Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Authenticated Scalable Port-Knocking

Daniel Sel

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Authenticated Scalable Port-Knocking

Authentifiziertes Skalierbares Port-Knocking

Author	Daniel Sel
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Sree Harsha Totakura, M. Sc.
	Dr. Heiko Niedermayer
Date	March 15, 2016

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, March 15, 2016

Signature

Abstract

Most attacks on public Internet services and their providers' infrastructure are not aimed at a specific target, but rather send huge amounts of packets on large IP ranges in order to find vulnerable computers to be exploited by the hypothetical attacker. These types of attacks typically start with *port scans* to determine the active hosts and the services running on them. One technique to counter such attacks to prevent the port scans from providing usable information to the attacker has been known for a while now and is called *port-knocking*. However, this security mechanism has not seen high adoption in today's industry.

This thesis is aimed at providing a specification for a secure and scalable solution for hiding services in a cloud environment using port-knocking. The specification describes a concept based on X.509 certificates with Elliptic Curve Cryptography (ECC) to enable decentralized authorization of clients via port-knocking. The presented approach also allows for deployment of this concept with minimal overhead on the server or provider side while eliminating any required configuration or visible complexity for the end user. Furthermore, the design is based on the requirement to keep the impact on the communication of protected applications to a minimum and therefore relies on a single UDP packet, which is small enough not to be affected by fragmentation.

sKnock, a python-based prototypic implementation of the described specification is also encompassed in the thesis to evaluate performance and reliability of this approach. The implementation runs completely in user-space on the server as well as on the client side and does not require elevated privileges of any kind from the end user. The prototype is targeted at the Linux platform with OpenSSL as cryptographic module and iptables as firewall on the server side. However, the implementation was developed with platform independence in mind to make its' extension by adding modules for other platforms simple.

The incorporated experiments indicate that the included implementation is already fast and reliable enough for a large-scale deployment. By extending the base functionality provided in this prototype, this solution can be adapted to numerous port-knocking scenarios with low deployment and management overhead in scalable environments.

All facts considered, the most important characteristic as well as the major design goal of the presented specification and *sKnock* is to provide a concept for *scalable* port-knocking mechanisms and therefore lay the foundations for increasing industry adoption of port-knocking.

Zusammenfassung

Die meisten Angriffe auf öffentliche Internetdienstleistungen und die Infrastruktur der verantwortlichen Provider haben kein vordefiniertes Ziel. Stattdessen sendet ein Angreifer große Mengen von Datenpaketen an eine großere Anzahl von IP Adressen, um ein potentielles Opfer auszumachen. Typischerweise beginnt er dazu mit einem Port Scan. Dieser ermöglicht die Erkennung von aktiven Hosts und verifiziert zudem welche Dienste auf dem potentiellen Ziel ausgeführt werden.

Das Hauptziel dieser Arbeit ist das bereitstellen einer Spezifikation für sicheres und skalierbares, authentifiziertes Port-Knocking. Die Spezifikation beschreibt ein Konzept basierend auf X.509 Zeritifikaten unter Nutzung von Elliptischer Kurven-Kryptografie (ECC), um die dezentrale Autorisierung von Clients durch Port-Knocking zu ermöglichen. Der präsentierte Ansatz ermöglicht desweiteren die umsetzung dieses Konzeptes mit minimalem Overhead auf der Provider-Seite, während die gesamte Komplexität vor den Endbenutzern verborgen bleibt. Desweiteren basiert das Design auf der Vorraussetzung, die Auswirkungen auf die Kommunikation der zu schützenden Applikationen so gering wie möglich zu halten und benötigt deshalb nur ein einziges UDP paket, welches klein genung ist um nicht von Fragementierung betroffen zu sein.

sKnock, eine auf Python basierende, prototypische Implementierung der beschriebenen Spezifikation ist ebenfalls im Umfang der Arbeit enthalten. Der Zweck hiervon ist die Einschätzung der Performance und Zuverlässigkeit unseres Konzeptes. Das Programm läuft vollständig im User-Space (sowohl auf dem Server als auch auf dem Client) und erfordert keine erhöhten Berechtigungen jeglicher Art vom Endbenutzer. Die Zielumgebung dieser Implementierung ist die Linux Platform mit OpenSSL als kryptographischem Modul und Iptables als as Server-Firewall. Nichtsdestotrotz wurde bei der Umsetzung hoher Wert auf Platformunabhängigkeit gelegt, was die Erweiterung auf andere Umgebungen durch das Hinzufügen von Platform-spezifischen Modulen sehr einfach gestaltet.

Die in der Arbeit enthaltenen Experimente verdeutlichen, dass die Implementierung bereits schnell und zuverlässig genug für eine großangelegte Auslieferung ist. Durch die Erweiterung der Basis Funktionalitäten, die in diesem Prototyp bereits enthalten sind, kann die Implementierung derart angepasst werden, dass sie unterschiedliche Port-Knocking Szenarien mit einem geringen Aufwand für Auslieferung und Management auch im großen Rahmen ermöglicht.

Dementsprechend bieten die hier präsentierte Spezifikation und *sKnock* ein Konzept für skalierbare Port-Knocking Mechanismen und legen damit den Grundstein für eine zunehmende Akzeptanz dieser Technologie in der Industrie.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Goals of the thesis

The major goal of this thesis is to provide a secure and scalable solution for hiding services in a cloud environment using port-knocking. In order to achieve this, the thesis aims to explore a variety of possibilities in implementing such a security layer and especially focuses on keeping the overhead for daily operations as low as possible. The scope of this thesis is to create a minimum working prototype to prove the viability of the port-knocking concept in general and evaluate its' strengths and weaknesses against realistic requirements for such a software component. This work should lay the foundation for following future research on the topic of converting the theoretical concept of port-knocking into a full-featured viable security solution for real world service providers and customers from other segments.

1.2 Outline

In Chapter 1 this work introduces the topic of port-knocking and explains the motivation of developing such a security layer. Following the introduction, the necessary scientific background information is provided in Chapter 2. All details regarding the implementation developed in the process of this thesis, *sKnock*, will be presented in Chapter 3. The evaluation of this implementation together with implications for the concept in general can be found in 4 which is followed by concluding this work with an outlook on further improvements and a comparison to related work in the final Chapter 5.

1.3 Motivation

1.3.1 Security in modern Network Design

Historically, computer networks were never conceptualized with security in mind. When people started interconnecting computers in the 1980's, they wanted their machines to be able to communicate with each other as easily and reliably as possible [3] — implementing security features would not only have slowed down this process, but would also have made it considerably more complex. A few decades later, however, after connecting billions of devices across every continent using an unbelievably large and complex global network — the Internet — security became a major consideration in network design [4]. Consider the architecture of cloud services today: the service providers have datacenters in different locations and use them to host nodes running essential computations for their service. To address the demands of a global audience, the providers may then choose to have several edge-nodes, residing at even more locations. These edge-nodes take care of user authentication and some preliminary caching, thereby relieving the compute nodes for important tasks [5].

However, securing the services offered by these cloud providers has been shown to be a great challenge [6]. Even giant IT-corporations with decades of experience in networking like Google, Amazon, etc. struggle to keep up with evermore new types of attacks threatening the very base of their business operations: their computer services and confidential data [7].

Since network communication is based on an architecture of multiple decoupled layers defined in the ISO-Standard "Open Systems Interconnection Model" (OSI) [8], it only makes sense to employ specialized security components, which individually cover each of the networking layers. This strategy enables security architects to introduce redundancy in network security components in order to reduce the risks of the weakest link, which is critical to every network.

Generally, public services have the security-related disadvantage that everyone can communicate with them. Security infrastructure on the networking layer, such as firewalls or intrusion prevention systems (IPS) [9], are only able to protect servers from network based attacks, for example, a Distributed Denial of Service attack (DDoS) [10]. However, every user behaving in a non-suspicious way from a network traffic perspective is able to initiate a communication with every public application, even when they have no legitimate interest in using the application. In most cases this cannot be prevented by aforementioned (hardware) firewalls, as the information available to them is not sufficient for effectively restricting access to legitimate users in a session-independent way [9]. As most modern services available to the public employ encryption as part of their security concept [11], most of the time the only data available to the network security infrastructure is metadata like IP addresses, ports, packet size, etc. Processing

this information clearly does not allow for a reasonable restriction of user access, since IP addresses, for example, can change very frequently for the same user and do not provide a reliable way of identifying authorized users. This exposure of services to almost unrestricted access on the network layer imposes significant security implications, since all higher-level security mechanisms have to be implemented by the application itself and a weakness at the application layer can be exploited by everyone on the Internet.

1.3.2 Port-Knocking

One possible and particularly interesting solution to fill the gap between network layer and application layer security in addition to providing an authorization mechanism at a much earlier stage is Port-Knocking. It stems from an authentication concept that was invented thousands of years ago, where members of certain secret communities were using special *knock sequences* when knocking on each others doors to prove their identity as part of the community. In the world of networking, the house is the server, the person outside the door is the client and the door is the firewall. The most basic implementation of the port-knocking concept would be a client sending packets to a number of closed ports on the server in a predefined sequence after which the server firewall opens the port for the actual application, granting the client access [12].

This concept provides an additional security layer to hide the services running on the server, which makes it harder for an attacker to exploit any application-level vulnerabilities without being successfully authorized through the port-knocking protocol. By concealing the running applications in this way, a cloud provider could not only minimize the risk of an attacker breaching their system using a *Zero-Day Exploit* [13] or unpatched vulnerability, but also significantly reduce the amount of random port-scans followed by attacks on well-known services. Even if all the running services are perfectly secure at a given time (meaning there are no known security holes), minimizing the attack surface on edge-servers using a strong authentication mechanism beneath the application layer may significantly reduce unnecessary load generated by above-mentioned random undirected attacks.

Of course the concept of *knock sequences* described above would not fulfill the requirements of any serious service provider on the Internet, since it is neither secure, nor scalable or granular in granting application-specific privileges. Generally speaking, all of the available products relying on this classic scheme of port-knocking are practically unusable for cloud service providers in terms of performance and security. Just by being forced to send multiple packets before being able to start the relevant application-level communication the overhead, and thereby delay, introduced in the communication is severe and definitely noticeable for the end-user. Taking into account the chance of packets getting lost, the probability of an even worse delay by retransmission of one or more packets scales proportionally with the number of knock-packets in the

sequence. Furthermore, the use of a single packet or a static sequence of packets does not represent much additional security as the only required secret, the sequence, is easily discoverable by sniffing the network traffic. Using a dynamic sequence derived from a secret for port-knocking would improve security, however, a solution based on a packet volume high enough to provide useful security would not be able to scale up to modern expectations of performance.

Therefore from this point on, this thesis focuses solely on a specific variant of port-knocking implementations, called *Single Packet Authorization* [14] or *Hybrid Port-Knocking* [15]. This variant relies on strong cryptography to reduce the required communication to a single packet while also enhancing security.

1.3.3 Related Implementations

To assess the current state and usage of this kind of additional security, the following implementations were researched and judged for real-world applications by relevant criteria such as security, performance, overhead and scalability. Out of hundreds of different interpretations of the port-knocking concept, five were chosen because of either their popularity or particular interesting characteristics in regard to any of our criteria: *knockdaemon* [16], *SilentKnock* [17], *fwknop* [18], *knockknock* [19], and *Knock* [20].

1. knockdaemon—This is the prototype provided by Martin Krzywinski, when he published his article about the port-knocking concept [12]. It has been updated several times until December 2004 enabling features such as encryption of knock-packets using a shared secret and configurable encryption modules, commands to be executed depending on the target port, sniffing of network traffic to listen for knocks, etc.

The prototype runs in user-space and its initial version was restricted to reading knocks from the firewall log. It is highly configurable, which allows for completely insecure operation up to a secure environment. As it follows the classical concept, it requires multiple packets to be sent to multiple closed ports which knockdaemon uses to authorize clients. The configuration allows for the packets to be encrypted, but only using a shared secret between server and all clients. The prototype also allows for incorporation of the client IP address in the knock packets [16].

Since every new connection requires multiple packets to be sent beforehand, it causes significant overhead and therefore a delay for the end-user. Encryption only works via a single shared secret for *all* users, which means that any user can become a liability for the whole security. Additionaly, the prototype was written in Perl, an interpreted language known for low performance, and is therefore unable to run at a high speed. Finally, it requires a knock-sequence to be pre-shared per-user on the server-side in

order to identify the port-knocking requests and therefore provides no scalability for adding new users.

2. SilentKnock—This approach by authors Eugene Y. Vasserman, Nicholas Hopper, John Laxson, and James Tyra is intended as proveably undetectable and high-performing implementation for realizing port-knocking without the necessity of changing the protected applications.

It works by using kernel-hooks via the netfilter and libIPQ API to intercept each TCP-SYN packet that is sent from the client machine to generate a MAC over the TCP Header. This hash is then encoded into the TCP initial sequence numbers of the SYN packet as steganographic information using a system by Murdoch and Lewis' [17] before letting it continue with the sending process. On the server side, the *sknockd* service intercepts the TCP-SYN packets for all configured ports and verifies the encoded MAC before passing the packets on to the TCP/IP stack. Other types of packets are not intercepted. The generation of the MAC checksums relies on shared secrets, which can also be unique per client. According to the published paper, the only supported target environment is Linux with a 2.6 kernel version [17].

The nature of this implementation allows to deliver excellent performance concerning the processing of incoming requests. However, it only supports TCP and affects *every* program running on the client side, even the ones that have no relation to port-knocking. Although it supports client-specific shared secrets, it also suffers from the scalability problem as there is no way of obtaining that secret on the server-side without pre-populating the configuration with every clients' key.

3. fwknop—Developed by Michael Rash from CipherDyne Security[1], fwknop is an extensive port-knocking framework supporting numerous features including symmetric and asymmetric encryption, server- and client-side NAT, processing knock requests by sniffing network traffic or reading a pcap dump, execution of remote shell commands, OS fingerprinting, and even TOR integration. The server is targeted at UNIX environments and supports *iptables* and *firewalld* under Linux, *pf* on OpenBSD, and *ipfw* on BSD and MacOS. These platforms are also supported by the client, which additionally is also compatible with Windows, Android and iOS as operating system [18].

This implementation is mature compared to other implementations and contains a large feature-set while being a prime example for supporting multiple platforms. As fwknop is written in C, it should be able to perform at a high level compared to most other port-knocking implementations. However, although it is one of the few port-knocking solutions to support asymmetric encryption, it still requires the RSA key of every user to be present on the server and therefore also suffers from greatly limited scalability.

[1] http://cipherdyne.org/

4. knockknock—Being annoyed with other peoples' interpretation of the port-knocking concept that led to, in his opinion, overly complex and bulky services instead of, for example, "us[ing] cryptography to *simplify* the initial port knocking concept, rather than making it more complex", Moxie Marlinspike wrote a lightweight implementation of a simple version of the port-knocking concept in Python. He uses the TCP SYN and ACK fields as well as the fields for window scaling and sequence numbers in the TCP Header to encode an encrypted and authenticated port-knocking request. In order to receive the requests the implementation relies on the firewall log and since the encryption is based on AES with shared keys, the server requires per-client configuration files containing these shared secrets [19].

Obviously, *knockknock* also has the same limit for scalability as the previously described implementations. Additionally its' performance is limited by using Python as programming language and relying solely on a log file for processing incoming requests. Finally, it has the serious disadvantage of requiring root privileges on the client-side in order to be able to manipulate the TCP Header fields of the sent requests.

5. Knock—Another work in the area of port-knocking is the software *Knock*, an implementation of the *TCP Stealth* concept targeted at the Linux platform. *TCP Stealth* was developed in the scope of Julian Kirchs' Master's Thesis at the Chair for Network Architectures and Services at the Informatics Faculty of the Technical University of Munich. Generally also applying the idea of authenticating TCP packets on the kernel-level, this approach has the unique characteristic of not only authenticating the first SYN packet or the TCP Handshake, but being able to protect the entire communication by also authenticating the client and first bytes of the TCP payload for subsequent packets.

However, this also means that *Knock* is affected by a similar overhead situation as *SilentKnock*, because it processes *every* TCP packet leaving the client and *every* TCP packet reaching the server. Apart from that, it also requires a kernel-patch and linking networking-enabled applications against a library called *libknockify* in order to deploy this solution. As well as the other implementations described above, *Knock* relies on a symmetric algorithm with shared secrets to provide authenticated messages and is therefore affected by the same scalability limitations.

Comparison—To compare the approaches described above, they were judged in terms of how high the expected processing performance would be in a real-world deployment (Performance), how much overhead the implementation requires in terms of communication overhead (Overhead), the expected resiliency against relevant and probable security threats, and finally in terms of the increase in deployment as well as management complexity and overhead caused by employing the given port-knocking concept (Scalability). In each category the compared implementations receive a rating between

++ and – to provide a basis for the later analysis of the implementation presented in Chapter 3.

As shown in Table 1.1, there are port-knocking implementations with good characteristics especially in regard to performance and security. However, providing these properties while maintaining a low overhead is not trivial. Following the evaluation of the listed designs, it is clearly visible that additionally realizing the port-knocking concept in a scalable way is a hard challenge and is therefore not considered in the examined implementations.

