Flowrider: Fast On-Demand Key Provisioning for Cloud Networks

Paladi, Nicolae; Tiloca, Marco; Nikbakht Bideh, Pegah; Hell, Martin

Published in:
SecureComm 2021 - Proceedings of the 2021 EAI International Conference on Security and Privacy in Communication Networks

2021

Document Version:
Peer reviewed version (aka post-print)

Link to publication

Citation for published version (APA):
Paladi, N., Tiloca, M., Nikbakht Bideh, P., & Hell, M. (2021). Flowrider: Fast On-Demand Key Provisioning for Cloud Networks. In *SecureComm 2021 - Proceedings of the 2021 EAI International Conference on Security and Privacy in Communication Networks* EAI.

Total number of authors:
4

Flowrider: Fast On-Demand Key Provisioning for Cloud Networks

Nicolae Paladi[1,2], Marco Tiloca[2], Pegah Nikbakht Bideh[1], and Martin Hell[1]

[1] Lund University, Lund, Sweden
{first.last}@eit.lth.se
[2] RISE Research Institutes of Sweden - RISE Cybersecurity
{first.last}@ri.se

Abstract. Increasingly fine-grained cloud billing creates incentives to review the software execution footprint in virtual environments. For example, virtual execution environments move towards lower overhead: from virtual machines to containers, unikernels, and serverless cloud computing. However, the execution footprint of security components in virtualized environments has either remained the same or even increased. We present Flowrider, a novel key provisioning mechanism for cloud networks that unlocks scalable use of symmetric keys and significantly reduces the related computational load on network endpoints. We describe the application of Flowrider to common transport security protocols, the results of its formal verification, and its prototype implementation. Our evaluation shows that Florwider uses up to an order of magnitude less CPU to establish a TLS session while preventing by construction some known attacks.

Keywords: Network Security, Software Defined Networking, Secure Communication, Key Management, Cloud Security

1 Introduction

Throughout the past decade, cloud computing has evolved to support a panoply of orchestration, deployment, and billing approaches. Notable trends are the use of resource description templates [11], emergence of serverless computing [24] and fine-grained resource billing [29,53]. Resource description templates allow to dynamically deploy workloads and provision them with cryptographic material or network and application configuration. Most major cloud providers offer serverless computing[3]. This defers the operation of the server platform to the cloud provider, while allowing developers to focus on the application code. Cloud users are billed for the number of function invocations and consumed computation resources, rather than for a pre-purchased unit of computation such as a bare-metal server or a virtual machine. Finally, serverless plans are billed based on the CPU, memory, and I/O operations that functions consume. Fine-grained billing provides strong incentives to develop and deploy applications that utilize a minimum amount of computing resources. This calls for a rigorous review of software development and deployment approaches to reduce the use of computing resources.

[3] See Amazon Lambda, Google Cloud Functions, Azure Functions, Salesforce Evergreen, etc.

Consider software-defined networking (SDN): separation of control and data planes helps network configuration and management; however, network operations security did not keep up with new capabilities enabled by SDN. So far, the distribution of cryptographic material to network endpoints leverage to a limited extent the logical centralization of network control [50]. As a result, public-key cryptography, rather than symmetric key cryptography, remains almost pervasively the tool of choice for enabling secure network traffic in virtualized deployments regardless of the network architecture. While public-key cryptography is robust and scalable, it introduces key management complexity and is relatively CPU-expensive; with fine-grained billing in place, this directly translates into additional financial costs. On virtualized hosts where tenants share a common entropy pool, generating asymmetric keys may slow down applications if sufficient entropy is not available [25]. Some network endpoints may even lack the computational capacity to generate cryptographic material without disrupting their own operations. Generating keys on a dedicated host with deep entropy pools can reduce the key creation overhead. On the other hand, while generating symmetric keys requires less computational power and has firmware support on many platforms, the use of symmetric keys leads to challenges such as secure key provisioning and key authentication. This introduces the research question: *can the SDN model be leveraged to conveniently provision symmetric keys and reduce computational resource consumption?*

We posit that the answer is *yes* and demonstrate this with Flowrider, a novel key provisioning mechanism for network endpoints in SDN deployments that considers the practicalities of cloud systems deployment. In particular, Flowrider takes a reactive, on-demand, and automatic approach that embeds key distribution into the network flow establishment. Furthermore, Flowrider makes key distribution agnostic of the network topology and communication patterns in the system, of which it does not require any early knowledge. Overall, Flowrider reduces the number of steps for providing symmetric key material to endpoints and the time required to set up secure communication.

By conveniently enabling the use of symmetric keys [46], Flowrider reduces by an order of magnitude the computation load of secure channel establishment on network endpoints and simplifies key management in SDN deployments [38], without compromising communication security properties. Minor modifications of network endpoints introduce another contribution - *flow-specific symmetric keys* - that enable per-flow cryptographic isolation of network traffic. Flowrider is compatible with common transport layer security protocol suites including (D)TLS v1.2 [12,40] and v1.3 [18,17]. Our contribution is three-fold:

- We describe a key provisioning mechanism that leverages the use of symmetric encryption keys in virtualized deployments within an administrative domain.
- We describe the mechanism and functioning of *flow-specific symmetric keys*, for establishing secure channels between network endpoints.
- We detail how the proposed mechanism works in the (D)TLS security suites v1.2 and v1.3, where it also prevents the “Selfie” attack [13] by construction.

Our Flowrider implementation shows reduced computation effort and fewer round-trips to generate authentication credentials and establish secure communication between endpoints. Note that Flowrider primarily targets controlled enterprise environments and does not focus on privacy for network endpoints.

The rest of this paper is organized as follows. We introduce the necessary background in Section 2 and describe the system model, threat model, and assumptions in Section 3. In Section 4, we introduce the Flowrider key provisioning mechanism. In Section 5, we describe the use of Flowrider with (D)TLS. In Section 6, we provide a formal security analysis of Flowrider with ProVerif, followed by an experimental evaluation in Section 7. We review the related work in Section 8 and conclude in Section 9.

2 Background

We first introduce the main concepts and context considered in the rest of the paper.

2.1 Deployment in Virtualized Environments

In modern distributed systems, workloads are commonly deployed using a resource orchestration system such as Kubernetes [14], Micado [28] or Rancher[4]. Workloads are deployed based on a resource description expressed in a template encoded in a domain-specific language such as TOSCA [49]. Based on the deployment template, an orchestrator creates and configures workload environments (virtual machine images, containers, or microservices), and deploys them on the underlying hardware. The orchestrator also deploys network components - such as the network controller and network functions - and implements a network configuration defining the communication topology between workload containers. Depending on operating considerations, the orchestrator may be co-located with the network controller. Orchestrators commonly maintain a control channel to patch and update the workloads, re-provision cryptographic material, and collect operation logs. Finally, deployments can be dynamically reconfigured depending on the availability of resources (such as memory, CPU, IO, and bandwidth).

