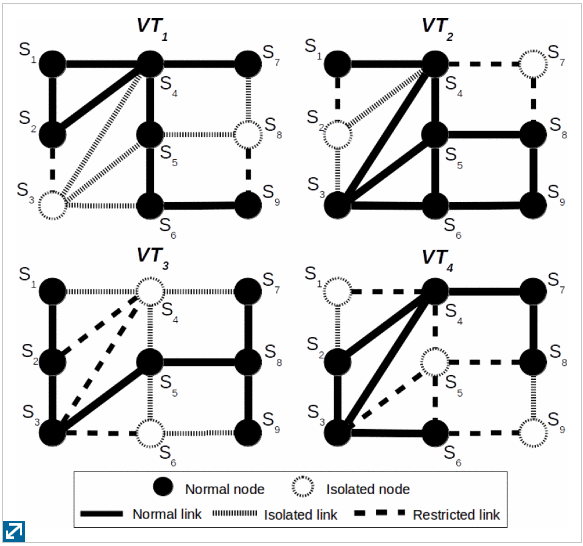
# **Multi Topology Routing based IP Fast Re-Route for Software Defined Networks**

The SDN data plane may experience link or node failures but it must be recovered fast and seamlessly to support real-time services. Multiple Routing Configurations (MRC) is a Multi Topology Routing (MTR) based IP fast re-route (IPFRR) technique which provides full coverage against failures by using virtual topologies (VTs). The technique proposed relies on MRC and provides a self-recovering SDN against single failures in the data plane, the central computation of the virtual topologies by the controller guarantees consistency among the routing tables.

Upon the detection of a failure, the disrupted traffic can be rerouted towards its destination according to the VT where the failed component is not used for traffic forwarding. (However, it provides alternate path coverage against single failures only). The MRC assigns one of three possible weights to a link in a VT, namely, the same weight as in the physical topology, a high weight called wr or restricted, and infinity or isolated. A restricted link may only be used to forward traffic if one of the end point nodes of the link is the source or destination of the traffic whereas an isolated link may not be used to forward any traffic at all. The set of the links in a VT which have the same weight as in the physical topology is called the backbone of the VT. The VTs constructed by the MRC satisfy the following properties: 1. Each link and node in the physical topology becomes isolated in exactly one of the VTs. 2. Each VT backbone is connected. 3. Each isolated node in a VT connected to the backbone via at least one restricted link

As shown in the figure, the VTs satisfy (*i), (ii*), and (*iii*). For example, when the link (S8,S9)  fails,  VT4 is selected for the recovery from the failure since (S8,S9) is isolated in VT4. In this case, any traffic sourced at S9 starts to be delivered towards the destination via the restricted link (S6, S9). (shown in the figure 2)

MTR bases IPFRR can be implemented as an application module which fast recovers from the failures using the restoration approach. It constructs a pre-configured number of virtual topologies based on the topology of the data plane and relies on the existing failure detection mechanisms of the controller. During the network initialization, through port\_status messages for link additions, the underlying topology is discovered and the VTs along with the routing tables are computed. When a anew traffic flow is detected via a packet\_in message, the primary route for the flow is activated by installing the rules to the corresponding switches. Upon receipt of a port\_status message for link deletion, the alternate paths for the disrupted flows are determined using the VT excluding the failed link, and the primary and alternate routing tables are re-computed.



**Failure Recovery using VLAN-Tag in SDN: High speed with low memory requirement**

Using the protection method, the basic idea proposed is:

1. Calculate the working path (using the shortest path algorithm) for a flow.
2. For each link composing the working path, calculate its backup path by one.

Fast Recovery of a Single Link failure:

The existing protection methods have a much larger requirements for TCAMs than the restoration approaches. When referring to TCAMs they can be divided into two parts- for working paths and backup paths respectively. For working paths, both restoration and protection are equivalent if using the same method (shortest path algorithm) to calculate routes. The difference arises from the TCAMs for backups. As the protection methods compute backups independently for each flow, then the TCAMs for protection paths raise with the increment of flows. The “protection using VLAN ID” method is used for distinguishing forwarding table entries for working and protection paths by assigning a VLAN ID for each link’s backup path(whereas source and destination IP are assigned for each working path). All the mapping relations between VLAN IDs, backup paths and the first output port of the backup path are stored into the controller. When the need for the broken link arises according to a source and destination IP match, the corresponding group table entry has two action buckets, the first is to forward to the next best hop according to the working path amd the second is to push the VLAN ID and send to the next hop on the backup link. For switches on the backup path, the match field is a VLAN ID. The actions on the last but one hop on the backup path are to pop the VLAN ID and send to the next hop. This computation of backup paths is required for each link during the initialization stage. In cases where nodes are added or removed in a high frequency, overheads of updating backup paths cannot be ignores. To relieve this problem, we can assign a flag for each link to distinguish whether it has a backup path calculated or not. Each time when the topology changes, our method only needs to compute backup paths for newly joined links. Hence, the cost of initialization is reduced.

Fast recovery of Multiple Link Failures using VLAN ID

For the algorithm proposed for single link failures, a modification is required. For a switch on the backup path we need an extra group table entry along with the flow table entry. In this group table entry, the first action bucket is send to the next hop of the backup path and the second one is to return to the previous hop of the backup path. Meanwhile for a switch on the backup path, another type of flow table entries is needed with the match field being a VLAN ID and in port, and the actions are to pop the VLAN ID and forward to the next hop once again. The reason for including an in port into the match field is to judge whether packets arrive from the positive (corresponding to the match field being just a VLAN ID) or negative direction (corresponding to the match field being a VLAN ID with an in port) of a backup path. For thw switches on the backup path, the flow table entry prepared for the negative direction has a higher priority than the one for the positive direction. Forwarding loops are not triggered after this modification.

**Software-BASED Failure recovery in load balanced SDN-based datacenter networks**

For network monitoring in SDN networks, we have been using active and passive probes. The probe station selection mechanism addresses the problem of selecting minimum subset of nodes in the managed network where probe station must be placed such a way that best monitoring can be achieved. Since our method requires that group modifications be performed locally by pre-configured instructions set by controller on the switches, each ToR switch must have a probe station. The probe set selection mechanism ensures that all the elements are probed and all the ToR switches must receive probes sent to each ToR switch in the same cluster, the minimum probe set is required for networking monitoring and has better performance in bandwidth usage and delay. When the ToR switch forwards the probe to its aggregator link, it floods the probe packet to all ToR switches inside the cluster. Furthermore, to reduce bandwidth exploited for probing, the probe only consists of an UDP packet and each link is distinguished by different UDP Port number carried in the header. The monitoring component expects to receive at least one probe packet in a predefined interval, if not it assumes the link is failed and the restoration phase begins sending OpenFlow group modification messages. Since we find the failure detection phase as the most time consuming phase if the initial failure recovery scheme, we improve our recovery mechanism by exploiting active probes. Active probes have several advantages over passive monitoring techniques, such as less instrumentation, capability to compute end-to-end performance, quicker localization etc.

This paper tries to investigate and provide an insight into deployment of an effective load balancing mechanism for latency reduction. Regarding failure detection, we exploit active probes sent from each Top-of-Rack (ToR) switch and flooded by aggregate switches (cluster switches) to all other ToR switches of the same cluster. By exploiting this technique, all of ToR switches can perform local configuration modifications and act independently of central controller, avoiding controller saturation and scalability issue.

The ability for a flow entry to point to a group enables OpenFlow to represent additional methods of forwarding. Each group is consisted of an ordered list of action buckets, where each bucket contains a set of actions to execute and associated parameters. The notion of Liveness in OpenFlow provides the switches with the capability to choose group buckets based on availability of other groups or ports. By exploiting this Liveness feature, also known as watch group and watch port, OpenFlow switches are able to perform automatic failure recovery. (for eg. if a link becomes unavailable, the adjacent switch automatically reroutes the traffic by choosing another output port defined in the group)

**DNS Tunneling Stateful Detection:**

In this scenario, a malicious user tries to abuse the DNS messages to bypass the access policies in order to send data. To detect a DNS tunneling attempt:

1. Assign a counter cH to each client H in order to keep track of all the resolved IP addressed for H.
2. Increase cH when the client receives a DNS response and decrease it per used resolved IP address i.e. when the client sends a packet to the corresponding IP address.
3. Consider a threshold for cH, and report as malicious the client whose counter is higher than the threshold.

In order to perform the second step, SNAP maintains all the resolved IP addresses destined to client H in an array based variable. This variable, orphan maps each pair of source and destination IP addresses to a Boolean value. If a client H receives a resolved response from the DNS for a destination IP address, the value of the variable is set to true and the value of cH will be incremented. When H sends a packet to I, the switch checks if the state associated to the variable is true, if it is true the value is set to false and value of cH is decreased meaning that H actually used the information contained in the received DNS record.

**HULA (data plane load balancing technique for data centers):**

In HULA, each switch keeps the best path utilization table with the information of the next best hop towards a destination Top of Rack switch (TOR). The entries of the utilization table are of the form <Dest IP, Best Hop, Path Util>. In order to find the next best hop, each TOR switch periodically sends probes throughout the network to gather global link utilization information. Based on the received probe, each switch proactively updates its best path utilization table with the next best hop, which balances the load towards a destination through the existing path.

Each probe packet contains 2 fields:

1. A 24 bit torld i.e. the identifier TOR switch that originated the probe.
2. An 8 bit minUtil, which carries the utilization of the best path for flowlets. Once a switch receives a probe from port I, it first computes maxUtil as the maximum value between minUtil and the measured link utilization on port i. Then the switch computes the minimum value between maxUtil and the previous corresponding PathUtil value recorded inside its utilization table. If maxUtil is the minimum value, the switch updates the fields PathUtil and BestHop with the values maxUtil and torld respectively in the path utilization table. Moreover, the switch produces a new probe specifying, inside the header, the latest torld and the PathUtil which is propagated throughout the network using a simple loop fire strategy.

**Detecting Packet Forwarding Anomalies in SDN: FADE**

SDN is prone to packet forwarding anomalies where packets are forwarded along the wrong paths. Flow rules installed in switches can be easily tampered by different entities intentionally or unintentionally. For instance, flow rules on compromised switches can be modified arbitrarily and communication channels between the control and data plane can be intercepted to tamper control messages and install wrong flow rules. Flow rules could even be mismatched due to hardware flaws resulting in forwarding anomalies. Current approaches such as probing packets do not capture all attacks and the statistics approaches induce high communication overheads since they collect statistics of all flows. A novel scheme called FADE was proposed to resolve these issues, FADE detects forwarding anomalies by accurately analyzing flow statistics or rule paths(sequence of flow rules consecutively processing the same flow) of a minimal set of flows. It leverages packet labels to identify flows under analysis and verify the consistency of these flows so as to identify the anomalies. FADE uses a flow selection algorithm to select a small set of flows whole rule paths cover all existing rule paths. It generates a small number of dedicated flow rules associated with these flows and installs them in the data plane to accurately measure their flow statistics. It also controls the installing and timeout of these dedicated flow rules so that all dedicated flow rules generated for the same flow operate on the same set of packets. This method has been able to detect all forwarding anomalies and has negligible impact on the throughput.

**Attacks on Host Tracker in SDN Controller:**

In an SDN network, locations of all hosts can be monitored with the host tracking service (HTS) by monitoring Packet-In messages that the controller has received from the switches, however this service does not support authentication mechanisms. This vulnerability of HTS may lead to host impersonation attack, man-in-the-middle attack and denial-of-service attacks. SPHINX was proposed which builds and continuously updates flow graphs for each traffic flow observed in the network in order to detect anomalies in the network topology. However, this method needs a flow graph for every flow which does not scale to large data center networks. Also, the anomaly detection approach generates false alarms in case of frequent network topology changes. TopoGuard was also proposed as a security extension for the SDN controller to protect the SDN networks from network topology poisoning attacks but it did not cover all possible attack scenarios. The following attacks can be launched by an attacker based on the vulnerability of the host tracking service in SDN controller:

1. Host impersonation attack: The attacker will receive the packets intended to the victim and can reply to these packets on behalf of the victim.
2. Man-in-the-middle attack: The attacker will be able to monitor all the traffic between two communicating hosts.
3. Denial-of-service attack: The attacker will prevent the two communicating hosts from getting connected to each other.

It is difficult to use passive configuration to solve the problem, hence an extension to the SDN controller is proposed to dynamically protect the attacks on host tracking service. The extension includes the following three elements:

1. Port Manager: Detects the host generated traffic and contains the host list. The host list is based on MAC address of each host.
2. Host Probing: To check the availability of a network host (reachable/unreachable). The host probing supplies a host probing packet (e.g. ICMP Echo request) to the h0st and waits for a response within a reasonable timeout.
3. Host Checker: To verify the right of a host migration and prevent the ARP poisoning. The condition for each event of host migration are below: Join event (a new host must not be the same switch port with a current active host), Move event (the host tracking service must receive a Port Shutdown signal from a host through the Port Manager. This host is also unreachable by checking through the host probing in the previous location after the completion of host migration).

**NetCo: Reliable Routing with unreliable routers**

In general, an adversarial router may perform the following attacks: rerouting (forwarding a packet to the wrong port), mirroring (duplicating a packet, sending one to the correct port and one to an incorrect port), packet modification (deleting packets, modifying the header like changing the VLAN field, modifying the payload of packets), low rate Denial of Service (generate new packets, drop packets, overload the network by overloading the link resources as well as resources of nearby network elements). Using NetCo, it is shown how we can obtain a reliable routing system and efficient networking infrastructure even if we have untrusted routing hardware and this is based on two key concepts:

1. Leverage redundancy and diversity: Given a certain hardware heterogeneity which renders collusion among different routers unlikely, we can assemble and connect these routers in a manner which allows us to detect and prevent misbehavior.
2. Trusted but simple components: While high performance routers are complex and in house production often out of the question, the design of simpler and trusted components may be feasible at low costs.

Two simple trusted components, a so-called hub and a so-called compare can be used to emulate a trusted router using untrusted routers. The basic idea is to apply the robust combiner approach to networks, where we replace each router in the network with a robust combiner construction that’s consists of a number k of different routers. These routers are organized in a parallel circuit and the traffic originally entering the router is now forwarded by the hub to each of the k routers. After the routers process the packets and forward them to their outports, they reach the compare, the logic which lies at the heart of NetCo. The compare simply performs a majority decision (in general more than k/2 routers) where they may be compared bit by bit, or just based on the header or hashing can be used. The resilience is guaranteed assuming that in a given time at least majority of the candidates function correctly. The intuition is that backdoored devices controlled by an attacker in reality do not exhibit the same adversarial behavior i.e. they would be preconfigured with different malicious software generating different attack traffic patterns.

In SDN, we can get rid of the compare device and still ensure secure routing. Instead of comparing the packets locally, the switches perform the usual per-flow reporting to the controller, which ensures that the reported content is consistent for the majority of the switches. Integrating the compare functionality into the controller allows to guarantee availability using only two candidate network devices. The controller can detect a misbehavior if the two network devices differ in the data they report to the controller (this may induce some communication overhead and latency). For resilience, we replace some of the network devices with robust construction or fast failover mechanisms so that upon detection of misbehavior in some network location, we can reroute the packets through a different route.

**FRESCO (Framework for Enabling Security Controls in OpenFlow networks):**

FRESCO is an OpenFlow application designed specifically to facilitate the development of security applications for SDN networks. It is composed of the following components:

* Application Programming Interface (API) for scripting.
* Reusable models designed in python for providing security features.
* FRESCO-DB: database for storage and management of session’s information.

The OpenFlow specification defines only three mandatory actions (Drop, Forward and Group). FRESCO decomposes it into three sub actions each relating to a particular security need:

* Redirect (eg. in case of need to use a honey-pot feature)
* Mirror: copy traffic to another module for further analysis (eg. IDS)
* Quarantine: Insulation network traffic (infected mail)

The architecture of FRESCO is as follows:

* Application Layer: FRESCO development environment (DE): modules FRESCO modules, FRESCO DB and FRESCO scripting language.

FRESCO Resource Controller (RC): monitors the OpenFlow switches and keeps track of their states. The RC ensures immediate insertion of flow rules issued by the FRESCO applications seeing that they are urgent. It ensures the eviction of outdates rules in case of lack of space to insert FRESCO rules (Switch monitor and garbage collection).

FRESCO DE and RC enable developers to use scripting language to instantiate and define interactions between security modules.

Kernel Layer: FRESCO SEK (Security Enforcement Kernel) ensures that the rules resulting from the critical security applications are applied with priority over to other rules.

FRESCO DB module represents a knowledge base containing valuable information that are useful for various security applications. It can still be improved by pre-processing and correlation capabilities in order to allow applications to have quickly the relevant information.

**Moving Target Defense (MTD**) abandons the way of building a defect free defense system to protect network security, and develops a mechanism by irregularly changing network parameters over time to improve the attacker’s complexity and cost. In this way, MTD can limit the leakage of vulnerabilities and reduce the success rate of attacks. Hence, MTD revolves about the movement of network reconfiguration securely communicating reconfiguration specifications to other network nodes as required and ensuring that connectivity between nodes is uninterrupted. The MTD approach eliminates adversaries targeting known static attributes of network devices and systems such as network randomization for TCP/UDP ports, for IP addresses and for network paths.

1. Network randomization for TCP/UDP ports: The TCP/UDP port hopper is implemented at the host level. The port hopper follows a distributed system model that relies on time synchronization for all nodes communicating in the network. Communications between endpoints “hop” between network port numbers at specified intervals of time that are configurable. These intervals include tolerance thresholds to account for delays incurred during transmission and traversal.
2. Network randomization for IP addresses: The implementation of the IP address randomizer consists of a SDN controller that manages communication among network devices that pass IP-addressed traffic. As traffic ingresses the network-layer device, its source destination pairs are validated and rules are installed to encode/decode randomized IP addresses. The duration of these rules can be made static, random or be forced to change from a trusted third party. Since, the IP address randomizer is implemented at the network layer, it is transparent to the communicating endpoint processes and may be implemented without endpoint modification.
3. Network randomization for network paths: The network path randomizer uses overlay networks to impede traffic analysis, given that traffic analysis is a technique often used by an adversary to identify endpoints. The underlying network consists of several nodes that form a physical mesh topology. Routing through the mesh is coordinated among the nodes via a SDN controller. For each flow through the network, the controller may assign asymmetric forward/reverse paths, with the ability to modify the given specified parameters (time, bitrate etc.). This approach also improves network resiliency to eavesdropping attacks and denial of service attacks by providing multiple possible communication paths between nodes.

**Path Hopping based SDN network defense technology**

To solve the problems of traditional route selection and enhance network and system security, path hopping technologies can make the communication paths between the source and destination sides randomly hop. In the reverse AODV based path hopping scheme, multiple paths can be established from source node to destination node which is suitable for ad hoc networks but is difficult to be implemented in wired networks. A proactive random route mutation technique, which can simultaneously change the routes of multiple flows to defend against sniffing, eavesdropping and DoS attacks was proposed. To proactively resist DoS attack, a flexible multi path routing method which combines game theory and constraint satisfaction optimization was proposed for better QoS. Existing path hopping technologies can disperse traffic across multiple transmission paths or routes and have a good preventive effect on local attackers who can intercept only a part of the network links and nodes. However, if a global attacker can capture all related network packets of target flow and thus effectively get the real traffic and their order by correlation analysis, because these dispersed flows send by above path hopping technologies belong to a single flow, whose source and destination addresses and ports keeps the same, they are not efficient enough for defending against network interception and analysis attackers. However, to defend against the same, the PH-SND technology (Path Hopping based SDN network defense) was proposed.