Implementation	Performance	Overhead	Security	Scalability
knockdaemon	o	--	-	--
SilentKnock	++	o	o	--
fwknop	++	++	++	--
knockknock	+	++	++	--
Knock	++	-	++	--

Table 1.1: Comparison of well-known port-knocking implementations

All of the evaluated implementations rely on some kind of client-specific data being present on the server. This requirement provides an easy way to realize strong cryptography and therefore simplifies the implementation of a high security level, but eliminates the possibility of scaling port-knocking to a cloud-sized environment. Since cloud providers often employ edge servers to handle the immediate client connections, deploying one of the described implementations would require a synchronization of this client-specific data and therefore impose an immense management overhead from an administration point-of-view.

We believe this lack of scalable implementations to be one of the key reasons for the low adaption of port-knocking as part of real-world security concepts in the industry.

1.3.4 Summary

As shown by a few of the aforementioned implementations the compromise between security and acceptable overhead in terms of introduced latency and deployment overhead is a hard challenge.

For port-knocking to be usable in a production environment it needs to considerably improve the overall security of the system without a noticeable impact on the user-experience and without severe disruption of the existing cloud provider infrastructure. This requires a zero-configuration deployment for the client, which should become part of the application in the form of a library so the use of port-knocking is completely transparent to the user. Additionally, a distributed verification system to authenticate clients without having to synchronize client-specific data over all edge-servers is necessary.

Extensive research suggests that no such implementation is diffused in industry or academia. The following thesis will present an approach to unite the aforementioned requirements in an implementation usable in a real-world industry deployment.

Chapter 2

Background

The following Chapter presents the most important criteria for a production viable implementation of the port-knocking concept along with different approaches to fulfill these requirements using industry-proven algorithms and technologies. Each approaches will then be compared against others in respect to delivering the expected results regarding the given criteria. Additionally, every approach will be judged by the influence exerted on other parts of the architecture regarding aspects such as implicated restrictions or imposed performance degradations.

2.1 Security

The security considerations of the port-knocking layer are among the hardest challenges of choosing a specification for the targeted cloud-sized scale environment. Since an increase in security has a direct effect on performance and complexity, one has to consider thoroughly which technologies are worth their trade-off regarding their impact on latency, bandwidth, and processing power. It is thus important to make a thoughtful decision which of the many available concepts and technologies could and should be used while developing a solution for this new layer of network security.

2.1.1 Encryption

Data confidentiality is often desired in secure systems. This also applies to port-knocking, since an adversary sniffing on the network should not be able to read the clients' identity from the port-knocking protocol. When it comes to keeping prying eyes away from confidential data, it is almost guaranteed that the discussion converges towards a solution using encryption, which has two generally accepted concepts: Symmetric or Asymmetric Encryption [21].

Symmetric Encryption—For symmetric encryption, the involved parties need to establish a shared secret first, since by definition both need to have the same key to encrypt or decrypt the data [22]. This approach is widely used, including hardware support for algorithms like "Advanced Encryption Standard" (AES) [23] and therefore very simple, secure, and cheap in terms of processing power and bandwidth requirements. This requires the secret to be shared with the clients, which comes at the cost of deploying and managing these shared secrets across servers. We describe this approach in detail in Section 2.3.

Asymmetric Encryption—Choosing not to accept the downsides of an architecture based on distributing shared secrets across servers leaves us with an approach based on asymmetric cryptography [22]. This scheme includes two different keys based on the same generation scheme: A public key for encryption and a private key for decryption. Using this way of providing confidentiality for the transmitted request provides the possibility of handling user authentication and/or authorization at the edge-servers without the need of deploying any client-specific configuration on these servers by basing this process on cryptographic verification measures such as X.509 certificates [24].

Typical algorithms that are widely believed to be secure, and on which this so called public-key cryptography is based on in modern environments are RSA (Rivest, Shamir, Adleman) [25] and ECC ("Elliptic Curve Cryptography") [26]. Following the path of asymmetric security entails two major challenges that need to be solved in order to reasonably apply this type of cryptography. Foremost, public-key techniques are unable to securely encrypt message sizes larger than the chosen key-size of the key-pair [27]. Furthermore, they also require secure transportation of the recipient's public key to the sender of the message via either another communication channel or in a secure way along with the message, which inevitably increases either deployment complexity or the total packet size and therefore latency.

Hybrid Encryption (Asymmetric & Symmetric)—In light of these obstacles, the general consensus on how to make the use of asymmetric cryptography viable for transmission of messages with lengths exceeding the key-size is by combining it with symmetric encryption [27]. This technique works by exploiting the capabilities of public-key mechanisms to enable both parties to agree on a secret and then using a "Key-Derivation Function" (KDF) to generate an encryption key for use with a symmetric encryption algorithm like AES [27]. This procedure is known as *Key-Exchange* and can be done out of the box using the RSA cryptosystem. ECC however needs to be extended with the well-known and proveably secure "Diffie-Hellmann" (DH) [28] algorithm to be able to guarantee a secure key agreement based on elliptic curves.

The combination of both methods of encryption requires a small additional overhead

in message size compared to using only symmetric or asymmetric cryptography. It also has the advantage of providing relatively high performance without impacting the flexibility of the intended authorization mechanism [21].

2.1.2 Integrity

Ensuring that no other party except the server and client components involved in the port-knocking communication can successfully decrypt and interpret the request is an important step on the way to a secure solution. However, achieving this goal requires the implementation of a few additional security principles. A critical property of every secure system, which encryption alone cannot guarantee, is the integrity of the request data. Attack scenarios exist, in which an attacker can defeat a security mechanism without ever having access to the clear-text of the communication by tampering with predictable ciphertext. Vulnerabilities of this type can typically be mitigated by authenticating the message using a cryptographic hash function or by cryptographically signing the message, which causes any change to the message to inevitably invalidate the hash or signature [29].

The most commonly used technologies are "Hash-based Message Authentication Code" (HMAC) [30] and digital signatures using either RSA [25] or ECDSA ("Elliptic Curve Digital Signature Algorithm") [26] which is a variant of the well-known DSA algorithm, enhanced with the strong security capabilities of elliptic curves. Choosing signatures over message authentication codes has the benefit of additionally including a way to verify the authenticity of the message by guaranteeing the sender's identity, but comes at the cost of a larger message size, and, as mentioned in Section 2.1.1, the overhead of transporting the sender's public key to the receiver.

2.1.3 RSA vs. ECC, ECDSA, ECDH & ECIES

In the preceding sections, RSA & ECC were mentioned as industry-standard cryptosystems for asymmetric encryption, as well as for signing and verification of messages. While both of them can be used to achieve the same goals, the low-level technical implementations are based on completely different computations: While RSA relies on the mathematical assumption that the multiplication of large prime numbers is much faster and easier than reversing the process using factorization [25], ECC relies on the assumption that point addition on elliptic curves over finite fields is easy and fast but reversing it, known as the elliptic curve discrete logarithm problem (ECDLP), is hard [26].

RSA—According to the specification of the RSA cryptosystem, the creation of a key pair consisting of the *private key d* and the *public key e* with a key-size of *n* bits follows

the equation:

$$\forall m : \quad (m^e)^d \mod n = m$$

RSA: Encryption & Decryption—The encryption therefore works by applying a defined secure padding function onto the clear-text message M and converting it into an *Integer* m. Using the public key of the receiver e and the formula

$$c \equiv m^e \mod n$$

the sender can then obtain the ciphertext c, which is sent to the recipient for decryption.

For the decryption the recipient employs his private key d and the equation

$$c^d \equiv (m^e)^d \equiv m \mod n$$

to derive the original padded and converted cleartext m from the received ciphertext c, from which he is then able to reconstruct the original message M under the assumption that sender and recipient are using the same padding function [25].

Since plain RSA is susceptible to several types of context-based attacks including the *chosen-ciphertext attack*, which is very popular in the IT-Security community, it should always be used in combination with a secure padding-scheme such as the "Optimal Asymmetric Encryption Padding" (OAEP) to provide semantic security under IND-CPA ("Ciphertext Indistinguishability Under Chosen Plaintext Attack") [31].

RSA: Signatures—RSA directly specifies algorithms for generating and verifying cryptographic signatures. For this procedure, the sender needs to hash the message M using a secure cryptographic hash function $h(x)$ and append the signature calculated using the formula

$$s = h(M)^d$$

(where d corresponds to the sender's private key) to the original message. Assuming once again that both parties use the same hash function, the recipient can then verify the transmitted signature using the sender's public key e by checking if it satisfies the equation [25]:

$$h(M) \equiv s^e$$

ECC—Unlike RSA, ECC itself does not specify procedures for neither signature generation nor key exchange. Instead it defines a mathematical framework for cryptographic functions by defining the point addition operation. Given that both input points are on the curve, the result of point addition will be another point on the same curve. Combining this operation with a specified elliptic curve and a corresponding *base point* (G)

on the curve — both chosen for their cryptographically valuable mathematical properties — creates a cryptosystem allowing for all necessary operations to fully implement asymmetric cryptography.

This yields the formula:

$$Q = dG$$

where the public key Q is a point on the curve obtained by adding the base point to itself d times. d is an integer of the given key-size representing the private key [26].

ECC, ECDH, ECIES: Key Exchange & Encryption—For asymmetric encryption using elliptic curves, the concept needs to be extended with a technique to establish a secret with another party. This needs to be done while only knowing the other party's public key. Once a secret is established, it is possible to derive a symmetric encryption key from it, which in turn can be used with AES, for instance. To calculate a secret only known to the involved parties, both need to obtain the public key of their counterpart via a reliable but not necessarily confidential channel, for example using certificates. Assuming both are using the same elliptic curve as the basis for their calculations, multiplying their own private key with the public key of the counterpart would give both parties the same point on the curve. If now both participants interpret the x-coordinate of the calculated point as the secret, they have successfully completed a key-exchange using the *Diffie-Hellman* algorithm based on elliptic curves — called ECDH [32].

A small example to illustrate: Given the counterparts A with private key d_A, public key Q_A, and B with private key d_B, public key Q_B both parties can calculate the secret:

$$(x_s, y_s) = d_A Q_B$$

$$(x_s, y_s) = d_B Q_A$$

This calculation is correct based on:

$$d_A Q_B = d_A d_B G = d_B d_A G = d_B Q_A$$

with G being the base point of the same elliptic curve for A and B. In case identification of only one party is required, this hybrid encryption technique can be extended by using an ephemeral key-pair which is intended for short-term use and newly generated at the beginning of the encryption process. Using this variation of the ECDH algorithm called ECDHE ("Elliptic Curve Diffie-Hellman Ephemeral"), the whole cryptosystem gains the property of being IND-CCA ("Indistinguishability Under Chosen Ciphertext Attack") and IND-CCA2 ("Indistinguishability Under Adaptive Chosen Ciphertext Attack") [33] secure and is standardized under the name "Elliptic Curve Integrated Encryption Scheme" (ECIES) [34]. The identification of the party using the ephemeral key pair for encryption can be done using a certificate-based mechanism contained

in the encrypted message which implies a communication overhead for the benefit of increased security.

ECC, ECDSA: Signatures—The process of generating and verifying signatures using ECDSA, an implementation of the DSA ("Digital Signature Algorithm") [35] algorithm using the mathematic properties of elliptic curves, is more complex than its RSA-based counterpart. In the following example, we re-use the two parties from above with their respective private and public keys. This time, however, A is signing a message and B is verifying the signature. In addition to the parameters already given, A needs the bitsize L_n of the integer order n of the basepoint G, which equals the key-size in bits. Having prepared all necessary inputs, the generation of an ECDSA signature is explained by the following scheme [26]:

1. Calculate a secure cryptographic hash over the message m: $e = HASH(m)$
2. Take only the L_n most significant bits of e, save them as integer in z
3. Choose a random integer k from $[1, n-1]$
4. Calculate $(x_k, y_k) = kG$
5. Calculate $r = x_k \mod n$
6. Calculate $s = k^{-1}(z + rd_A) \mod n$
7. The signature consists of the tuple (s, r) and can now be appended to the message

While executing this scheme, it is very important to ensure that k is a truly random value generated by a cryptographically secure random number generator. Using the same k for two different messages known to an attacker allows for simple calculation of the private key, thereby compromising the entire security system based on this key. This exact mistake has been made by Sony Computer Entertainment and led to the recovery of the private key used to sign Playstation 3 console games, rendering the entire security ineffective and enabling uncontrollable piracy opportunities [36, 37]. Another way of preventing this kind of exposure is by implementing deterministic signatures. This approach removes the risk of insufficient entropy by deriving k from a combination of the message m and the private key d, ensuring that different messages will never be signed using the same secret integer k [37].s Another important algorithmic peculiarity is that neither r nor s are allowed to equal 0. If this rule is broken, the calculation has to start over from choosing a new random integer k and recalculating r and s until both of them differ from 0 [26].

Assuming once again that A and B are using the same curve, basepoint, and hash algorithm and B obtained the public key Q_A from A in a secure and reliable way, B can verify the signature generated by A using the following scheme [26]:

Bits of Security	**Symmetric algorithms**	**RSA key-size [bits]**	**ECC key-size [bits]**
80	2TDEA	1024	160
112	3TDEA	2048	224
128	AES-128	3072	256
192	AES-192	7680	384
256	AES-256	15360	512

Table 2.1: Comparison of security strengths of symmetric cryptography, RSA, and ECC. Adapted from Certicom Corp et al. [1]

1. Calculate a secure cryptographic hash over the message m: $e = HASH(m)$
2. Take only the Ln most significant bits of e and save them as integer in z
3. Calculate $w = s^{-1} \mod n$
4. Calculate $u_1 = zw \mod n$
5. Calculate $u_2 = rw \mod n$
6. Calculate $(x_k, y_k) = u_1 G + u_2 Q_A$
7. Verify if $r \equiv x_k \mod n$

To make this process reliable and secure, there are obviously many necessary checks, such as *Verify if r and s are in the valid interval* $[1, n-1]$, etc. These details, together with the mathematic proof of the correctness of the ECDSA signing and verification algorithms, have been omitted at this point, since they fall out of this thesis' scope due to their extent and mathematical complexity.

Comparison—Consequently, both cryptosystems have advantages in different types of operations. For Instance, to achieve a security level of 128 bits, the required ECC key-size would be 256 bits, while RSA would require a key with a length of 3072 bits length. A comparison of widely used key-sizes and their security level for RSA and ECC is shown in Table 2.1.

Since signatures for RSA as well as ECC based cryptography are dependent on the private key size, the required minimum signature size for the same security level for both, RSA and ECDSA signatures, differs by orders of magnitude. The expected signature size for a security level of n bits equals — depending on the chosen saving format —

RSA key-size [bits]	RSA signature size [bytes]	ECC key-size [bits]	ECDSA signature size [bytes]
1024	128	160	40
2048	256	224	58
3072	384	256	64
7680	960	384	96
15360	1920	512	128

Table 2.2: Signature sizes for RSA & ECDSA [2]

at least $\frac{n}{8}$ for RSA and $\frac{2n}{8}$ for ECDSA [2]. This fact results in the signature sizes for comparable security levels as seen in Table 2.2.

Of course, the reduced space requirements of the ECC based algorithms come at a cost: Computational complexity and thus performance. While ECC can outperform RSA in terms of key generation speed and for very large key sizes even in terms of time required for the signing operation, the verification of signatures takes at least 20 times longer for ECDSA based signatures. [38]

2.1.4 Other security features

Apart from focusing on the cryptographic principles found in almost every security-related implementation, one should take care not to forget about some very simple and often quite effective generic vulnerabilities such as *replay attacks* [39] or *hijacking the connection*. To prevent a "Man in the Middle" [40] from hijacking the open port by, for instance, changing the source IP address in the port-knocking packet, it is of critical importance to include the client's address in the payload, which cannot be modified by an attacker in an undetectable way since it is encrypted and authenticated using strong cryptography. In order to prevent unauthorized parties from recording and replaying a legitimate port-knocking packet it is recommended to rely on one of the proven countermeasures widely known in the IT-Security community, for example including a timestamp in the authenticated payload [41], using *OTPs* (One-Time Password) [42], etc.

Finally, depending on the architecture and exact implementation of the port-knocking layer, one should keep in mind that the port-knocking service itself could become a security liability. If the code responsible for handling the knock requests is directly accessible from remote machines and/or runs with elevated privileges, hardening this

critical piece of software and maintaining its secure state is of utmost importance. A compromised best-case implementation should not be able to interfere with the overall system security in a way that renders it less secure than it would be without port-knocking.

2.2 Performance

Apart from security considerations, the next highly important property of the resulting specification is the impact on performance. As seen in the list of existing implementations presented above, there is more than one way to transmit and process the knock-requests, which calls for a comparison of the possible and reasonably implementable techniques for this scenario.

2.2.1 TCP vs. UDP

The first item on the list of performance-relevant decisions to be made is the transport layer protocol to use for the port-knocking protocol. For requests that have to travel through a multitude of network hardware it is not possible to abuse any standard-protocol with a specific purpose to implement our scenario. This is because well-developed security infrastructure will notice this illegal use of standardized communication techniques, which will probably result in refused or dropped packets and therefore loss of reliability [9]. This leaves the two general-use transport protocols: The “Transmission Control Protocol” (TCP) [43] and the “User Datagram Protocol” (UDP) [44].

