EXECUTIVE SUMMARY OF THE THESIS

SDN implementation of zero-knowledge intrusion prevention system

TESI MAGISTRALE IN TELECOMMUNICATION ENGINEERING – INGEGNERIA DELLE TELECOMUNICAZIONI

AUTHOR: Giulio Siano

ADVISOR: Giacomo Verticale

ACADEMIC YEAR: 2022-2023

1. Introduction

In the scenario of network security, firewalls are basic component to demarcate the boundary between a network and external environment, such as the Internet, and foster integrity, confidentiality and availability of data and services.

Several solutions with diverse features and functionalities are available to meet security objectives. Choosing a firewall solution is a critical decision as it affects traffic flows and end-to-end performance.

The very first generation of firewalls were built upon the principle of applying sets of rules to incoming data, denying traffic with certain characteristics and potentially dangerous. Acting as filters, the criteria adopted to state whether incoming traffic is legit includes IP header values, protocol types, port numbers and others [1]. A HTTPS firewall product implementing this solution would accept connection on default port (443) to allow such protocol, resulting in a simple firewall configuration and easy maintenance. Moreover, since criteria consider few header fields and thus non-personal information, the cost in terms of time required to process the packet is low and non-privacy affecting. However, two main concerns influence this solution: HTTPS traffic is not allowed in non-standard ports and non-HTTPS traffic is allowed on the opened port (443).

Other solutions add the additional feature of inspecting TLS packet content: data is decrypted, and content analyzed in search for threats [2]. Some intrusion prevention systems use this option to apply further inspection [3]. While more accurate analysis can be performed, this solution is the most intrusive one raising critical concern on users' privacy.

These two products highlight how choosing a network security appliance involves deciding the right trade-off between system effectiveness and users' privacy, even though both properties are critical to achieve.

However, with the introduction of Zero Knowledge proofs, an advanced cryptographic notion, a party can convince a second of their possession of specific information while preserving the confidentiality of said information. Thus, a system using such tool makes unnecessary the exchanging of sensitive information (or their disclosure by decrypting) to demonstrate they comply with specific requirements. This technology can be implemented in security appliances to remove the limitation dictated by the trade-off decision between system effectiveness and users' privacy and instead allows to achieve both [4]. In this scenario the first party is the prover (a client that provide the proof along with data packet) and the second party is the verifier (the middlebox that check the validity of the received proof), while server is kept free from such process. A third party, the key generator, computes needed data to make this process work. The entire mechanism is transparent to the final server. The resulting topology is following:

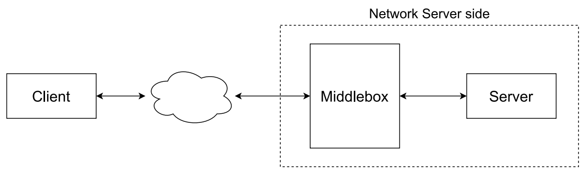

Figure 1: Network topology in [4]

A middlebox is placed at the perimeter of the network, and its role is to intercept incoming proofs and their related packet (the packet on which proof is computed). If the verification process is successful, data packet can be forwarded to the final server.

Nevertheless, beyond the concept of middlebox as security appliance able to support ZK proofs, no practical implementation has been provided. Its structure has not been further developed opening the opportunity to investigate its internal processes. Moreover, the key generator, prover and verifier has been programmed as a single entity embedded in the middlebox, without reflecting a real-case scenario where the three are distributed across the network. Finally, the advantage that a ZK proof does not reveal any sensitive information neither they can be somehow extracted, could be a risk for security: when the verification process is performed and the prover detains both data packet and proof, no method is provided to check whether the proof has been actually computed on the received data packet since the code provided foresees the ciphertext as the private input. In fact, a malicious client can compute a proof on a legit data packet and send it instead along with a non-valid one.

This thesis addresses these problems providing a SDN ZK-based intrusion prevention system implementation focusing on the HTTP-over-TLS traffic.