The PH-SND technology

The main challenge for path hopping is randomly changing routes between given source and destination addresses while fulfilling some constraints. Hopping routes and paths can be pre-calculated and be deployed in routing configuration. However, here we model path hopping constraint based on the satisfiability modulo theory (SMT) formalization and find the satisfied routing paths by using the SMT solver. For modeling the path, the duration of a flow is divided into a lot of slots and PH-SND aims at finding different routes for each hopping slot between the source IP address host (S) and destination IP address host (D) while satisfying the following constraints:

1. Overlap constrain: To increase the unpredictability and provide better load balance, new routes should avoid containing recently used intermediate node and hence less overlaps. 2. Capacity constraint: New routes should not contain the node or link which is already overloaded or can’t satisfy bandwidth requirements. The hopping constraint based on SMT formalization defines rules which guarantee the balance between incoming and outgoing edges, the correct source and destination nodes and also make sure that a previously used route will not be used in a current node. After every hopping slot, the old route is deleted and the new route installed. A reverse route is also installed for each route to achieve bidirectional communication path hopping. Not only does the PH-SND technology randomly hop paths, the source and destination addresses and ports of every protected packet are randomly modified by every OpenFlow switch along the communication path (by using different variables: real and virtual for ports and IP addresses)

**Fingerprinting of SDN networks**

Fingerprinting implies the ability of a remote adversary to identify whether an interaction between the controller and the switches (and a subsequent rule installation) has been triggered by a given packet. The absence of a controller-switch interaction typically provides evidence that the flow rules that handle the received packet are already installed at the switches. However, if the communication between the controller and the switches is triggered then the received packet requires further examination by the controller, i.e. it might not have a matching entry stored at the switch’s flow table or because the controller requires additional information before installing a forwarding decision at the switches. In the given proposal, both active and passive adversaries are considered. The active adversary can compromise a remote client by injecting probe packets of her choice and capture the corresponding responses issued by the server. However, a passive adversary cannot inject packets but can monitor the exchanged traffic between the server and the client. Hence, passive adversaries are hard to detect by standard intrusion detection systems since they do not generate any extra network traffic.

The proposed countermeasure focuses on processing packets which pertain to existing flows. We leverage the group table and the internal timer maintained by a switch to identify whether this flow has recently appeared. The group tables are used in OpenFlow switches to describe per-packet forwarding conditions. A group table contains one or more buckets, which in turn contain an action set, similar to the one contained in the flow rules. A group table is further associated with a bucket selection logic, which is related to the group table type. For eg. a group table of type “fast failover” implements a selection logic that associates each bucket to a switch’s port. Then, the logic selects the first bucket in the table whose associated switch’s port status is live. The countermeasure defines a new bucket selection logic for the group table, such that packets of active flows are immediately forwarded, while packets of inactive flows are forwarded onto a special port that connects the switch to a network delay element. Our selection logic considers a flow inactive if no packets for such a flow were received by the switch in a threshold amount of time which is measured by the switch’s internal timer. The first received packet of an inactive flow is delayed which gives the adversary little advantage in identifying whether the additional delay measured by the RTT feature is caused by the controller-switch interaction or is artificially introduced by our countermeasure. Moreover, all the packets of the same flow received within a short time window are also delayed by a small time.

**AuthFlow: Authentication and access control mechanism for Software Defined Networks based on Host credentials**

The paper proposes a host authentication mechanism just above the MAC layer in an OpenFlow network which guarantees a low overhead and ensures a fine-grained access protocol. A credential based authentication to perform an access control according to the privilege level of each host, through mapping the host credentials to the set of flows that belong to the host. A new framework for control applications, enabling SDN controllers to use the host identity as a new flow field to define forwarding rules. AuthFlow denies the access of hosts either without valid credentials or with revoked authorization. The proposed scheme also allows different levels of access to network resources according to its credentials.

**FortNOX (Implements role based authentication for determining the security authorization of each OF application):**

A new flow rule that conflicts with an existing flow rule will be detected by FortNOX. If the new (conflicting) flow rule request was generated by a higher priority author, then the existing flow rule will be replaced. However, if the new flow rule is produced by a lower priority author, then it will be ignored. Three flow rule producer roles are defined: OF Operator, OF Security and OF Application. A limitation of this approach is the determination of appropriate security authorization level and priority enforcement.

**ROSEMARY**

Each OF application is run within an independent instance of ROSEMARY on a micro NOS architecture effectively sandboxing the application to protect the control layer from any vulnerability or malicious operation of the application. The solution separates network applications from the trusted Computing base of the NOS, monitors and controls network resources consumed by each application, monitors and controls application operations such as privileged system calls, and implements a safe NOS restart process to carefully restart each service improving the resilience of the control plane.

**LegoSDN**

This method was proposed to increase controller reliability even in the case of SDN application failures. In order to avoid the crash of an SDN application leading to the crash of an SDN controller, the authors proposed an isolation layer between SDN apps, a network wide transaction system to support atomic updates and efficient rollbacks, and a fault tolerance layer to overcome crash triggering events.

**Securing Distributed Control in SDN**

We assume that the central control element is secured by TLS. The author proposes a hybrid control model to increase network efficiency. They then present a signature algorithm to securely transmit flow installation requests from network device to network device to prevent any malicious user from obtaining any information related to that flow entry or disclosing its contents thus preventing the user from obtaining knowledge about the network or its operation or control. The system requires a centralized trust manager and introduces significant overhead in message passing and signature checking.

**Best Recommended Practices to be considered in SDN deployment today**

1. **Policy Conflict Resolution/ Network Invariant Detection**: When application modules manipulate the network state, the controller should identify misconfigurations, unauthorized access, and irregularities to ensure correct functioning of the entities. Therefore, a policy conflict resolution subsystem is recommended to avoid network logic manipulation issues.
2. **Mutual Authentication**: SDN components should enable authentication solutions both within and across trusted domains to avoid the introduction of insecure access to network resources. This prevents data manipulation attacks, impersonation of components and ensures secure identification of network entities.
3. **Control Plane Isolation via Slicing**: Unlike network virtualization, slicing the network resources will partition the resources allocated for tenants/users sharing the infrastructure. From a security standpoint, this isolated environment that can be securely instantiated and protected from unauthorized access, data manipulations and data leakages.
4. **Containerized Applications**: Assessing the different controller implementations, network applications are either statically compiled with the controller code, instantiated as a dynamic module with the controller software or integrated as a third party software with remote access to the controller. To prevent or restrict the impact of malicious application behavior, it is recommended to support application containerization, which can authenticate the application during setup, control the application’s access rights on the infrastructure (bandwidth, latency, counters to monitor etc), and limit, account for and isolate the resource usage for each application.
5. **Rate-Limiting, Flow Aggregation and Short Timeouts:** the correct use of flow and switch attributes such as flags, timeouts, modes of operation as well as the inherent security features such as metering to rate-limit the data flow to the control plane can ensure correct packet forwarding behavior by avoiding overlaps, notify flow deletes and operating securely when the connection is lost with the controller.
6. **Logging/ Forensics for IDS/IPS**: Network services and applications with monitoring capabilities require logging critical information and positively benefit from the logged information when troubleshooting and debugging the infrastructure. They should be able to determine network state at any given time and be able to track back to identify the network state at a previous point in time.

**Security Proposals:**

For switch memory saturation attacks: we have to give a precise definition of memory allocation functions, platforms should limit the available memory space on a per application basis (eg. in terms of rows of a stateful table) and use reactive approaches to monitor the memory usage of applications and react in case of excessive occupation (eg. by notifying the control plane )

CPU exhaustion attack: Designers should carefully identify potential CPU intensive operations (eg. computing cryptographic functions) and make sure they cannot be triggered by attackers manipulating messages.

State Inconsistency Check: Designers should introduce API’s for trust management and mechanisms for authentication between switches. As an example, switches may utilize Message Authentication Codes (MACs) in flows tags and/or control messages to guarantee the authenticity of information. Hardware acceleration integration may be considered, in order to speed up the execution of potentially expensive cryptographic functions. Mechanisms and infrastructures should be provided to manage distributed states between data plane and control plane to ensure consistency. API’s can be written to define fine grained access to policies on state table, single entries or state variables.

Secure inbound signaling:

Leveraging stateful SDN data plane capabilities, recent proposals have provided solutions for failure recovery, load balancing and other applications which use inbound signaling (same channel for data and control channels). However, the proposals generally lack mechanisms for verifying authenticity and integrity of information exchanged between switches. Therefore, we can provide APIs for trust model definition and management between switches or we can develop mechanisms for information authenticity and integrity verification. The lack of authentication mechanism between switches allows an adversary to spoof and inject malicious control messages into the network resulting in many attacks. Lightweight authentication mechanisms for the signaling packets can be employed. But since cryptographic operations are clearly expensive and might reduce overall network performance, designers may investigate the use and integration of hardware support.

Local Monitoring

Switches should perform local monitoring on the resources allocated to each stateful application, or run more sophisticated anomaly detection algorithms

State Consistency Check:

To avoid local inconsistency between the switch and the controller due to the stateful data plane it has to retain some form of lose control on how these local states evolve even if it does not have an active role in enforcing local states inside the switches. However, the situation in which the switch needs to update the controller should be precisely defined, since otherwise the updating procedure may lead to CPU failure (in case of per state transition update). We could use the policy enforcing methodology proposed by SNAP. In SNAP, the central controller enforces some access policies based on which the switches decide where and when to access and modify the state variables. SNAP also proposes centralized state storage in one switch which requires careful ordering of the packets and state transitions and ensuring that the packets will pass by the correct switch.

AuthFlow

Due to centralization provided by SDN, misused behavior of a unique component can compromise operation of the entire network such as performing denial of service attacks against the network controller. Hence, an access control and authentication mechanism of end hosts and switches are essential for SDN security, through the use of Secure Sockets Layer (SSL) and Public Key Infrastructure (PKI). In cases of malicious behavior, the node authentication can be revoked and the node may be isolated from the network. The main contributions are:

1. A host authentication mechanism just above the MAC layer in an OpenFlow network, guarantees a low overhead and ensures a fine-grained access control by mapping authentication credentials of the supplicant host into the flow set belonging to each host.
2. A credential based authentication to perform an access control according to the privilege level of each host, through mapping the host credentials to the set of flows that belongs to the host.
3. A new framework for control applications, enabling Software Defined Network controllers to use the host identity as a new flow field to define forwarding rules.

AuthFlow Mechanism

AuthFlow applies the IEEE 802.1X standard (widely adopted, requires no change on end hosts and works just above the MAC layer) and Extensible Authentication Protocol (EAP). EAP encapsulates authentication messages exchanged between the supplicant host and RADIUS authentication server. The translation of IEEE 802.1X messages into RADIUS packets is performed by the authenticator. AuthFlow authenticator also called the radius client is a software module that communicates with the AuthFlow application running on top of the controller. This application allows or denies traffic from a host depending on the result of authentication between the host and the authenticator. It also sends a confirmation message of authentication success over a secure, encrypted and authentication channel using SSL. The authentication server i.e. the RADIUS server extracts the information encapsulated in EAP and validates the credentials presented by virtual routers against a database.

EAP allows the use of several different authentication methods, the method adopted in the reference paper was MS-CHAP v2 (Microsoft Challenge Handshake Authentication Protocol), which authenticates virtual routers against a database using username and password as credentials. Lightweight Directory Access Protocol (LDAP) is used to store username password pairs and other parameters that define the privileges of the router for network access.

The mechanism of authentication works as follows: A virtual router sends an authentication request, standardized by IEEE 802.1X and the controller redirects it to the authenticator. The authenticator responds to it and the host sends it credentials. The authenticator then checks the credentials against the RADIUS server, running the authentication method defined in EAP. If the authentication procedure succeeds, the authenticator sends a success message for the host and an authorization and confirmation message to the controller. The AuthFlow application on top of the controller then allows the host to access network resources. In case of revocation of the authentication of the host , the authenticator informs the application which immediately denies the host access to the network, erasing and blocking all flow entries from and to the banned host.

The AuthFlow application controls access of end hosts by denying or allowing virtual machines to access virtual links and services. The main idea is to block or to allow the creation of new flow entries that uses the virtual links. These links do not need to perform an authentication process to allow their traffic, AuthFlow performs its topology discovery of the network core via Link Layer Discovery Protocol (LLDP). LLDP packets are forwarded link to link and the controller generates and verifies each LLDP packet transmitted, the controller is able to identify which links are between OpenFlow switches and which links are connecting end hosts. LLDP packets generated by the controller are tagged with a nonce, an arbitrary number that is only used once, so as to avoid replay or spoofing attacks. (This is only possible because switches are assumed to be trustworthy entities which have already been authenticated through the OpenFlow secure channel).

Through the proposed access control, allowing or denying end hosts traffic depends on the source/ destination MAC address and the incoming/ outgoing port, if they are equal to those in the authentication tuple, the authentication credentials and identity of the host are assigned to this flow. Forwarding decisions takes these and the the credentials into consideration, making access control fine grained.

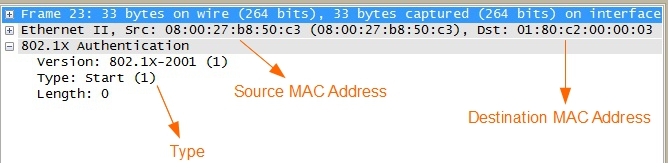
Problems:

The EAP- MSCHAP v2 is considered weak in the realm of authentication, and whilst it does allow configurators to select their own authentication and encryption methods, any hacker with knowledge in traditional attacks should be able to force an entry into the network as a rogue data path.

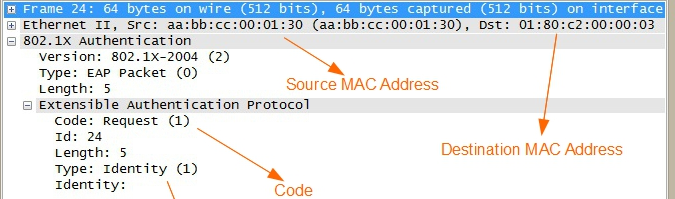
IEEE 802.1X

IEEE 802.1X uses EAP to exchange messages during the authentication process. In a wired Ethernet LAN, EAP over LAN (EAPoL) is used to transport packets between supplicant and an authenticator over Local Area Network. Before authentication, the identity of the endpoint is unknown and all traffic is blocked except EAPoL. Once, the user credentials are successfully verified, other user traffic is permitted. EAPoL frames have ether type 0x888e. IEEE 802.1X provides a way for the supplicant and authenticator to negotiate an EAP authentication method which defines credential type and how the credentials are submitted from the supplicant to the authentication server through RADIUS (Remote Authentication Dial In User Service).

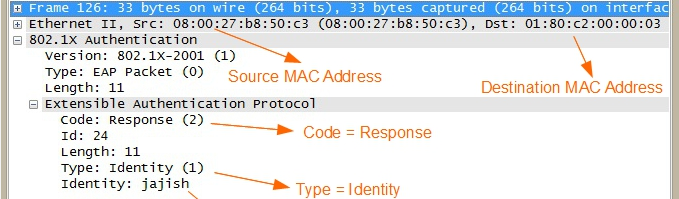
Step 1: When the supplicant first connects to the LAN, it will send EAPoL-Start message to a multicast group (special destination muticast address 01:80:c2:00:00:03) to identify the authenticator.



Step 2: Authenticator will send back EAP-Request Identity message (in response to the EAPoL start message) which I sued to request identity from the supplicant such as the username. Authenticator sends out EAP request identity periodically even before receiving an EAP-Ol start message.

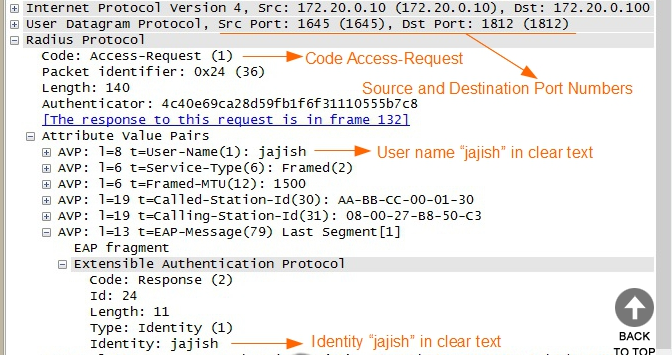


Step 3: As a response to EAP-Request identity message, the supplicant provided identity (example user name) in an EAP-Response message to the authenticator.

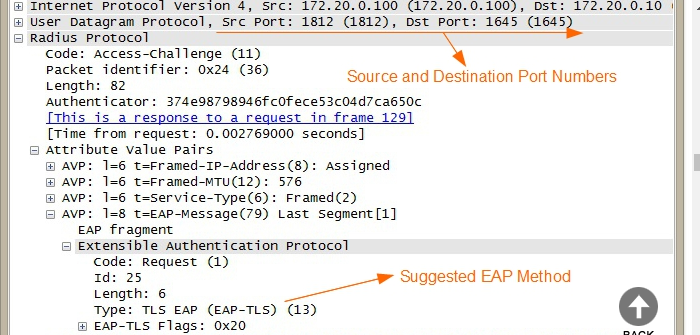


Step 4: The authenticator forwards the information received from the supplicant to the authentication server (RADIUS server, in this case). The RADIUS uses UDP as transport layer protocol between authenticator and authentication server (portnumbers: 1812 for authentication 1813 for accounting in RADIUS and 1645 for authentication and 1646 for accounting in UDP).

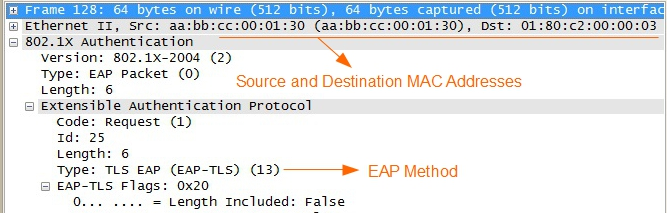
The authenticator creates a RADIUS access request message and forwards it to the authentication server.



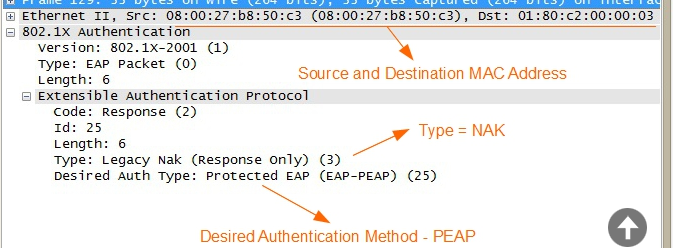
Step 5: Once the authentication server (RADIUS server) received the access request RADIUS message, the authentication server will send back a RADIUS access-challenge message to the authenticator. The access challenge message from the server contains not only the challenge, but also the authentication method to be used for further communication. The supplicant can reject the authentication method and request another which it supports.