2.2 SDN and OpenFlow

SDN emerged in response to the increasing complexity of network deployments, facilitating operation and management of virtualized networks [1]. Its operational advantages lead to wide adoption in enterprise deployments [42]. We next introduce several relevant components of the SDN model.

The *data plane* contains hardware and software routing components and implements routing policies that satisfy network administrator goals. It is optimized for forwarding speed but may contain logic for in-network processing [39]. The *Southbound API* is a vendor-agnostic set of instructions implemented by the data plane, allowing two-way communication between the data and the control planes. In this paper, we consider the OpenFlow protocol [36]. The *control plane* is an abstraction layer transforming high-level network operator goals into discrete routing policies based on a global network view. *Network functions* are used by network administrators to express their network configuration goals using a set of high-level commands. Examples of such applications are firewalls, intrusion detection systems, traffic shapers, etc. In this paper, we use a custom network function to generate symmetric keys for establishing secure channels between network endpoints.

[4] https://rancher.com/

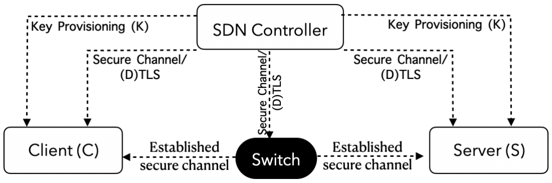

Fig. 1: High-level system architecture and components

2.3 Secure Channels

To establish a secure channel, two parties authenticate with each other and derive key material to protect their communication. To this end, two fundamental approaches exist.

The first approach uses a symmetric *pre-shared key* held by both parties. Advantages of this approach include the typically small size of keys, computationally efficient operations needed to use those keys, and resilience against cryptanalysis using quantum-based algorithms. On the other hand, pre-shared keys are typically more difficult to manage, requiring dedicated management procedures to provide, distribute, and revoke them. Management tasks become especially complicated in large and dynamic systems.

The second approach is based on public-key cryptography, where each party acquires the other's public share of a key pair. In practice, this is a bare *raw public key*, or a public *certificate* including the public key and signed by a trusted certification authority. This approach is widely used: since only public information is shared, management tasks are simpler compared to pre-shared keys and can be automated through dedicated Public Key Infrastructure. On the other hand, this approach results in much larger key material, heavier computation load when performing cryptographic operations, and higher entropy requirements on the communication parties.

3 Network Scenario

Consider the network scenario illustrated in Figure 1: an orchestration node collocated on a network controller deploys two endpoints, i.e. C as Client and S as Server, as well as an OpenFlow Switch on the communication path between C and S. Also, it configures the network controller to establish and manage the network flows between the endpoints. For monitoring and patch management purposes, the orchestrator node establishes at deployment time and maintains a secure channel with the endpoints. Note that this approach is in-line with the industry best-practice recommendations [20].

We assume that the network controller established at deployment time three secure communication channels: with C, with S, and with the Switch. These can practically be enforced through (D)TLS sessions. The Switch is able to forward network traffic between C and S, according to the established flows.

For simplicity and with no loss of generality, we hereafter focus on the scenario in Figure 1. Nevertheless, the solution presented in this paper seamlessly works also in more complex and scalable scenarios, where multiple switches, as well as multiple pairs of client and server peers, are deployed. We assume that the network deployment follows best practices in terms of capacity for network flows, flow establishment rate, and the number of peers engaged in acceptable traffic shapes. This includes proper allocation of bandwidth resources and a sufficient number of deployed switches to prevent bottleneck points and congestion.

With reference to Figure 1, C intends to securely communicate with S. As discussed in Section 2.3, typical approaches to establish a secure communication channel rely on either: a symmetric key pre-shared between C and S; or asymmetric key material either pre-provisioned at orchestration time, exchanged during the secure channel establishment, or acquired out-of-band, such as through a custom PKI infrastructure. We argue that, currently, the above approaches display at least the following limitations.

First, if C and S use multiple network flows, communications on each network flow occur over secure channels created with the same pairwise set of key material. Thus, compromising the single set of key material leads to endangering the data security on all network flows between the two endpoints.

Second, asymmetric key material, e.g. raw public keys and public certificates require computationally- and resource-demanding operations on the endpoints. This becomes critical in virtualized environments and serverless model with fine-grained resource billing and limited entropy pools.

Third, while use of symmetric keys is computationally lightweight and faster than public-key approaches, they are rarely used to establish secure communication between endpoints due to constraints in key provisioning and management. Symmetric keys are harder to distribute and revoke, especially in large-scale and dynamic distributed workload deployments.

Fourth, provisioning of symmetric key material must occur before communication between the endpoints can start. Moreover, it requires pre-knowledge of the network topology and of the communication patterns expected from the two endpoints, further complicating the management of symmetric key material. We describe an alternative solution allowing to: (i) provide per-flow key material, where a single key compromise does not affect the security of other flows; (ii) distribute symmetric key material in a way that is fast, dynamic, and automatic. This approach does not require a priori knowledge of the network topology and communication patterns among the involved endpoints; (iii) facilitate centralized maintenance of software and hardware for cryptographic operations and key generation.

Flowrider achieves this by provisioning the Client and Server with a *flow-specific* symmetric key used to establish a secure communication channel. Key provisioning is done *at flow installation time*, whenever the Client initiates a new communication session with the Server. This approach, further described in Section 4, can be used with various protocol suites for secure channel establishment. In Section 5, we additionally detail how it can be implemented in the (D)TLS suite without transcending the isolation between the transport and encryption layers of a communication session.

4 Key Provisioning Method

We next describe *Flowrider*, a novel key distribution method for cloud networks. In particular, Flowrider enables fast, automatic on-demand provisioning of symmetric pre-shared keys to peer endpoints. Pre-shared keys are distributed contextually with the establishment of a network flow between two endpoints and is associated with that respective network flow. Once received, the endpoints can use the pre-shared key to establish a secure channel for communicating over that network flow.

Flowrider builds on the following rationale: each time the Client initiates a session with the Server and triggers the establishment of a new network flow, the network controller generates a new symmetric pre-shared key associated to that flow, and provisions it to both endpoints. To convey the concept, in this paper we assume that the control plane operates in a reactive mode. However, this is not a hard requirement: packets can be matched on the switch while matching packets can be mirrored and upstreamed to the controller.

In the network scenario illustrated in Figure 1, the network Controller provides the Client and the Server with symmetric per-flow keys. Key provisioning is done over the secure channel between the Controller and the Client (C) and Server (S), pre-established at deployment time. Key provisioning is contextual to establishment of a new network flow between C and S, involving the Switch and the Controller.