TCP—TCP is able to ensure that packets reach their destination by providing mechanisms for packet loss detection and automatic retransmission of lost data, while in the case of UDP, the higher-level protocol — in this case the port-knocking layer — would have to implement it itself. While this would generally speak in favor of TCP as the transmission method of choice, one should not forget that this reliability comes at a cost: Before any data can be sent, TCP requires an established connection — meaning a handshake requiring at least 3 packets has to take place first [43]. Out of these three packets only one is able to transport useful data, which implies that the mean loss ratio is $\frac{2}{3}$. These unnecessary packets increase the establishment delay of *every* connection protected via port-knocking. So, depending on the reliability of the underlying network, UDP can be more efficient if packets don’t get lost too often, since it does not require any connection establishment before being able to transmit data [44].

The only way to send a knock-request in a single TCP packet would be to abuse the *SYN* packet, which is not supposed to contain any data since it only represents the

initialization of the handshake [43]. Therefore, it would most probably be dropped by firewalls between client and server [9], making the whole specification unreliable.

UDP—The UDP based implementation however is not affected by this specific problem but turns detecting lost packets into a considerable challenge. A possible solution would be for the server to confirm received port-knocking requests, which not only creates a security threat by exposing unnecessary information to a possible attacker, but also has a negative impact on performance by doubling the time of the communication for the average case. The other solution would require the client to verify the success of the port-knocking and in case of failure, retry after a timeout. Defining an acceptable threshold for this timeout, however, is not a trivial task since one can never be certain about the network conditions for every client affected by this security measure and using high timeouts impacts the performance negatively.

2.2.2 Packet size

Besides the choice of the transportation protocol, another highly important metric is the general size of the request packet. Independently from using TCP or UDP, very large requests could result in fragmented packages, which would impose new problems for the overall system performance. Fragmentation, by definition, increases the communication latency because of the fact that multiple packets need to be routed to the recipient. Additionally, these fragments can arrive in the wrong order and — especially in the case of UDP — require additional processing to be reassembled correctly [45]. Furthermore, splitting a request into multiple fragments obviously increases the probability that the request has to be retransmitted since the loss of one fragment requires resending the whole packet.

The challenge is to find a secure implementation that minimizes the probability of being affected by the fragmentation threshold for every involved network device in the route since the performance penalty resulting from fragmentation would render the whole concept too expensive (from a latency based point-of-view) for the additional security that it provides.

2.2.3 Processing incoming packets

While solving the challenge of ensuring the requests reach their destination in an optimal way, one should also consider how to process the received packets efficiently. As seen in Section 1.3.2 of the introduction, there are very diverse techniques for capturing and processing these requests. Leaving aside strategies that would cause significant deployment or administration overhead in a real-world scenario like, for instance, patching the kernel on the server-side to filter out the port-knocks [20], there are three possible

implementations: Using the *Firewall log* to dump incoming packets that match a defined pattern, running a *full-featured service* on the server, or using a *RAW socket* [46] provided by the operating system.

Firewall log—Firewalls supporting logging could be configured to log refused packets in their logfiles. These logfiles could then be monitored for port-knocking protocol packets by a lightweight daemon on the server without any need for elevated privileges with the only exception of the thread responsible for opening the ports on the firewall after a successful port-knock. In addition this approach also prevents any direct contact between clients and the port-knocking software, thereby minimizing the probability of an attacker successfully exploiting a vulnerability of the port-knocking implementation itself. However, consistently using a file mostly placed on a hard-drive for frequent write and read operations will probably affect the overall system performance by generating a considerable amount of unnecessary load on the storage system. Additionally the log file might grow uncontrolledly — if not handled in an efficient and secure way — filling up the partition it is located on, which could have security implications of its own.

Dedicated service—The approach of running a service dedicated to only handling port-knocking requests would probably be the most comfortable one from a development perspective and also the fastest implementation in regards to packet processing speed since the filtering of non-relevant packets would be done by the operating system. On the other side, such an implementation would need to have its own port open at all times which makes it obvious that there is a port-knocking service running on the server and therefore creates a new attack surface [47]. While this might not be considered a significant security risk by many, one should not forget the initial goal of this security layer: Hiding services running on the server. Giving away the information, that there are hidden services — which could easily be detected by running a port-scan, the very technique port-knocking aims to deflect — could motivate a resourceful attacker to further pursue this particular server as target instead of moving on to the next possible victim.

RAW socket—While the last viable option — using a RAW socket — is the most complex to implement and imposes a great challenge when it comes to efficiently filtering out the irrelevant packets, it also enables an enormous flexibility in how exactly the incoming traffic gets processed. Provided that this approach is implemented with skill and caution, it is possible to implement an extremely secure solution while maintaining a high level of performance without interfering with the operating system or kernel itself. Acquiring a raw socket requires elevated privileges, which can however be dropped immediately after initialization. This leaves only the port-opening thread running with elevated privileges, which would be necessary in every implementation anyway [48].

2.3 Deployment Complexity

Finally, after carefully considering all security and performance relevant properties, it should not be forgotten that there is an important abstract attribute highly relevant for success in a real-world industry: The complexity of deploying the product on the server infrastructure, as well as the complexity of integrating it into existing client software.

As the additional security benefit resulting from deploying a port-knocking solution is restricted to hiding services, the associated impact on normal operation for the customers' application needs to be kept as small as possible. This is because in a modern environment, the services additionally protected by this new security layer are mostly considered secure from a business point-of-view, the expected allocation of resources to integrate port-knocking into the existing environment will probably be proportionally low.

Server side—The decision affecting server side complexity the most is definitely how user identity management is implemented. In general one can assume that there are two major distinct ways to identify or authorize users: Storing per-client information on the server [49] or using certificates [50].

Depending on the infrastructure of the provider, using customized client profiles can make sense if they already synchronize data sets over all relevant edge servers. In this case, extending the synchronization to include the client configurations for the port-knocking layer would be a very secure, cheap, and high performing solution. However, this would probably apply to only a few of the possible customers for such a product. In most cases, a deployment requiring fast and reliable data synchronization between all edge servers would render the complete security solution inviable.

For these very common environments, the only viable alternative would be an authorization technique based on a trust anchor that does not require any further client-specific information. This implementation could be achieved by using, for instance, X.509 certificates backed by strong cryptography, which would provide a standardized environment for employing user authorization [51]. Since most service providers already use certificates for some type of secure communication, they already have the infrastructure in place for managing X.509 certificates, leaving a very minor cost position for extending their certification authorities to issue certificates for the port-knocking security layer. However as illustrated in Section 2.1.1, the inevitable consequence of this approach is a considerably higher packet size compared to using symmetric encryption and client profiles. It also suffers from a synchronization problem: keeping the certificate revocation list up-to-date on *every* involved node. We argue that this is a minor drawback as the infrastructure for it is already available as part of the PKI (Public Key Infrastructure) [52].

After careful consideration of the high-level requirements of this specification, a thought-

ful decision about the low-level implementation has to be made: The programming language. Since many online service providers use hardware-based security appliances right in front of their edge server, it could be a requirement to deploy the port-knocking service on these hardware firewalls. Since these devices mostly run a stripped-down and hardened operating system with very basic support for only a few programming languages [53], this possible requirement should be considered before starting development.

Client—Whichever approach is chosen for the server-side implementation, there are a few very important considerations for the complete deployment scenario regarding the client software. If, for example, the resulting client part needs elevated privileges or manual user configuration, this could scare away affected users from using the provided service and consequently make the implementation less probable to be adapted by the industry, resulting in a niche existence for the product.

Assuming the customer has access to the source code of the client software in use — or is just generally able to induce a change of its implementation — the most favorable realization of the client side for a port-knocking concept would probably be a library in one or more widely used programming language(s), which could be easily linked to the original client software to execute the port-knocking just before the initial application communication. In the rare case that the service provider has no direct influence on the client software, the client deployment would have to fall back to a separate launcher, which first ensures that the port is accessible by executing the port-knocking sequence and launches the actual client software once the port has been opened. Another possible solution for the cases related to legacy-software could be to employ a variant of a proxy based on a *VPN* (Virtual Private Network) protected by the port-knocking security layer.

Chapter 3

Implementation

In this Chapter a full specification for a port-knocking solution based on the principles and technologies introduced in Chapter 2 is proposed. Following the specification, the architecture for our implementation of this specification, called *sKnock*, will be described in detail.

3.1 Requirements & Specification

As previously explained, the goal of this work is to develop a viable concept for port-knocking as an additional layer of security that scales up to the demands of today's large online service providers.

In the following section major requirements of such a concept will be presented along with the implemented or suggested approach. While these decisions add up to a complete specification, every aspect of this design will be reviewed according to the requirements initially stated.

As the most important aspect for making port-knocking a viable security component for large-scale service providers, all design choices presented in the following sections were made while considering the ability of deploying this software on millions of devices without causing excessive management overhead on the provider's side. Since these choices imply restrictions on the available processing and security mechanisms, a great effort has been made to create a design that is consistent with most scalable compromises which still keep a high level of security and performance.

3.1.1 Network Communication

One of the first decisions was choosing UDP over TCP as a communication protocol for port-knocking. The main argument for this choice was the limitation of TCP to only

be able to transmit actual data after successfully establishing a connection — which basically translates to (assuming no packet loss) wasting at least two packets for each knock request, if the operating system is able to combine the last handshake packet (client ACK) with useful data, and at least three packets otherwise [54]. With at least three sequential packets between server and client for every port-knocking request, the average delay caused by the TCP-based implementation would not be proportional to the expected security improvements and therefore not economically justifiable. Using UDP for transmitting the port-knocking requests yields the advantage of being able to send the port-knocking request immediately in the first packet. Nonetheless, this also implies that the port-knocking client software is responsible for detecting lost packets. Depending on the configuration of the server firewall, the client can detect lost packets by not receiving any response in case the firewall is configured to send out notifications for rejected communication requests. If the firewall silently drops packets related to communication requests targeted at closed ports, the client has no way of differentiating between lost packets and failed authorization attempts and is left with no choice but to wait for a configurable *timeout* and retry. Since the Internet has been improving the reliability of its network connections substantially during the last decades [55], we are optimistic that the cases in which a delay is caused by lost packets will account for only a small fraction of all port-knocking attempts and that thus, in spite of the less efficient packet loss handling, a lower average latency can be achieved by using UDP. Together with the parametrization of the optimal timeout for retrying port-knocking attempts this assumption is evaluated in the next Chapter.

3.1.2 User Authorization

For our design, there was no need for implementing a full-featured user identity management. The relevant user-related design consideration was to solve the challenge to efficiently authorize legitimate users while keeping unknown entities from contacting the protected services.

The approach implemented in this specification is based on X.509 certificates to authorize clients, because this technique requires only one certificate of a *Certification Authority* (CA) and the most recent *Certificate Revocation List* (CRL) to be stored on the server-side device responsible for handling the port-knocking. To mitigate the risks implied by using a trust system based on certificate authorities, our design employs *certificate pinning*, a technique where only certificates from one or more given authorities are accepted for a specialized use-case.

The actual authorization data contained in these client certificates is a UTF-8 encoded, comma-separated list of allowed port and protocol combinations as, for example, *1.2000,1.3000,0.8000*. In this scheme, 1 corresponds to TCP, while 0 represents UDP as protocol. This authorization string is stored under the *OID* (Object Identifier) for the

TUM, *1.3.6.1.4.1.19518* as *Other Name* in the *Subject Alternative Name* (SAN) extension of the certificate and represents the only relevant data considered in the authorization process. The chosen OID was an arbitrary decision and can be freely replaced by any available OID to suit the needs of the target environment.

This feature gives the service provider the freedom to deploy their application using user-specific or application-wide certificates, depending on the requirements of their environment. Including additional information such as a user ID has no adverse effects on the port-knocking protocol except for increasing the certificate size and therefore requiring a larger overall packet size. Of course one has to keep in mind that significantly increasing the packet size leads to a higher probability of exceeding the *Maximum Transmission Unit* (MTU) and therefore a higher risk of dropped packets for some clients. Furthermore this approach imposes no restrictions on the choice of identity management systems or certification authorities for the customer, as long as the mentioned authorization string is correctly encoded in the resulting certificates.

3.1.3 Security Suite

All security-relevant properties of this specification were developed with a minimal impact on packet size and therefore performance — specifically latency — in mind. As a result of the previous decision towards certificates, a given prerequisite of the security concept is the involvement of asymmetric encryption. Following this, we chose elliptic curve cryptography based on the NIST-P 256 curve for generating the required keypairs to use with the described authorization infrastructure. The security level of our cryptosystem based on these 256 bit long keys is about equivalent to the security of AES-128 or 3072 bit RSA as shown in Table 2.1.

To secure the actual port-knocking requests, the following scheme of iteratively adding security measures is applied on the packet consisting of one zero byte followed by the protocol and port number.

Counter Man-in-the-Middle—The request is extended by adding the IP address of the client network interface from which the port-knock will be sent to mitigate a *Man-in-the-Middle* attack by just changing the source address in the IP header of the packet on the wire and then hijacking the open port. However, this measure greatly impacts the ability to traverse networks which employ *Network Address Translation* (NAT), as it is the case with most private home setups.

Counter replay attack—The client appends a *timestamp* to the clear-text request, which is checked by the server, allowing it to deviate from the current time by only a small, configurable threshold to prevent *replay* attacks. Finding the optimal value for the

allowed time-delta depends on the security requirements of the environment and the time-synchronization capabilities expected from the deployment.

Signature & Certificate—Following the timestamp, the *client certificate* is appended to the request and the contents of the whole packet up until now including the certificate are signed using the *ECDSA* algorithm with the private key corresponding to the client certificate to generate a signature. Before appending the signature to the request it is padded to a deterministic size of 72 bytes[1] since ECDSA signatures can vary in length.

Encryption—To protect the sensitive information contained in the port-knocking packet, all previously mentioned contents are padded and encrypted using AES-128. The symmetric key required for this process is derived from a secret established using *ECDH* with the server's public key and an ephemeral ECC key-pair which is based on the same curve and separately generated for each request. The technique used to derive the symmetric key from the secret is called *HKDF* ("HMAC-based Extract-and-Expand Key Derivation Function") [56]. In order for the server to be able to decrypt the packet, it is necessary to append the public key of the ephemeral key-pair to the message.

Conclusion—By integrating the above mentioned security features we have step-by-step implemented a cryptosystem secure in regard to IND-CPA, IND-CCA, and IND-CCA2. This scheme is a variation of the publicly known "Elliptic Curve Integrated Encryption Scheme" (ECIES) with the major difference of using the digital signature (ECDSA) to authenticate the message instead of a specified message authentication code, since the certificate and signature has to be included for authorization purposes anyway. By using 256 bit as the key-size for our security model we believe to achieve the optimal compromise between security level and performance, especially in terms of acceptance by the industry.

ECC curve recommendation—As an improvement in the near future, it is recommended to replace the NIST-P 256 curve with the newer curve25519 as soon as the X.509 standard incorporates it and the curve is supported by all major platforms and certification authorities. This recommendation is based on the fact that since the NSA scandal of 2013, all cryptography-related information and technologies coming from regulatory bodies of the United States should be treated with caution [57]. Apart from being developed independently from any governmental influence, curve25519 has some advanced cryptographic properties that make it superior to most other curves in terms of security and performance. The only reason we chose the NIST recommendation for

[1] This depends on the key-size of the ECC key-pair. For the curve chosen in our specification (NIST P-256), the maximum signature size is 72 bytes.

now is that curve25519 is not yet part of the X.509 standard and therefore unsupported by most certification authorities which renders it practically unusable on commercial platforms [58].

3.1.4 Packet Design

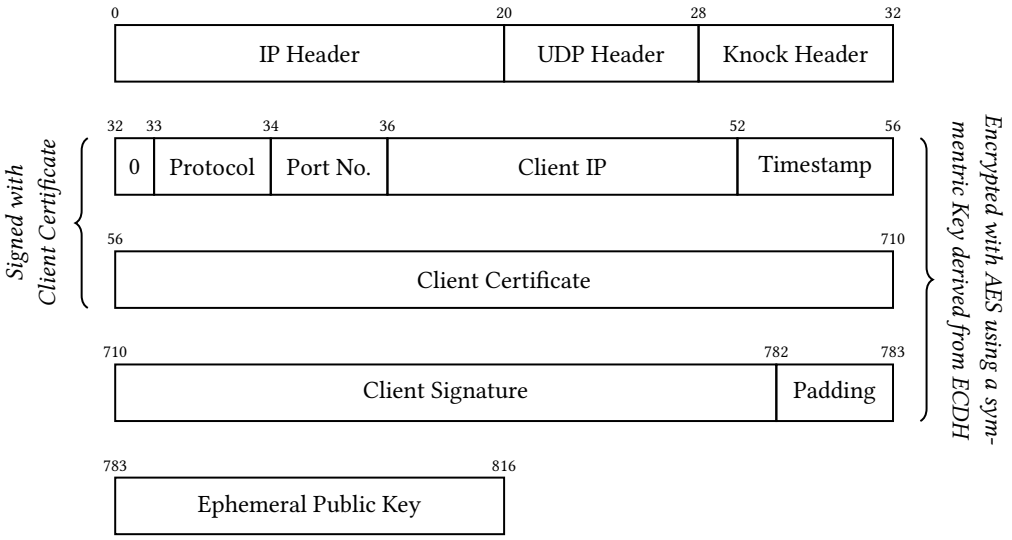

Figure 3.1: Example of a port-knocking request packet

Combining all solutions and measures necessary to fulfill the requirements stated above, there was one last optimization to achieve: Unify everything that has to be sent over the wire into a packet as small as possible. Since the packet design was generally optimized to yield the highest possible performance under the given environment, the second important parameter next to the overall size was the improvement of processing speed. This consideration led to the decision of including one extra byte of zeros in the encrypted payload, enabling the server to quickly verify if the decryption was successful. By implementing this extra check, the packet processing logic on the server can filter out most malformed, misdirected or incorrectly encrypted packets significantly more efficiently since an invalid packet would most certainly not contain the zero byte at the correct position.