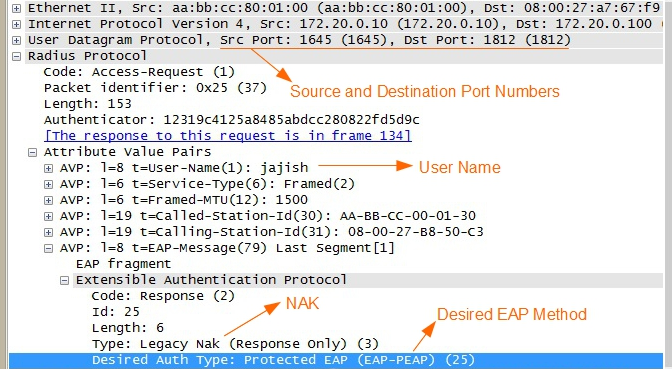
Step 6: The authenticator receives the access challenge message from the authentication server. The authenticator then prepares an EAP request message and forwards it to the supplicant. The purpose of this EAP request message is to verify that the supplicant can support the EAP method suggested by the RADIUS/authentication server.



Step 7: If the supplicant is not configured to support the suggested EAP method, it can reject the proposed method and request another EAP method by sending an EAP- Response Auth NAK message which also specifies the authentication method the supplicant wants to use.



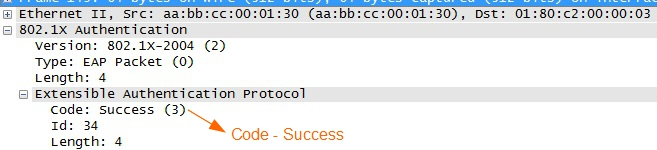
Step 8: Authenticator will forward the EAPoL NAK message and the desired authentication method sent form the supplicant to the authentication server as RADIUS access request message. Now the RADIUS server can start the authentication process based on desired authentication method.



Step 9: The authentication server then sends back a new access challenge message, based on the EAP authentication method supported by the supplicant. Once the EAP authentication method is agreed between the supplicant and authentication server, a few more messages are exchanged related to EAP authentication method.



Step 10: When the credentials of the user are successfully verified by authentication server after the authentication process, the authentication server will send a RADIUS accept-accept message to the authenticator indicating that this user request is valid. If the credentials are wrong, the authentication server will send a RADIUS accept-reject message to the authenticator indicating that this user request is invalid.



Step 11: Once the authenticator receives RADIUS accept-accept from authentications erver, the authenticator sends back a EAP success message to the supplicant. The switch port will be open for the client for network communication.

If there is a problem in evaluating the credentials of the user, the authenticator will send back a EAP failure message to the supplicant.

**BAN (Michael Burrows, Martin Abadi and Roger Needham) Logic**

Authentication protocols are the basis of security in many distributed systems, and it is therefore essential to ensure that these protocols function currently. Unfortunately, most protocols found in literature contain redundancies or security flaws. A simple logic, such as BAN has allowed the description of the beliefs of trustworthy parties involved in authentication protocols and the evolution of these beliefs as a consequence of communication.

This formal method of analysis helps us in answering the questions:

* Does this protocol work? Can it be made to work?
* Exactly what does this protocol achieve?
* Does this protocol need more assumptions than another protocol?
* Does this protocol do anything unnecessary?

In BAN logic, symbols A,B and S denote specific principals, Kab, Kas and Kbs denote specific shared keys, Ka, Kb and Ks denote specific public keys and Ka-1,Kb-1 and Ks-1 denote the corresponding secret keys and Na, Nb and Ns denote specific statements.

Semantics and rules of BAN are as follows:

*  : P is entitled to believe X and can take it as true.
* : Someone has sent a message containing X to P, who can read and repeat X (mostly after some decryption). Therefore, P sees X.
* : The principal P at some time sent a message including the statement X. It is not known whether the message was sent long ago or during the current run of the protocol, but it is known that P believed X then.
* : P has jurisdiction over X. The principal P is an authority on X and should be trusted on this matter.
* Fresh(X): X has not been sent in a message at any time before the current run of the protocol. Usually true for nonces, which might commonly be a timestamp or a number that is used only once. The expression was invented for the purpose of being fresh.
* P Q : P and Q use the shared key K to communicate. The key K will not be discovered by any principal except P or Q or principal trusted by either P or Q.
* .  P : P has K as a public key. The matching secret key (K-1) will never be discovered by any principal except P or a principal trusted by P.
* P  Q: The formula X is a secret known only to P and Q, and possibly to principals trusted by them. Only P and Q may use X to prove their identities to one another. (such as passwords)
* **:** This represents the formula X encrypted under the key K.
* **:** This represents X combined with the formula Y, it is intended that Y be a secret and that its presence prove the identity of whoever utters . In implementations, X is simply concatenated with Y.
* **:** P has a good public key Kp
* **:** P has a good private key KP-1
* **:** X signed with P’s private key KP-1.
* **:** The certificate of P issued by CA.
* **:** (t1, t2 ) is a good time interval

Logic Postulates

1. For private keys:



If P believes that the key is shared with Q and sees X encrypted under K, then P believes Q once said X. (For this to be sound, we must first guarantee that P did not send the message himself).

For public keys:



For shared secrets:



If P believes that the secret Y is shared with Q and sees  then P believes that Q once said X given that it was not uttered by P himself.

1. The nonce-verification rule expresses the check that a message is recent and hence the sender still believes in it.

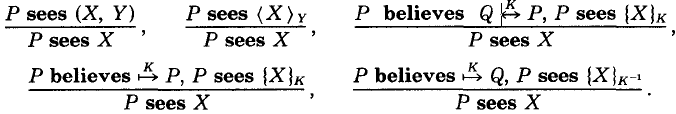


If P believes that X could have been uttered only recently and Q once said X, then P believes that Q still believes X.

1. The jurisdiction rule states that if P believes that Q has jurisdiction over X, then P trusts Q on the truth of X.

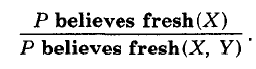


1. If a principal sees a formula, then he also sees its components, provided he knows the necessary keys:



Note that if P sees X and P sees Y, it does not follow that P sees (X,Y), since that means X and Y were uttered at the same time.

1. If one part of the formula is fresh, then the entire formula must also be fresh:



1. Message meaning (for public key) rule:

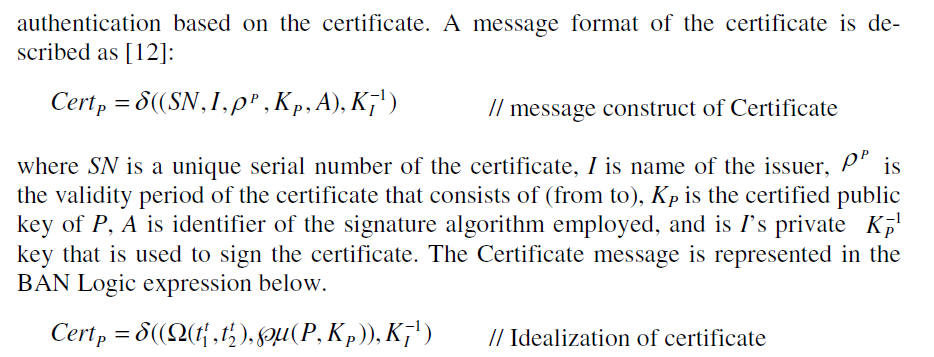


If Q has good public and private keys and P has seen X encrypted with Q’s private key, then Q has seen X.

1. Message format for the certificate used in authentication is:



Where SN is a unique serial number of the certificate, I is the name of the issuer  is the validity period of the certificate,  is the certified public key of P, A is the identifier of the signature algorithm employed and I’s private key is used to sign the certificate. In BAN logic expression, it is represented as:



Idealized Protocols:



This denotes that the principal P sends a message to the principal Q. A message in the idealized protocol is called a formula.

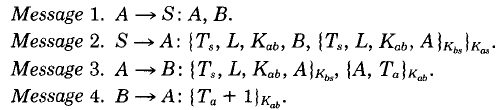
, this step tells B(who knows the key Kbs), that Kab is a key to communicate with A. This step should be written as 

The goals of authentication, formalized: Initial assumptions must invariably be made to guarantee the success of each protocol. Typically, the assumptions state what keys are initially shared between the principals, which principals have generated fresh nonces, and which principals are trusted in certain ways. Once all the assumptions have been written, the verification of a protocol amounts to proving that some formulas hold as conclusions.

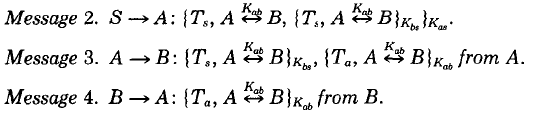
1. The Kerberos Protocol

The Kerberos protocol establishes a shared key between two principals with help from an authentication server. It is based on the shared key Needham-Schroeder protocols and makes use of timestamps as nonces, both to remove security problems and to reduce the total number of messages required.

Here, A and B are the two principals, Kas and Kbs are their private keys and S is the authentication server. S and A generate the timestamps, Ts and Ta respectively and S generates the lifetime L. The fourth message is used only if mutual authentication is required.



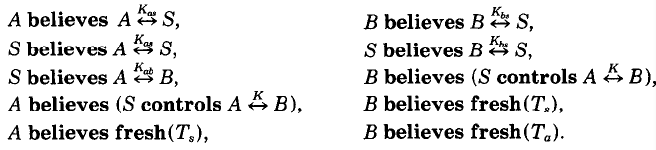
First, A sends a cleartext message to S stating his desire to communicate with B. The server responds with an encrypted message containing a timestamp, a lifetime, a session key for A and B, a ticket that only B can read. This ticket also contains the timestamp, lifetime and the key. A forwards the ticket to B together with an authenticator (a timestamp encrypted with the session key). B first decrypts the ticket and checks the timestamp and lifetime. If the ticket has been created recently enough, he uses the enclosed key to decrypt the authenticator. Then if the authenticator’s timestamp is recent, he uses the session key to return the timestamp, which he checks. Once the principals are satisfied, they can proceed to use the session key which can be idealized as:



For simplicity, the lifetime L has been combined with the timestamp Ts, which is treated like nonce. The first message is omitted since it does not contribute to the logical properties of the protocol.

Kerberos Protocol Analyzed:

To analyze this protocol, we first give the following assumptions:



The first four are about shared keys between the clients and the server. The fifth indicates that the server initially knows the key for communication between A and B. The next two indicate the trust that A and B have in the server to generate a good encryption key. The final three assumptions show that A and B believe that timestamps generated elsewhere are fresh, this indicates that the protocol relies heavily on the use of synchronized clocks. The analysis of the idealized version of Kerberos is done by applying our rules to the assumptions.

When A receives message 2, the rules yield that holds , since we have the hypothesis , the message meaning rule for shared keys applies and yields:, and breaking conjunctions we get: and we have the hypothesis:  and this the nonce verification rule applies and yields: and on breaking conjunctions again, we obtainand Finally, the jurisdiction rule applies and yields:and as A passes the ticket to B in the third message along with a timestamp and we can obtain  in the same way as was done for message 2 via the message meaning, jurisdiction and nonce verification postulates, this decryption of the message leads to . The fourth message simply assures A that B believes in the key and received A’s last new message. After the applications of postulates to these messages, we obtain:

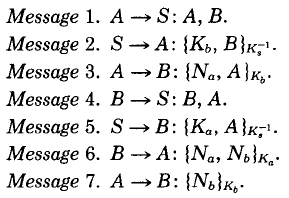


If only the three messages are used, A is not convinced of B’s existence and observes the same messages whether B is running or not.

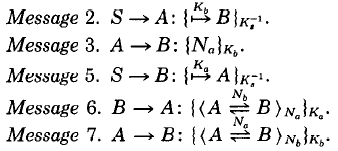
1. The Needham Schroeder Public Key Protocol:

This protocol allowed two principals to exchange two secret numbers based on public-key cryptography. A weakness in the protocol permits a replay attack in the interactions with the certification authority if a key is compromised.

Here, S, whose public key is Ks, operates only as a certification authoritybetween A and B, whose public keys are Ka and Kb, Na and Nb  are nonces. The message exchange is as follows:



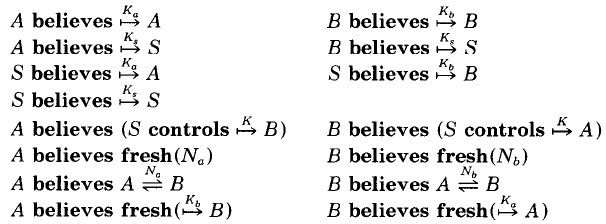
It is expected initially, that A and B hold S’s public key Ks.  Therefore, the principals A and B can obtain each other’s public key from S and then use the public keys obtained to communicate the secret nonce identifiers Na and Nb. These secrtes can be used later for signing further messages. For example, if B receives a message {X, Na} Kb, then B may deduce that A has sent X. The idelaized protocol is as follows:



Messages 1 and 4 are omitted as they do not contribute to the logical properties of the protocol. Here, Na and Nb. are used as secrets later to exchange messages with encryption.

The protocol analyzed:

The assumed intial beliefs are:



Each principal knows the public key of the certification agent S, as well as their own keys. In addition, S knows the public keys of A and B. Each principal trusts the certification agent to correctly sign certificates giving the public key of each other. Also, each principal believes that each secret they generate is fresh. Moreover, they also have to assume that the message containing the public key of the other principal is fresh, which can be resolved by adding timestamps. This leads to the final beliefs:



Each principal knows the public key of the other and has knowledge of the shared secret that he believes the other will accept as being shared only by two principals. A and B can now continue to exchange messages using Na, Nb and public key encryption. In this way they can transfer data and other keys securely.

**Scyther**

Scyther assumes that all cryptographic functions are perfect: the adversary learns nothing from an encrypted message unless he knows the decryption key. However, the tool can be used to find problems that arise from the way the protocol is constructed. This language describes protocols through roles which are in turn described by events that denote the sending or receiving of terms. Scyther has semantics for defining constants, variables, symmetric and asymmetric keys, hash functions, events and claims.

The developer’s predefined the adversaries model according to Dolev-Yao’s model to save user’s effort, but it also brings a minor drawback by limiting the diversity of the adversaries abilities. An adversary in this model has five abilities: eavesdropping messages, deleting messages, carrying out cryptographic analysis to intercepted messages to gain knowledge, creating new messages based on its knowledge and inserting them into communication channels.

Basic Syntax:

1. Comments start with // or # for single line comments and /\* and \*/ for multi line comments.
2. The language is case sensitive.
3. The language is based on protocol definitions:

Protocol ExampleProtocol (I, R) {

Role I {};

Role R {};

};

This implies that we are defining a protocol called ExampleProtocol which has two roles I and R. The events that describe these roles are defined within the curly braces.

1. Freshly generated values: Many security protocols rely on generating random values which are specified in Scyther through the keyword fresh. Eg. To generate a random value Na of type nonce:

Role X (…){

fresh Na: Nonce;

send\_1 (X,Y, Na);

};

1. Variables: Agents can use variables to store received terms. Eg. To receive nonce into a variable with name Na, we can write:

Role Y(…){  
var Na: Nonce;

Recv\_1(X,Y, Na);

};

Local declarations are local to the role. Hence, we can specify a freshly generated nonce Na in one role and a variable Na in another role without any conflicts. Variables are rigid, once the event in which they occur has been executed, they are assigned a value which cannot be changed.

1. Pairing: Any two terms can be combined into a term pair. Eg. (x,y) for x and y. We can also write n-tuples such as (x,y,z) which is interpreted as ((x,y),z).
2. Symmetric keys: Any term can act as a symmetric key for encryption. Unless kir is explicitly defined as being part of an asymmetric key pair, it is assumed as symmetric encryption. It is defined as k(X,Y) which denotes the long term symmetric key shared between X and Y.

Eg. {ni}kir which implies encryption of ni with the term kir.

1. Asymmetric keys: A public key infrastructure (PKI) is denoted by sk(X) and pk(X) which denote the private key and public key respectively. {ni}pk(I) denotes ni being encrypted with the public key pk(I) which can only be decrypted with the secret key sk(I).
2. Hash functions: They are essentially encryptions through a function whose inverse is not known to anyone. Since all agents and protocols should have access access to such a function, the hashfunction is defined globally i.e. declared outside any protocol definition.

Eg. hashfunction H1;

Inside the protocol messages: H1(ni)

1. Ticket are variables which can be substituted by any term.
2. Usertypes are used to define a new user defined type:

Usertype MyAtomicMessage;

Protocol X(I,R)

{

Role I

{

var y: MyAtomicMessage;

recv\_1(I,R,y);

};

Variables of this type can only be instantiated with messages m of this type i.e. they should be declared by the global declaration const m: MyAtomicMessage or the freshly generated fresh m: MyAtomicMessage.

1. Events: They mark the sending and receiving of messages by giving them the same label. In some protocols, we may want to send or receive to the adversary directly, in which case there will be no event. However, in the absence of a corresponding send or receive event, Scyther will output a warning which can be avoided by prefixing the label with !.

Eg. send\_!1 (I,I, LeakToAdversary);

Claim events and security properties: Claim events are used in role specifications to model intended security properties. Eg. claim(I,secret,ni)

Types of claim types:

* Secret: Requires a parameter term which is meant to be kept secret.
* SKR: This is equivalent to the secret claim, but it additionally marks the parameter term as a session-key.
* Alive: aliveness of all roles
* Weakagree: Weak agreement of all roles
* Commit, Running: This non-injective claim is used to model agreement over data. Commit requires the existence of a corresponding running signal in the trace.
* Nisynch: Non-injective synchronization
* Niagree: Non-injective agreement on messages
* Reachable: When this claim is verified, Scyther checks whether this claim can be reached at all. It is true if and only if there exists a trace in which this claim occurs. This can be useful to check if there is no obvious error in the protocol specification, and it is infact inserted when the check mode of Scyther is used.
* Empty: This claim is not verified, but simply ignored.

13. Internal computation/ Pattern match events:

Match event: match (pt,m): This implies that we can replace pt by m throughout the specification which could be useful in modelling equality tests, delayed decryption and checking commitments.

Eg. var x: nonce;

var y;

recv (R,I,X);

match (Y,hash(X,I,R));

send (I,R,Y,{Y}sk(I));

Not match event: not match (pt,m): This means that if there is no substitution for pt as m, then the event can be executed, this can be used to model inequality constraints.

Eg. role A{

Not match (A,B);

Send (A,B,m1);

};

Match and not match can also be used together to model if- else constraints.

1. Macros are abbreviations for particular terms used to simply protocol specification when they contain complex messages or repeating elements. They have a global scope.