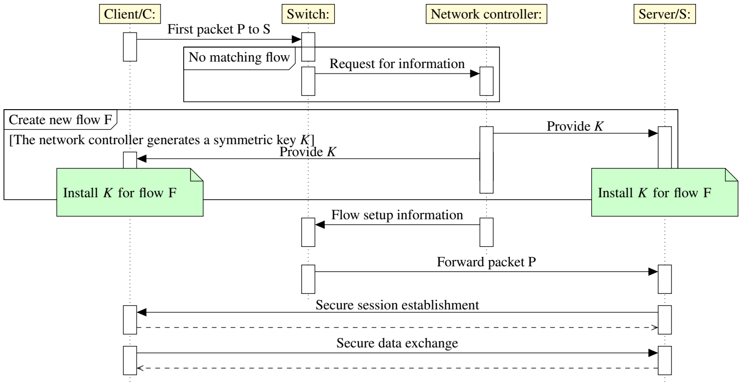

Fig. 2: Step-by-step general execution

We illustrate a run-through of Flowrider in Figure 2, with the following steps:

1. C sends the first packet *P* addressed to S. The packet reaches the Switch.
2. The Switch does not find in its flow table a flow rule matching with packet *P*.
3. The Switch sends a control message to the network controller.
4. The network controller:
   (a) Generates a flow rule F to handle traffic between C and S matching packet *P*.
   (b) Generates a cryptographic symmetric key K associated to F, together with a related key identifier[5].
5. The network controller provisions the key K and the related key identifier to both C and S, through the respective pre-established secure channel. The network controller may additionally provide C with the IP address of S, echoing what is specified in the control message from the Switch.

[5] As a possible optimization, the network controller may have generated in advance a number of symmetric keys, which would thus be immediately available to distribute.

6. C and S install the received key K and related key identifier. If the message from the controller includes also an IP address, C verifies that to be the destination address of its original request to S. This prevents possible internal adversaries from carrying out misbinding attacks based on IP-spoofing.
7. The network controller communicates to the Switch the new flow rule F.
8. The Switch forwards the packet P to S, according to the flow rule F.
9. C and S use the key K to establish a secure session, for example using the (D)TLS Handshake protocol (see Section 5).
10. C and S use the flow F to exchange packets over the established secure channel.

Note that steps 5 and 7 occur concurrently.

4.1 Discussion

In Flowrider, the network controller distributes symmetric keys ad-hoc and on-demand when installing network flows between C and S. Flowrider generates and provisions symmetric keys on a per-flow basis. Hence, different flows between two peer endpoints are related to different and independent security domains. Therefore, compromising the symmetric key associated with a flow does not endanger the security of any other flow between the two endpoints. Note that provisioning symmetric key material is embedded in the OpenFlow control traffic to upstream matching packets and install network flows.

The symmetric key material provided with Flowrider is an alternative to state-of-the-art use of certificates and asymmetric cryptography. Flowrider reduces computational efforts on network endpoints, and hence lowers economic costs. It also reduces entropy requirements for the network endpoints, which is particularly important in virtualized networks.

Section 5 describes how Flowrider can be embodied in versions 1.2 and 1.3 of the security protocol suites TLS and DTLS, without transcending the isolation between the transport layer and (D)TLS. Flowrider is easily and effectively deployable in existing network scenarios that use (D)TLS. Further optimizations are possible, such as indirect provisioning of pre-shared keys to the Server endpoint, through local key derivation on the Server. Section 5.4 describes this optimization with (D)TLS.

The process in Figure 2 refers to a common execution pattern, i.e. where the establishment of the network flow between C and S is triggered by C sending a first packet *P*. Flowrider supports alternative execution patterns, where the SDN deployment is not configured in reactive mode and establishing the network flow - and the consequent key provisioning to C and S - is triggered by the Switch or the network controller, forcing the installation or change of a flow rule. This can happen when enforcing management network policies at deployment time, or when dynamically addressing changes in the network topology and traffic load.

In case of a compromise, the Controller will revoke every flow key issued to a pair of peers. Determining if a peer was compromised can be achieved through intrusion- and anomaly-detection, which are out of the scope of this work. When the Controller determines that one peer P was compromised, the Controller promptly revokes each per-flow key K issued to P which is not yet expired, and notifies any other peer than P that has been provided with K, over the respective secure control channel. This requires the Controller to store at least the key identifier of each non-expired per-flow key.

5 Compatibility with (D)TLS

While Flowrider can be used with various common transport security protocols, we next discuss compatibility with the TLS and DTLS security suites. In Sections 5.2 and 5.3, we describe the embodiment in version 1.2 and 1.3 of (D)TLS, allowing Flowrider to be immediately deployable without breaking existing security standards.

5.1 Transport Layer Security

Most of the network traffic exchanged today, especially on the Internet, is protected at the transport layer. That is, two communicating peers establish a secure channel, namely *session*, and use it to secure the entire application message. The protected message is then handed over to the transport layer, e.g. to the TCP or UDP protocol, for delivery to the other peer. Such secure communication is typically achieved using the protocol suites *Transport Layer Security* (TLS) and *Datagram Transport Layer Security* (DTLS).

The TLS 1.2 protocol suite [12] secures the exchange of application data over TCP among two peers, namely *Client* and *Server*, by preventing the eavesdropping, tampering, and forgery of exchanged messages. The two main protocols composing the TLS suite are the *Handshake* protocol and the encapsulation *Record* protocol. The Client initiates the Handshake execution with the Server, by sending a *ClientHello* Handshake message. Following the Handshake protocol, the two peers agree on a number of security parameters and establish key material to later secure their communications.

The Handshake execution is fundamentally based on two possible approaches, depending on the type of security material pre-installed on the two peers and used during the secure session establishment. In the first approach, the two peers own one or more symmetric pre-shared keys [19], and the Client can suggest to the Server which key it intends to use during the Handshake. In the second approach, the peers rely on asymmetric key pairs, and public keys are exchanged either as conveyed in public certificates [8] or as raw public keys [51] generated by manufacturers and installed on nodes before deployment. A node must use out-of-band means for validating raw public keys received from other peers, and usually retains a list of trusted peer identities. Upon successful Handshake completion, peers can exchange application data messages over the established secure session, using the Record protocol.

The DTLS 1.2 protocol suite [40] provides secure communication of application data over unreliable datagram protocols such as UDP. DTLS is based on TLS, provides equivalent security guarantees, and relies on analogous Handshake and Record protocols. The DTLS protocol suite has several differences from TLS, to deal with the unreliable underlying datagram transport protocols it runs on. In particular, it does not support stream ciphers, admits preserving secure sessions upon silently discarding invalid incoming messages, and includes an explicit fresh sequence number in every protected message. This allows to correctly distinguish and process incoming DTLS messages, also in case of out-of-sequence delivery due to the unreliable transport service.

Finally, DTLS introduces an optional additional exchange of a stateless *Cookie* between the Client and Server, as a first step of the Handshake. Upon receiving a first ClientHello message, the Server can reply with a *HelloVerifyRequest* message, including a locally generated value as Cookie. The Client must then reply by sending a second ClientHello, which includes the same Cookie. The Handshake further continues only if the Server successfully verifies the Cookie received in this second ClientHello. This forces the Client to prove its alleged source IP address, and, possibly in combination with additional means such as [33], complicates possible Denial of Service attacks against the Server performed by an active adversary able to spoof IP addresses.