An example of a port-knocking request packet can be seen in Figure 3.1 including description and byte sizes. The overall communication infrastructure was developed in a way to allow for a variable certificate size, since the length of an X.509 certificate can vary considerably depending on the usage of extensions and the employed certification authority. Also, in Table 3.1 an explanation of the relevant fields for this port-knocking specification can be found.

Field	Description
Knock Header	Contains a magic number to identify sKnock (1 byte) and the used version (3 bytes)
Protocol	Boolean value for either TCP (1) or UDP (0)
Port Number	The requested application port number
Client IP	IP address of the client to mitigate Man-in-the-Middle attacks. Needs to be 16 bytes because of IPv6 support
Timestamp	Client time to prevent replay attacks
Client Certificate	ECDSA certificate used by client to sign the request. Contains the clients' authorization for port-knocking
Client Signature	Signature to authenticate the request
Padding	Actual length of the signature, since ECDSA signatures are not deterministic and can be smaller
Ephemeral Public Key	Public key of the temporary ECDH key-pair generated by the client to encrypt the request

Table 3.1: Explanation of relevant Port-knocking packet fields

3.2 Architecture

In the following Section all major modules used in our implementation of the design specified in the previous section are presented. Every module is explained in detail by describing its requirements and functionality. At some points general design decisions are explained and the current state of development is assessed together with the intentions for future extensions at the time the code was written. An overview of high-level server and clients components can be taken from Figure 3.2.

3.2.1 General considerations

The work done in the scope of this thesis is intended as a proof-of-concept for a scalable port-knocking implementation that could satisfy the requirements of cloud-scale online service providers. To obtain a high probability of industry acceptance, every module described in this Chapter was designed with platform independence in mind. Since a great variety of devices are connected to the Internet, no security solution that aims to be deployed on a large scale can ignore any major operating system. While this property is absolutely crucial for the client side, where a platform-independent implementation would not require a lot of platform-specific code, the topic becomes more complex for the port-knocking server. As it contains a considerable amount of operating system dependent code, such as the startup module, the socket, or the certificate management, all of these modules need to be abstracted and implemented multiple times to enable compatibility across all relevant server systems. Even more challenging is the imple-

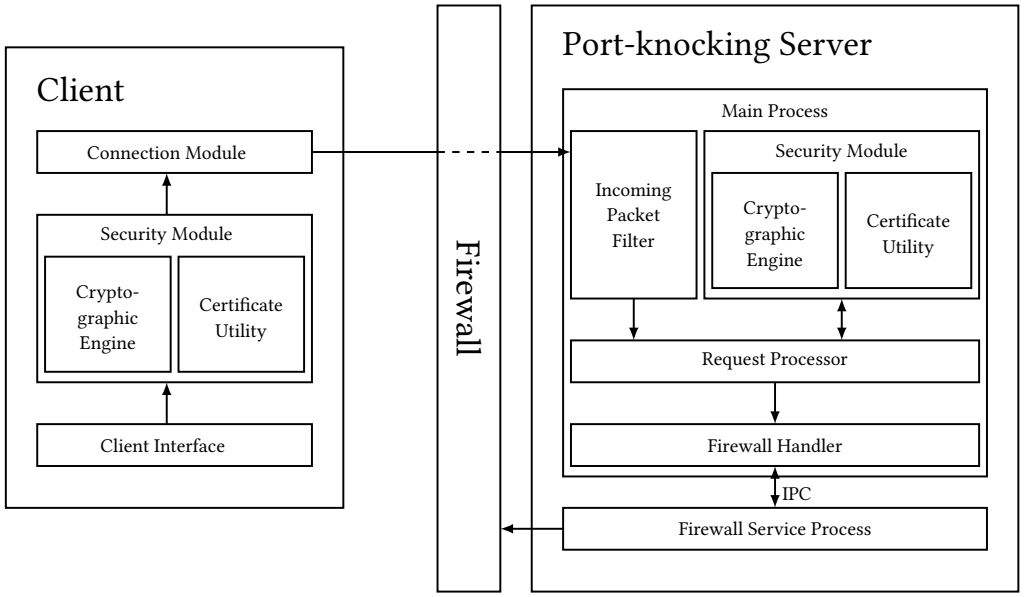

Figure 3.2: Important Components of the Implementation

mentation of the *Firewall* module, since it has to be flexible enough to support different firewall software on different platforms.

3.2.1.1 Programming language

In order to realize as many of the proposed features and components as possible, *sKnock* was written in Python, a programming language renown for its excellent efficiency in quickly developing software prototypes. However, for realizing a specification similar to the presented design, our recommendation would be to use a lower-level programming language such as C, as this allows deployment on hardware firewalls, security hardening, and extensive performance tuning up to the limit of the respective platform among other things.

3.2.1.2 Platform independence

To achieve the aforementioned goal of platform independence, the *Platform* module was created, among others, to provide general, platform-specific abstractions for all the relevant target environments. The goal of this concept is to enable the final version to be agnostic of all the platform-dependent implementation details by decoupling the high-level logic from the low-level implementation details from a software architecture point-of-view. This empowers users from every operating system with every imaginable firewall software to extend the code presented in this document with their platform specifics and to immediately deploy a port-knocking solution in their environment.

However, keeping in mind the time-constraint and scope of this work, the implementation presented in the following subsections focuses on environments based on the Linux operating system with OpenSSL as a main cryptographic library and iptables as system firewall. Although the low-level encapsulations were only written for the mentioned environment, the code is designed to allow for easy extension by only implementing these abstractions for other environments while not having to change the high-level logic of this port-knocking concept.

3.2.1.3 Firewall: iptables

As mentioned before, our implementation relies on the iptables firewall software to open and close the ports for legitimate port-knocking requests. The iptables firewall is configured using *chains* and *rules*. Every rule belongs to exactly one chain and is used to match packets to certain parameters and execute an action in case of a positive match. These matching parameters depend on the modules and extensions loaded at startup of iptables and include, for example, protocol, source and destination port, source and destination IP address, connection state, packet header flags, and many more. The same is also true for the available actions, which can reach from simple operations such as *ACCEPT*, *REJECT*, or *DROP* up to complex routines such as applying NAT to incoming packets or manipulating TCP Header fields. The main difference between these standard actions is, that *ACCEPT* will, as the name suggests, let the packet pass on to the application while *REJECT* will not let the packet through and additionally informs the sender by issuing a *icmp-port-closed* to the sender. *DROP* will behave just as *REJECT* with the difference, that it silently stops the packet without giving any information back to the source.

Iptables starts up with a set of default chains: *INPUT*, *OUTPUT*, and *FORWARD*. Modules and extensions are able to provide additional default chains that are associated with the respective module only, such as loading the *nat* module results in the *PREROUTING* and *POSTROUTING* chains being added. Packets traveling through the system are processed by iptables depending on their origin and destination. For instance, incoming packets which are destined for the local machine arrive at the INPUT chain. By using the name of different chains as action parameter, it is possible to create complex rule sets by conditionally concatenating multiple chains. In case the packet reaches the end of a chain it returns to the chain it came from; if the chain is a default chain, the default policy of that chain is applied. The three default policies of a regular iptables deployment are the simple operations mentioned before: *ACCEPT*, *REJECT*, and *DROP*.

3.2.2 Server

The following Subsections explain the core modules of the server architecture, ranging from the way incoming packets are captured, up to the logic responsible for finally opening the ports by modifying the current firewall rule set.

3.2.2.1 Server Interface

To provide a point of entry with easy access to the high-level functions of the server, we are using the abstraction methods located in the class *ServerInterface*. This class compacts the initialization process into a single startup method, which accepts all necessary and optional inputs as parameters. By using these parameters, it then instantiates all modules required for operation with respect to the necessary logical order and dependencies. The signal handling in the server interface also provides the possibility to run the port-knocking server as a background service using, for example, *start-stop-daemon* under Linux or the included service framework under Windows.

3.2.2.2 Configuration

As it is best-practice to provide a headless start for server-side services, a mechanism to initialize the required parameters from a configuration file is necessary. The reading and parsing of the config-file for the port-knocking service is done in the *Configuration* module, which then holds all the the user-specified settings in a global variable. For the options not set by the user or in case there is no configuration file at all, the implementation provides default values for all required parameters. The loaded settings object is then distributed by the Server Interface to every instance of classes requiring configurable parameters in their operation.

At the current status, following parameters are configurable:

3.2.2.3 Security

The security module located in the *Security* class encapsulates the certification utility described in Subsection 3.2.4.1 and the cryptographic engine described in Subsection 3.2.4.2 to provide high level interfaces for decrypting and verifying legitimate port-knocking requests. The verification process for incoming requests is implemented using the following scheme:

1. Call the *CryptoEngine* to decrypt the message
2. Verify that the first byte equals zero

Setting	Default Value
Minimum Knock-Packet length [bytes]	800
Time in seconds that the port remains opened	15
Threshold for the timestamp verification in seconds	7
Receive Buffer size [bytes]	1600
Signature size [bytes]	72
Certificate Revocation List file path	`certificates/devca.crl`
Certificate Revocation List URL	`https://home.in.tum.de/ sel/BA/CA/devca.crl`
Certificate Revocation List Update Interval in Minutes	30
Server Certificate file path	`certificates/devserver.pfx`
Server Certificate pass-phrase	`portknocking`
Firewall Policy	reject

Table 3.2: Port-knocking server configuration settings with default values

3. Verify signature and client certificate using the *CertUtil*

4. Check the timestamp considering the configured threshold

5. Extract the authorization string from the certificate and verify that the user has been granted access

Apart from checking incoming port-requests this module is also responsible for constantly updating the *Certificate Revocation List* from a deployment server. For this task, the security engine schedules the *UpdateCRLThread* to run repeatedly in a configurable interval.

3.2.2.4 Request Processing

Module: Listener—The logic responsible for processing the incoming packets was too extensive to be contained in a single class, which is why the *Listener* is one of two modules in this implementation containing multiple classes and requiring more than one thread. An overview over the processing logic is given by Figure 3.3.

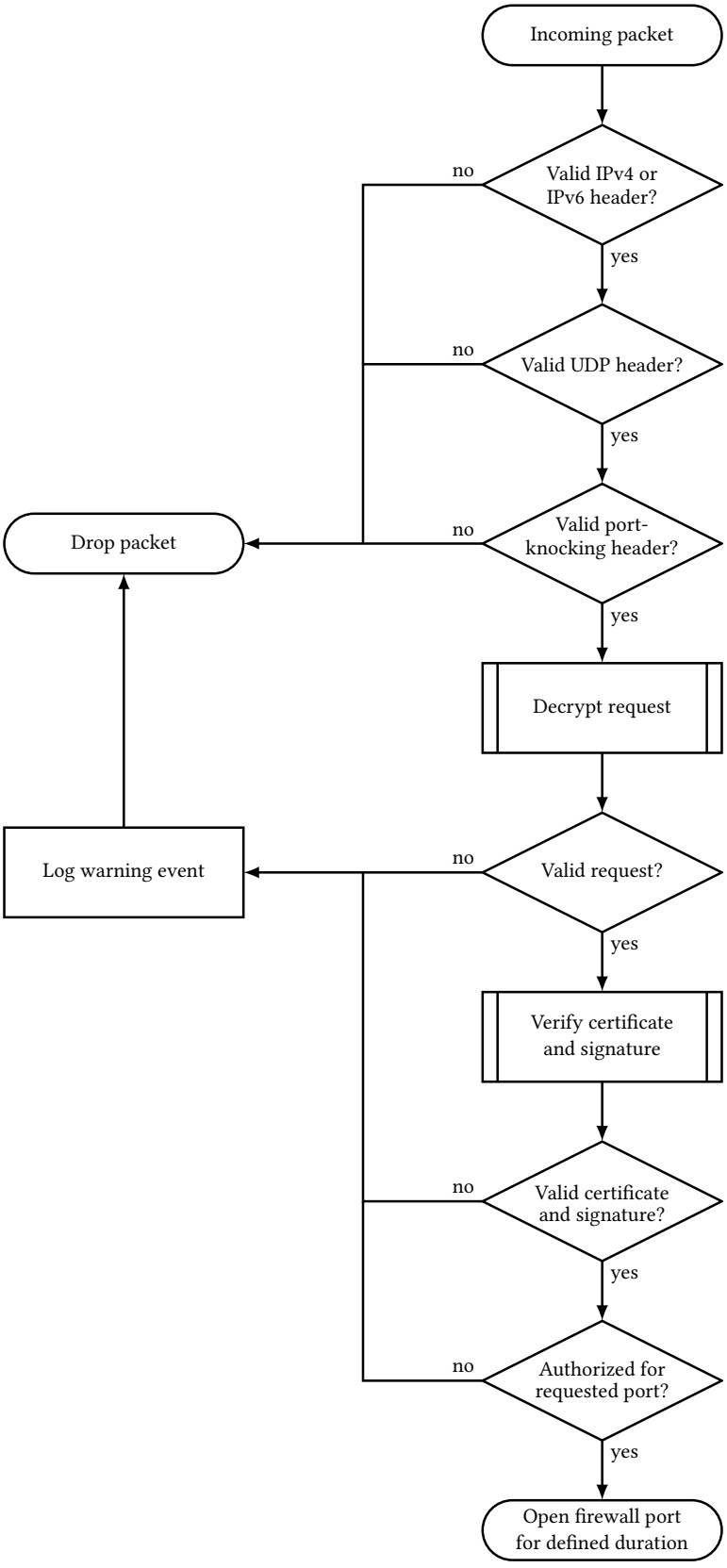

Figure 3.3: Packet Processing Algorithm

Knock Processor—At startup, the *KnockProcessor* class is responsible for setting up a raw socket using the python socket interface, which intercepts all incoming packets. To maximize throughput, only packets with a size larger than the minimum expected length for a valid knock requests are passed on to their own *New Packet Thread* for further filtering. This approach provides the possibility to implement a high-performance packet-filter without requiring the port-knocking service to be bound to a specific port.

New Packet Thread—The logic to efficiently separate possible port-knocking requests from usual server traffic is implemented in the *NewPacketThread* class. After a sizable amount of irrelevant packets has been filtered out just by matching their size, the packets that are big enough to potentially contain a port-knocking request are processed using the following scheme:

1. Skip the Ethernet Header
2. Process the IP Header
    - Determine if the packet is an IP packet at all
    - Check the protocol field to verify it's an UDP packet
    - Determine the IP version of the packet
    - In case of IPv4: Extract the IP header length
3. Skip the UDP Header
4. Check if the packet contains the magic number in the port-knocking header (first 4 bytes of the payload)
5. Verify that the Knock-Version (last 3 bytes of the port-knocking header) is compatible with the current version

If the packet survives this procedure it is considered a port-knocking request with very high probability and is therefore relayed to the *Process Request* module, which is designated for further processing.

Process Request Thread—Each of the threads assigned with the processing of a possible request has to decrypt and verify the received packet using the security engine first. Assuming this operation completed successfully, the logic in the *ProcessRequestThread* class is responsible for verifying that the client IP address contained in the port-knocking request matches the source IP address from the packet header. In case of a mismatch, a Man-in-the-Middle attack is assumed and the packet is dropped while writing a warning message to the configured logging environment. However, if the request is considered valid, the module verifies if the requested port has either already been opened or is in the process thereof, and spawns a thread to do so if necessary. This precaution is

implemented by storing a hash over the combination of port, IP version, protocol and client IP from each valid request in a list owned by the *KnockProcessor* instance. This list is checked for every request before kicking off the port-opening logic and the hash is only removed from the list by the port-closing logic after the port has been successfully closed.

3.2.2.5 Firewall Operations

Module: Firewall—The module responsible for handling the firewall operations to actually open and close the requested ports is the most complex part of the implementation, requiring a separate process and multiple distinct threads to fulfill all requirements regarding security and performance. Although every line of code in this implementation was written with platform independence in mind, time constraints led to a focus on Linux server environments with iptables as main firewall software. While this code can easily be extended for other firewalls on other operating systems, the following description focuses on the implemented solution.