2. HTTPS ZK-based IPS in SDN

Implementing a ZK system is intrinsically pervasive due to the computation effort required by this cryptographic tool. It has impact on the client side, which must be equipped with a specific program and ZK-related data, and on the node that previously was a firewall and that now must act as a verifier. Thus, the overall system architecture must be carefully designed to support customized computational tasks. In fact, the processing steps required to verify a ZK proof influences network traffic as the action to be performed on the data packet, that is whether it is allowed to pass or dropped, depends on the verification process result.
This requires effective management of traffic and the ability to perform more complicated or customized computations, as the check of binding between proof and data packet. The middlebox must have the capability of supporting such tasks.

These requirements can be met by structuring the middlebox as an SDN environment, where a centralized controller is the single node handling the control plane. To handle traffic passing through the data plane and to execute proof verification process, the controller can be precisely programmed to accomplish such objectives. Furthermore, this solution provides the additional advantage that the system can be further enhanced with supplementary features, due to flexibility and versatility that such architecture offers. This is the reason why the security appliance proposed by this thesis is an intrusion prevention system: the inherited modularity of such middlebox, provides the opportunity to support additional computations that are also performed by common IPS, such as inspecting traffic patterns, traffic monitoring, search for intrusion attempts and so on.

Thus, the topology that addresses these challenges is the following:

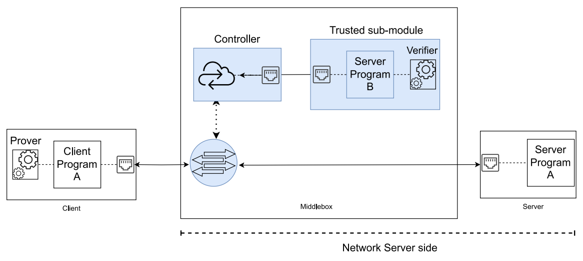

Figure 2: HTTPS ZK-based IPS architecture (in blue the elements introduced)

As depicted in the figure above, the middlebox consists of a switch in the data plane and of both controller and a processing module in the control plane. The need for the second module in the control plane lies in the fact that the controller is in charge of handling data plane device and traffic flowing through it, thus the computation it performs must be as lightweight as possible. For more complex tasks, such as proof verification or other IPS monitoring and controlling activities, SDN controller should outsource them to this second module.

The client is equipped with a program able to initiate TLS communication and generate ZK proof. Thus, according to the particular stage of its lifecycle, it acts as both HTTPS requester and as a prover. Its functions are initiating handshake with server, encrypting HTTP requests, generating proofs, storing TLS information, resuming previous TLS sessions.

The second block, the middlebox, comprises multiple components belonging to both data and control plane. As the data packet and proof arrives, they are forwarded to the second module for verification and, according to the result, the underlying switch is instructed to let pass or drop the HTTPS request. Therefore, the middlebox functions allow TLS handshake, verifying proofs and whether they have been computed on the received data packet, and packet control (that is instructing the switch).

Finally, the server is a minimalist component that runs an HTTPS server accepting incoming TLS connection request and does not participate in any way to ZK processes. It only completes TLS handshake and participates in HTTPS exchange.

3. Network lifecycle

The network and its elements are designed to pass through stages based on HTTP-over-TLS and ZK protocols. These stages form the network's lifecycle, involving data and control plane interactions for packet and connection handling.

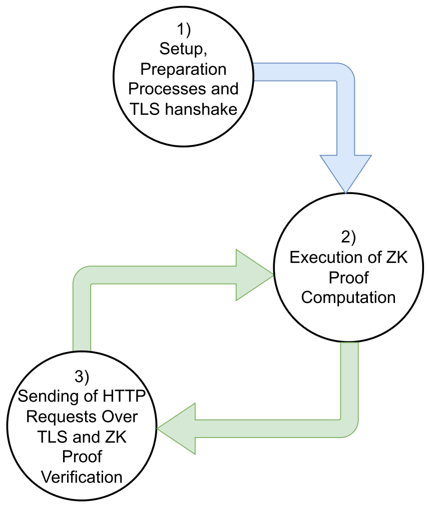
1) Setup, Preparation Processes and TLS hanshake
2) Execution of ZK Proof Computation
3) Sending of HTTP Requests Over TLS and ZK Proof Verification