Eg. hashfunction h;

Macro m1= {I, R, ni, h(R,ni)} pk(R);

Protocol macro\_example\_two (I, R)

{

Role I

{

Fresh ni: nonce;

send\_1 (I,R,m1);

}

Role R

{

Var ni: nonce;

Recv\_1 (I,R,m1);

}  
}

1. Include: It is possible to import other files in a protocol specification.

Eg. include “filename”

1. One-role-per-agent: The operational semantics allows agents to perform multiple roles, even in parallel. This elevates the security concerns as the adversary has many options to exploit. To avoid such attacks we use:

Option “- - one – role – per - agent”// disallows agents in multiple roles

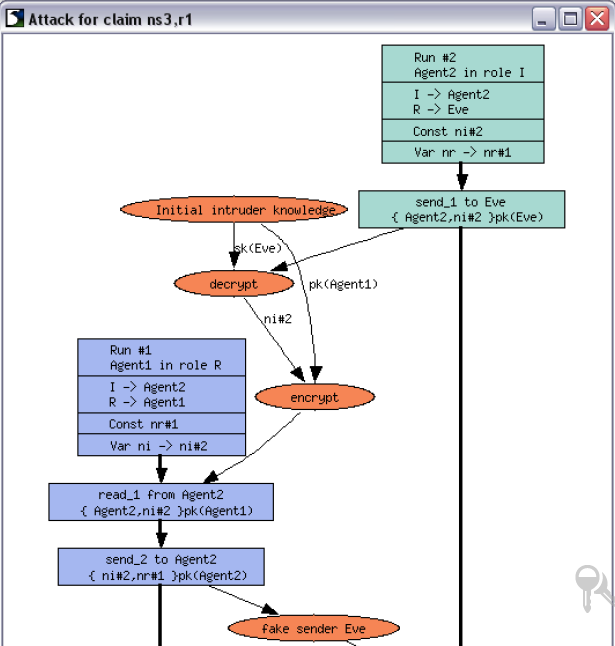
1. Comments:

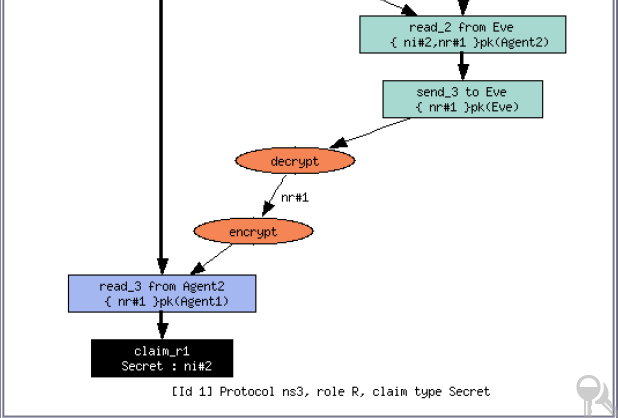
* Atleast X attacks: Some attacks were found in the state space, however due to the undecidability of the problem, we cannot be sure that there are no other attack states.
* Exactly X attacks: Within the statespace, there are exactly X attacks and no others.
* At least X patterns
* Exactly X patterns
* No attacks within bounds: No attacks were found within the bounded statespace, but there can possibly be an attack outside the bounded statespace.
* No attacks: No attack was found within the bounded or unbounded statespace, thus the protocol has been successfully verified.
* Bounding the statespace: During the verification process, the Scyther tool explores a proof tree that covers all possible protocol behaviors. The default setting is to bind the size of this tree in some way, ensuring that the verification procedure terminates.
* No attacks within bounds can be interpreted as Scyther did not find any attacks, but because it reached the bound, it did not explore the full tree and it is possible that there are still attacks on the protocol.

Way of bounding: Settings, maximum no of runs or protocol instances.

1. Attack Graphs

Each vertical axis represents a run, it starts with a diamond shaped box and is used to give information about the run.





Eg. Attacks for claim ns, R1 (each claim is assigned a run identifier – here 1, which is an arbitrary number that enables us to uniquely identify each run). This run executes the R role of the protocol. It is being executed by an agent called agent 1, who thinks he’s talking to agent 2. Note that although run 2 is being executed by agent 2, it does not believe he is talking to agent 1. He thinks that the responder role is being executed by the untrusted agent Eve. Additionally, the run headers contain information on the freshly generated values (eg. run 1 generated nr#1 and the information on the instantiation of local variables such as ni with with ni#2 or run 2).

Communication events: The incoming arrow does not indicate a direct sending of the message rather it denotes an ordering constraint which implies that this message can only be received after something else has happened. In this case, we can see that the message can be recived only after run 2 sends his initial message. The reason for this is the nonce ni#2, since the intruder cannot predict this nonce, it has to wait until run 2 has generated it.

In the graph the connecting arrow is red and has a label “construct” with it, which is caused by the fact that the message sent does not correspond to the message that is received. Other possibilities include a green and yellow arrow. A yellow arrow indicates that a message was sent, and received in exactly the same form, however the agents disagree about who was sending a message to whom. It is therefore labeled with redirect because the intruder must have redirected the message. A green arrow indicates that a message is received exactly the same way it was sent, representing a normal communication between two agents. A recv event without an incoming arrow indicates that a term is received that can be generated from the initial knowledge of the intruder

Point-to-Point Protocol (PPP)

PPP is a data link (layer 2) protocol used to establish direct connection between two nodes without any host or networking device in between. PPP provides a standard method for transporting multi-protocol datagrams over point-to-point links. It can also provide authentication, transmission encryption and compression. PPP uses LCP (Link Control Protocol), CHAP (Challenge Handshake Authentication Protocol) and EAP (Extensible authentication protocol) for self-configuration and authentication.

Extensible Authentication Protocol (EAP)

EAP is an authentication framework for providing the transport and usage of keying material and parameters generated by EAP methods. Today, deployments of IEEE 802.11 wireless LAN’s are based on EAP and use several methods with some common functions and negotiation. Eg. EAP-TLS, EAP-TTLS, PEAP and EAP-SIM. These methods support authentication credentials that include digital certificates, usernames and passwords, secure tokens and SIM secrets. Other credential types that may be used include public/private key (without necessarily requiring certificates) and asymmetric credential support (such as passwords on one side and public/ private key on the other).

EAP authentication methods must satisfy the following criteria:

* Symmetric key derivation security claims
* Effective key strength of minimum 128 bits (the master session key and extended master session key are of atleast 64 bytes), the keys must also be fresh and algorithmically independent.
* Mutual authentication support
* Shared state equivalence (such as method version number, cryptographic keys, specific attributes negotiated, shared credentials)
* Resistance to dictionary attacks
* Protection against man-in-the-middle attacks (cryptographic binding, integrity protection, replay protection and session independence).
* Protected Ciphersuite negotiation
* End user identity hiding (confidentiality security claim)
  1. **MS-CHAP v2 (Microsoft Challenge Handshake Authentication Protocol Version 2)**

MS-CHAP v2 was proposed to provide security for remote access connections. The process of authentication is described below in brief, followed by its descriptive representation:

1. The new client/ application first requests a login challenge from the authentication server.
2. The remote access server sends a challenge message to the remote access client that consists of a session identifier and a 16-byte arbitrary challenge string.
3. The remote access client sends a response message that contains the user name, a 16 byte random peer authenticator challenge, a 8 byte challenge by hashing the received challenge, peer authenticator challenge and the client’s username, a 24 byte reply by encrypting the 8 byte challenge with the MD-4 hashed version of the client’s password.
4. The server then uses the hashes of the client’s password stored in the LDAP database, to decrypt the replies. If the decrypted blocks match the challenge, the client is authenticated. The server then sends an indication of the success or failure of the connection attempt, by using the 16 byte peer authenticator challenge as well the client’s hashed password.
5. The remote access client also computes the authenticator response and if the computed response matches the received response, mutual authentication is successfully completed and the connection is used, else it is terminated.

Xa : 0 to 256 char User Name

Nb : 16 byte Arbitrary challenge string

Na: 16 byte Peer challenge string

Xb : Session Identifier (it is a one byte field and aids in matching requests and responses)

SHA: Secure hash algorithm version 1

C: 8 byte challenge

R: 24 byte response

Challenge Hash function( ):

Input: Authenticator challenge, Peer challenge, user name.

Output: 8 byte challenge (by using SHA)

Challenge Response input( ):

Input: 8 byte challenge, 16 byte password hash

Output: 24 byte challenge

Zpasswordhash: We will pad 5 bytes of zeroes, to make a 21 byte password hash.

DESencrypt( ):

Input: 8 byte clear, 7 byte key

Output: 8 byte cipher

DESencrypt (challenge, 1st 7 bytes of Zpasswordhash giving 1st 8 bytes of response)

DESencrypt (challenge, 2nd 7 bytes of Zpasswordhash giving 2nd 8 bytes of response)

DESencrypt (challenge, 3rd 7 bytes of Zpasswordhash giving 3rd 8 bytes of response)

Generate Authenticator Response( ):

Input: username, password, response, peer challenge, authenticator challenge

Output ( 42 byte authenticator response): 16 byte password hash, 16 bytes password hash hash, 8 byte challenge

Check authenticator response( ):

Input: username, password, response, peer challenge, authenticator challenge, 42 byte received response

Output: Boolean (Response OK), by generating its own authenticator response and checking if giving myresponse is same as received myresponse.

Verification through BAN

Assumptions:

A believes fresh Na  B believes fresh Nb

A believes A  B B believes A  B

Protocol Messages:

Message 1: A→B: A

Message 1: B→A: Nb

Message 2: A→B: Xa, Na, H (Xa,Na,Nb), {H (Xa,Na,Nb)} H(P)

Message 3: B→A: H(H((H(H(H(P)), {H (Xa,Na,Nb)}H(P)),H (Xa,Na,Nb))

The client also computes the auth response and if it matches the message received from the server, then mutual authentication is successful and the port is opened for communication, else the peer must end the session.

Do we have to describe different hashing functions such as SHA, PRF, md-5 or any others with different names in BAN and Scyther or will a simple h suffice for all hashing.

Verification on Scyther

Here, we are assuming that the initial password and username allocation to client and server is done securely. (and that no third party has this information before the protocol commences and that the only way to obtain this information is by deciphering the protocol)

hashfunction h1;

hashfunction h2;

usertype String;

const p: String;

const x: String;

const magicservertoclientsigningconstant: String;

const padtomakeitdomorethanoneiteration: String;

const authenticationsuccessful: String;

const authenticationfailed: String;

macro m1= h1(x,nr,ni);

macro m2= h2(p);

macro m3= magicservertoclientsigningconstant;

macro m4= padtomakeitdomorethanoneiteration;

macro m5= h1(h2(m2), {m1}m2, m3);

macro m6= h1(m5,m1,m4);

macro m7= authenticationsuccessful;

macro m8= authenticationfailed;

protocol mschap (I,R) {

role I {

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I,R,ni);

recv\_2 (R, I, x, nr, m1, {m1}m2);

send\_3 (I,R, m6 );

claim\_i1 (I, Secret, p);

claim\_i2 (I, Niagree);

claim\_i3 (I, Nisynch);

}

role R {

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I,R,ni);

send\_2 (R, I, x, nr, m1, {m1}m2);

recv\_3 (I,R, m6);

match(h1(h1(h2(m2), {m1}m2, m3),m1,m4), m6);

send\_4 (R, I, m7);

not match((h1(h1(h2(m2), {m1}m2, m3),m1,m4)), m6);

send\_5 (R, I, m8);

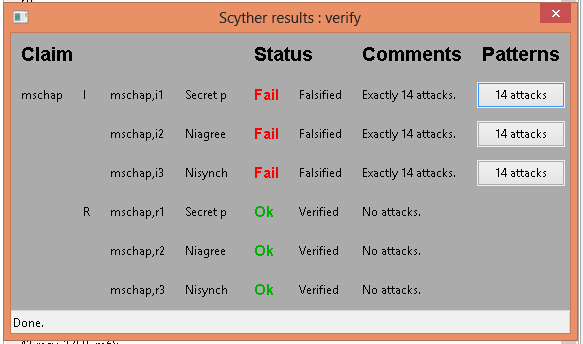
claim\_r1 (R, Secret, p);

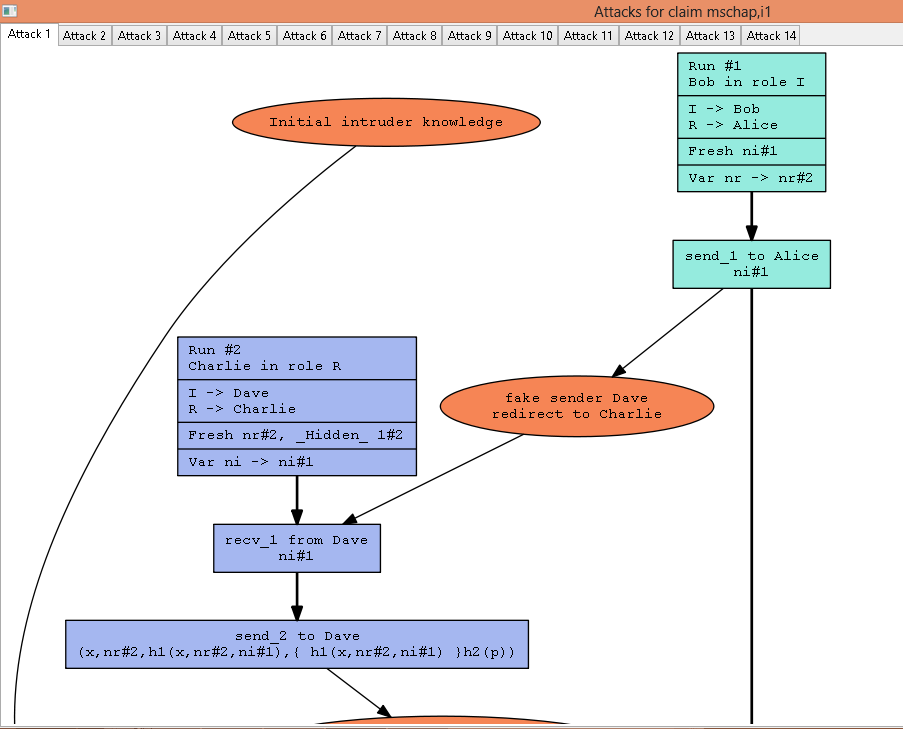
claim\_r2 (R, Niagree);

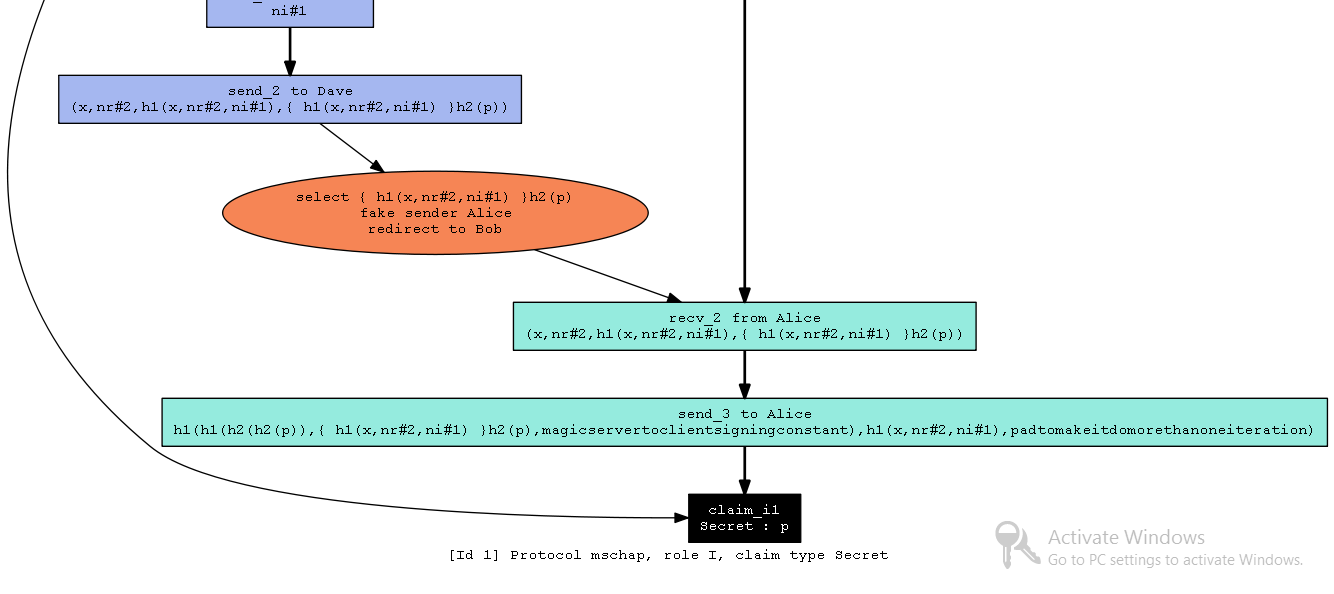
claim\_r3 (R, Nisynch);

}

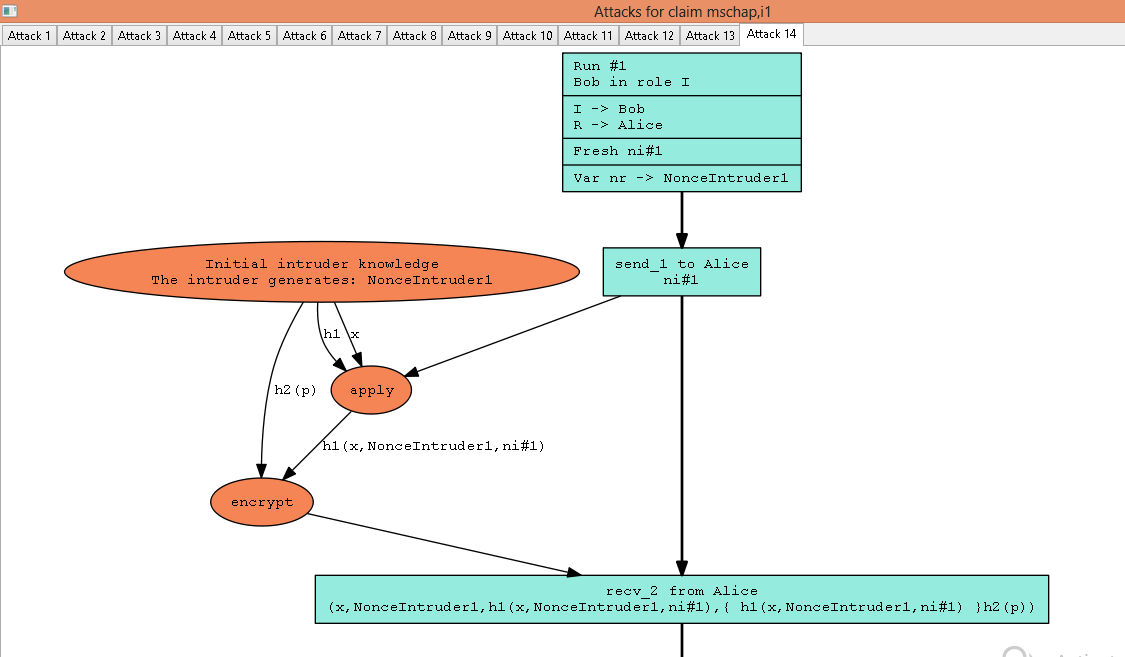
}

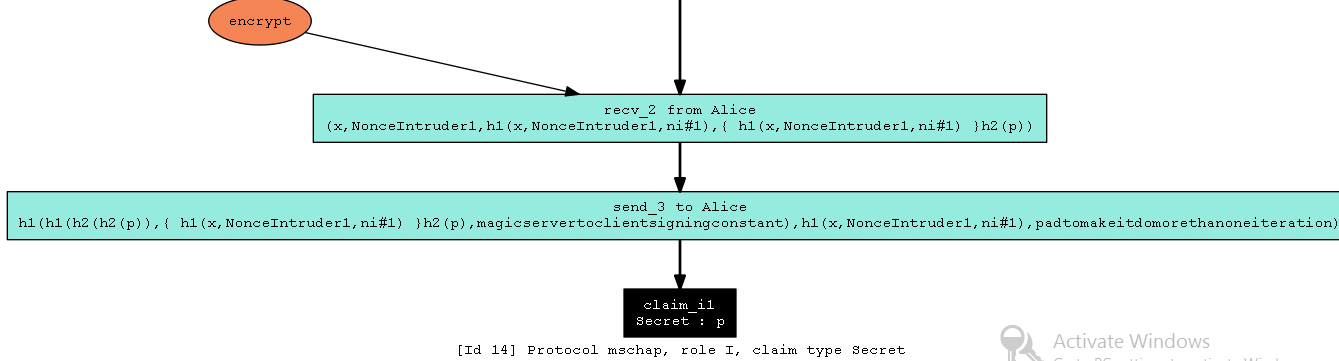






We can see that since the nonce, ni#1 is sent in an unencrypted format, it can be seen and redirected to an attacker. (Sniffing). The reply sent also sends the username and nonce nr#2 in cleartext, and hashing of such variables can easily be done by a third party, making a man-in-the-middle attack a very high possibility, like:





The most important change to be made is encrypting the nonces and the username so that sniffing does not have an impact on the protocol anymore.