Port Opening Thread—This thread is intended to be spawned by the thread responsible for processing a request in case of successful verification of the port-knocking request's legitimacy. Upon execution, the logic implemented in the *PortOpenThread* class appends the previously mentioned hash over the combination of the requested port parameters to the list of running port-related tasks stored in the KnockProcessor instance. It then tries to open the corresponding port by sending the respective request for a port-opening task to the high-level firewall interface and handles possible exceptions arising during this process. After successfully opening the port, it spawns the thread responsible for closing the same port after a period of time.

Port Closing Thread—Spawned by the thread described in the paragraph above, the logic contained in the *PortCloseThread* class starts by sleeping a configured period of time. After the waiting time is up, it tries to close the port opened by aforementioned logic via the same firewall interface. Upon successful completion of the port-closing operation, the thread removes the hash belonging to the current port parameters from the running tasks list in the Knock Processor before terminating itself.

Firewall—The high-level interface for executing firewall-related tasks is implemented in the *Firewall* class. It provides methods for initialization and safe tear-down of the complete Firewall module, while abstracting the complexity necessary for opening and

closing the firewall ports in an atomic[2] and thread-safe[3] way.

At startup, the *Firewall* module spawns the process responsible for executing the actual commands leading to firewall operations together with the communication pipes required for sending tasks to this process. Furthermore, the current active rule-set of the firewall is backed up and an emergency rule is inserted to enable remote access in case of an unexpected service failure (for debugging purposes).

During normal operation, the interface provides methods for asynchronously opening and closing ports on the current firewall. These methods are engineered to guarantee that the firewall always resides in a consistent state while the requested tasks are executed sequentially although the incoming requests are being received in parallel.

At shutdown of the port-knocking service, this module ensures to reverse all actions performed during the operational time by restoring the firewall rules to the previously backed-up state.

Linux Service Wrapper—This wrapper located in the *LinuxServiceWrapper* file contains the logic for listening and processing the commands sent over the pipe from the main process. The aforementioned firewall process uses the methods contained in this wrapper to decode the requested tasks and execute them using a helper class for the respective firewall. By decoupling the firewall operations into a separate process, the risk for an attacker gaining elevated privileges through a possible vulnerability in the port-knocking implementation is significantly reduced, since only these tasks can be executed by the firewall process.

Other platforms—The way multithreading, multiprocessing, access control, resource distribution, etc. works differs substantially between the available platforms. Due to this fact, the encapsulation of the processing logic for the firewall operations has to be developed separately for each platform since leveraging python's multiprocessing library only allows for the abstraction of basic inter-process communication and process management.

Safe inter-process communication—Every requested task is assigned a task ID consisting of 32 characters before being sent to the firewall process through the interface in the *Firewall* class. The process encapsulating the firewall service then answers with the task ID on the reverse communication pipe on successful completion of a command. By ensuring that the task ID returned by the firewall process matches the one sent with the last command, the interface can verify that the tasks were executed in the right order

[2] Every operation is either executed completely and consistently or not executed at all.

[3] Multiple parallel threads are not able to interfere with each other when executing firewall commands, so the state of open ports is always consistent.

without interference of other processes. Additionally, the method *executeTask*, responsible for assigning the task ID's and the only way to communicate with the firewall process, uses a decorator called *synchronized*, which — inspired by the Java equivalent — was developed to ensure thread-safety of certain methods.

IPTables Helper—A collection of helper methods located in the file *IPTablesHelper* provides an encapsulation of the low-level commands necessary to reconfigure the iptables firewall. It is based on the *python-iptables (iptc)* python library by Vilmos Nebehaj, which provides basic iptables bindings to enable direct access to the binary via python code [59]. Using these bindings, the methods provide high-level abstractions to execute the commands necessary to: back-up or restore the currently active iptables rule-set, open a TCP or UDP port for a specific client IP address, and to remove an open port from the rule-set, which is equivalent to closing it. Additionally it allows for injection of an emergency access exception rule to allow administrators SSH access to the server in case of a malfunction in the port-knocking service.

The restriction of server access to ports opened by the port-knocking daemon works by redirecting all incoming traffic from the *INPUT* chain to the new *knock* chain, which is created by this service on startup using a method from the *IPTablesHelper* class. Every allowed port equals an iptables rule granting the authorized client access for as long as it is present in the chain. Furthermore, these helper methods provide the abstraction for the IP version in use by relaying the requested commands to either the *iptables* or the *ip6tables* binary.

3.2.3 Client

This section revolves around the implementation of the port-knocking request generation as well as the handling of errors and the important possibility of including the port-knocking client as library for other client software. A high-level abstraction of the client-server-interaction is shown in Figure 3.4.

3.2.3.1 Client Interface

Similar to the aforementioned server interface there is a class, called *ClientInterface*, on the client side responsible to provide high-level abstractions for the initialization and core functionality of the client library. This interface significantly simplifies the deployment process by providing an easy way to implement the port-knocking protocol using just two method calls[4]. Furthermore, it also serves as the connection point for a wrapper that allows access to the functionality of the client from code written in C.

[4]One for initialization; One for the actual port-knocking

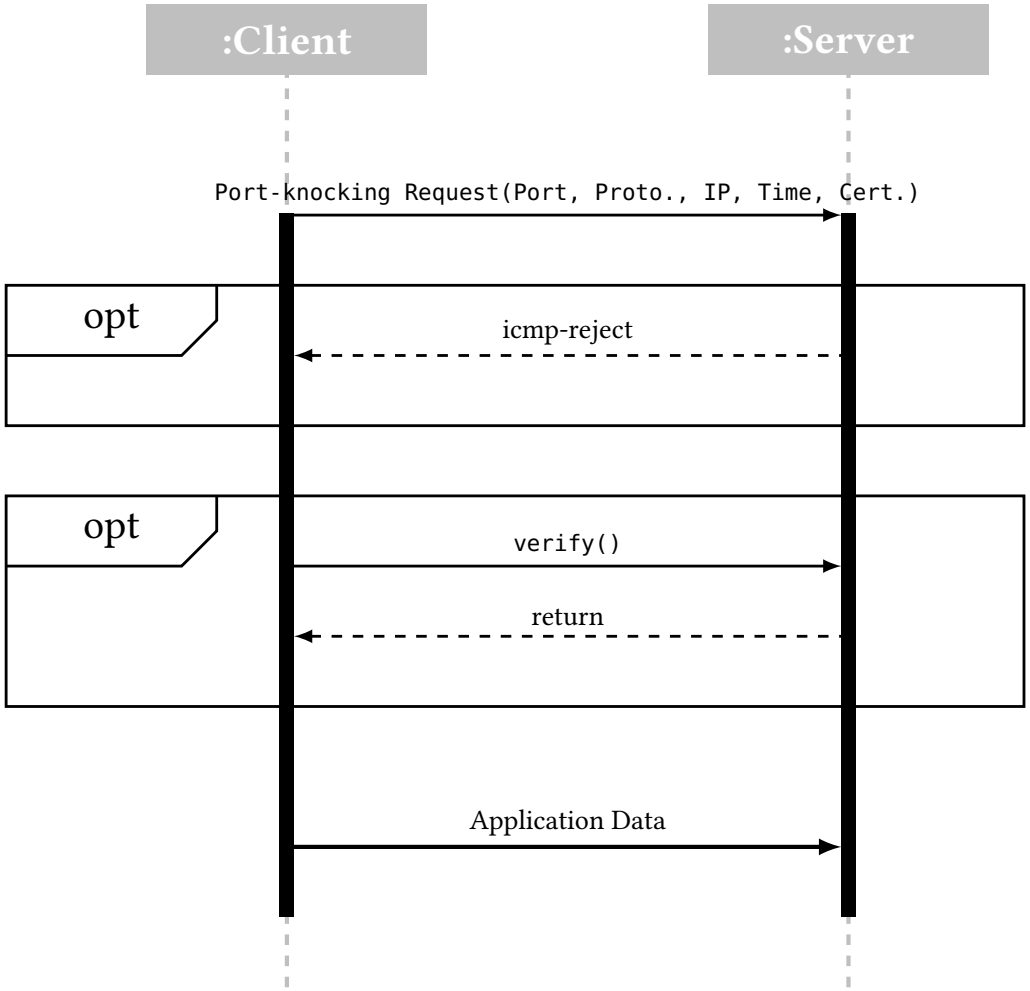

Figure 3.4: Overview of Port-knocking sequence

3.2.3.2 Connection Provider

Apart from the initialization process, most of the high-level logic is implemented in the *Connection* class. Before crafting the port-knocking request, the connection provider tries to determine the preferred interface and IP version for the route to the target server. It then uses the resulting IP address together with the other required parameters such as port number, protocol, or timestamp to assemble the clear-text knock request. This data is then signed and encrypted by the security engine returning the secure request to the connection module, which adds the Knock-Header containing the magic number and version of the protocol. Finally the complete knock packet is sent to the server and a verification logic is kicked off in case the requested port is a TCP port and the verify flag was set upon initialization. This verification process works by trying to establish a connection with the service running behind the requested port and in case of failure retrying the port-knocking process while respecting a configurable timeout

and a specified number of retries.

Additionally, this interface provides error handling and useful messages to the user, but depending on the environment and the current configuration, it is possible that the only information that the client can infer from the situation is that the port-knocking had failed. In this case, there is no way to inform the user of the exact reason for the failure, meaning that the compromise between performance, informativeness to the user and security is a decision that has to be made by the service provider based on their environment and is fully configurable through the port-knocking parameters and the general firewall configuration.

3.2.3.3 Security

Just like the server, the client also employs the *CryptoEngine* and *CertUtil* modules to build up its' security system. This security module contained in the similarly named class provides a high level abstraction for securing the sensitive data in the port knocking request. The process generally follows the following scheme:

1. Prepend a Zero-Byte at the beginning of the request to improve decryption and verification performance
2. Calculate a timestamp and append it to the clear-text port-knocking data
3. Encode it as binary data
4. Append the client certificate and sign the whole request (including the certificate)
5. Encrypt the signed request using an ephemeral key and append the public key to the encrypted request

3.2.4 Common Modules

As referenced before, there are obviously definitions, constants, utility functions, and modules which are required for both server and client. The implementation of these modules can be found in a separate *common* package.

3.2.4.1 Certificate Utility

Implemented in the *CertUtil* class, this module is responsible for every operation involving certificates. At startup it loads the required operation certificates as well as the signing certificate of the Certification Authority (CA) responsible for issuing the port-knocking certificates. Using *certificate pinning* the validation of all certificates interacting with this module is performed against only this specific CA certificate as

opposed to a multitude of different trust providers, effectively establishing a single trust anchor for the port-knocking service. In normal operation the core functions of the certificate utility include validating certificates, checking their revocation status, and generating and verifying signatures using these certificates.

The ability of executing these tasks without abysmal performance metrics comes from the use of the native OpenSSL library by basing most of the code responsible for loading, decoding, and encoding of certificates on the pyOpenSSL wrapper library, an open-source project started by Jean-Paul Calderone [60].

3.2.4.2 Cryptographic Engine

As the name suggests, the cryptographic engine in the class *CryptoEngine* is a complex module exploiting the API of the M2Crypto python library [61] to implement the cryptographic functionality required for secure communications between client and server. This module provides a high-level abstraction for encrypting and decrypting the port-knocking request by establishing a secret, deriving a symmetric key from it and using it to perform the requested operation on the given data.

Since, in our implementation, ephemeral key-pairs for each encrypted message are used to provide a higher level of security, this process involved the generation of these keys for encryption as well as the extraction of the public key from the message for decryption. This ephemeral key is then used together with the servers key-pair to establish the secret using ECDH. Then a 128-bit AES key is derived from the secret using HKDF as key derivation function to encrypt or decrypt the request including its respective authentication data. The whole scheme is a variant of the ECIES cryptosystem, which is proven to be secure against adaptive chosen chiphertext attacks as discussed in Section 2.1.3 in Chapter 2.

3.3 Limitations

As the scope of this work was focused on a prototype to demonstrate the viability of a scalable port-knocking concept, there are a few limitations encountered during the development process that need to be solved for a feature-complete version of this implementation.

3.3.1 Network Address Translation (NAT)

A consequence of the security measures employed against Man-in-the-Middle attacks described in Section 3.1.3 is the restrictions it imposes on the ability to traverse NAT-ed networks. Since NAT requires the rewriting of the source IP address in the IP header

of every packet, the server will definitely recognize a client behind a NAT setup as malicious attacker and therefore refuse to accept the request. Keeping this in mind, there is practically no other way to effectively counter Man-in-the-Middle attacks in this scenario since the client IP address is the only data the firewall has available to identify the authorized client for subsequent requests.

The adoption of IPV6, which provides *significantly* more addressing space than IPv4, is constantly increasing and will most probably completely replace IPv4 at some point. Since one of the major design goals of IPv6 is to provide every device with its' own public IP address, the need for NAT will certainly decrease in the future. However, at the time of this work IPv4 still remains the dominant IP version used in the Internet and therefore a workaround for this limitation is definitely required to enable reasonable use of this port-knocking implementation.

3.3.2 Tracking of Established Connections

One significant limitation of the current state of our implementation is the way packets are handled after a successful port-knocking. With the current design, UDP based applications will be cut off after the configured duration after successful port-knocking. So either this duration needs to be extremely large to allow uninterrupted access to these applications or the application would have to repeat the port-knocking process as soon as the port is closed again.

The first option is not an acceptable state, since this approach creates a possible security issue while not ensuring that *every* possible application will finish before the port closes again. For example a video-on-demand streaming service could be used by binge-watching users for numerous *hours* non-stop, which in turn would be an unacceptable duration for a default timeout setting.

The second option causes an unnecessary high overhead on the overall communication since the client has already provided the required authorization information for using the application and should not be bothered renewing the authorization for the same session over and over again. Even worse, since as of now there is no logic for actually renewing an open port, the client has to wait for the port to be closed before a new request would be accepted.

This restriction obviously requires a reliable solution before the presented implementation can be used in a productive environment.

3.3.3 Chosen Implementation Language

Python is an excellent language for quick prototype development, however, this capability comes at a cost: performance. With the interpreter architecture, inefficient

cryptographic libraries and limitations concerning multi-threaded execution because of the *Global Interpreter Lock* (GIL), the achievable processing power is significantly inferior compared to low-level languages such as C or Rust [62].

The impact of the reduced processing capabilities of the chosen programming language is clearly visible, especially in computational expensive operations such as encryption, decryption, verification of certificates or signatures, and cryptographic operations in general. Additionally, the fact that there is no condensed cryptographic library available for python that contains all necessary operations for this work reduced the achievable performance even further. This not only forced the use of multiple libraries using different internal representations of the same data objects, but we also had to hack some own implementations to achieve compatibility between the representations.

3.3.4 Multi-platform support

Although the implementation was designed with special attention to keeping as much code as possible platform independent, due to time constraints and the scope of this work, the developed prototype supports only one specific environment. It depends on OpenSSL being available as a cryptographic module as well as iptables for managing the firewall and requires to be run on the Linux Operating System. Since major design goals are scalability and ease of use, the lacking of multi-platform support is an unacceptable restriction.

However, all the logic necessary to employ the port-knocking protocol is completely decoupled from the modules encapsulating the platform-specific dependencies and operations. This means, that even at the current state this implementation is easily extendable to other software components as well as operating systems just by providing the required wrappers for the desired platform specifics.

3.3.5 UDP

One of the greatest strengths of this concept could also be a great weakness depending on the use-case one is considering. If a deployment scenario involves protecting services hosted at third-party infrastructure providers, where the configuration of their security infrastructure is out of reach, the use of UDP as a protocol could become an issue. As some cloud providers block UDP communication at their hardware firewalls, it is possible that this implementation could not be used to protect services running there. However, in order to support the huge variety of the applications deployed by their customers, large cloud providers have very relaxed and flexible firewall configurations as default setup, so this limitation should only apply in very few scenarios.

Chapter 4

Evaluation

The following Chapter is dedicated to test the implementation described in Chapter 3 based on the initial requirements and expectations. The data generated by these tests is then used to assess how well the requirements have been fulfilled. Additionally, the real-world viability of this implementation is evaluated and suggestions for possible improvements are made.

4.1 Per-Module Performance Analysis

The first analysis consists of benchmarking every module with a considerable impact on processing performance independently. In the following Sections, the test environment and test cases for each of these components will be explained. For every test case the collected results will be presented with an evaluation of their effect on the implementation and possible implications for the port-knocking concept in general.

4.1.1 Test environment

System—All of the module-specific benchmarks were performed on a DELL OptiPlex 9020M machine equipped with a quad-core Intel(R) Core(TM) i5-4590T CPU @ 2.00GHz and 16GB of memory. The operating system running on the machine was Ubuntu Linux 12.04.5 LTS with all recent updates installed before any tests were executed. Therefore, the crptography provider used in the evaluation process was *OpenSSL version 1.0.1* while the firewall software was *iptables v1.4.12*.

Measurements—Whenever possible all time-critical measurements were performed using the *timeit* Python library, which is designed to use the most precise timing functionality available at the given platform. In some cases, however, specific restrictions of

this library required us to fall back to the *time* module, provided by the standard Python environment, to create our own timing mechanism.