Figure 3: Network lifecycle

Identifying macro-stages creates a structured framework for understanding network operations in various phases and makes easier handling TLS and ZK proofs communication complexities.
The advantage this lifecycle offers is the only one setup phase execution, which is also the one with the higher cost in terms of computation. Once the initial parameters are generated, they can be reused for future interactions, so its cost is amortized.
Following, each step of network lifecycle is described.

3.1. Setup, preparation processes and TLS handshake

Since different technologies and protocols are used in this architecture, the setup phase needs a distinction for each of them.

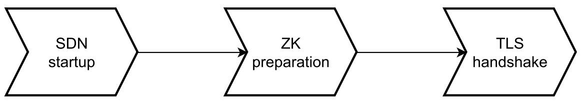

Figure 4: setup steps

1) **SDN startup:** this step involves the middlebox only. Both switch and controller are booted up and, using the *Openflow* protocol, they exchange network traffic information. Moreover, the module running the verifier is started too. The connection between switch and controller is essential for the controller to orchestrate network behavior. In fact, controller issues specific instructions to the underlying switch, instructing to handle certain packet. The main rules applied are the following: a) the default behavior is let traffic flows without interruption; b) when a HTTP-over-TLS packet is detected, specifically when one contains a http request destined for the server, the controller retains such packet for further process; c) UDP packet with specific destination port are forwarded to controller. They are ZK proofs sent by the client.
   The two will be sent to the verifier for verification process.
2) **Zero-Knowledge preparation:** the key elements to implement the ZK solution are a) circuit, which is the program that runs the firewall (i.e., check that the packet content complies with network policy rules), b) circuit input file, that contains public input, private input for the circuit and circuit-related data (i.e., gates, wires and their connection), c) proving key and verification key. So, when it comes to distributing such elements, the client receives the circuit, proving key, and input file, while the verifier receives the verification key and input file only. Indeed, verification process consists in checking whether a certain equality is verified [5], therefore no circuit is needed. In this phase, input file detained by both prover and verifier contains circuit data only.
3) **TLS handshake:** this step requires the boot up of both HTTPS client and server. While the server listens for incoming TLS connections, the client starts TLS handshake. Connection related parameters are stored to be later used for proof computation or to reuse that TLS session.

Eventually, information detained by all parties are shown in the following figure:

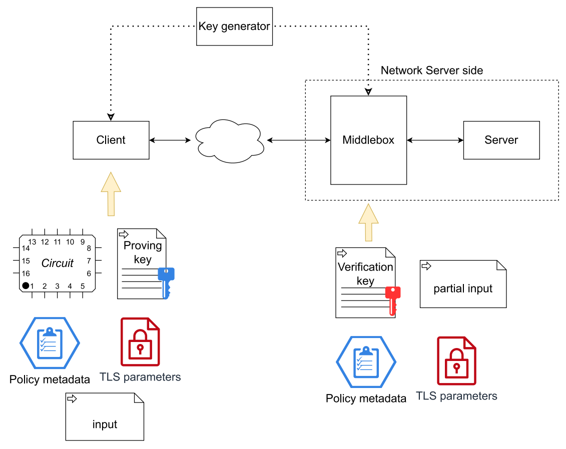

Figure 5: files detained by each party at the end of the setup phase

The policy metadata file contains the rules applied in the network. When the circuit runs, it decrypts HTTPS packet using TLS session information and checks its content against the rules. If it complies, the proof can be computed.