TLS 1.3 was released to improve both performance and security assurances [18]. While fundamentally providing the same security guarantees as TLS 1.2, TLS 1.3: i) reduces the handshake by one round trip, while having more handshake messages also encrypted; ii) provides new functions for key material derivation, with improved key separation and facilitating cryptographic analysis; iii) always provides perfect forward secrecy if peers run the handshake through public-key based key establishment; iv) supports the latest key establishment, cipher, and signature algorithms, deprecating insecure or obsolete ones; and v) enables the exchange of early secure data at the beginning of the handshake, at the cost of sacrificing a subset of security properties for such data. While TLS 1.3 has been increasingly adopted since its release, TLS 1.2 is expected to continue being used for a long time, as (a dominant) protocol suite for secure communication.

5.2 Flowrider with (D)TLS 1.2

Assume the Client and Server intend to securely communicate using the TLS 1.2 [12] or DTLS 1.2 [40] protocol suite. With reference to the steps in Section 4 shown in Figure 2, Flowrider can be embedded in the (D)TLS Handshake protocol as follows.

At Step (1), the first packet P from C addressed to S is either a TCP SYN (for a TLS handshake) or a *ClientHello* Handshake message (for a DTLS handshake). In either case, C performs the (D)TLS Handshake with S in pre-shared key mode [19].

Later on during the Handshake execution, i.e. at Step (9) of the Flowrider execution (see Section 4), C points S to key K to be used as a pre-shared key for mutual authentication and as input for deriving the (D)TLS session key material. C specifies the key identifier of the key K in the *PSK identity* field of the *ClientKeyExchange* Handshake message sent to S.

5.3 Flowrider with (D)TLS 1.3

Assume that the Client and Server intend to securely communicate using the TLS 1.3 [18] or DTLS 1.3 [17] protocol suite. Flowrider can be embedded in the (D)TLS Handshake protocol as follows (see Section 4, Figure 2).

At Step (1), the first packet P from C addressed to S is either a TCP SYN (for a TLS handshake) or a *ClientHello* Handshake message (for a DTLS handshake). In either case, C performs the (D)TLS Handshake with S in pre-shared key mode. That is, as per (D)TLS 1.3 [17][18], C has to include in the *ClientHello* Handshake message:

1. A *psk_key_exchange_modes* ClientHello extension, which specifies the *psk_ke* or *psk_dhe_ke* key exchange mode.

2. A *pre_shared_key* ClientHello extension, present as the last extension and including a collection of offered pre-shared keys. This collection is structured as follows: (1) a list of key identifiers; (2) a list of *key binders*, one for each pre-shared key and in the same order as the key identifier list. Each key binder is an HMAC computed with a binder key derived from the corresponding pre-shared key. The key binder is computed over the *ClientHello* message up to and including the key identifier list of the *pre_shared_key* ClientHello extension.

S expects a valid hint of the pre-shared key already at the first (D)TLS *ClientHello* message. However, if DTLS is used, C does not have the key K and its key identifier from the network controller already at Step (1) of the Flowrider execution, where the first packet P addressed to S is already the *ClientHello* message. Thus, when starting a new communication flow in the DTLS case, the Client cannot produce a *ClientHello* message, as the per-flow symmetric key is not available yet. Intuitively, this is overcome by the network controller finalizing the original and incomplete *ClientHello* message, before the Switch eventually forwards it to the Server as per the newly established traffic flow. In particular, C stores a dummy pre-shared symmetric key and a related key identifier, which is not associated with any corresponding server. Then, the following adaptation of the Flowrider execution is performed, as also shown in Figure 3.

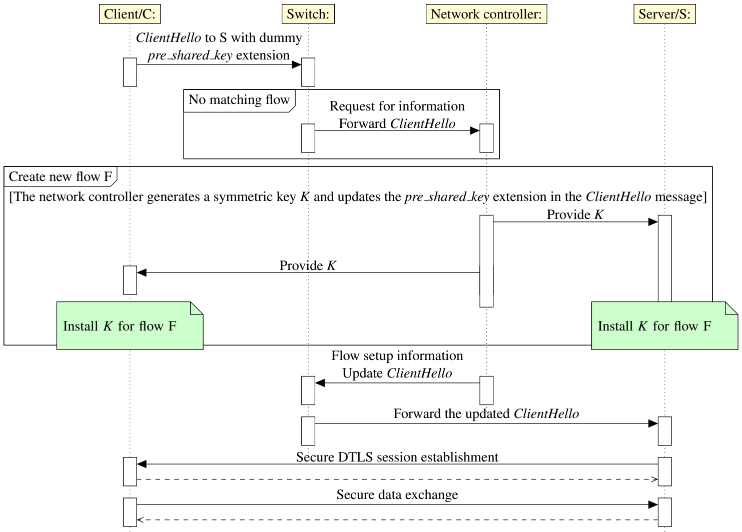

Fig. 3: Step-by-step execution for implementing Flowrider in DTLS 1.3

1. C sends the *ClientHello* message in the first packet *P* addressed to S. In particular, the *pre_shared_key* ClientHello extension offers only the dummy pre-shared key used by C for this purpose. Then, the packet reaches the Switch.
2. The Switch fails to find in its flow table a flow rule matching with packet *P*.
3. The Switch sends a control message to the network controller, asking for information about setting up a new flow between C and S and also forwards the entire packet P, including *ClientHello*, to the network controller.

4. The network controller:
   (a) generates a new flow rule F to handle traffic between C and S akin to packet *P*;
   (b) generates a cryptographic symmetric key K associated to flow F, together with a related key identifier;
   (c) builds a new *pre_shared_key* ClientHello extension for the *ClientHello* message in the packet *P*. The new extension offers only the key K associated to flow F, and includes one consistently recomputed key binder. The recomputed extension replaces the one originally included in the *ClientHello* message in the packet *P*.
5. The network controller provisions the key K and the related key identifier to both C and S, through the respective pre-established secure channel.
6. Both C and S install key K and related key identifier.
7. The network controller replies to the Switch with:
   (a) information on handling packets in the new flow F;
   (b) packet *P* including the updated *ClientHello* message.
8. The Switch forwards the packet P to S, as per the newly installed flow F.
9. C and S establish a secure session/channel, by using the key K, as per the DTLS 1.3 Handhshake protocol.
10. C and S use the flow F to exchange packets over the established DTLS 1.3 channel.