Here, we can see that since the authentication success or failed strings are encrypted through the public key, any third person/attacker can send an authentication failed message falsely indicating a security breach closing the port for communication. We can encrypt with the public key for secrecy and then with the secret key for authenticity.

Modified MS-CHAP v2 using asymmetric keys (removing unnecessary encryption and hashing functions):

hashfunction h1;

hashfunction h2;

usertype String;

const p: String;

const x: String;

const authenticationsuccessful: String;

const authenticationfailed: String;

const accesslevel0, accesslevel1: String;

macro m1= h1(x,nr,ni);

macro m2= h2(p);

macro m3= h1(m1, m2);

macro m4= authenticationsuccessful;

macro m5= authenticationfailed;

protocol mschap (I,R) {

role I

{

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I,R,{ni}pk(R));

recv\_2 (R, I,{ x, nr}pk(I), {m1}m2);

claim (I, Running, R, ni, nr);

send\_3 (I,R, m3 );

recv\_4 (R, I, {{m4}pk(I)}sk(R));

send\_5 (I, R, {{accesslevel1}pk(R)}sk(I));

recv\_6 (R, I, {{m5} pk(I)}sk(R));

send\_7 (I, R, {{accesslevel0}pk(R)}sk(I));

claim\_i1 (I, Secret, p);

claim\_i2 (I, Commit, R, ni, nr);

claim\_i3 (I, Niagree);

claim\_i4 (I, Nisynch);

}

role R

{

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I,R,{ni}pk(R));

send\_2 (R, I,{x, nr}pk(I), {m1}m2);

claim (R, Running, I, ni, nr);

recv\_3 (I,R, m3);

match(h1(m1, m2), m3);

send\_4 (R, I, {{m4}pk(I)}sk(R));

recv\_5 (I, R, {{accesslevel1}pk(R)}sk(I));

not match(h1(m1, m2), m3);

send\_6 (R, I,{{m5}pk(I)}sk(R));

recv\_7 (I, R, {{accesslevel0}pk(R)}sk(I));

claim\_r1 (R, Secret, p);

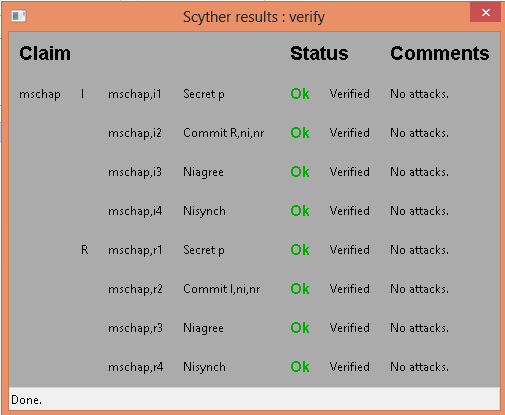
claim\_r2 (R, Commit, I, ni, nr);

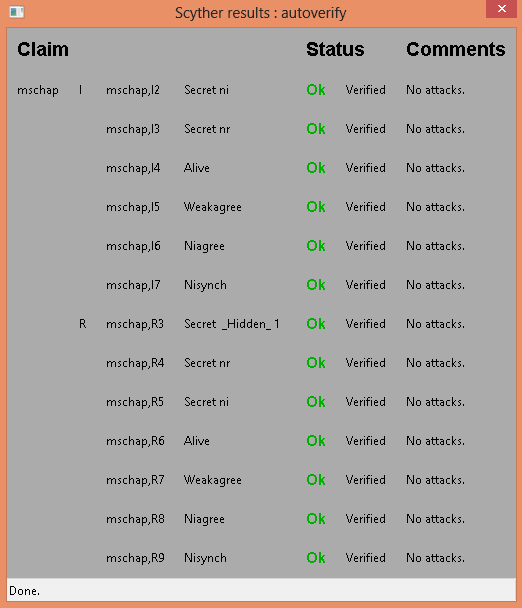
claim\_r3 (R, Niagree);

claim\_r4 (R, Nisynch);

}

}





LEAP

Lightweight Extensible Authentication protocol uses dynamic wired equivalent privacy (WEP) keys that are changed with more frequent authentications between a client and a RADIUS server. The LEAP is an enhanced version of MD-5 with dynamic key rotation and mutual authentication added. However, LEAP’s reliance upon a version of the MS-CHAPv2 (Microsoft Challenge Handshake Authentication Protocol Version 2) protocol means that user credentials are not adequately protected as they do not use a salt(a random string of data that modifies a password hash), use a weak 2 byte DES key and send usernames in clear text. Because of this, offline dictionary and brute force attacks can be made much more efficient by large databases of likely passwords with pre-calculated hashes. Through methods like ASLEAP, an adversary can now crack the vast majority of enterprise wireless LAN’s in a few minutes with virtually zero chance of detection.

Assumptions:

A believes fresh Na  B believes fresh Nb

A believes A  B B believes A  B

Protocol Messages:

A → B: A

B → A: Nb , SID

A → B: H (Nb, H (p)), SID

B → A: String, SID

A → B: Na , SID

B → A: {Na}H(H(p)), (String’, H(H(H(p))), Na, {Na}H(H(p)), Nb, H (Nb, H (p)))

Original LEAP

hashfunction h;

hashfunction h1;

hashfunction h2;

usertype String;

const authenticationfailed: String;

const authenticationsuccessful: String;

const p: String;

const Sharedsecret: String;

const Auth: String;

const SID: String;

macro m1= h1(p);

macro m2= h(nr, m1);

macro m3= authenticationfailed;

macro m4=authenticationsuccessful;

macro m5= h1(h1(p));

macro m6= h2 (m5, ni, {ni}m5, nr, m2);

macro m7= (Sharedsecret, Auth, m6);

protocol leap (I, R)

{

role I

{

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, nr, SID);

send\_3 (I, R, m2, SID);

recv\_4 (R, I, m3, SID);

recv\_5(R, I, m4, SID);

send\_6 (I, R, ni, SID);

recv\_7 (R, I, {ni}m5, m7);

claim\_i1 (I, Secret, m2);

claim\_i2 (I, Secret, m6);

claim\_i3 (I, Niagree);

claim\_i4 (I, Nisynch);

}

role R

{

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, nr, SID);

recv\_3 (I, R, m2, SID);

not match (m2, h(nr, h1(p)));

send\_4 (R, I, m3, SID);

match (m2, h(nr, h1(p)));

send\_5 (R, I, m4, SID);

recv\_6 (I, R, ni, SID);

send\_7 (R, I, {ni}m5, m7);

claim\_r1 (R, Secret, m2);

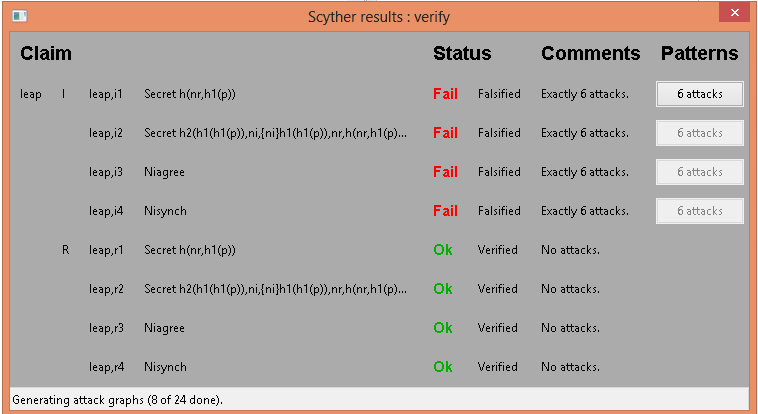
claim\_r2 (R, Secret, m6);

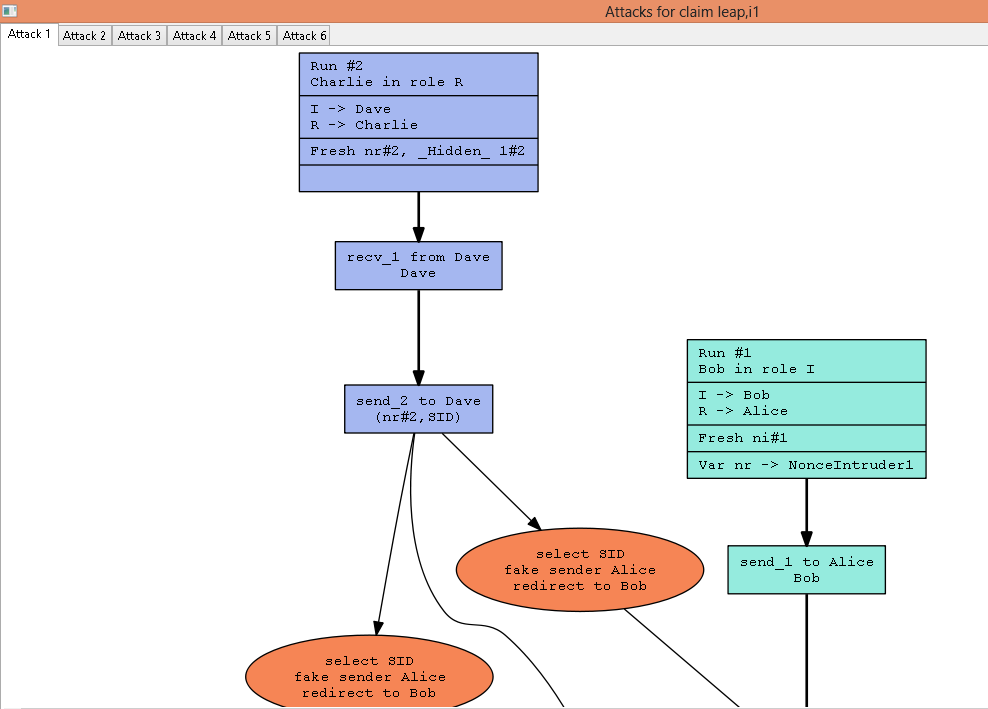
claim\_r3 (R, Niagree);

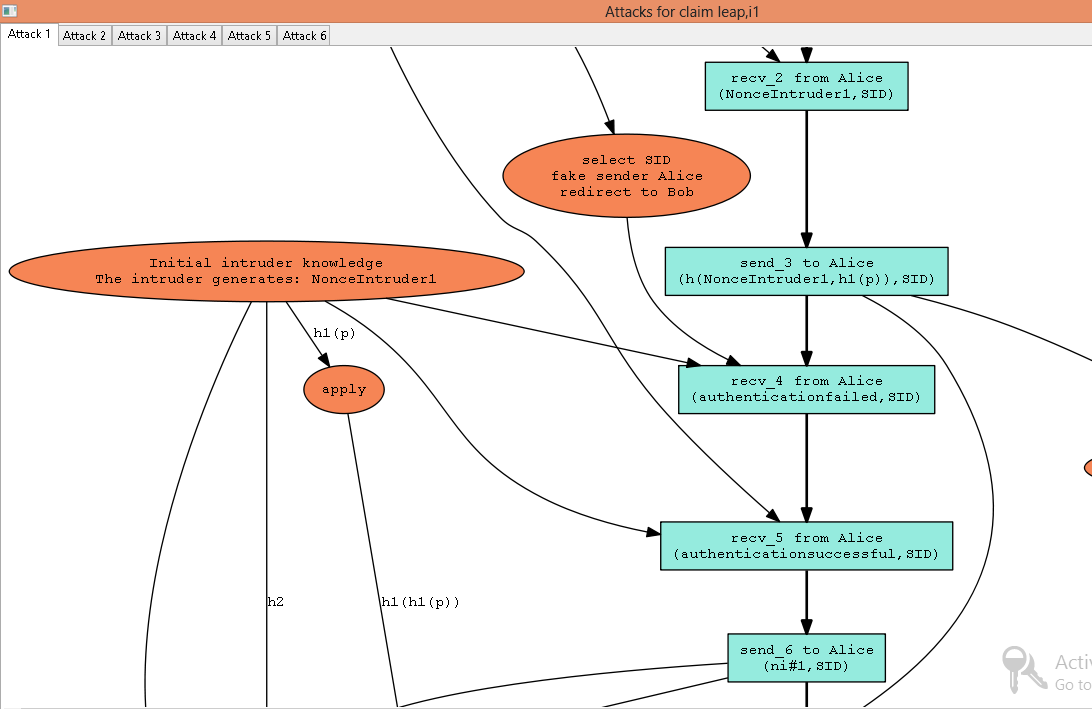
claim\_r4 (R, Nisynch);

}

}







We clearly notice that that the problems are caused mainly due to the nonces and SID being sent in clear text. This can be rectified by using asymmetric keys for encryption.

hashfunction h;

hashfunction h1;

hashfunction h2;

usertype String;

const authenticationfailed: String;

const authenticationsuccessful: String;

const p, Auth, SID : String;

const Sharedsecret: String;

macro m1= h1(p);

macro m2= h(nr, m1);

macro m3= authenticationfailed;

macro m4=authenticationsuccessful;

macro m5= h1(h1(p));

macro m6= h2 (m5, ni, {ni}m5, nr, m2);

macro m7= (Sharedsecret, Auth, m6);

protocol leap (I, R)

{

role I

{

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, {nr, SID}pk(I));

send\_3 (I, R, {m2, SID}pk(R));

recv\_4 (R, I, {{m3, SID}pk(I)}sk(R));

recv\_5(R, I, {{m4, SID}pk(I)}sk(R));

send\_6 (I, R, {ni, SID}pk(R));

recv\_7 (R, I, {{ni}m5, m7}pk(I));

claim\_i1 (I, Secret, m2);

claim\_i2 (I, Secret, m6);

claim\_i3 (I, Niagree);

claim\_i4 (I, Nisynch);

}

role R

{

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, {nr, SID}pk(I));

recv\_3 (I, R, {m2, SID}pk(R));

not match (m2, h(nr, h1(p)));

send\_4 (R, I, {{m3, SID}pk(I)}sk(R));

match (m2, h(nr, h1(p)));

send\_5 (R, I, {{m4, SID}pk(I)}sk(R));

recv\_6 (I, R, {ni, SID}pk(R));

send\_7 (R, I, {{ni}m5, m7}pk(I));

claim\_r1 (R, Secret, m2);

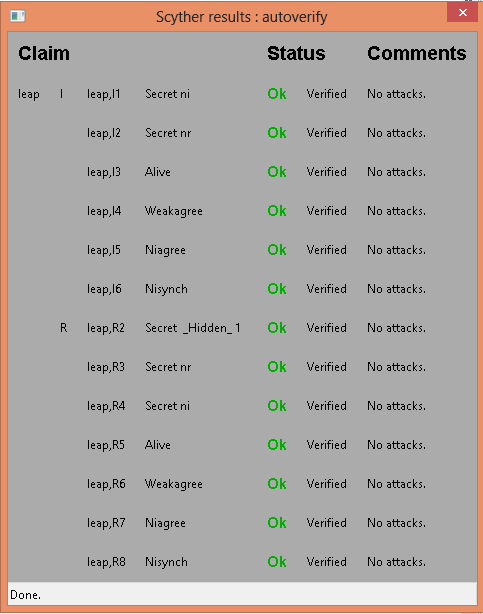
claim\_r2 (R, Secret, m6);

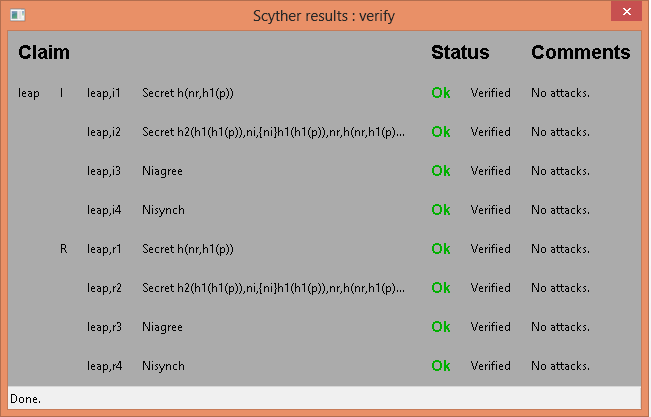
claim\_r3 (R, Niagree);

claim\_r4 (R, Nisynch);

}

}





The encryption of the exchanged messages can also be done through symmetric keys:

hashfunction h;

hashfunction h1;

hashfunction h2;

usertype String;

const authenticationfailed: String;

const authenticationsuccessful: String;

const p: String;

const Sharedsecret: String;

const Auth: String;

const SID: String;

macro m1= h1(p);

macro m2= h(nr, m1,SID);

macro m3= authenticationfailed;

macro m4=authenticationsuccessful;

macro m5= h1(h1(p));

macro m6= h2 (m5, ni, {ni}m5, nr, m2);

macro m7= (Sharedsecret, Auth, m6);

protocol leap (I, R)

{

role I

{

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, {nr, SID}k(I,R));

send\_3 (I, R, m2);

recv\_4 (R, I, {m3, SID}k(I,R));

recv\_5(R, I, {m4, SID}k(I,R));

send\_6 (I, R, {ni, SID}k(I,R));

recv\_7 (R, I, {ni}m5, {m7}k(I,R));

claim\_i1 (I, Secret, m2);

claim\_i2 (I, Secret, m6);

claim\_i3 (I, Niagree);

claim\_i4 (I, Nisynch);

}

role R

{

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, {nr, SID}k(I,R));

recv\_3 (I, R, m2);

not match (m2, h(nr, h1(p)));

send\_4 (R, I, {m3, SID}k(I,R));

match (m2, h(nr, h1(p)));

send\_5 (R, I, {m4, SID}k(I,R));

recv\_6 (I, R, {ni, SID}k(I,R));

send\_7 (R, I, {ni}m5, {m7}k(I,R));

claim\_r1 (R, Secret, m2);