3.2. Execution of ZK proof computation

The second step of the lifecycle (*Figure* 3) involves the prover (client) only. First, as stated in the previous section, the input file contains circuit-related data only, consequently it must be filled with private and public inputs. This file is structured as follows:

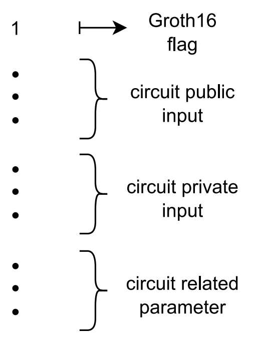

Figure 6: input file structure

The analysis performed on such file is what allowed the separation, from a programming point of view, of the three ZK entities (key generator, prover, and verifier). Its interpretation and its filling with public and private input is performed

manually (by a custom function, outside the used libraries), and by splitting and converting them into the appropriate representation.
As a second step, circuit and input are loaded and ready to be executed. Proof is then computed, and the circuit output is a binary file.

3.3. Sending of HTTP request over TLS and ZK proof verification

Now that proof is ready, the client encapsulates it in a UDP packet with specific destination port that controller uses to intercept proofs. Then, using the previous TLS session, send HTTP request over the channel. The switch forwards both UDP and HTTPS request to the controller which in turn sends them to the verifier module (the HTTPS packet is encapsulated into a UDP packet).
Here, the input file is filled with public input only, such as TLS public session data, policy rules and others. Then both input and verification key files are loaded, and proof received verified. If the result is PASS, the controller instructs the underlying switch to forward the HTTPS packet to server, otherwise to drop it. Finally, the server receives the request and replies with ack, and the resources content requested.
Together with proof verification, the check about HTTPS request and proof binding is performed. Because verifier does not fill input file with private input (since they are the secrets), public input can be exploited. A proof verification is successful if the proof is valid and if the public content used is the same for both prover and verifier (for example, network rules). Thus, configuring the circuit at client side to include HTTPS request as public input, and at middlebox side to include in the input file the HTTPS request received, the prover is forced to send out the same HTTPS request used to compute the proof.
Indeed, the circuit has been programmed to use the HTTPS request in public input to compute the proof and if it is invalid the proof computation fails. Instead, if a valid proof is sent out along with another HTTPS request, the verifier would insert the received HTTPS in the public input but since it is different from what used by the prover to compute the proof, the verification process fails.
Summarizing, public inputs used by the prover must be the same as those used by the verifier. By requiring the HTTPS request as a public data, prover must provide to the verifier the same HTTPS packet used to compute the proof. Moreover, such modification does not affect client privacy because the packet content is encrypted and, since the scenario can be the Internet, everyone can sniff such (encrypted) content.

4. Testbed

The overall network has been implemented on a machine with quad-core processor (AMD FX-8350, 4 GHz) and 20 Gb of DDR3 memory. Each block (client, middlebox, server) has been deployed in a separate virtual box and interfaces appropriately set to let them communicate as happens in the real case scenario.
Client is equipped with two main programs: the first contains the steps described for the prover (a C++ program using the *libsnark* library), the second is a Python script that runs the prover program, initiates TLS handshake, and sends out HTTPS request and proof. Server is simply equipped with a Python script that opens for incoming TLS connection and replies to HTTPS request.
Finally, the middlebox is made of an OpenVSwitch (OVS) switch that, through the *Openflow* protocol, communicates with the controller. This last has been implemented with the ONOS controller due to its ability to house new customized applications using the language Java. Due to this flexibility, it has been possible to implement the controller behavior as described in the section *HTTPS ZK-based IPS in SDN*.