5.4 Optimization through key derivation

As an optimization, the network controller may not explicitly provide S with the key K. Instead, S can *derive* the key K from its key identifier, provided by C as a hint during the (D)TLS Handshake, allowing to further reduce the communication overhead. The optimization requires that:

- The network controller and S share a pairwise symmetric key-derivation key K^*.
- The network controller maintains a counter N_S, which is uniquely associated with S and incremented upon generating a new per-flow key K associated to S.
- The network controller generates the key K by means of a secure key derivation function $PRF(\cdot)$ that takes as input the key-derivation key K^* and a nonce N set as the current value of the counter N_S. $PRF(\cdot)$ can be based on a $HMAC$ function [30] and rely on the same data expansion scheme described in [12].
- Nonce N used to generate the key K is also used as the key identifier of that key.

In (D)TLS 1.2, the Client C simply specifies the nonce N as a key identifier for the key K in the *PSK identity* field of the *ClientKeyExchange* Handshake message. Upon receiving the *ClientKeyExchange* Handshake message, S derives the key K by means of $PRF(\cdot)$, using the retrieved nonce N and the key-derivation key K^*. This approach was discussed in [21].

In (D)TLS 1.3, the nonce N is used as the key identifier for the key K in the *pre_shared_key* ClientHello extension for the *ClientHello* message. In TLS 1.3, this is directly specified by C, after having received the key K from the network controller. In DTLS 1.3, this is specified by the network controller, when building the new *pre_shared_key* ClientHello extension for the *ClientHello* message in the packet *P* (see step (4c) in Section 5.3). In either case, upon receiving the *ClientHello* message, S derives the key K by means of $PRF(\cdot)$, using the retrieved nonce N and the key-derivation key K^*.

5.5 On Preventing the Selfie Attack

Flowrider prevents the reflection attack (“Selfie” [13]) against (D)TLS, which tricks a session peer into processing messages generated by itself, assuming they come from the other peer. This exploits the use of the same pre-shared key in two secure sessions, as (D)TLS client and (D)TLS server.

In an SDN deployment, a peer A (B) acting as (D)TLS client (server) results in one flow, as an exact combination of source address/port and destination address/port. Instead, peer A (B) acting as (D)TLS server (client) results in a different flow, with a flipped combination of source and destination address/port.

In Flowrider, the SDN Controller generates and provides two different pre-shared keys to peers A and B, one for each of the flows. A and B never use the same pre-shared key for both combinations of roles, as they always result in different flows, and distributed pre-shared keys are per-flow. Thus, a given peer gets one different pre-shared key for each role that such peer has with the other peer sharing the same key, and the Selfie attack is prevented by construction.

6 Formal Security Verification

We verified the security properties of Flowrider, using ProVerif [3]. ProVerif is based on the *applied pi calculus* modeling language and can represent processes, their interactions, and available security channels. ProVerif considers an active adversary (Dolev-Yao model [9]) that cannot decrypt encrypted messages without accessing the secret keys.

6.1 ProVerif Modeling

To model Flowrider with ProVerif, we started by declaring types, cryptographic functions, security assumptions, queries, and processes. Throughout the model, we maintain the assumption of a pre-established secure channel between the network controller and the endpoints (Client and Server), consistently with the network scenario presented in Section 3). The channels were securely established using key material assumed to be inaccessible and infeasible to derive for the adversary. The Client, the Server, the Switch, and the network controller are each modeled as independent, top-level processes.

We verified[6] the following security properties of Flowrider: i) the secure provisioning and resulting secrecy of key *K*, i.e. the key associated with the flow between the Client and Server (see Section 6.2); and ii) the mutual secure possession of key *K* by Client, Server, and controller (see Section 6.3). Note that we *do not* verify security properties that are assumed to be already satisfied, such as the security of the (pre-)established secure sessions and the security of session establishments themselves. In particular, the security of the TLS session establishment has been formally verified in [4].

[6] ProVerif scripts available at *https://anonymous.4open.science/r/8e9da3de-6ccd-4f49-b925-389fbcc9bca6/*

6.2 Key secrecy

In the protocol model we assume that, upon receiving from the controller the key *K* associated with the flow, C and S use it to derive the key material for the secure session. While this consists of executing a session establishment protocol, the model assumes that a cryptographically secure Key Derivation Function (KDF) is used to derive a single session key K_S as key material. The KDF takes as input the flow key *K* and context information related to the secure session. Once the Client-Server session is established, the Client sends a message M to the Server, encrypted using the session key K_S. We verified that the adversary cannot access the secret message, with the following query:

$$query(attacker(M)) \tag{1}$$

The model successfully verified the secrecy of message M. Since *K* was used as input to securely derive the session key K_S, in turn used to protect the message M, we conclude that the secrecy of key *K* is also preserved.

6.3 Mutual secure key possession

In Flowrider, only the Client and Server with access to the flow key *K* can successfully establish a secure session with each other in a symmetric mode, over that flow. We verify that the parties that possess *K* can establish a secure session over the flow associated with *K*.

To this end, we verified that, if the Server receives an encrypted message M from the Client over a flow, then i) the Client has previously established a secure session with the Server over that flow; and ii) the Client has sent the message M to the Server, encrypted with the session key K_S derived from the flow key *K* associated to that flow.

ProVerif allows specifying send and receive operations, as well as to initiate and terminate communications between the Client and Server, by means of events such as Initclient(K_S), Termserver(K_S), Initserver(K_S,Ack) and Termclient(K_S,Ack), where K_S is the session key K_S derived from the flow key *K*. The session establishment is successfully completed by both parties when each of them received an acknowledgment from the other party, over that session. The formalized queries for the above events are:

$$\begin{aligned} inj - event(termclient(K_s, Ack)) \\ ==> inj - event(initserver(K_s, Ack)) \end{aligned} \tag{2}$$

$$\begin{aligned} inj - event(termserver(K_s)) \\ ==> inj - event(initclient(K_s)) \end{aligned} \tag{3}$$

Queries 2 and 3 verify that for all Initserver(K_S,Ack) and Initclient(K_S), events Termclient(K_S,Ack) and Termserver(K_S) were previously executed. ProVerif successfully verified both correspondence properties in queries 2 and 3. This implies that only the Client possessing the flow key *K* can connect to the Server over that flow, protecting messages with the session key K_S derived from *K*. This also implies that the Server accepts communications over that flow only from the Client corresponding to the flow key *K*, i.e. exchanging messages encrypted with the session key K_S derived from *K*.

The injective correspondence in query 2 and 3 verifies that the relation between correspondence events is one to one, implying that the Client with access to flow key *K* can successfully open a dedicated session with the Server. Injectivity holds and ProVerif verified the injective correspondence since the Server should complete the session establishment using the flow key only once for the session initiated by the Client.