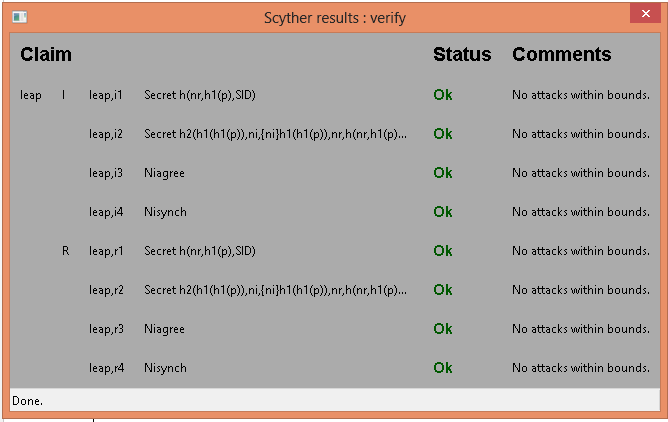
claim\_r2 (R, Secret, m6);

claim\_r3 (R, Niagree);

claim\_r4 (R, Nisynch);

}

}



Modified LEAP (by removing unnecessary encryption and hashing while ensuring the secrecy of exchanged messages):

hashfunction h;

hashfunction h1;

hashfunction h2;

usertype String;

const authenticationfailed: String;

const authenticationsuccessful: String;

const p, SID: String;

const Sharedsecret, Auth: String;

const accesslevel1, accesslevel0: String;

macro m1= h(nr, p);

macro m2= authenticationfailed;

macro m3=authenticationsuccessful;

macro m4= h2 (p, ni, nr, SID);

macro m5= (Sharedsecret, Auth, m4);

protocol leap (I, R)

{

role I

{

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, {nr, SID}pk(I));

send\_3 (I, R, m1);

recv\_4 (R, I, {{m2, SID}pk(I)}sk(R));

recv\_5(R, I, {{m3, SID}pk(I)}sk(R));

send\_6 (I, R, {ni, SID}pk(R));

recv\_7 (R, I, {m5} pk(I));

match ((Sharedsecret, Auth, h2 (p, ni, nr, SID)), m5);

send\_8 (I, R, {accesslevel1}pk (R));

not match ((Sharedsecret, Auth, h2 (p, ni, nr, SID)), m5);

send\_9 (I, R, {accesslevel0}pk (R));

claim\_i1 (I, Secret, m1);

claim\_i2 (I, Secret, m4);

claim\_i3 (I, Niagree);

claim\_i4 (I, Nisynch);

}

role R

{

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, {nr, SID}pk(I));

recv\_3 (I, R, m1);

not match (m1, h(nr, p));

send\_4 (R, I, {{m2, SID}pk(I)}sk(R));

match (m1, h(nr, p));

send\_5 (R, I, {{m3, SID}pk(I)}sk(R));

recv\_6 (I, R, {ni, SID}pk(R));

send\_7 (R, I, {m5}pk(I));

recv\_8 (I, R, {accesslevel1}pk (R));

recv\_9 (I, R, {accesslevel0}pk (R));

claim\_r1 (R, Secret, m1);

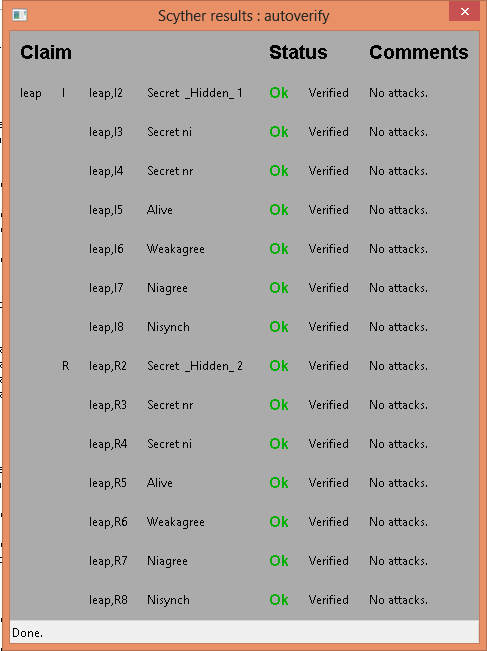
claim\_r2 (R, Secret, m4);

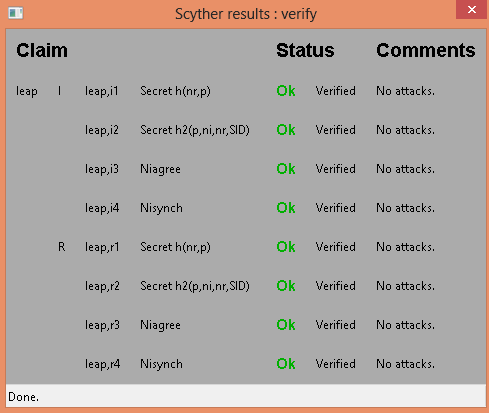
claim\_r3 (R, Niagree);

claim\_r4 (R, Nisynch);

}

}





* 1. **EAP-TLS (The Extensible Authentication mechanism - Transport Layer Security protocol)**

The EAP-TLS protocol, based on the Secure Sockets Layer allows applications to communicate securely through the exchange and verification of certificates. TLS specifies a framework that enables mutual authentication through symmetric or asymmetric encryption, negotiation of the specific encryption algorithm (the cipher suite) and a secured exchange of keys to be used for encrypting messages. After the authenticator and the peer negotiate EAP. The authenticator typically sends an EAP Request/Identity packet to the peer and the peer responds with its user ID in an EAP response/ identity packet. From now on, the conversation occurs between the authentication server and the peer and the authenticator just acts like a pass-through device which encapsulates the packets for transmission. The protocol proceeds as follows:

1. Once the EAP server receives the peer’s identity, it must respond with an EAP-TLS start packet with the start bit set and no data.
2. The peer then sends an EAP Response packet, the data field of which will contain a TLS client\_hello handshake message encapsulating the peer’s TLS version number, a session ID, a random number and a set of ciphersuites supported by the peer.
3. The EAP server responds with an EAP\_Request packet which contains a TLS server\_hello handshake message (with the TLS version number, another random number, as session ID and ciphersuite), TLS server\_certificate (which contains unique serial number of the certificate, name of issuer, validity period of the certificate, certified public key of the server and the private key of the certifying authority signs the certificate), server\_key\_exchange (the public key is sent in the message to compute the premaster secret if it wasn’t sent in the server\_certificate), certificate\_request and a server\_hello\_done message.
4. Once the peer receives the certificate request, unless it is configured for privacy it must send the client certificate, client\_key\_exchange (the client computes a premaster secret using the server\_key\_exchange and then encrypts it with the server’s public key), certificate verify, change\_cipher\_spec and TLS finished.
5. After receiving this packet, the EAP server will verify the peer’s certificate and digital signature and then send the TLS finished message. (After the client\_key\_exchange, we use a random ID in client hello, a random ID in server hello and the premaster secret to compute the master secret).

SID : Session ID

Na : Nonce, random number generated by client.

Nb : Nonce, random number generated by server.

Kca-1 : Private key of the certifying authority

Ka : Public key of the client

Kb : Public key of the server

PMS: Pre-master secret

KMS : Master secret key

Assumptions:

A   Na (A believes fresh Na) B   Nb (B believes fresh Nb)

A   Xa  (A believes fresh SID) B   Xa (B believes fresh SID)

CA  CA (certifying authority believes that Kca is its public key)

A   (Kca-1), A believes CA has a good private key Kca-1

B   (Kca-1), A believes CA has a good private key Kca-1

A   A (A believes Ka is it’s public key)

B   A (B believes Ka is A’s public key)

B  B (B believes Kb is its public key)

A  B (A believes Kb is B’s public key)

A  A  B (A believes PMS is a shared secret between A and B)

B  A B (B believes PMS is a shared secret between A and B)

B  A  B (B believes MS is a shared secret between A and B)

A  A  B (A believes MS is a shared secret between A and B)

A  B   (Kb, B) (B has a good public key Kb)

B  A   (Ka, A) (A has a good public key Ka )

A  , A believes (t1, t2) is a good time interval, i.e. the certificate is valid for this duration.

B , B believes (t1, t2) is a good time interval, i.e. the certificate is valid for this duration.

Goals:

A  A  B B  A  B

Protocol:

Message 1: A→B: SID, Na

Message 2: B→A: SID, Nb, ,  (B, Kb), Kca-1)

Message 3: A→B: ,  (A, Ka), Kca-1), {PMS}Kb, {H(KMS, B)}Kb

Message 4: B→A: {H(PMS, Na, Nb)}KMS

usertype String;

const ccs, csu, shd: String;

hashfunction h, prf;

macro kms= prf(PMS, ni, nr);

macro ikey = h( I, ni, nr, prf (kms));

macro skey= h (R, ni, nr, prf (kms));

protocol eaptls (I,R,CA)

{

role I

{

const SID, CerI, CerR,PMS, S: Data;

fresh ni: Nonce;ms-chap

var nr:Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni, csu);

recv\_6 (R, I, SID, nr, {CerR,pk(R)}sk(CA), shd);

send\_7 (I, CA, I);

recv\_8 (CA, I, {I, {CerI, pk(I)}sk(CA)}sk(CA));

send\_9 (I, R, {CerI, pk(I)}sk(CA), {PMS}pk(R), {h( kms, R)} sk(I),

{h( prf( kms), ni, I, R, SID) } ikey) ;

recv\_10 (R, I, {h( prf (kms), I, R, ni, SID)} skey);

claim\_I1 (I, Secret, CerR);

claim\_I2 (I, Secret, CerI);

claim\_I3 (I, Secret, PMS);

claim\_I4 (I, Secret, kms);

claim\_I5 (I, Nisynch);

claim\_I6 (I, Niagree);

}

role R

{

const SID, CerI, CerR, PMS, S: Data;

fresh nr: Nonce;

var ni:Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR,pk(R)}sk(CA)}sk(CA));

send\_6 (R, I, SID, nr, {CerR,pk(R)}sk(CA), shd);

recv\_9 (I, R, {CerI, pk(I)}sk(CA), {PMS}pk(R), {h( kms, R)} sk(I),

{h( prf( kms), ni, I, R, SID) } ikey);

send\_10 (R, I, {h( prf (kms), I, R, ni, SID)} skey);

claim\_R1 (R, Secret, CerR);

claim\_R2 (R, Secret, CerI);

claim\_R3 (R, Secret, PMS);

claim\_R4 (R, Secret, kms);

claim\_R5 (R, Nisynch);

claim\_R6 (R, Niagree);

}

role CA

{

const CerI, CerR: Data;

recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)}sk(CA));

recv\_7 (I, CA, I);

send\_8 (CA, I, {I, {CerI, pk(I)}sk(CA)}sk(CA));

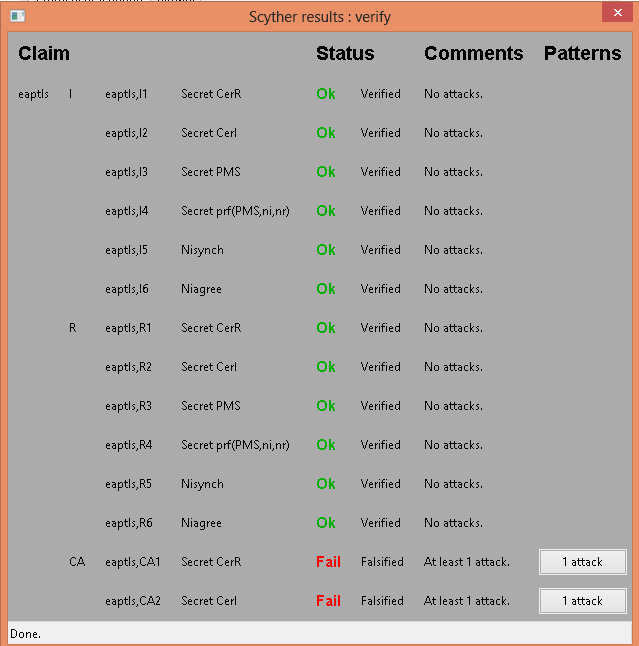
claim\_CA1 (CA, Secret, CerR);

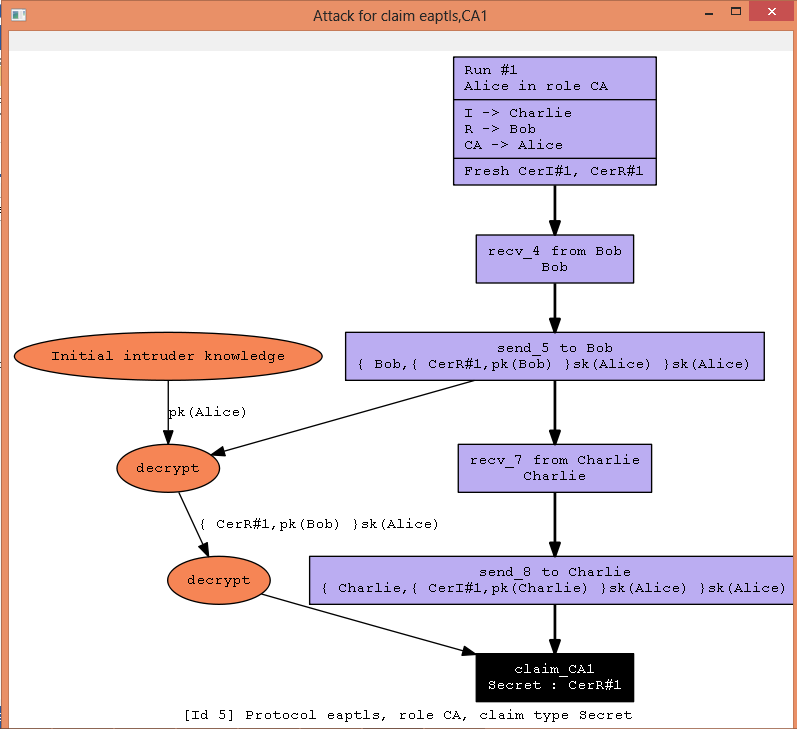
claim\_CA2 (CA, Secret, CerI);

}

}

Scyther Verification





usertype String;

const ccs, csu, shd, accesslevel1: String;

hashfunction h;

hashfunction prf;

macro kms = prf (PMS, ni, nr);

macro ikey = h( I, ni, nr, prf (kms));

macro skey= h (R, ni, nr, prf (kms));

protocol eaptls (I,R,CA)

{

role I

{

const SID, CerI, CerR,PMS, S: Data;

fresh ni: Nonce;

var nr:Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni, csu);

recv\_6 (R, I, SID, nr, csu, {CerR,pk(R)}sk(CA), shd);

send\_7 (I, CA, I);

recv\_8 (CA, I, {I, {CerI, pk(I)}sk(CA)}pk(I));

send\_9 (I, R, {CerI, pk(I)}sk(CA), {PMS}pk(R), ccs, {h ( R, kms)} sk(I), {h( prf (kms), ni, I, R, SID)} ikey);

recv\_10 (R, I, {h (prf ( kms), I, R, ni, SID)} skey, {accesslevel1}kms);

claim\_I1 (I, Secret, CerR);

claim\_I2 (I, Secret, CerI);

claim\_I3 (I, Secret, PMS);

claim\_I4 (I, Secret, kms);

claim\_I5 (I, Nisynch);

claim\_I6 (I, Niagree);

}

role R

{

const SID, CerI, CerR, PMS, S: Data;

fresh nr: Nonce;

var ni:Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR,pk(R)}sk(CA)}pk(R));

send\_6 (R, I, SID, nr, csu, {CerR,pk(R)}sk(CA), shd);

recv\_9 (I, R, {CerI, pk(I)}sk(CA), {PMS}pk(R), ccs, {h ( R, kms)} sk(I), { h ( prf (kms), ni, I, R, SID)} ikey);

send\_10 (R, I, {h (prf ( kms), I, R, ni, SID)} skey, {accesslevel1}kms);

claim\_R1 (R, Secret, CerR);

claim\_R2 (R, Secret, CerI);

claim\_R3 (R, Secret, PMS);

claim\_R4 (R, Secret, kms);

claim\_R5 (R, Nisynch);

claim\_R6 (R, Niagree);

}

role CA

{

const CerI, CerR: Data;

recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)}pk(R));

recv\_7 (I, CA, I);

send\_8 (CA, I, {I, {CerI, pk(I)}sk(CA)}pk(I));

claim\_CA1 (CA, Secret, CerR);

claim\_CA2 (CA, Secret, CerI);

}

}



The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random

values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the

client and server to agree on a premaster secret.

- Exchange certificates and cryptographic information to allow the

client and server to authenticate themselves.

- Generate a master secret from the premaster secret and exchanged

random values.

- Provide security parameters to the record layer.

- Allow the client and server to verify that their peer has

calculatesd the same security parameters and that the handshake

occurred without tampering by an attacker.

Modified EAP-TLS

usertype String;

const ccs, csu, shd, accesslevel1: String;

hashfunction h;

hashfunction prf;

macro kms = prf (PMS, ni, nr);

macro ikey = h( ni, kms);

macro skey= h (nr, kms);

protocol eaptls (I,R,CA)

{

role I

{

const SID, CerI, CerR,PMS, S: Data;

fresh ni: Nonce;

var nr:Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni, csu);

recv\_6 (R, I, SID, nr, csu, {CerR,pk(R)}sk(CA), shd);

send\_7 (I, CA, I);

recv\_8 (CA, I, {I, {CerI, pk(I)}sk(CA)}pk(I));

send\_9 (I, R, {CerI, pk(I)}sk(CA), {PMS}pk(R), ccs, {h ( R, kms)} sk(I), {h(kms, SID)} ikey);

recv\_10 (R, I, {h (kms, SID)} skey, {accesslevel1}kms);

claim\_I1 (I, Secret, CerR);

claim\_I2 (I, Secret, CerI);

claim\_I3 (I, Secret, ikey);

claim\_I4 (I, Secret, skey);

claim\_I5 (I, Nisynch);

claim\_I6 (I, Niagree);

}

role R

{

const SID, CerI, CerR, PMS, S: Data;

fresh nr: Nonce;

var ni:Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR,pk(R)}sk(CA)}pk(R));

send\_6 (R, I, SID, nr, csu, {CerR,pk(R)}sk(CA), shd);

recv\_9 (I, R, {CerI, pk(I)}sk(CA), {PMS}pk(R), ccs, {h ( R, kms)} sk(I), { h ( kms, SID)} ikey);

send\_10 (R, I, {h (kms, SID)} skey, {accesslevel1}kms);

claim\_R1 (R, Secret, CerR);

claim\_R2 (R, Secret, CerI);

claim\_R3 (R, Secret, ikey);

claim\_R4 (R, Secret, skey);

claim\_R5 (R, Nisynch);

claim\_R6 (R, Niagree);

}

role CA

{

const CerI, CerR: Data;

recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)}pk(R));

recv\_7 (I, CA, I);

send\_8 (CA, I, {I, {CerI, pk(I)}sk(CA)}pk(I));

claim\_CA1 (CA, Secret, CerR);

claim\_CA2 (CA, Secret, CerI);

}

}



EAP-TLS is a superior form of EAP and was used as an alternative, it is completely password cracking resistant as it does not rely on user passwords. EAP-TLS relied on digital certificates on both the server and the client end to facilitate mutual authentication and secure key exchange. Unfortunately, the need for public key infrastructure deployment on the server end and the installed user base was a barrier for many organizations. Hence, a new IETF standard called EAP-TTLS (Tunneled Transport Layer security) was proposed to ease the deployment requirements by producing a standard that only required digital certificates on the authentication server end and not on the client end.