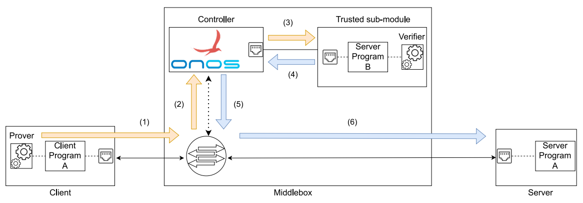

Figure 7: network implementation

Instead, following the IP and MAC addresses planning:

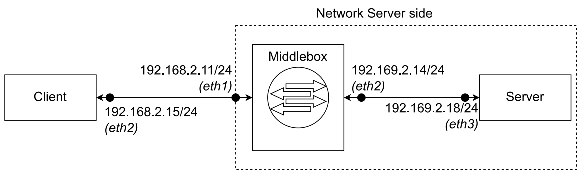

Figure 8: IP network planning

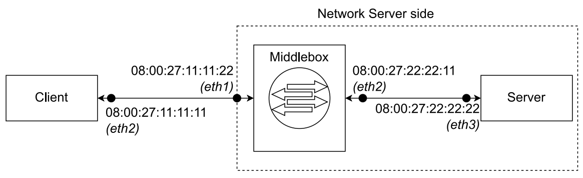

Figure 9: MAC network planning

Therefore, according to such plannings the rules applied to OVS switch are the following:

Table 1: OVS rules

ID	Condition	Action
1	ICMP, ip_dst = 192.169.2.18	Set mac_dst = 08:00:27:22:22:22, out port=eth2
2	ICMP, ip_dst = 192.168.2.15	Set mac_dst = 08:00:27:11:11:11, out port=eth1
3	TCP, ip_dst =192.169.2.18, port = 443	Set mac_dst = 08:00:27:22:22:22, out port=eth2
4	TCP, ip_dst =192.168.2.15, port = 443	Set mac_dst = 08:00:27:11:11:11 out port=eth1
5	UDP, ip_dst =192.169.2.18, port = 49152	Do not forward. Send to Controller
6	TCP, ip_dst =192.169.2.18, port = 443, HTTPS request	Do not forward. Send to Controller

Finally, the practical ZK implementation used is Groth16.

5. Results

The experiment conducted consists of 5 testbed runs to test both scenario consistency and resiliency, and proof computation/verification times. SRS (proving and verification keys) and circuit size have been considered as well. Running the entire testbed means:

1. Set up the environment as described in the previous sections.
2. Keeping the middlebox running with its controller, switch, and verification unit.
3. Run the server script which listens for incoming HTTPS requests.
4. Run the client script which starts TLS connection, proof computation and sends out both proof and HTTPS request.

The testbed has used the following ZK-related data which size is:

- Circuit file size: **327,5 MB**
- Circuit input file size for prover (*partial_info.in*): **186 KB**
- Circuit input file size for verifier (*input.in*): **2 KB**
- Proving key size: **655 MB**
- Verification key size: **17 KB**
- Proof size: **1019 bits**

The testbed results in terms of time spent for each ZK step are the following:

Table 2: time required by each ZK step.

Step	Time (s)
Circuit parsing and evaluation	126.9799
Key generator	21.1368
Proof computation	14.2502
Proof verification	0.0221

Note that to achieve this result the *libsnark* library has been compiled by setting the flag DMULTICORE=ON, thus the computation uses all processing units available.
However, the key generator and prover runs more than one step to achieve their goals. Following, the overall time each party spend:

1. The overall time the generator spends is given by the sum between "*Circuit passing and evaluation*" and "*key generator*" (approximately **154,689s**)
2. The overall time the prover spends is given by the sum between "*Circuit passing and evaluation*" and "*Proof computation*" (approximately **141.2301s**).
3. The overall time the verifier spends is given "*Proof verification*" only (approximately **22ms**).

Therefore, according to *Table 2* and the times reported above, it comes out that the verification process is the lightest one. Key generation is run only once, and its cost is amortized over time. Instead, proof generation process is the heavy and costly one and must be run each time a HTTPS request is emitted.

Moving the focus from ZK proof system to TLS performance, the scenario considered are two: in the first the switch only forwards packets from one side to the other, thus no proof verification or other systems are applied; in the second the scenario proposed so far with proof verification is applied. The logic behind this choice is to gather statistics about the impact of ZK proofs.

Table 3: comparing overall performance.