6.4 Verifying the optimization through key derivation

Flowrider can be further optimized for certain (D)TLS protocol use cases (see Section 5.4). In this optimization, the controller does not send the flow key *K* to the Server. Instead, the Server locally derives the flow key *K* using a nonce generated by the controller and a long-term symmetric key shared with the controller. The nonce is used as a key identifier for the flow key *K* and is specified in the session establishment message addressed to the Server. We verified the optimized version of Flowrider and included the nonce in the first message sent out by the Client to the Switch.

$$\begin{aligned} &event(termclient(K_s, Ack)) \\ ==> \ &event(initserver(K_s, Ack)) \end{aligned} \tag{4}$$

We verified the security properties discussed in Sections 6.2 and 6.3. In this case, we verified only the correspondence, since we considered also multiple flows between the Client and the Server. For non-injective correspondence, the one to one relation between events is not required, but only the event after the arrow is executed prior to the event before the arrow. The formalized queries are:

$$event(termserver(K_s)) ==> event(initclient(K_s)) \tag{5}$$

ProVerif verified the security properties of the optimized Flowrider version.

7 Experimental Evaluation

In order to understand the practical implementation aspects, trade-offs and performance of Flowrider, we implemented[7] it in a distributed virtualized environment. We ran the experiments on Google Compute Platform [44] in a `g1-small` virtual machine (VM) instance (1 vCPU, 1.7 GB memory).

The test bed is distributed between four Docker containers with the following roles (see Figure 4): (a) Client, (b) Server, (c) Controller, (d) Open vSwitch (OvS). The endpoints (Client and Server) use TLS 1.3 [18] implemented with the GnuTLS library [34], version 3.6.5. Two distinct but closely related Client and Server implementations were created for using symmetric keys and certificates respectively. The controller container runs Ryu 3.12 and a custom Python application, that defines packets to be matched and

[7] Implementation code available at `https://anonymous.4open.science/r/8e9da3de-6ccd-4f49-b925-389fbcc9bca6/`

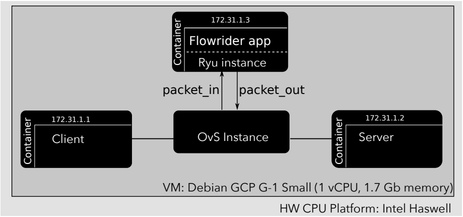

Fig. 4: System test bed

Table 1: **Overview of the performance measurements data set**

Type	PSK			PKI		
	Task clock, msec	**CPU utilized**	**Time elapsed, msec**	**Task clock, msec**	**CPU utilized**	**Time elapsed, msec**
Minimum	3.17	0.034	0.087	13.71	0.183	0.056
Maximum	4.43	0.049	0.097	16.06	0.269	0.080
Mean	3.37	0.037	0.089	14.31	0.24	0.058
Median	3.34	0.037	0.089	14.23	0.24	0.057
Stddev	0.096	0.001	0.0006	0.32	0.005	0.001
Variance	0.00923	0.000001	0.0000004	0.1081	0.00002	0.000002

subsequently generates and delivers keys to the endpoints. The OvS container runs an instance of Open vSwitch that routes packets between endpoints and forwards predefined packet types to the controller.

We measured the performance of establishing a TLS session in two scenarios. We ran the TLS handshake in asymmetric mode using PKI certificates (vanilla scenario) and in pre-shared key (PSK) mode using symmetric keys (Flowrider scenario) consistently with the Flowrider embodiment for TLS 1.3 (see Section 5.3). The Client established a TLS session with the Server in the considered mode and terminated the session immediately afterward. We ran the experiment 10,000 times. In both cases, the OvS flow table did not contain any flows between the Client and the Server; as a result, the first Client message (TCP SYN) was forwarded to the controller in each scenario run.

We illustrate the results of our experimental evaluation in Figure 5 and Table 1. Figure 5 shows that the 'PSK' scenario (representing Flowrider) performs better in terms of time spent on the task and CPU utilization. Time elapsed is longer for the 'PSK' scenario, partly due to the overhead introduced by the communication between the switch and the controller. However, the overhead is mostly offset by distributing the pre-shared keys after the first TCP packet, before the TLS session negotiation starts.

Table 1 presents a more detailed view. The mean task clock is lower in the Flowrider ('PSK') scenario (3.37 msec compared to 14.31 msec). The CPU utilization is *significantly* lower in the Flowrider ('PSK') scenario, with a mean of 0.037 versus 0.24 CPU.

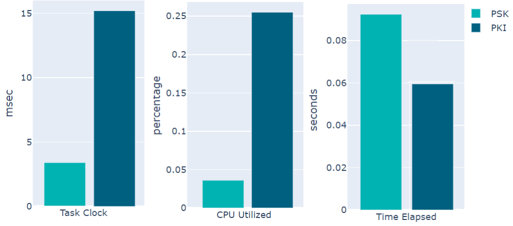

Fig. 5: Task Clock, CPU utilisation and Time elapsed for PSK/PKI scenarios

The time elapsed is about 30% higher in the case of the Flowrider ('PSK') scenario. However, considering that this delay occurs *once* at setup time and is not recurrent, we consider that this is an acceptable overhead.

The Flowrider scenario highlights an order of magnitude lower CPU consumption, due to the use of symmetric key material when establishing the TLS session. Note that the overall time to establish the secure channel does not change significantly. In fact, the very first step of the TCP session establishment triggers the distribution of the symmetric key, which is used to establish the TLS session in PSK mode.

8 Related Work

Protocols such as Kerberos [35] are widely used for symmetric key distribution. This involves a Key Distribution Center - a Trusted Third Party generating and distributing ephemeral keys to clients, without disclosing the secret shared key of the server. Internet applications often rely on an Authorization Server providing trusted assertions to servers about requesting clients [23].

Flowrider key distribution can be viewed as a three-party setting, with the switch acting as a relay and middleman. Three-party authenticated key exchange has received much attention. Its security was formalized by Bellare and Rogaway in [31] and much research has focused on the password-based variant, introduced in [41] and given for the three-party case in [32]. In the three-party password-based authenticated key exchange (3PAKE), low entropy secrets shared with the server are used to negotiate a session key between two parties. This protocol in [32] was shown to have weaknesses [52,6] and many variants have been proposed since then, some using a server public key [22,47], and some that do not [5,48]. More recent work on PAKE protocols include making them post-quantum secure, both in the two-party setting [26] and for three parties [7]. While authenticated key exchange protocols assume an unprotected channel and pre-shared keys, Flowrider uses TLS as the underlying protocol for distributing the key from the cluster manager to the involved parties. This can be accomplished either through symmetric pre-shared keys or public keys and is not predetermined by the Flowrider protocol. Flowrider adapts the problem of three-party key distribution to the SDN setting. The cluster manager is a natural part of the network, and not an otherwise added trusted third party (Key Distribution Center) as in the case with e.g, Kerberos. Since TLS is already used e.g., for deploying jobs to the endpoints and configuring the switch, there is no need to implement additional key exchange protocols. Moreover, new cryptographic primitives incorporated into TLS can be used by Flowrider to take advantage of improved ECC [2] and post-quantum resistant algorithms [15].