4.1.2 Firewall

As mentioned in Chapter 3, this work focuses on the iptables firewall software running under Linux as operating system to provide a proof-of-concept. Consequently, the handling of iptables operations in our software is one of the core modules influencing overall performance and therefore deserves to be analyzed in detail.

Test Case—To measure the capability for opening and closing ports using the iptables firewall, a suitable benchmark was written to explore the limits for reasonable addition and removal of packet-filtering rules to and from the iptables chains. The following test measures the time necessary to open all available ports for a single client, meaning all possible combinations of ports *0* up to *65535* for *TCP* and *UDP* as well as *IPv4* and *IPv6*. Additionally, the performance for closing the open ports afterwards is also considered in this test. All of the ports are opened for a single random source IP address since during the test runs it became clear that using different IP addresses vs. the same IP address for each rule has no impact on performance as long as only the last rule actually matches to the request. A typical iptables rule for a port-knocking client contains the *protocol*, *destination port*, and *source IP* address as parameters and an incoming request has to match all of these parameters for the rule to apply.

Results—Crunching the data resulting from adding in total 131072 rules (65536 TCP ports, 65536 UDP ports) for an IPv4 client to our "knock" iptables chain, clearly shows significantly decreasing performance proportional to the size of the iptables rule-set as shown in Figure 4.1.

Looking at this relation it becomes clear that the time required to execute a single iptables operations increases with the number of rules already present. After running a second analysis and processing the resulting data, it is clearly visible how the number of active rules influences the execution time for following operations as seen in Figure 4.2.

As expected, the behavior for closing the ports is consistent to the relation described above. As in the initial state of the port-closing measurement we have 131072 active IPv4 port-knocking rules, the time taken for executing the port-closing operations, which effectively remove the rules added before, decreases while the size of the current rule-set diminishes as shown by Figure 4.3.

At this point it is worth mentioning that the average execution time for closing a port is lower than for opening a port, which among other reasons could be due to memory deallocation being faster than memory allocation.

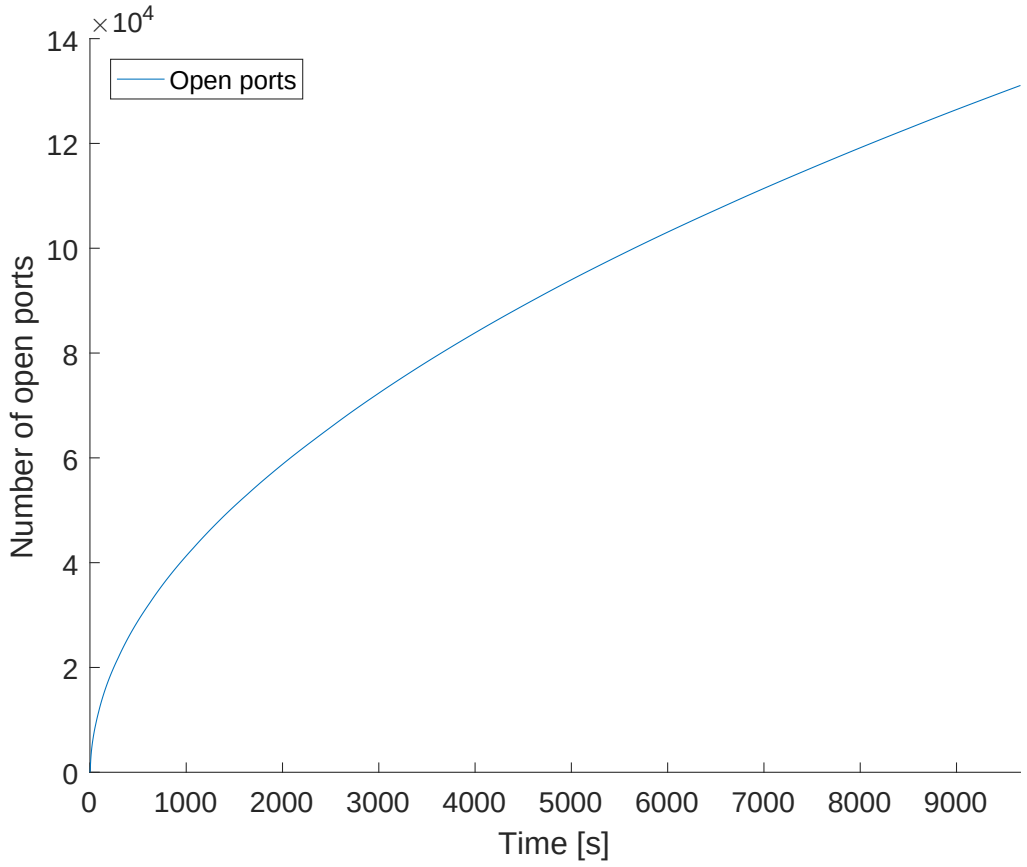

Figure 4.1: Time for opening 131072 ports (65536 TCP + 65536 UDP) for an IPv4 client

As the iptables kernel-modules responsible for performing these operations for IPv6 addresses are completely separated from their IPv4 pendants, the total number of active IPv4 rules does not influence the performance of ip6tables, the IPv6 pendant of iptables. Apart from this observation, the performance-related behavior of ip6tables in this test scenario is equivalent to the results regarding iptables presented above and can be found as Figure A.4 and Figure A.5 under Section A in the appendix.

Interpretation—The results indicate that the number of active rules has to be limited for sustained operation. The exact threshold depends on the requirements of the respective service provider as much as on the hardware available for the machines running the port-knocking service. Additionally, the limit should be chosen in relation to the processing demands of the other performance-relevant modules to prevent starvation of these components by exhaustion of the available CPU and memory resources by the firewall handler.

For our test environment, we would recommend a soft limit of about 5000 active rules per IP version (as *iptables* and *ip6tables* do not directly influence each other). At this

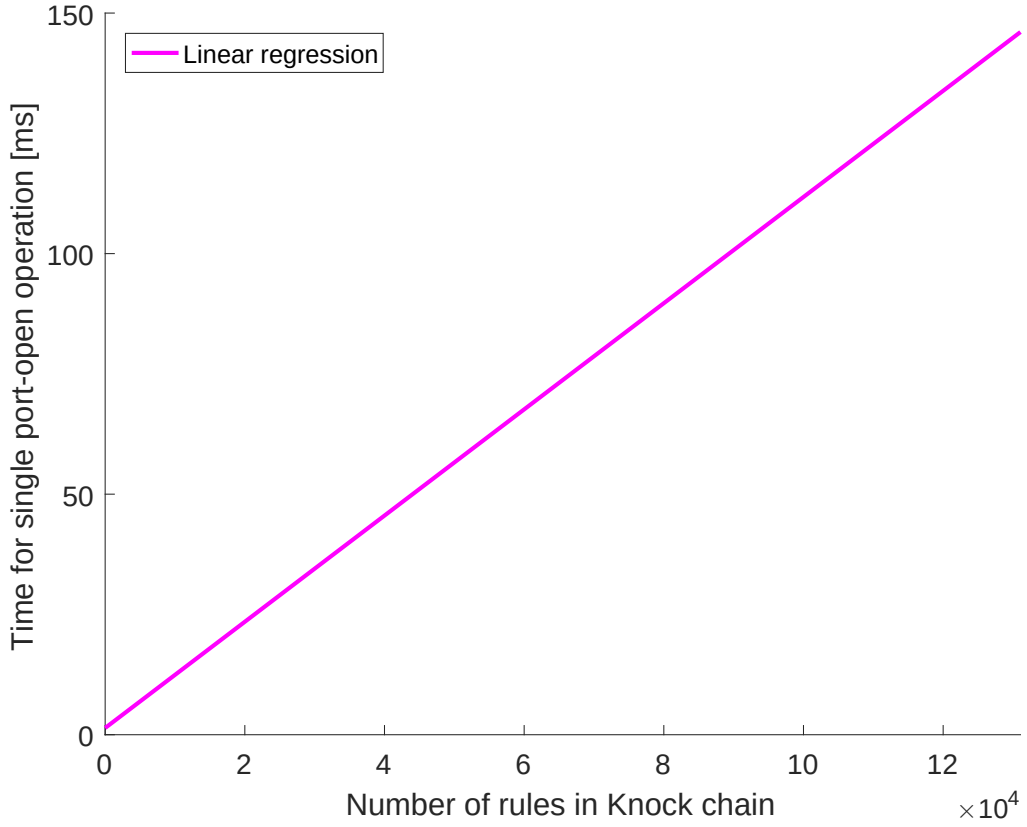

Figure 4.2: Execution time for adding a single rule to an iptables chain in relation to the number of active rules (IPv4)
Lin. regression taken from Figure A.1

threshold the processing capability of our hardware manifests around *235 port-opening requests per second* with an average of about *4ms per operation*. This delay caused by the firewall seems acceptable for a viable real-world solution and could be used as a starting point metric for tuning the performance to the requirements of a specific application. An important fact to note at this point is that this delay occurs only once per connection until the rule is removed again.

Potential Optimization—Obviously, there has to be room for improving this performance degradation caused by a high number of active rules. One approach would be to extend the iptables firewall with the *IPSet* module, which is capable of processing combinations of ports and ip addresses. According to the Linux networking community, this iptables with the ipset module can outperform a vanilla iptables deployment by orders of magnitude in terms of processing performance in matching packets to a high number of IP and port tuples [63]. By eliminating the degrading performance problem using this extension a much higher limit of concurrent clients can be achieved, which would obviously boost the scalability properties of the implementation significantly.

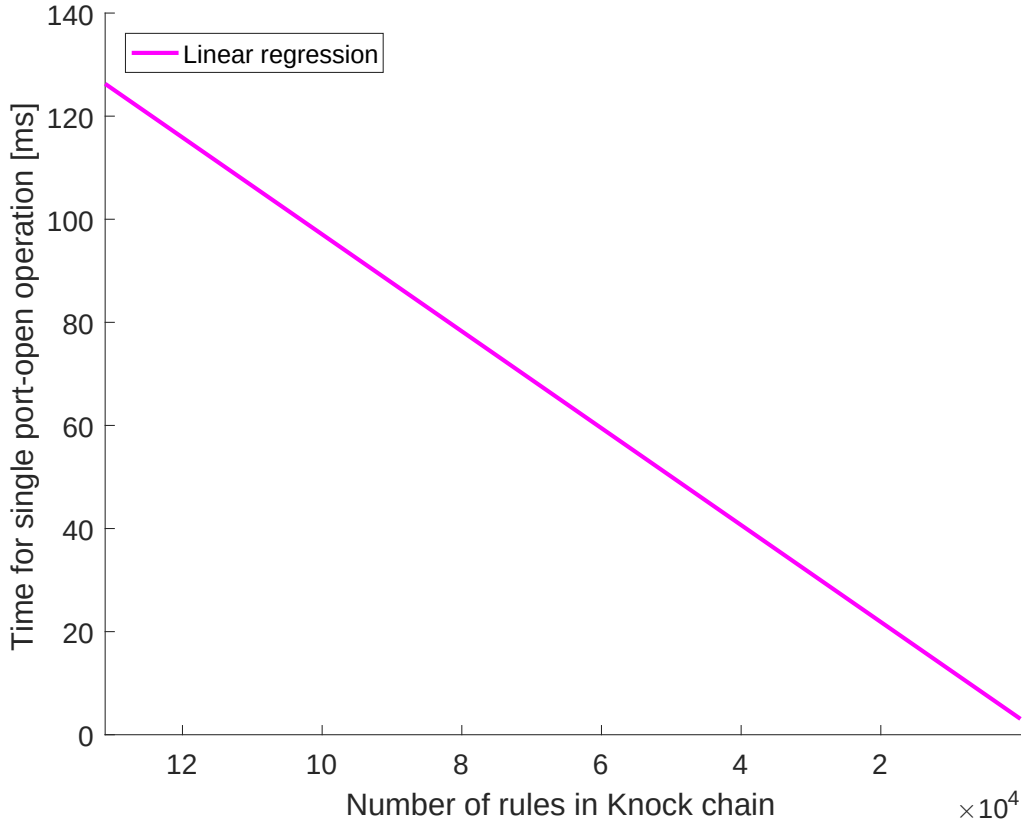

Figure 4.3: Execution time for removing a rule from an iptables chain in relation to the number of active rules (IPv4)
Lin. regression taken from Figure A.2

Another possibility lies in deploying iptables' successor, *nftables* [64]. Since nftables supports the matching of packets to sets of values such as IP addresses or ports out of the box, it yields the same potential performance improvement as the IPSet approach mentioned before. However, the current nftables release is still at an early development stage, which implies possible weaknesses in terms of reliability and therefore may not be suitable for most production environments.

4.1.3 Packet Processing

As specified before, our implementation relies on intercepting all packets using a raw socket to provide the possibility of a fully locked-down firewall configuration since a raw socket receives all Layer 2 network packets reaching the host. However, as a consequence of this design choice our implementation needs to be able to efficiently filter out all packets irrelevant to the port-knocking service at a high processing speed. As this property greatly influences the achievable overall system performance, its implementation has to be comprehensively evaluated.

Test Cases—In order to measure the capability for processing incoming packets, we decided to create a static test case where pre-generated port-knocking requests and usual packets including the complete Ethernet frame with IP header and UDP header are loaded and processed for a specified number of iterations. This test excludes the influence on the processing time of the raw socket itself and does not take into account the operations performed by the cryptographic engine after the packet was recognized as port-knocking request. These exclusions do not impose a limitation on the validity of the results since it can safely be assumed that the processing performance of the Linux networking stack exceeds our processing capabilities by orders of magnitude and the cryptographic engine will be tested by its own benchmark in Section 4.1.4.

In our test environment, we decided to run two different measurements: One synthetic worst-case scenario, which simulates that all incoming packets are valid port-knocking requests and a realistic scenario with 1% of port-knocking traffic while the rest of the packets are irrelevant to our service. Additionally the test data for the second scenario contains 5% of packets bigger than the configured minimum knock request size. Of these packets the TCP to UDP ratio is set at 5:1 in order to resemble the Internets' traffic patterns as close as possible [65].

This configuration ensures, that the incoming non-portknocking packets get dropped at different stages in the filter logic (as described in Section 3.2.2.4), therefore providing more realistic and detailed results for evaluation.

For the first scenario we decided to cap the test data-set at size of 3 million while the second test was performed with 5 million simulated incoming packets.

Results—The test-run performed to evaluate the worst-case processing power of our implementation yielded a result of roughly *5300 pps* (packets per second), which translates to a processing time of less than *0.19ms* per packet for valid port-knocking requests.

Analyzing the performance of our second test case, which is an obviously closer approximation for real-world operation, yields even higher processing capabilities. With an average throughput of over *6100 pps* our implementation can achieve computation times of about *0.16ms* per packet.

Finally it should be added that during both test cases the Python interpreter completely exhausted the systems' memory resources. Since the working set of neither test case comes even close to the 16GB of available RAM, we have to assume that the high memory consumption is most probably a result of the thread handling in our implementation.

Interpretation—As shown by the raw numbers in the presented test cases, the processing performance delivered by the filtering mechanism in our implementation is able to handle large quantities of traffic influx. Even in the worst-case scenario of the entire

server communication consisting of valid port-knocking requests, our packet filter performed well. Combined with the evaluation of the processing capabilities delivered by the other critical modules, it is safe to assume that this part of the implementation will hardly become a bottleneck for most environments.

4.1.4 Cryptographic Engine

From the very beginning we estimated that the cryptographic engine would contain the most processor-intensive operations. Although AES calculations can be done very efficiently in hardware on most modern CPU's, there is no hardware support for elliptic curve cryptography aside from very specialized cryptographic devices. To assess the performance penalty of using bleeding-edge cryptographic techniques in our implementation, we engineered the tests described in the following paragraphs.

Test Cases—To measure the raw processing performance of the crypto-engine for recognized port knocking requests we decided to create two benchmarks — single-threaded and multi-threaded — based on the same test. The most important metric for assessing the general performance of our cryptographic module is the throughput achievable during the full decryption and verification process utilizing the whole spectrum of our implemented cryptographic operations.

For our environment the test data had to be limited to 300,000 valid port knocking requests as our implementation starts a new thread for every incoming packet exceeding the minimum length and too many threads quickly exceed our test setups' capabilities. Since the engine has to fully decrypt every request in order to verify its validity, the difference in computational demand between valid and invalid requests is negligible.

Results—With our test setup we were able to achieve a throughput of *620 pps* for the single-threaded implementation while running the test using one thread per processing task resulted in only approximately *260 pps*.

The according log files from the performance test can be found in Listing C.1 for the single-threaded test and in Listing C.2 for the multi-threaded test under Section C in the appendix.

This result shows problems in our implementation of the thread-handling for cryptographic operations and again underlines the widely known fact of significant inefficiencies in Pythons' multi-threading capabilities.

Aside from these performance issues, we discovered that a segmentation fault originating from the employed OpenSSL wrapper can cause the port-knocking service to reject a valid signature. This is a known and unfixed bug [66].

Generally, we expect the results of this particular performance test to differ tremendously depending on the hardware and operating environment it is running on.