	No ZK proof	ZK proof
Handshake time (s)	~ 0.170	~ 0.170
E2E time (s)	~ 0.006	~ 0.050

In the table above, the handshake time does not vary whether one solution is used or the other. This fits with the fact that in the second scenario no TLS handshake packet is sent to the controller, so the behaviour in the two cases is the same. E2E time means the time passed from the HTTPS request sending to the server response receiving, thus this time has been measured at client side. As reported in the *Table 3*, the E2E time if ZK proof is used is 8.3x times higher than the first case. Further consideration can be done: knowing that the proof verification time is 22ms, considering the *Figure 7* the time required for the step *1*, *2*, *3*, *4*, *5* and *6*, is *28ms*. Moreover, from the first case where the controller does not take over during HTTPS traffic, is known that step *1* and *6* of the *Figure 7* requires on average *6ms* (see *Table 3*), thus it comes that for steps *2*, *3*, *4*, *5* of the *Figure 7* the time spent is *22ms*.

Therefore, using the proposed solution in the testbed described, the additional time (once the proof is computed) spent is:

1. **22ms** to verify the proof.
2. **22ms** for the steps *2*, *3*, *4*, *5* of the *Figure 7*

For a total of **44ms** more with respect to the first case (no firewall).

In general, the time this architecture requires intended as the time passed from when the proof and HTTPS request are sent to when a response from server is received, is 8.3x more the case where no firewall, ISP or other security applications are used.

6. Conclusion

Using zk-SNARK proofs in a communication network means providing additional privacy preserving option with respect to current ones due to their intrinsic theoretical structure. This work proposes a network design strictly compliant with theoretical results and starting from the elements and actors foreseen by the adopted solution (i.e., Prover, Verifier, Middlebox in Groth16). Step by step, this framework has been implemented and extended with additional tools (such as ONOS controller) to let theoretical solutions meet technical implementation and make it practical.
To show that such a solution can be adopted, an example of computation has been designed and implemented. This consists in an intrusion prevention system accepting HTTPS messages if related proof verification process is successful. Clients can generate proof based on the message they are willing to send if they respect policy rules provided by the destination network, where allowed methods and paths are specified.
All this is orchestrated by an ONOS controller in which a custom application is installed.
The results analysis confirms how the major effort in terms of resource consumption such as storage and computation, is up to provers (or clients). The effort the generator makes is amortized over time and over message exchanges, and thus not considered. While verifier (middlebox) needs 20 KB of storage and spends 22ms to verify a proof, the prover (client) needs ~ 980MB of storage and spends 142 seconds to compute a proof.
Finally, the E2E time intended as the time passed from when the proof and HTTPS request have been sent to when the server response have been received, is of **0.050ms** in the proposed testbed which is 8.3x higher to the case where middlebox is deployed. However, this approach provides both firewall effectiveness without affecting users' privacy, with the possibility to further extend the controller capabilities in an easy way (i.e., implement ONOS application such as malware detection using Java).

7. Bibliography

[1] Cisco, "The Future of the Firewall White Paper," 2019, [Online]. Available: https://www.cisco.com/c/en/us/products/c

ollateral/security/firewalls/ngfw-futureoffirewalling-wp.html

[2] Cisco, “SSL Inspection (SSLi) Bundles for Scalable Inspection of SSL/TLS Encrypted Traffic.” 2022. [Online]. Available: https://www.cisco.com/c/en/us/products/collateral/security/ssli-bundles-wp.html

[3] T. Radivilova, L. Kirichenko, D. Ageyev, M. Tawalbeh, and V. Bulakh, “Decrypting SSL/TLS traffic for hidden threats detection,” in *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 2018, pp. 143–146.

[4] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish, “Zero-Knowledge Middleboxes,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4255–4272.

[5] J. Groth, “On the Size of Pairing-Based Non-interactive Arguments,” M. Fischlin and J.-S. Coron, Eds., in Lecture Notes in Computer Science, vol. 9666. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. doi: 10.1007/978-3-662-49896-5.