Key distribution for SDN deployments was explored in several contexts. Li *et al.* proposed a symmetric key generation and distribution for content delivery network interconnections using SDN and application-layer traffic optimization [45]. The mechanism relies on key generation on the endpoints and a central entity for matching and distributing key pairs. Similar to Flowrider, this relies on a central authority. However, it neither reduces the computational load on the endpoints nor improves the performance of the key exchange. Cloud frameworks commonly rely on a central authority to provision authentication material to virtual instances (either virtual machines or containers) before deployment [10,43]. Provisioning authentication credentials before instantiation reduces the computational load on the endpoints and reduces the entropy requirements. However, the use of public keys certificates for key establishment requires more round trips compared to protocols using symmetric keys.

Provisioning cryptographic material to network endpoints by storing it in trusted execution environments (TEEs) was explored in both academia and industry [27].While this approach leverages hardware security guarantees to store the provisioned cryptographic material, it also introduces additional overhead on accessing the cryptographic material. This includes both provisioning the material to TEEs and retrieving it from TEEs. Finally, other less common approaches rely on information that may be public or not unique, such as the serial number of the device [16], or require manual steps that do not scale in production settings [37].

Flowrider builds on earlier work and leverages the OpenFlow protocol to enable symmetric key provisioning. In contrast to existing approaches, Flowrider drastically reduces the computational requirements for supporting end-to-end encryption; it reduces the number of steps for providing symmetric key material to two endpoints and hence for them to set up secure communication; finally, it allows granular cryptographic isolation of network flows. While Flowrider does not require TEE support on network endpoints, it is complementary to approaches provisioning credentials to TEEs.

9 Conclusion

We have presented Flowrider, a novel approach to distribute cryptographic symmetric keys to endpoints in software networks, contextually with network flow establishment. Flowrider efficiently provisions symmetric key material and significantly reduces the number of CPU cycles needed to establish a secure communication channel between two endpoints. Flowrider leverages the logical centralization of software-defined networks to enable efficient use of symmetric keys.

Furthermore, Flowrider makes key distribution agnostic of the network topology and communication patterns in the system, of which it does not require any early knowledge. Finally, Flowrider is compatible with the (D)TLS 1.2 and 1.3 security protocol suites, with only minor modifications to endpoint implementations. Our experimental performance evaluation shows that Flowrider requires up to an order of magnitude less CPU for a TLS session establishment. Future work will focus on the embodiment and evaluation of Flowrider in alternative protocols for secure channel establishment.

Acknowledgments

This work was financially supported in part by the Swedish Foundation for Strategic Research, with the grant RIT17-0035; by the H2020 project SIFIS-Home (Grant agreement 952652); and by VINNOVA and the CelticNext project CRITISEC.

References

1. A. Greenberg and G. Hjalmtysson and D. A. Maltz and A. Myers and J. Rexford and G. Xie and H. Yan and J. Zhan and H. Zhang: A Clean Slate 4D Approach to Network Con-

trol and Management. SIGCOMM Comput. Commun. Rev. **35**(5), 41–54 (October 2005). https://doi.org/10.1145/1096536.1096541

2. A. Langley and M. Hamburg and S. Turner: Elliptic Curves for Security. RFC 7748 (Jan 2016)
3. B. Blanchet: ProVerif: Cryptographic protocol verifier in the formal model (2020), https://prosecco.gforge.inria.fr/personal/bblanche/proverif/
4. C. Cremers and M. Horvat and J. Hoyland and S. Scott and T. van der Merwe: A Comprehensive Symbolic Analysis of TLS 1.3. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. pp. 1773—-1788. Association for Computing Machinery, New York, NY, USA (2017)
5. C.-L. Lin and H.-M. Sun and M. Steiner and T. Hwang: Three-party encrypted key exchange without server public-keys. IEEE Communications letters **5**(12), 497–499 (2001)
6. C.-L. Lin and H.-M. Sun and T. Hwang: Three-party encrypted key exchange: attacks and a solution. ACM SIGOPS Operating Systems Review **34**(4), 12–20 (2000)
7. C. Liu and Z. Zheng and K. Jia and Q. You: Provably secure three-party password-based authenticated key exchange from rlwe. In: S.-H. Heng and J. Lopez (ed.) Information Security Practice and Experience. pp. 56–72. Springer International Publishing, Cham (2019)
8. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard) (May 2008). https://doi.org/10.17487/RFC5280, https://www.rfc-editor.org/rfc/rfc5280.txt, updated by RFC 6818
9. D. Danny and C. Y. Andrew: On the security of public key protocols. IEEE Trans. on Information Theory **29**(2), 198–208 (March 1983)
10. D. S. Dodgson and R. Farina and J. A Fontana and R. A. Johnson and D. Maw and A. Narisi: Automated provisioning of virtual machines (January 2014), US Patent App. 13/547,148
11. D. Weerasiri and M. C. Barukh and B. Benatallah and Q. Z. Sheng and R. Ranjan: A Taxonomy and Survey of Cloud Resource Orchestration Techniques. ACM Comput. Surv. **50**(2) (May 2017). https://doi.org/10.1145/3054177
12. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (Aug 2008). https://doi.org/10.17487/RFC5246, https://www.rfc-editor.org/rfc/rfc5246.txt, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919
13. Drucker, N., Gueron, S.: Selfie: reflections on TLS 1.3 with PSK. Cryptology ePrint Archive, Report 2019/347 (2019), https://eprint.iacr.org/2019/347
14. E. A. Brewer: Kubernetes and the Path to Cloud Native. In: Proc. of the Sixth ACM Symposium on Cloud Computing. p. 167. SoCC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2806777.2809955
15. E. Crockett and C. Paquin and D. Stebila: Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. In: NIST 2nd Post-Quantum Cryptography Standardization Conf. 2019. pp. 1–24. NIST, Gaithersburg, MD, USA (2019)
16. E. Moret and R. Hubbard and K. A. Watsen and M. Murthy and N. Beauchesne: Systems and methods for provisioning network devices (April 2013), uS Patent 8,429,403
17. E. Rescorla and H. Tschofenig and N. Modadugu: The Datagram Transport Layer Security (DTLS) Protocol Version 1.3 (April 2021), https://tools.ietf.org/html/draft-ietf-tls-dtls13-43, work in Progress
18. Eric Rescorla: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Aug 2018). https://doi.org/10.17487/RFC8446
19. Eronen, P., Tschofenig, H.: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard) (Dec 2005). https://doi.org/10.17487/RFC4279, https://www.rfc-editor.org/rfc/rfc4279.txt