Extensible Authentication Protocol – Tunneled Transport Layer Security (EAP-TTLS)

EAP-TTLS is an EAP method that encapsulates a TLS session, consisting of a handshake phase and a data phase. During the handshake phase, the server is authenticated by the client (or client and server are mutually authenticated) using standard TLS procedures and keying material generated in order to create a cryptographically secure tunnel for information exchange in the subsequent data phase. During the data phase, the client is authenticated to the server, using an arbitrary authentication mechanism such as EAP, PAP, CHAP, MS-CHAP or MS-CHAP v2 encapsulated within the secure tunnel. Considering CHAP authentication in the TLS session established.

Challenge Handshake Authentication Protocol (CHAP)

CHAP uses PPP and extends the user authentication functionality provided on Windows networks to remote workstations (by integrating their encryption and hashing algorithms). It is used to periodically verify the identity of the peer using a 3 way handshake. The process of authentication is as follows:

1. After the link establishment phase (exchange of LCP packets), the authenticator sends a “challenge” message to the peer.
2. The peer responds with a value calculated using a “one way hash” function such as MD5.
3. The authenticator checks the response against its own calculation of the expected hash value. If the values match, authentication is acknowledged, otherwise the connection should be terminated.
4. At random intervals, the authenticator sends a new challenge to the peer and repeats the process.

Therefore, EAP allows legacy password based authentication protocols to be used against existing authentication databases, while protecting the security of these legacy protocols against eavesdropping, man-in-the-middle and other attacks. The data phase may also be used for additional arbitrary data exchange.

EAP-TTLS

BAN

Assumptions:

A   Na (A believes fresh Na) B   Nb (B believes fresh Nb)

A   SID (A believes fresh SID) B   SID(B believes fresh SID)

CA  CA (certifying authority believes that Kca is its public key)

A   (Kca-1), A believes CA has a good private key Kca-1

B   (Kca-1), B believes CA has a good private key Kca-1

A   A (A believes Ka is it’s public key)

B   A (B believes Ka is A’s public key)

B  B (B believes Kb is its public key)

A  B (A believes Kb is B’s public key)

A  A  B (A believes PMS is a shared secret between A and B)

B  A B (B believes PMS is a shared secret between A and B)

A  B   (Kb, B) (B has a good public key Kb)

B  A   (Ka, A) (A has a good public key Ka )

A  , A believes (t1, t2) is a good time interval, i.e. the certificate is valid for this duration.

B , B believes (t1, t2) is a good time interval, i.e. the certificate is valid for this duration.

Goals:

A  A  B B  A  B

Protocol messages:

Message 1: A→B: SID, Na.

Message 2: B→A: SID, Nb, ,,  (B, Kb), Kca-1)

Message 3: A→B: {PMS}Kb, {H (H(Na, Nb, PMS), A, B, Na, SID)} H(A, Na, Nb, H(Na, Nb, PMS))

Message 4: B→A: {H(PMS, Na, Nb), A, B, Na, SID} H(A, Na, Nb, H(Na, Nb, PMS))

Message 5: A→B:{Xa, H(H(Na, Nb, PMS), ttlschallenge, Na, Nb, string), H(H(Na, Nb, PMS), ttlschallenge, Na, Nb, string’)} H(A, B, Na, H(PMS, Na, Nb))

hashfunction prf;

hashfunction keygen;

hashfunction h1;

hashfunction h2;

usertype String;

const ccs, csu, success, failure: String;

const ttlschallenge, shd, chapresponse,uname, accesslevel1, accesslevel0: String;

usertype numeric;

const 16,17: numeric;

macro ms= prf (pms, ni, nr);

macro finished= h1(ms, I, R, ni, csu, SID);

macro ikey= keygen (I, ni, nr, ms);

macro skey= keygen (R, ni, nr, ms);

macro chapchallenge= h2 (prf (ms, ttlschallenge,ni,nr), 1, 16);

macro chapid= h2 (prf (ms, ttlschallenge, ni, nr), 17, 17);

protocol eapttls (I, R, CA)

{

role I

{

const SID, CerR, pms, S: Data;

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni, csu);

recv\_6 (R, I, SID, nr, csu, {CerR, pk(R)}sk(CA), shd);

send\_7 (I, R, {pms}pk(R), ccs, {finished}ikey);

recv\_8 (R, I, ccs, {finished}skey);

send\_9 (I, R, {uname, chapchallenge, chapid, chapresponse}ikey);

recv\_10 (R, I, success, accesslevel1);

recv\_11 (R, I, failure, accesslevel0);

claim\_i1 (I, Secret, ni);

claim\_i2 (I, Secret, nr);

claim\_i3 (I, Secret, CerR);

claim\_i4 (I, Secret, ms);

claim\_i5 (I, Secret, ikey);

claim\_i6 (I, Secret, skey);

claim\_i7 (I, Secret, chapid);

claim\_i8 (I, Nisynch);

claim\_i9 (I, Niagree);

}

role R

{

const SID, CerR, pms, S: Data;

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)}pk(R));

send\_6 (R, I, SID, nr, csu, {CerR, pk(R)}sk(CA), shd);

recv\_7 (I, R, {pms}pk(R), ccs, {finished}ikey);

send\_8 (R, I, ccs, {finished}skey);

recv\_9 (I, R, {uname, chapchallenge, chapid, chapresponse}ikey);

send\_10 (R, I, success, accesslevel1);

send\_11 (R, I, failure, accesslevel0);

claim\_r1 (R, Secret, ni);

claim\_r2 (R, Secret, nr);

claim\_r3 (R, Secret, CerR);

claim\_r4 (R, Secret, ms);

claim\_r5 (R, Secret, ikey);

claim\_r6 (R, Secret, skey);

claim\_r7 (R, Secret, chapid);

claim\_r8 (R, Nisynch);

claim\_r9 (R, Niagree);

}

role CA

{

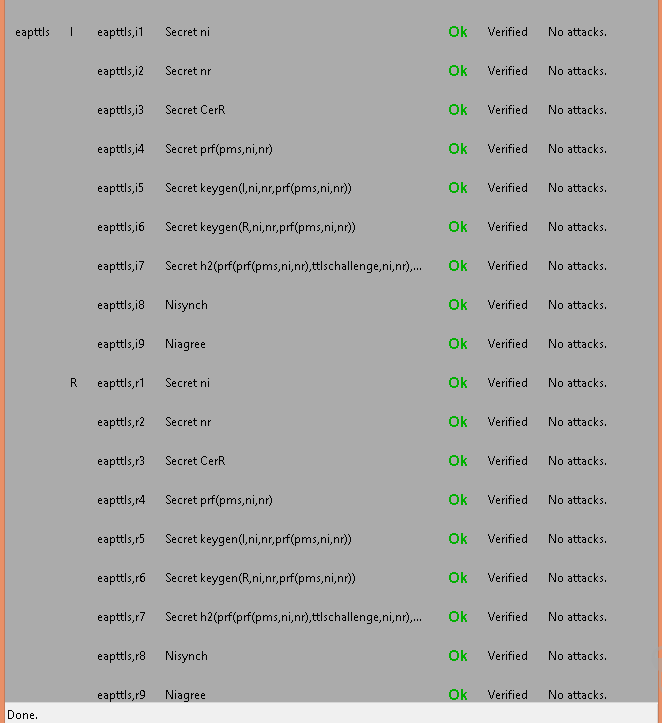
const CerR: Data;

recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)} pk(R));

}

}



Modified EAP-TTLS

hashfunction prf;

hashfunction keygen;

hashfunction h1;

usertype String;

const ccs, csu, success, failure: String;

const ttlschallenge, shd, chapresponse,uname, accesslevel1, accesslevel0: String;

usertype numeric;

const 16,17: numeric;

macro ms= prf (pms, ni, nr);

macro finished= h1(ms, I, R, csu, SID);

macro ikey= keygen (I, ms);

macro skey= keygen (R, ms);

macro chapchallenge= prf (ms, ttlschallenge,ni,nr, 1, 16);

macro chapid= prf (ms, ttlschallenge, ni, nr, 17, 17);

protocol eapttls (I, R, CA)

{

role I

{

const SID, CerR, pms, S: Data;

fresh ni: Nonce;

var nr: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni, csu);

recv\_6 (R, I, SID, nr, csu, {CerR, pk(R)}sk(CA), shd);

send\_7 (I, R, {pms}pk(R), ccs, {finished}ikey);

recv\_8 (R, I, ccs, {finished}skey);

send\_9 (I, R, {uname, chapchallenge, chapid, chapresponse}ikey);

recv\_10 (R, I, success, accesslevel1);

recv\_11 (R, I, failure, accesslevel0);

claim\_i1 (I, Secret, ni);

claim\_i2 (I, Secret, nr);

claim\_i3 (I, Secret, CerR);

claim\_i4 (I, Secret, ms);

claim\_i5 (I, SKR, ikey);

claim\_i6 (I, SKR, skey);

claim\_i7 (I, Secret, chapid);

claim\_i8 (I, Nisynch);

claim\_i9 (I, Niagree);

}

role R

{

const SID, CerR, pms, S: Data;

fresh nr: Nonce;

var ni: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)}pk(R));

send\_6 (R, I, SID, nr, csu, {CerR, pk(R)}sk(CA), shd);

recv\_7 (I, R, {pms}pk(R), ccs, {finished}ikey);

send\_8 (R, I, ccs, {finished}skey);

recv\_9 (I, R, {uname, chapchallenge, chapid, chapresponse}ikey);

send\_10 (R, I, success, accesslevel1);

send\_11 (R, I, failure, accesslevel0);

claim\_r1 (R, Secret, ni);

claim\_r2 (R, Secret, nr);

claim\_r3 (R, Secret, CerR);

claim\_r4 (R, Secret, ms);

claim\_r5 (R, Secret, ikey);

claim\_r6 (R, Secret, skey);

claim\_r7 (R, Secret, chapid);

claim\_r8 (R, Nisynch);

claim\_r9 (R, Niagree);

}

role CA

{

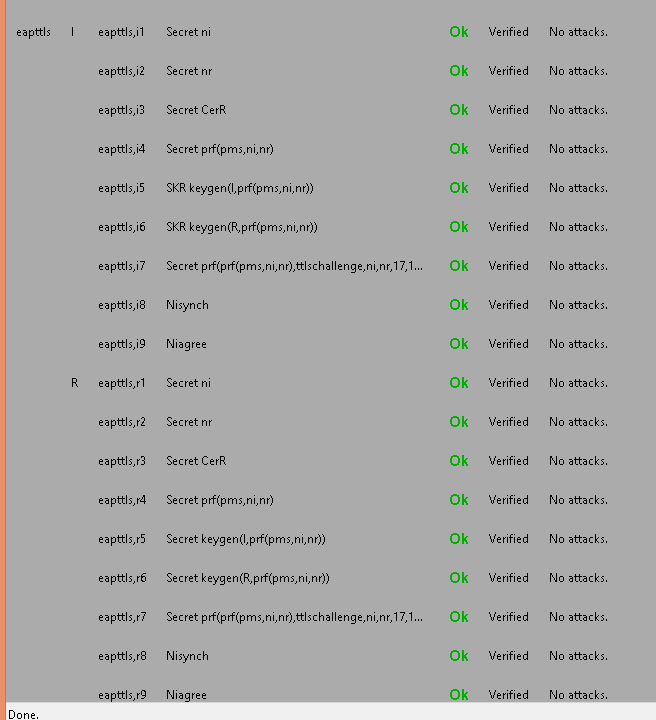
const CerR: Data;

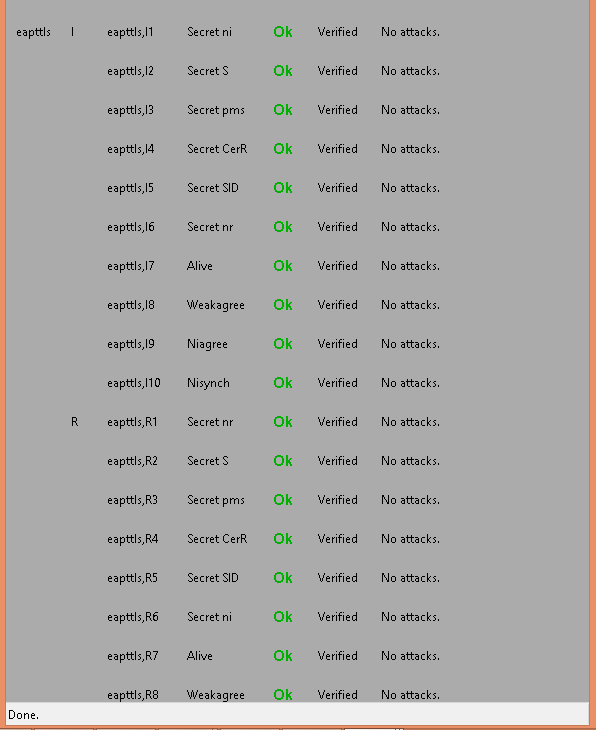
recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)} pk(R));

}

}





Microsoft, Cisco and RSA collaborated and created a light version of EAP-TTLS called protected extensible authentication protocol (PEAP).

PEAP

PEAP does not specify an authentication method, but provides additional security for other EAP authentication protocols, such as EAP-MSCHAP v2, that can operate through the TLS encrypted channel provided by PEAP. To enhance both the EAP protocols and network security, PEAP provides:

* A TLS channel that provides protection for the EAP method negotiation that occurs between client and server. This TLS channel helps prevent an attacker from injecting packets between the client and the network access server to cause the negotiation of a less secure EAP type. The key that is derived during this negotiation is used to encrypt all subsequent communication, including network access authentication that allows the user to connect to the organization network.
* Clients with the ability to authenticate the NPS or other RADIUS server. Because the server also authenticates the client, mutual authentication occurs.
* PEAP fast reconnect, which reduces the delay between an authentication request by a client and the response by the RADIUS server. Fast reconnect also allows wireless clients to move between access points that are configured as RADIUS clients to the same RADIUS server without repeated requests for authentication. This reduces resource requirements for both client and server, and minimizes the number of times that users are prompted for credentials.

BAN

Assumptions:

A   Na1 (A believes fresh Na) B   Nb1 (B believes fresh Nb)

A   Na2 (A believes fresh Na) B   Nb2 (B believes fresh Nb)

A   SID (A believes fresh SID) B   SID(B believes fresh SID)

CA  CA (certifying authority believes that Kca is its public key)

A   (Kca-1), A believes CA has a good private key Kca-1

B   (Kca-1), A believes CA has a good private key Kca-1

A   A (A believes Ka is it’s public key)

B   A (B believes Ka is A’s public key)

B  B (B believes Kb is its public key)

A  B (A believes Kb is B’s public key)

A  A  B (A believes PMS is a shared secret between A and B)

B  A B (B believes PMS is a shared secret between A and B)

A  B   (Kb, B) (B has a good public key Kb)

A , A believes (t1, t2) is a good time interval, i.e. the certificate is valid for this duration.

B , B believes (t1, t2) is a good time interval, i.e. the certificate is valid for this duration.

A A B (A and B share key Kab)

B  A B (A and B share key Kab)

Goals:

A  A  B B  A  B

Protocol Messages:

Message 1: A→B: SID, Na1

Message 2: B→A: SID, Nb1, ,  (B, Kb), Kca-1)

Message 3: A→B: {PMS}Kb

Message 4: B→A: {H(PMS, Na1, Nb1), A, B, Na1,SID}

Message 5: A→B:{Xa} H(A, Na1, Nb1, H (Na1, Nb1, PMS))

Message 6: B→A: {Nb2} H(B, Na1, Nb1, H((PMS, Na1, Nb1))

Message 7: A→B: {Na2, H (Kab, Na2, Nb2, A)} H (A, Na1, Nb1, H (Na1, Nb1, PMS))

Message 8: B→A: {H (Kab, Na2)} H(B, Na1, Nb1, H(PMS, Na1, Nb1))

Message 9: A→B: {ack} H (A, Na1, Nb1, H (Na1, Nb1, PMS))

hashfunction prf;

hashfunction h1;

hashfunction h2;

hashfunction h3;

usertype String;

const ccs, csu, ack, shd : String;

const PEAPsuccess : String;

const accesslevel1 : String;

macro ms= prf (pms, ni1, nr1);

macro ikey= h2 ( I, ni1, nr1, ms);

macro skey= h2 (R, ni1, nr1, ms);

protocol PEAP (I, R, CA)

{

role I

{

const SID, CerR, pms, S: Data;

var S: Data;

fresh ni1, ni2: Nonce;

var nr1, nr2: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni1, csu);

recv\_6 (R, I, SID, nr1, csu, {CerR, pk(R)}sk(CA), shd);

send\_7 (I, R, {pms}pk(R), ccs);

recv\_8 (R, I, ccs, h1 (ms, I, R, ni1, csu, SID));

send\_9 (I, R, {I} ikey);

recv\_10 (R, I, {nr2}skey);

send\_11 (I, R, {ni2, h3 (k(I,R), ni2, nr2, I)}ikey);

recv\_12 (R, I, {h3 (k(I, R), ni2)}skey);

send\_13 (I, R, {ack}ikey);

recv\_14 (R, I, {PEAPsuccess, accesslevel1}skey);

claim\_i1 (I, Secret, CerR);

claim\_i2 (I, Secret, pms);

claim\_i3 (I, Secret, ms);

claim\_i4 (I, Secret, ikey);

claim\_i5 (I, Secret, skey);

claim\_i6 (I, Secret, k(I,R));

claim\_i1 (I, Niagree);

claim\_i2 (I, Nisynch);

}

role R

{

const SID, CerR, pms, S: Data;

fresh nr1, nr2: Nonce;

var ni1, ni2: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni1, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)} sk(CA));

send\_6 (R, I, SID, nr1, csu, {CerR, pk(R)}sk(CA), shd);

recv\_7 (I, R, {pms}pk(R), ccs);

send\_8 (R, I, ccs, h1 (ms, I, R, ni1, csu, SID));

recv\_9 (I, R, {I} ikey);

send\_10 (R, I, {nr2}skey);

recv\_11 (I, R, {ni2, h3 (k(I,R), ni2, nr2, I)}ikey);

send\_12 (R, I, {h3 (k(I, R), ni2)}skey);

recv\_13 (I, R, {ack}ikey);

send\_14 (R, I, {PEAPsuccess, accesslevel1}skey);

claim\_r1 (R, Secret, CerR);

claim\_r2 (R, Secret, pms);

claim\_r3 (R, Secret, ms);

claim\_r4 (R, Secret, ikey);

claim\_r5 (R, Secret, skey);

claim\_r6 (R, Secret, k(I,R));

claim\_r1 (R, Niagree);

claim\_r2 (R, Nisynch);