Profiling—In order to determine the reason for the low performance of the cryptographic module, we decided to analyze the time distribution between the subsequent method calls using Robert Kerns' *line_profiler* [67].

After few profiling runs to find the relevant method calls that need more detailed profiling, the last execution of the profiler yielded the results shown in Listing B.1 under the appendix Section B.

The analysis of the profiling result clearly shows that most of the execution time is spent in wrapper methods of the incorporated cryptographic libraries (pyOpenSSL and m2Crypto) and the library-specific methods for converting input data into their respective internal format.

Interpretation—As seen in the results from the profiling, it is obvious that the biggest performance issue of the cryptographic engine is a result of using unoptimized cryptographic libraries. Additionally, a significant amount of processing time is spent on converting the exact same data between different internal representations of the incorporated libraries, which renders the overall implementation unnecessarily inefficient.

Summarizing the observations made during the test-runs of the cryptographic module, it seems clear that this is the part of the implementation with most weaknesses in regards of performance as well as reliability. There is also a non-zero probability of security issues arising from the combination of multiple un-audited cryptographic libraries and their interoperability mechanics.

Finally, we conclude that future improvements should focus on enhancing the crypto-engine first.

Potential Optimization—Replacing the M2Crypto and the pyOpenSSL should be the first goal in optimizing the crypto performance. However, implementing these operations in pure Python would most probably result in even worse processing power since Python's overhead is too enormous to efficiently implement low-level mathematical operations. Our recommendation would be to implement the cryptographic functions in a separate module using a highly optimizeable low-level programming language such as C. Not only would this yield the possibility to significantly increase efficiency but it also enables proper multi-core processing for the most computation-intensive part in the port-knocking service.

4.2 Firewall Filtering

As mentioned in Section 4.1.2, the number of active clients has a direct influence on the overall system performance. As for the iptables firewall, each client connecting to a service requires a new rule to be added to the knock chain. Apart from increasing the expected execution time for future iptables operations, the growing knock chain also impacts the filtering performance for *every* incoming packet.

Test Environment—Since the metric to be tested in this case requires packets to be sent over the network, the hardware of our test setup had to be upgraded. In order to provide reliable and simple measurements, we chose to add a second identical DELL OptiPlex 9020M machine and connect it with the first OptiPlex as well as the management network using a 100 Mbit/s Ethernet switch and CAT5e Ethernet cables.

As opposed to the previous test cases, this scenario required the development of a server & client in Python to simulate an application protected by port-knocking. Since the tests required us to time operations involving interactions between server and client, we decided to use timestamps as messages: the client sends the time measure just before the packet was sent and the server replies with the time when it received the packet. Additionally the server needs to contain the ability to call a callback function whenever a packet is received to enable the calling evaluation framework to record and analyze the data passed to the test server application.

This approach naturally requires very precise synchronization between the system clocks of server and client. In our test setup they were installed on two of the aforementioned DELL OptiPlex 9020M machines with the only connection between the computers being a local ethernet network. Although there was an *Network Time Protocol* (NTP)[1] server locally available at the local network, it quickly became clear, that the achievable precision was limited to errors in the millisecond range [68]. Additionally, the NTP implementations available for our test environment (Ubuntu Linux 12.04) allowed for synchronization only every 16 seconds at the minimum, which generated an unacceptable clock drift.

In lack of high-precision hardware clocks we decided to use PTP, the *Precision Time Protocol* to provide synchronized clocks with an accuracy high enough for sub-millisecond measurements. This protocol uses time-stamping features of the underlying network adapter to provide clock synchronization with a precision of less than *1 microsecond* depending on the available hardware [69]. The Intel(R) I217LM ethernet card present in our test environment allowed us to achive a precision of about 5 microseconds in average. An excerpt from the PTP logfiles showing statistics of the synchronization can be found in Listing C.5 under Appendix A.

[1] De-facto standard for time-synchronization in computer networks

Test case—The purpose of this test is to measure the impact on traffic unrelated to our port-knocking service. This traffic interacts indirectly with our service especially in two ways: by going through our raw socket where the unrelated packets are filtered out and by traversing the iptables rules before being delivered to their target application. Since the filtering capabilities of our packet processing only limits the number of requests the port-knocking server can handle, the most significant impact on other communication is the possible performance degradation of the iptables packet filter.

In order to measure the time taken by the iptables firewall to process a packet, we designed a scenario where the test client application keeps sending UDP packets to the server machine. Every packet contains the timestamp observed at the client just before the packet was sent to the server. The server then measures the time when the packet arrived at the socket and calculates the resulting delay. After timing a configurable number of packets, the server proceeds to open another port before measuring incoming traffic again. Following this strategy we are able to generate the desired amount of delay measurements over a given number of rules in the iptables knock chain.

Results—This measurement obviously incorporates a significant error because of external influences on the delay caused by other independent operations such as queues in kernel or networking stack, network hardware, or even the Ethernet cables used. Since the processing time of the firewall makes up only a fraction of the total delay measured, we had to provide a large dataset containing 100 measurements per data-point.

Additionally, the error of these measurements is amplified by time-synchronization issues. These arise from the observation that the PTP protocol becomes increasingly unreliable if the system is under heavy load.

To ensure the production of conclusive data, we decided to measure the increase in delay over the addition of 220,000 rules to the knock chain by generating data-points every 1000 added rules. As the number of available ports is historically fixed to 65536, we used two different source ip addresses combined with rules for TCP and UDP per port in order to reach over 200,000 rules in the knock chain.

As can be seen in Figure 4.4, the linear increase of the overall packet delay is proportional to the growth of the number of rules.

In order to see the relative impact of the rule-set size on the firewall processing time more clearly, Figure 4.5 shows a linear regression over the raw measurements normalized to 0 ms delay at 0 rules in the port-knocking chain.

The standard deviation of this experiment can be seen in Figure A.6 under Section A.2 of the appendix.

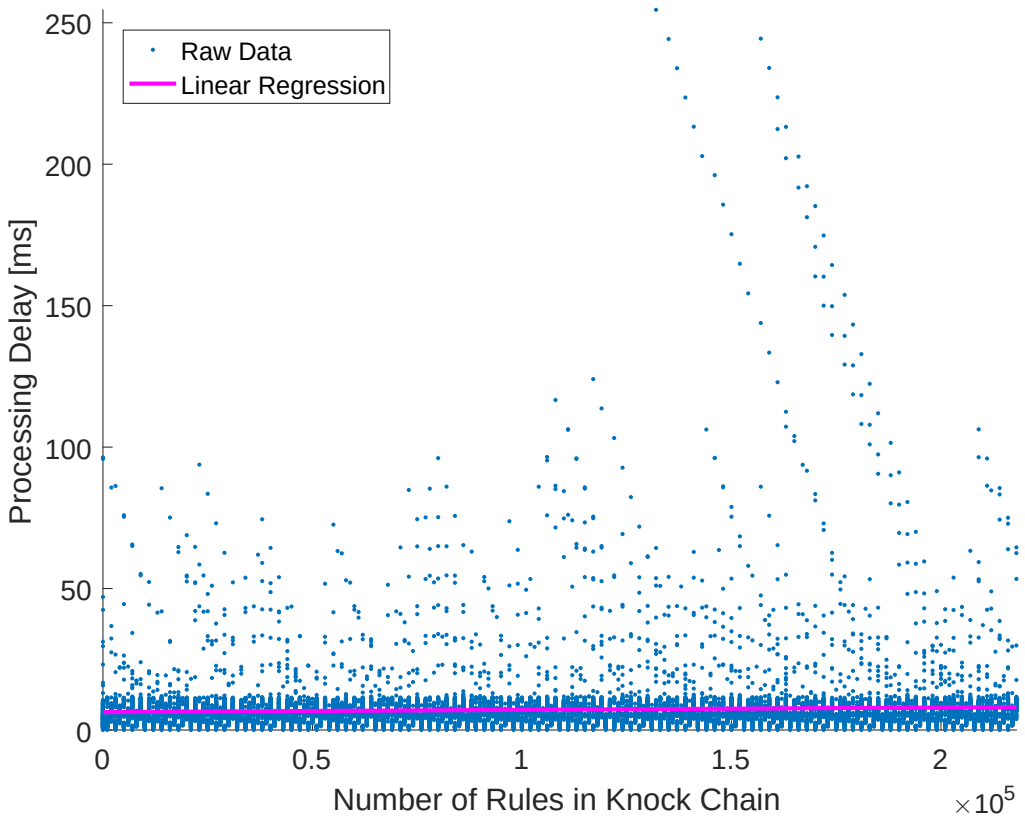

Figure 4.4: Raw measurements for packet delay introduced by growing iptables rule-set

Interpretation—Although the variance in the data is relatively high and a large number of measurements as well as mathematical smoothing of the resulting data were necessary to make it unambiguous, it is definitely clear, that the number of rules present directly impacts the delay caused by the iptables firewall.

This side effect of the port-knocking security layer was expected from the beginning and the analysis performed during this test clearly shows that the impact of this particular metric is almost negligible. In daily operation, most applications run on the TCP transport protocol, effectively meaning that these connections can be tracked and handled using a single firewall rule to allow established TCP connections. For all the other non-trackable communications such as UDP, the processing capabilities of iptables result in a performance hit of about *2.5 milliseconds* for over 220 000 active rules. We strongly believe that the number of manageable concurrent connections will be restricted to a much lower limit because of other bottlenecks.

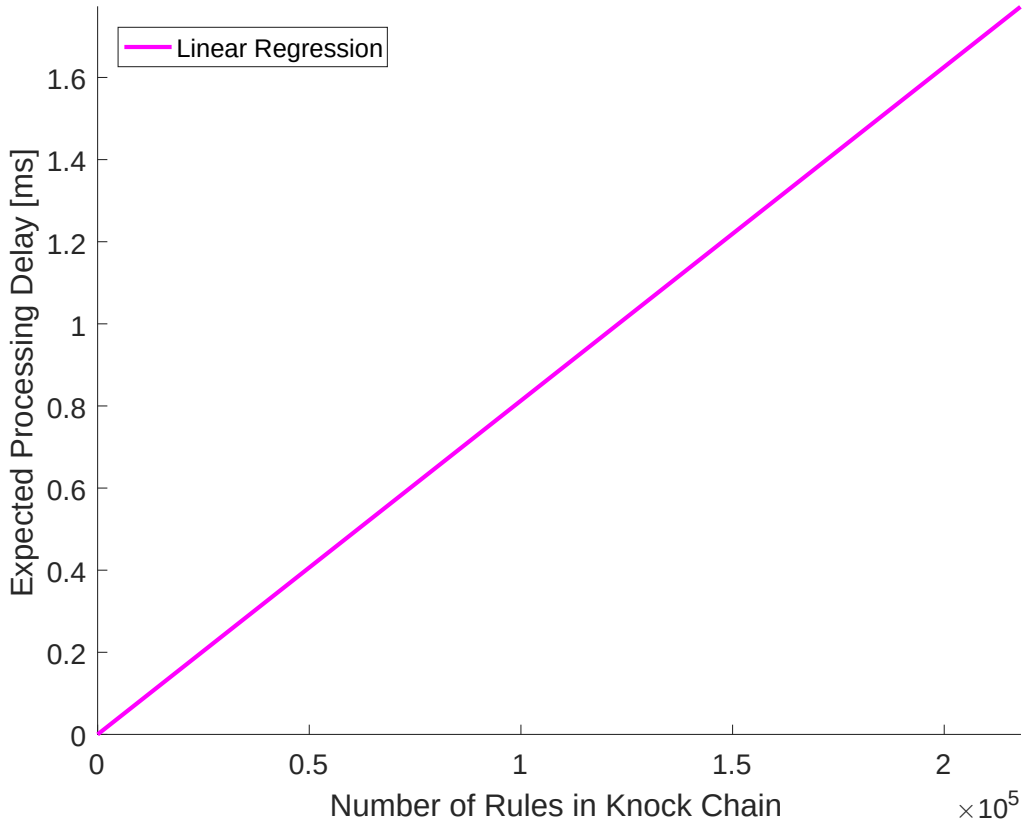

Figure 4.5: Normalized Linear regression for packet delay introduced by growing iptables rule-set

4.3 Connection Overhead

A very important metric for evaluating the overall performance impact of deploying the specified port-knocking solution is the additional delay caused on the overall application communication. Since our implementation does not directly influence any packets after establishing the connection, it is obvious that for any communication using a connection for a long time the impact is negligible. Therefore, the property to be evaluated in this Section is the overhead caused on the connection establishment.

Test Environment—In order to establish the environment for this test, it was necessary to extend the test server and client from Section 4.2 to allow for port-knocking before the test application starts its own communication. Additionally, the client needs to take into account a certain wait period between sending the port-knocking request and the first application packet, since the server needs this time to process the port-knocking request and open the respective port. Otherwise the subsequent application connection would just get refused, resulting in an unnecessary high delay and therefore overhead.

To determine the optimal waiting time, a calibration script was developed. The strategy employed by the script logic is to base the wait period for the first connection attempts on a configured start value and from there on adjust the wait period until the rate of failed requests becomes acceptable according to the given parameters.

Test case—Since the time required to establish a simple UDP connection differs significantly from the time necessary to complete the whole TCP handshake, this test case involved two test runs with identical parameters, except for the test application protocol.

To evaluate the overhead, we employed a simple yet reliable test scenario: running alternate executions of the test communication described above[2] with and without prior port-knocking. For the client requests omitting the port-knock, we permanently opened port 60001 on the server via iptables rule. To generate a significant data set allowing for conclusive results, the test was run continuously over several hours per protocol to minimize impact of other environment-dependant variables.

As we expect the wait time to be optimized for the target environment in a real-world deployment, the wait time was calibrated to a reasonable value beforehand.

Results—The aforementioned calibration yielded an optimal value of *11 milliseconds* for the wait period allowing for a request failure rate of under *0.2%* with an accuracy of *99%* as shown by the calibration log in Listing C.6 and the client log in Listing C.7. This means that for our test setup the probability for the application request to arrive before the port is opened by the port-knocking service and therefore failing is less than 2 per 1000 requests with this value being wrong by at most 1%. A visualization of the test run for UDP containing 14,000 measurements for requests with and without port-knocking enabled can be seen in Figure 4.6.

Using this conservative recommendation as parameter for our overhead simulation, we were able to achieve an average absolute overhead delay of *16.27 ms* for TCP and *16.63 ms* for UDP as visible in Table 4.1.

Interpretation—The measured 16 ms of average added delay to establish a single connection to the application server could be seen as significant. Before jumping to conclusions however, one should consider that this delay is an absolute metric that impacts only the very first request and most users are used to a slightly higher latency for starting applications and even tolerate unresponsiveness at startup depending on the service.

As this metric is independent of networking variables except for some border-cases of traffic congestion, the delay is an absolute value that can be just added as static

[2] Client sending a timestamp, server responding with his own timestamp

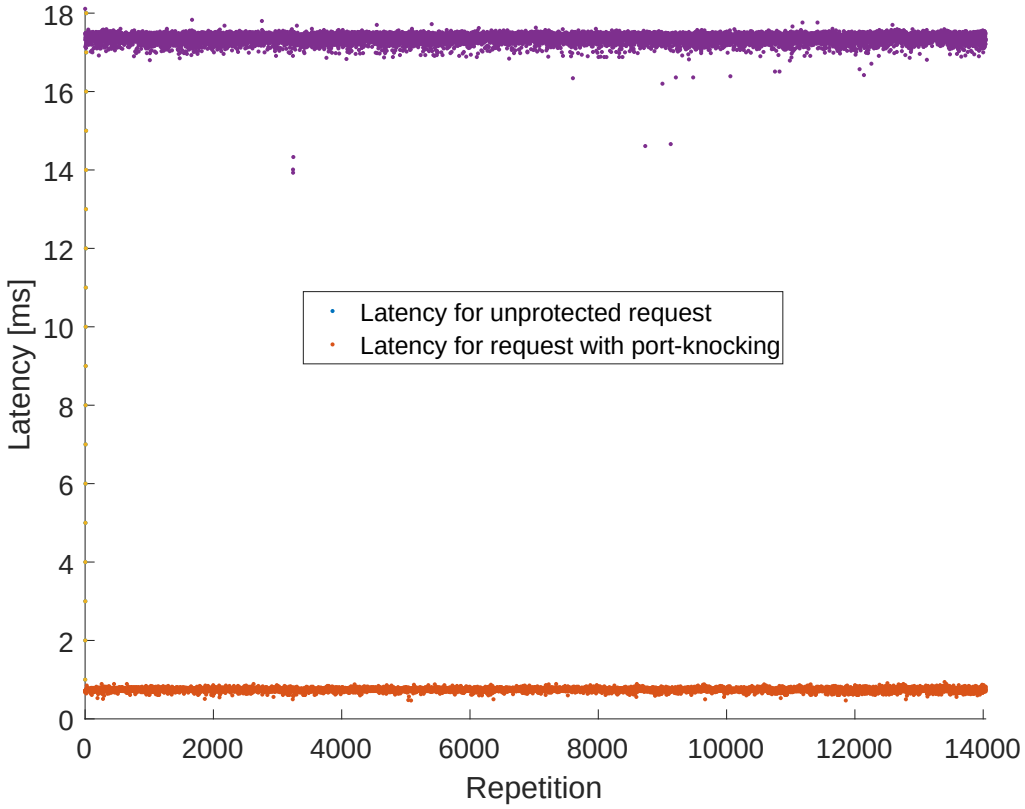

Figure 4.6: Latency overhead caused by *sKnock* (UDP)

parameter to any required latency calculation.