20. European Telecommunications Standards Institute: ETSI GS NFV-SEC 014 V3.1.1 (2018-04) - Network Functions Virtualisation (NFV) Release 3; NFV Security; Security Specification for MANO Components and Reference points (2018)
21. G. Selander: WO/2015/002581 Key establishment for constrained resource devices (2015)
22. H.-T. Yeh and H.-M. Sun and T. Hwang: Efficient three-party authentication and key agreement protocols resistant to password guessing attacks. Journal of Information Science and Engineering **19**(6), 1059–1070 (2003)
23. Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard) (Oct 2012). https://doi.org/10.17487/RFC6749, `https://www.rfc-editor.org/rfc/rfc6749.txt`, updated by RFC 8252
24. I. Baldini and P. Castro and K. Chang and P. Cheng and S. Fink and V. Ishakian and N. Mitchell and V. Muthusamy and R. Rabbah and A. Slominski and P. Suter: Serverless Computing: Current Trends and Open Problems, pp. 1–20. Springer Singapore, Singapore (2017)
25. I. Damgård and T. P. Jakobsen and J.B. Nielsen and J. I. Pagter: Secure Key Management in the Cloud. In: Stam, M. (ed.) Cryptography and Coding. pp. 270–289. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
26. J. Ding and S. Alsayigh and J. Lancrenon and RV Saraswathy and M. Snook: Provably secure password authenticated key exchange based on RLWE for the post-quantum world. In: Cryptographers' Track at the RSA Conf. pp. 183–204. Springer, Cham (2017)
27. K. Sood and J. B. Shaw and J. R. Fastabend: Technologies for secure inter-virtual network function communication (Aug 2 2016), US Patent 9,407,612
28. K. Tamas and P. Kacsuk and J. Kovacs and B. Rakoczi and A. Hajnal and A. Farkas and G. Gesmier and G. Terstyanszky: Micado, a microservice-based cloud application-level dynamic orchestrator. Future Generation Computer Systems **94**, 937 – 946 (2019). https://doi.org/https://doi.org/10.1016/j.future.2017.09.050
29. K. Thimmaraju and S. Schmid: Towards Fine-Grained Billing For Cloud Networking (2020)
30. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) (Feb 1997). https://doi.org/10.17487/RFC2104, `https://www.rfc-editor.org/rfc/rfc2104.txt`, updated by RFC 6151
31. M. Bellare and P. Rogaway: Provably Secure Session Key Distribution: The Three Party Case. In: Proc. of the Twenty-Seventh Annual ACM Symposium on Theory of Computing. p. 57–66. STOC '95, Association for Computing Machinery, New York, NY, USA (1995)
32. M. Steiner and G. Tsudik and M. Waidner: Refinement and extension of encrypted key exchange. ACM SIGOPS Operating Systems Review **29**(3), 22–30 (1995)
33. M. Tiloca and C. Gehrmann and L. Seitz: On Improving Resistance to Denial of Service and Key Provisioning Scalability of the DTLS Handshake. International Journal of Information Security **16**(2), 173–193 (April 2017)
34. N. Mavrogiannopoulos and S. Josefsson and D. Ueno and C. Latze and A. Pironti and T. Zlatanov and A. McDonald: GnuTLS Reference Manual. Samurai Media Ltd., London (2015)
35. Neuman, C., Yu, T., Hartman, S., Raeburn, K.: The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard) (Jul 2005). https://doi.org/10.17487/RFC4120, `https://www.rfc-editor.org/rfc/rfc4120.txt`, updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806, 7751, 8062, 8129
36. Open Networking Foundation: OpenFlow Switch Specification. Tech. Rep. ONF TS-025, Open Networking Foundation (March 2015), v.1.5.1
37. Open vSwitch: Open vSwitch with SSL (2019), `http://docs.openvswitch.org/en/latest/howto/ssl/`
38. Paladi, N., Tiloca, M., Bideh, P.N., Hell, M.: On-demand key distribution for cloud networks. In: 2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN). pp. 80–82 (2021). https://doi.org/10.1109/ICIN51074.2021.9385528

39. R. Bifulco and J. Boite and M. Bouet and F. Schneider: Improving SDN with InSPired Switches. In: Proc. of the Symposium on SDN Research. pp. 11:1–11:12. SOSR '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2890955.2890962
40. Rescorla, E., Modadugu, N.: Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard) (Jan 2012). https://doi.org/10.17487/RFC6347, `https://www.rfc-editor.org/rfc/rfc6347.txt`, updated by RFCs 7507, 7905
41. S. Bellovin and M. Merritt: Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. Security and Privacy, IEEE Symposium on **0**, 72 (04 1992). https://doi.org/10.1109/RISP.1992.213269
42. S. Jain, and A. Kumar and S. Mandal and J. Ong and L. Poutievski and A. Singh and S. Venkata and J. Wanderer and J. Zhou and M. Zhu and J. Zolla and U. Hölzle and S. Stuart and A. Vahdat: B4: Experience with a Globally-deployed Software Defined WAN. In: Proc. of the ACM SIGCOMM 2013 Conf. on SIGCOMM. pp. 3–14. SIGCOMM '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2486001.2486019
43. S. Martinelli and H. Nash and B. Topol: Identity, Authentication, and Access Management in OpenStack: Implementing and Deploying Keystone. O'Reilly Media, Inc., Sebastopol, CA, USA (2015)
44. S. P. T. Krishnan and J. L. U. Gonzalez: Google Compute Engine, pp. 53–81. Apress, Berkeley, CA (2015). https://doi.org/10.1007/978-1-4842-1004-8
45. Seedorf, J., Burger, E.: Application-Layer Traffic Optimization (ALTO) Problem Statement. RFC 5693 (Informational) (Oct 2009). https://doi.org/10.17487/RFC5693, `https://www.rfc-editor.org/rfc/rfc5693.txt`
46. Selander, G., Paladi, N., Tiloca, M.: Security for distributed networking. World Intellectual Property Organization - PCT/EP2019/051456 (July 2020)
47. T.-F. Lee and J.-L. Liu and M.-J. Sung and S.-B. Yang and C.-M. Chen: Communication-efficient three-party protocols for authentication and key agreement. Computers & Mathematics with Applications **58**(4), 641–648 (2009)
48. T.-Y. Chang and M.S. Hwang and W.-P. Yang: A communication-efficient three-party password authenticated key exchange protocol. Information Sciences **181**(1), 217–226 (2011)
49. Ts Binz and U. Breitenbücher and O. Kopp and F. Leymann: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer New York, New York, NY (2014). https://doi.org/10.1007/978-1-4614-7535-4
50. Wang, H., Zhao, Y., Nag, A.: Quantum-key-distribution (qkd) networks enabled by software-defined networks (sdn). Applied Sciences **9**(10), 2081 (May 2019). https://doi.org/10.3390/app9102081, `http://dx.doi.org/10.3390/app9102081`
51. Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., Kivinen, T.: Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250 (Proposed Standard) (Jun 2014). https://doi.org/10.17487/RFC7250, `https://www.rfc-editor.org/rfc/rfc7250.txt`
52. Y. Ding and P. Horster: Undetectable on-line password guessing attacks. ACM SIGOPS Operating Systems Review **29**(4), 77–86 (1995)
53. Y. Zhu and J. Ma and B. An and D. Cao: Monitoring and Billing of A Lightweight Cloud System Based on Linux Container. In: 2017 IEEE 37th International Conf. on Distributed Computing Systems Workshops (ICDCSW). pp. 325–329. IEEE, New York, NY, USA (2017)