}

role CA

{

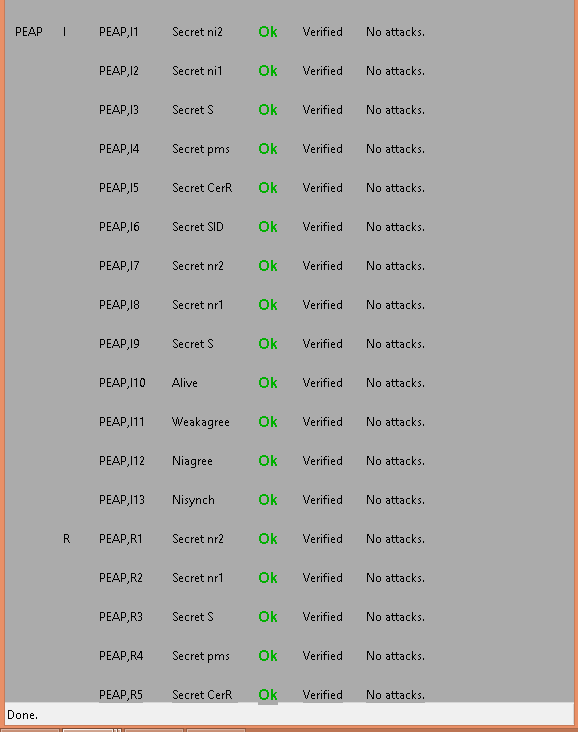
const CerR: Data;

recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)} sk(CA));

} }





Modified PEAP

hashfunction prf;

hashfunction h1;

hashfunction h2;

hashfunction h3;

usertype String;

const ccs, csu, ack, shd, username : String;

const PEAPsuccess : String;

const accesslevel1 : String;

macro ikey= h2 ( I, prf (pms, ni1, nr1));

macro skey= h2 (R, prf (pms, ni1, nr1));

protocol PEAP (I, R, CA)

{

role I

{

const SID, CerR, pms, S: Data;

var S: Data;

fresh ni1, ni2: Nonce;

var nr1, nr2: Nonce;

send\_1 (I, R, I);

recv\_2 (R, I, S);

send\_3 (I, R, SID, ni1, csu);

recv\_6 (R, I, SID, nr1, csu, {CerR, pk(R)}sk(CA), shd);

send\_7 (I, R, {pms}pk(R), ccs);

recv\_8 (R, I, ccs, h1 (prf (pms, ni1, nr1), I, R, csu, SID));

send\_9 (I, R, {username} ikey);

recv\_10 (R, I, {nr2}skey);

send\_11 (I, R, {ni2, h3 (k(I,R), ni2, nr2, I)}ikey);

recv\_12 (R, I, {h3 (k(I, R), ni2)}skey);

send\_13 (I, R, {ack}ikey);

recv\_14 (R, I, {PEAPsuccess, accesslevel1}skey);

claim\_i1 (I, Secret, CerR);

claim\_i2 (I, Secret, pms);

claim\_i4 (I, SKR, ikey);

claim\_i5 (I, SKR, skey);

claim\_i6 (I, Secret, k(I,R));

claim\_i1 (I, Niagree);

claim\_i2 (I, Nisynch);

}

role R

{

const SID, CerR, pms, S: Data;

fresh nr1, nr2: Nonce;

var ni1, ni2: Nonce;

recv\_1 (I, R, I);

send\_2 (R, I, S);

recv\_3 (I, R, SID, ni1, csu);

send\_4 (R, CA, R);

recv\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)} sk(CA));

send\_6 (R, I, SID, nr1, csu, {CerR, pk(R)}sk(CA), shd);

recv\_7 (I, R, {pms}pk(R), ccs);

send\_8 (R, I, ccs, h1 (prf (pms, ni1, nr1), I, R, csu, SID));

recv\_9 (I, R, {username} ikey);

send\_10 (R, I, {nr2}skey);

recv\_11 (I, R, {ni2, h3 (k(I,R), ni2, nr2, I)}ikey);

send\_12 (R, I, {h3 (k(I, R), ni2)}skey);

recv\_13 (I, R, {ack}ikey);

send\_14 (R, I, {PEAPsuccess, accesslevel1}skey);

claim\_r1 (R, Secret, CerR);

claim\_r2 (R, Secret, pms);

claim\_r4 (R, SKR, ikey);

claim\_r5 (R, SKR, skey);

claim\_r6 (R, Secret, k(I,R));

claim\_r1 (R, Niagree);

claim\_r2 (R, Nisynch);

}

role CA

{

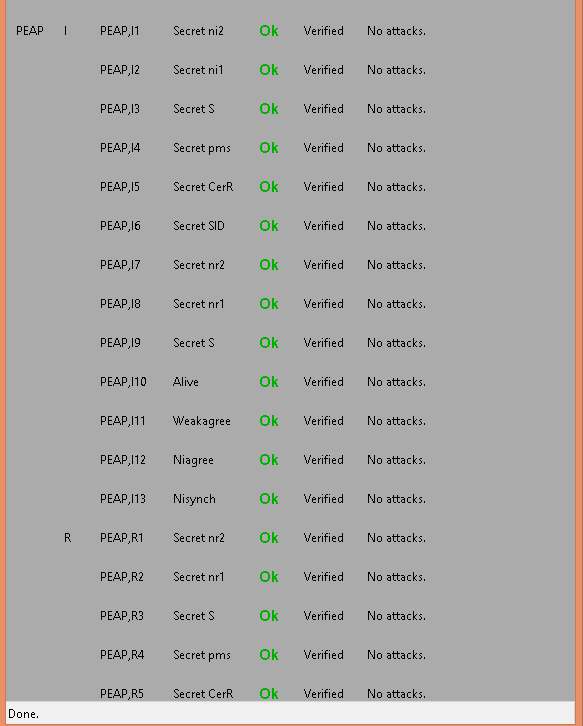
const CerR: Data;

recv\_4 (R, CA, R);

send\_5 (CA, R, {R, {CerR, pk(R)}sk(CA)} sk(CA));

} }





**References:**

1. Tooska Dargahi, Alberto Caponi, Moreno Ambrosin, Giuseppe Bianchi and Mauro Conti, “A Survey on the security of stateful SDN data planes,” IEEE Communication Surveys and Tutorials, Vol. PP, Issue 99, 2017.
2. S. Scott-Hayward, S. Natarajan, and S. Sezer, “A survey of security in software defined networks,” IEEE Communications Surveys & Tutorials, 2016.
3. Jakob Spooner, Dr. Shao Ying Zhu, “A review of solutions for SDN-exclusive security issues, International Journal of Advanced Computer Science and Applications,Vol.7, No.8 2016.
4. Jaouad Benabbou, Noureddine Idboufker, Khalid Elbaamrani, Raja Elassali, “Software-Defined Networks, Security Aspects Analysis,” 11th International Conference on Information Assurance and Security, 2015, pages 79-84.

**DoS**

1. Bin Yuan, Deqing Zou, Shui Yu, Hai Jin, Weizhong Qiang and Jinan Shen, “Defending against Flow Table Overloading Attack in Software-Defined Networks,” IEEE Transactions on Services Computing, Volume: PP, Issue 99, 2016.
2. Haopei Wang, Lei Xu and Guofei Gu, “FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks,” 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2015, pages 239-250
3. Peng Zhang, Huanzhao Wang, Chengchen Hu and Chuang Lin, “On Denial of Service Attacks in Software Defined Networks”, Network forensics and surveillance for emerging networks, Volume 30, Issue 6, 2016, pages 28-33.
4. Ali Hussein, Imad H. Elhajj, Ali Chehab and Ayman Kayssi, “SDN Security Plane: An Architecture for Resilient Security Services,” IEEE International Conference on Cloud Engineering Workshops, 2016, pages 54-59.
5. Lobna Dridi and Mohamed Faten Zhani, “SDN-Guard: DoS Attacks Mitigation in SDN Networks,” 5th IEEE International Conference on Cloud Networking, 2016, pages 212-217.
6. Tilman Wolf and Jingrui Li ,“Denial-of-Service Prevention for Software-Defined Network Controllers,” 25th International Conference on Computer Communication and Networks, 2016, pages 1-10.
7. Heng Cui, Ghassan O. Karame, Felix Klaedtke, and Roberto Bifulco, “On the Fingerprinting of Software-Defined Networks,” IEEE Transactions on Information forensics and security, vol.11, No.10, 2016, pages 2160-2173.
8. A. Wang *et al.*, “Scotch: Elastically Scaling Up SDN Control- Plane Using Vswitch Based Overlay,” Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, 2014, pages 403-414.
9. “Floodlight Project,” <http://www.projectfloodlight.org/floodlight/>.
10. D. K. Yau *et al.*, “Defending Against Distributed Denial-of-Service Attacks with Max-Min Fair Server-Centric Router throttles,” IEEE/ACM Transactions on Networking, vol. 13, Issue 1, 2005, pages 29–42.
11. S. Shin *et al.*, “AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks,” Proceedings of the 2103 ACM SIGSAC conference on Computer and Communication Security, 2013, pages 413-424.
12. Liron Schiff, Stefan Schmid2, “Study the past if you would define the future: Implementing Secure Multi-Party SDN Updates,” IEEE International Conference on Software Science, Technology and Engineering, 2016, pages 111-116.
13. Moreno Ambrosin, Mauro Conti, Fabio De Gaspari and Radha Poovendran, “LineSwitch: Tackling Control Plane Saturation Attacks in Software-Defined Networking,” IEEE/ACM transactions on networking, Vol. PP, Issue 99, 2016, pages 1-14.
14. Trung V. Phan, Truong Van Toan, Dang Van Tuyen, Truong Thu Huong and Nguyen Huu Thanh, “OpenFlowSIA: An Optimized Protection Scheme for Software-Defined Networks from Flooding Attacks,” IEEE Sixth International Conference on Communications and Electronics, 2016, pages 13-18.
15. Ying Qian, Wanqing You and Kai Qian, “OpenFlow Flow Table Overflow Attacks and Countermeasures,” European Conference on Networks and Communications, 2016, pages 205-209.
16. Amir Shoeb and Dr T. Chithralekha,  **“**Resource Management of Switches and Controller During Saturation Time to Avoid DDoS in SDN,”2nd IEEE International Conference on Engineering and Technology (ICETECH), 2016, pages 152-157.
17. Ricardo Macedo, Rafael de Castro, Aldri Santos, Yacine Ghamri-Doudane and Michele Nogueira, “Self-Organized SDN Controller Cluster Conformations Against DDoS Attacks Effects”, IEEE Global Communications Conference (GLOBECOM), 2016, pages 1-6.
18. Liancheng Zhang, Qiang Wei, Kejun Gu, Huiqiang Yuwen, “Path Hopping Based SDN Network Defense Technology”, 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery, 2016, pages 2058-2063.
19. Osamah Ibrahiem Abdullaziz, Yua-Jia Chen, Li-Chun Wang, “Lightweight Authentication Mechanism for Software Defined Network using Information Hiding,” IEEE Global Communications Conference, 2016, pages 1-6.

**Port Knocking Attack**

1. Liancheng Zhang, Yi Guo, Huiqiang Yuwen and Yu Wang, “A Port Hopping Based DoS Mitigation Scheme in SDN Network,”12th International Conference on Computational Intelligence and Security, 2016, pages 314-317.
2. Juan Wang, Yong Wang, Hongxin Hu, Qingxin Sun, He Shi and Longjie Zeng, “Towards a security enhanced Firewall application for OpenFlow Networks”, Cyberspace Safety and Security, 5th International Symposium, Vol. 8300, 2013, pages 92-103.
3. Vikas Srivastava, Alok Kumar Keshri, Abhishek Dutta Roy, Vijay Kumar Chaurasiya and Rahul Gupta, “Advanced Port Knocking Authentication Scheme with QRC using AES”, International Conference on Emerging Trends in Networks and Computer Communications, 2011, pages 159-163.
4. Mariam Khader, Ali Hadi and Amjad Hudaib, “Covert Communication Using Port Knocking,” Cybersecurity and Cyberforensics Conference, 2016, pages 22-27.
5. Fakariah Hani Mohd Ali, Rozita Yunos and Mohd Azuan Mohamad Alias, “Simple Port Knocking Method Against TCP Replay Attack and Port Scanning”, International Conference on Cyber Security, Cyber Warfare and Digital Forensic, 2012, pages 247-252.
6. Mehran Pourvahab, Reza Ebrahimi Atani and Laleh Boroumand, “SPKT: Secure Port Knock-Tunneling, an Enhanced Port Security Authentication Mechanism,” IEEE Symposium on Computers & Informatics, 2012, pages 145-149.

**Rerouting attacks/ fast failure recovery**

1. Shujuan Zhang, Ying Wang, Qichao He, Jinke Yu, Shaoyong Guo, “Backup-Resource Based Failure Recovery Approach in SDN Data Plane,” 18th Asia-Pacific Network Operations and Management Symposium, 2016, pages 1-6.
2. Pankaj Thorat, Rajesh Challa, S. M. Raza, Dongsoo S. Kim and Hyunseung Choo, “Proactive Failure Recovery Scheme for Data Traffic in Software Defined Networks”, IEEE NetSoft Conference and Workshops, 2016, pages 219-225.
3. Hengtong Li, Qing Li, Yong Jiang, Ting Zhang and Lei Wang, “A Declarative Failure Recovery System in Software Defined Networks,” IEEE International Conference on Communications, QoS, Reliability and Modeling Symposium, 2016, pages 1-6.
4. Hung-Chuan Wei, Yung-Hao Tung and Chia-Mu Yu, “Counteracting UDP Flooding Attacks in SDN”, IEEE NetSoft Conference and Workshops, 2016, pages 367-371.
5. Laura Mutu, Rania Saleh and Ashraf Matrawy, “Improved SDN Responsiveness to UDP Flood Attack,” IEEE Conference on Communications and Network Security, 2015, pages 715-716.
6. Thuy Vinh Tran and Heejune Ahn, “A Network Topology-aware Selectively Distributed Firewall Control in SDN”, International Conference on Information and Communication Technology, 2015, pages 89-94.
7. Selcuk Cevher, Mustafa Ulutas, Sedat Altun, Ibrahim Hokelek, “Multi Topology Routing Based IP Fast Re-Route for Software Defined Networks,” IEEE Symposium on Computers and Communication (ISCC), 2016, pages 1224-1226.
8. Jue Chen, Jinbang Chen, Junchen Ling, Wei Zhang, “Failure recovery using VLAN tag in SDN: High speed with low memory requirement,” IEEE 35th International Conference on Performance Computing and Communications, 2016, pages 1-9.
9. Behrouz Raeisi, Alessio Giorgetti, “Software-based fast failure recovery in load balanced SDN-based datacenter networks”, 6th International Conference on Information Communication and management, 2016, pages 95-99.
10. Capone A, Cascone C, Nguyen AQT, Sanso B, “Detour Planning for fast and reliable failure recovery in SDN with Open State,” 11th International Conference on the Design of Reliable Communication Networks, 2015, pages 25-32.

**Others**

1. Chunhui Pang, Yong Jiang, Qi Li, “FADE: Detecting Forwarding Anomaly in Software-Defined Networks,” IEEE International Conference on Communications, 2016, pages 1-6.
2. Tri-Hai Nguyen, Myungsik Yoo,“Attacks on Host Tracker in SDN Controller: Investigation and Prevention”, International Conference on Information and Communication Technology, 2016, pages 610-612.
3. Sean Peisert, Adrian R. Chavez and William M.S. Stout, “Techniques for the Dynamic Randomization of Network Attributes,” 49th International Carnahan Conference on Security Technology, 2015, pages 1-6.
4. Anja Feldmann, Philipp Heyder, Michael Kreutzer2, Stefan Schmid, Jean-Pierre Seifert, Haya Shulman, Kashyap Thimmaraju, Michael Waidner, Jens Sieberg, “NetCo: Reliable Routing With Unreliable Routers,” 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, 2016, pages 128-135.
5. Stefano Vissicchio, Luca Cittadini, “FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates,” IEEE INFOCOM The 35th Annual IEEE International Conference on Computer Communications, 2016, pages 1-9.
6. Seugwon Shin1, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu1, Mabry Tyson. “FRESCO: Modular Composable Security Services for Software-Defined Networks”, In Proceedings of Annual Network and Distributed System Security Symposium (NDSS), 2013.
7. Diogo Menezes Ferrazani Mattos, Otto Carlos Muniz Bandeira Duarte, “AuthFlow: Authentication and access control mechanism for Software Defined Networking”, Article in anals of Communication, 2016
8. P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, “A security enforcement kernel for OpenFlow networks,” Proceedings of the first workshop on Hot topics in software defined networks. ACM, 2012, pp. 121–126.
9. S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang, “Rosemary: A Robust, Secure, and HighPerformance Network Operating System,” Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014, pp. 78–89.
10. B. Chandrasekaran and T. Benson, “Tolerating SDN application failures with LegoSDN,” Proceedings of the 13th ACM Workshop on Hot Topics in Networks. ACM, 2014.
11. O. O. MM and K. Okamura, “Securing Distributed Control of Software Defined Networks,” International Journal of Computer Science & Network Security, vol. 13, no. 9, 2013.
12. Akshay Baheti, “Extensible Authentication Protocol Vulnerabilities and Improvements”, Master’s Thesis and Graduate Research, San Jose State University.

**BAN Logic**

1. Michael Burrows, Martin Abadi and Roger Needham, “A logic of authentication”, ACM Transactions on Computer Systems, Vol. 8, No. 1, 1990.
2. Kening Liu, Junyao Ye, Yinglian Wang, “The security Analysis on Otway-Rees Protocol Based on BAN logic”, Fourth International Conference on Computational and Information Sciences, 2012., pages 341-344.
3. Jan Wessels, “Applications of BAN logic”, 2001.
4. Li Tingyua, Liu Xiaodong, Qin Zhiguang, Zhang Xuanfang, “An improved security protocol formal analysis with BAN logic”, International Conference on Electronic Commerce and Business Intelligence, 2009, pages 102-105.
5. Ron Rivest, Lecture on Kerberos, BAN logic proof of Kerberos, BAN logic for proving the security of protocols, Computer and Network Security, 1997.
6. Bill Young, Lecture on “Foundations of Computer Security: The BAN logic”, Department of Computer Science, University of Texas at Austin.
7. Ruikang Zhou, Yingxu Lai, Zenghui Liu, Yinong Chen, Xiangzhen Yao and Jiezhong Gong, “A security authentication protocol for trusted domains in an autonomous decentralized system”, International Journal of Distributed Sensor Networks, Article No. 4,2016.
8. Kai Fan, Hui Li, Yue Wang, “Security Analysis of the Kerberos protocol using BAN logic”, Fifth International Conference on Information Assurance and security, Volume 2, 2009, pages 467-470.
9. L Acton, University of Cambridge Computer Library, Security Engineering: A guide to building Dependable Distributed Systems.
10. Christian Haack, Verification of Security Protocols: BAN logic, 2008.
11. <http://www.techrepublic.com/article/ultimate-wireless-security-guide-an-introduction-to-leap-authentication/>
12. https://technet.microsoft.com/en-us/library
13. <https://tools.ietf.org/html/rfc2759> (MS-CHAP v2)
14. <https://tools.ietf.org/html/rfc5216> (EAP-TLS)
15. <https://tools.ietf.org/html/rfc1994> (PPP-CHAP)
16. <https://tools.ietf.org/html/rfc5281> (EAP-TTLS)

**Scyther**

1. Cas Cremers and Sjouke Mauw. “Operational Semantics and Verification of Security Protocols”, Information Security and Cryptography, 2012.
2. Cas Cremers, “Scyther User Manual”, 2014.