Finally, a very important limitation of this test needs to be taken into consideration: the programming language of this prototype and the heterogeneous implementation of the cryptographic operations distributed over multiple libraries. A properly implemented and optimized cryptographic framework in a low-level programming language such as C would greatly improve the overall system performance and therefore reduce the delay to a fraction of the value resulting from this test.

4.4 Reliability Under Packet Loss

Another interesting property of every security system is how it affects users with unreliable Internet connections, since they still make up a large fraction of some target audiences. To provide an indication of the overall system behavior, a test scenario evaluating the impact of our port-knocking implementation on the ability of an application to establish a connection under packet loss is necessary.

	TCP	UDP
Unprotected Request	1.97 ms	0.74 ms
With Port-knocking	18.25 ms	17.37 ms
Port-knocking Overhead	16.27 ms	16.63 ms

Table 4.1: Measurement results for the connection overhead test

Test Environment—The setup for this test scenario required major changes to the employed hardware as well as the test applications. Since there is no reliable way to emulate packet loss locally at either the server or client machine, a third computer was necessary to connect the two other machines via a connection whose unreliability would be under our control. In our case this machine was a server containing an Intel(R) Xeon(R) CPU E3-1265L V2 @ 2.50GHz, 32 GB of Memory, and an Intel 82580 network interface controller. This system was running Debian 8.3 including all mainline updates released at the time of writing as operating system. This Linux distribution incorporates the *iproute2* software which provides the ability to manually configure a synthetic packet loss[3] using the *netem* utility.

As for the software changes related to this test scenario, the server as well as the client were extended to add the ability of simultaneously handling TCP and UDP communications[4] in order to reduce the necessary manual administration and therefore speed up the test runs.

Since in this test case the client is responsible for collecting and recording the relevant measurement data, some extensions had to be implemented. The client now also received the ability to call given callback methods on the event of receiving a response from the server. Furthermore the measurement capabilities as well as the logging and data collection mechanisms in the client were improved to satisfy the new requirements.

Test case—For this test case the server and client machine were connected via two ports on the dedicated networking device of the routing machine. These two ports were bridged together and with the help of aforementioned *netem* module different packet loss probabilities between 0% and 90% were configured during the test runs.

At each test run the client repeatedly sends port-knocking requests to the server, retrying the port-knocking process for a configurable number of attempts after a timeout

[3] And numerous other network-testing related parameters

[4] Before this change the protocol was given as parameter to server as well as client and the results were saved in different files. This effectively required to run every test twice if the metric to be measured was dependent on the protocol.

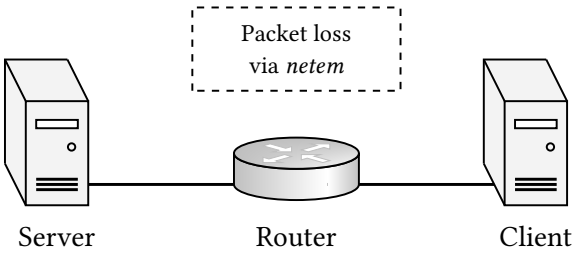

Figure 4.7: Test setup for packet loss evaluation

occurred. After all available attempts for a single repetition are used up, it is considered as failed and the next iteration starts. All test runs are done for test client running on the TCP protocol first and after completion repeated using UDP.

After each run the packet loss probability is increased by 10% up until the reasonable limit of 90%. Then the whole test is repeated after switching the default firewall policy from *REJECT* to *DROP*.

Results and Interpretation—For this test we chose relatively aggressive parameters in order to be able to give a recommendation for fast failure compensation. For slower networks these values should obviously be adjusted accordingly.

Every test was performed with *1000 iterations* for TCP as well as UDP, totaling 2000 iterations. The timeout was set to *1 second* and a total of *3 attempts* per iteration were allowed before it was considered as failed.

As expected, the reliability of the entire application degrades with increasing packet loss. It is visible from Figure 4.8 and Figure 4.9 that the measurements show the behavior we would expect from a simple test application like the one deployed in this test.

Since for TCP we provide a verification function to check if the port was opened correctly, the port-knocking client can handle the loss of the knock-packet for a TCP application independent from the application itself. Because our implementation only adds one single packet before the TCP handshake sequence begins, the impact of a lost port-knocking request is not visible in the number of failed attempts, since in case the handshake fails at the first attempt, both packets are retransmitted.

For UDP the application obviously has to employ its own mechanism of handling packet loss since the protocol does not provide it. The only change necessary to this mechanism is to repeat the port-knocking together with sending the first application-related packet. Following this strategy the same conclusion as for TCP applies: the impact of the port-knocking packet preceding the first application packet is of only minor significance.

Another relevant metric is the average time that was necessary for the iterations that resulted in a successful connection. As shown in Figure 4.10, the time to complete the

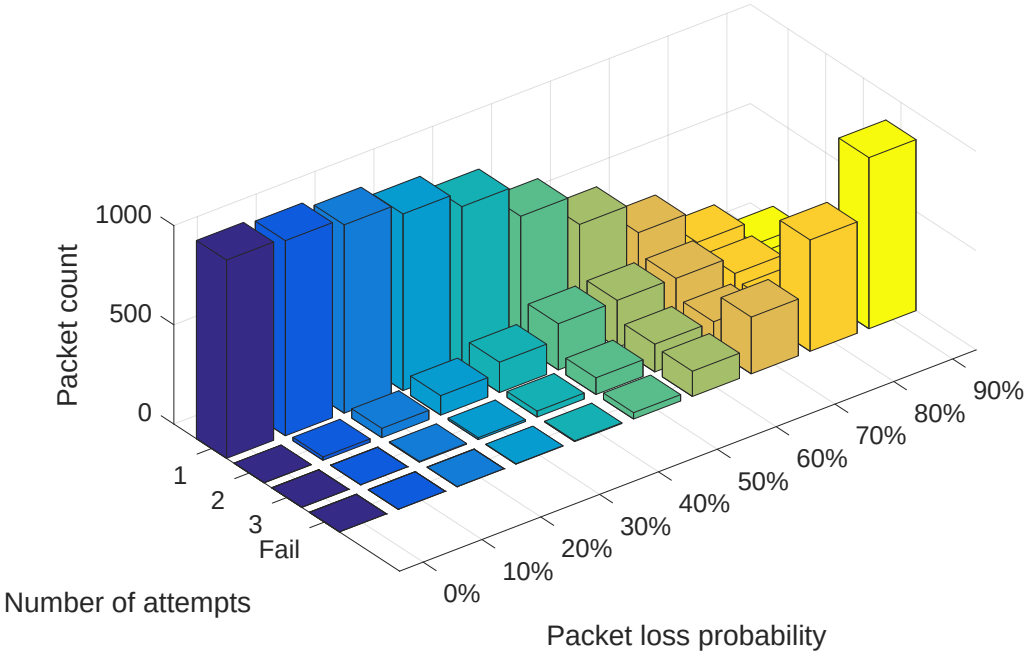

Figure 4.8: Number of attempts for the TCP test application using port-knocking in relation to the packet loss probability with firewall policy set to REJECT.
Timeout: 1s; Number of retries per iteration: 3; Number of iterations: 1000

establishment of a connection increases proportionally with the probability of occurring packet loss.

This measurement can be used as a general reference to determine the connection parameters according to the requirements of the deployment scenario. Since a high delay caused by packet loss renders many applications unusable and the resiliency of port-knocking against packet loss highly depends on the implementation of the application client, the optimal values depend completely on the targeted environment.

The results of the test runs performed with the Firewall configured to use the DROP policy can be found in the Appendix under Section A.3 and are left out at this point, because the data is almost identical to the measurements using the REJECT policy. This was an expected result; our test application does not expect ICMP notifications of reject packets and therefore always behaves as if the Firewall was dropping the requests silently.

However, even more sophisticated applications would most probably not show a significantly different performance in this test scenario, since a lost packet is almost equivalent to a dropped packet from the clients' point of view.

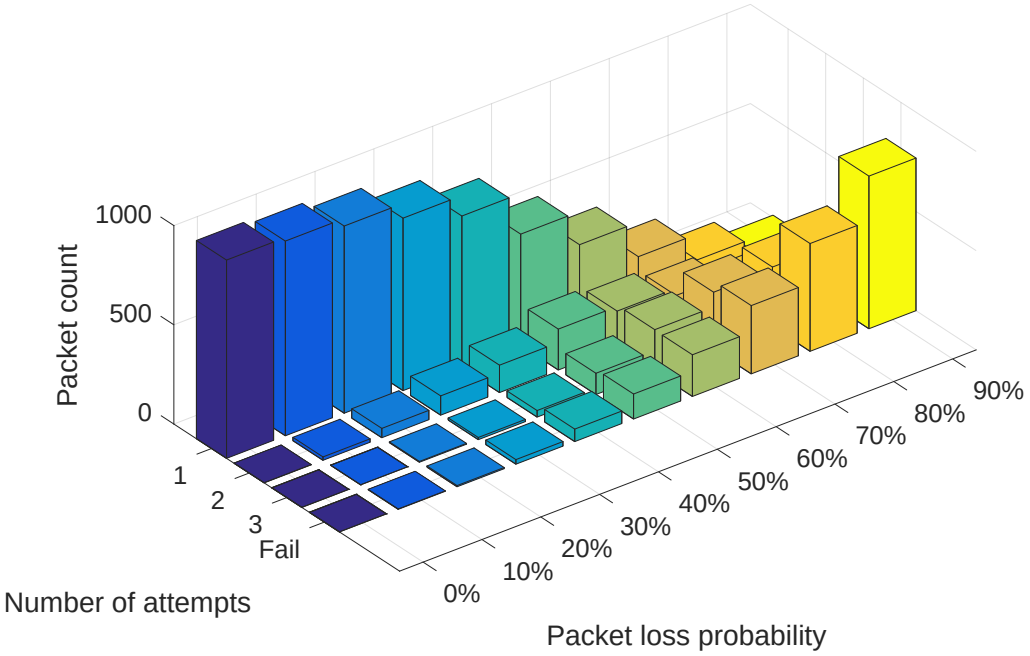

Figure 4.9: Number of attempts for the UDP test application using port-knocking in relation to the packet loss probability with firewall policy set to REJECT.
Timeout: 1s; Number of retries per iteration: 3; Number of iterations: 1000

Recommendation—Because of the dynamic and unpredictable nature of unreliable networks, our recommendation would be to include a rather aggressive timeout in the default configuration but additionally provide a heuristic for determining more effective parameters for unreliable networks. This heuristic should be loosely based on similar measurements as the ones seen in 4.10 to enable a quick convergence towards parameters providing reliable connection establishment with a high probability.

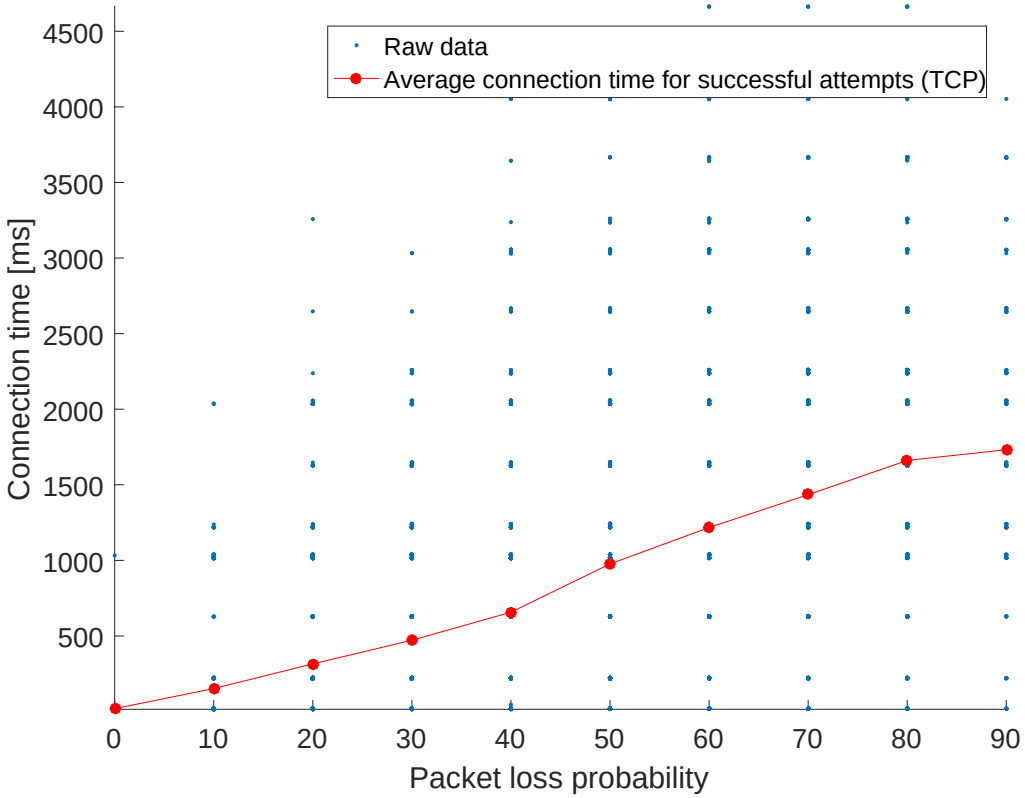

Figure 4.10: Average time needed to establish a connection (for successful attempts) using TCP and a REJECT policy

Chapter 5

Conclusion

In the course of this work we have presented an interpretation of the port-knocking concept with a focus on scalability in order to provide a solution that could significantly improve the adoption of port-knocking in real-world scenarios. We have described the strength and weaknesses of this concept and evaluated the performance and limitations of our implementation. In this last Chapter of the thesis, it is time to describe possible future work that could be done to improve *sKnock* and finally compare this implementation to the related work described in Chapter 1.

5.1 Future Work

First and foremost the limitations described in Section 3.3 should be addressed as the highest priority of future continued development. If the achievable performance of this implementation as shown in Chapter 4 is the biggest downside for the chosen deployment scenario, we recommend that before addressing other limitations the proposed specification be realized in C to lay the ground for a performance optimized port-knocking solution. In the process of re-implementing the design, one should carefully consider the implementation of the cryptographic operations, especially the choice of respective frameworks.

Another high-priority extension for the current concept should be the possibility of tracking the state of established connections to allow reasonable normal operation for UDP based applications (because of the port-closing timeout) and to enhance the security and performance for applications using TCP protocol. This would require to process all incoming packets to determine if they belong to an already established connection protected by the port-knocking security layer. In case of TCP the rule allowing the handshake to occur can be removed immediately after the sequence completes, as all subsequent traffic related to the connection could be permitted by a catch-all rule for

matching packets in related and established state. For UDP the timer responsible for closing the open port should be reset as long as related UDP packets are arriving at the socket. This locks down the time-window for the open UDP ports as much as possible without compromising the ability of the application to communicate as long as necessary.

Another major limitation of the current prototype is the lack of support for NAT-enabled environments. Since most home-setups rely on NAT to connect multiple devices to the Internet over a single connection, leaving out the support for Network Address Translation is not viable. At the current stage, the determination of the correct *public* client IP address has to be done by the application calling the port-knocking client library. It would definitely improve the ease-of-use characteristics to include a logic in the port-knocking client to determine the public IP address in a reliable way by itself. Determining the public IP address of the client could be achieved by implementing one of the proven and widely known algorithms for NAT-traversal such as STUN [70].

In order to prevent a slowdown of the general firewall operating speed, iptables should be extended with the *IPSet* module or replaced by the *nftables* variant as discussed in Section 4.1.2 of Chapter 4.

Furthermore, as conclusive from Section 3.2.2.4 starting a new thread for every incoming packet matching the minimum length is causing an unnecessarily high overhead. The behavior under load could be easily improved by replacing this part of the architecture using *queues* and *worker threads* to more efficiently distribute the workload over multiple threads. Additionally, a logic should be implemented to reduce the maximum number of worker threads if the server is running under high load in order to free resources for the actual services.

Moreover, the platform independence of the current state of the implementation is not sufficiently mature. Although the architecture is designed to simplify the modularization of platform-specific code, as of this writing the prototypic implementation is targeted only at operating systems based on Linux with iptables as firewall and OpenSSL as cryptographic library. The first step to improve this state would be to port the client to all major platforms, since the code complexity is only a fraction of the servers'. As soon as this task is accomplished one can focus on extending server-side platform support by adding one software- or platform-specific wrapper module at a time.

Finally, one consideration should go to legacy applications, where one could possibly have no access to their source code (anymore). In order to be able to protect these possibly vulnerable services, a launcher could be written to first perform the port-knocking and then launch the application, so it is decoupled completely from the port-knocking protocol and thus does not require any modification to sources or configuration.

5.2 Summary

In this work we specified a scalable design and developed a prototype implementation for authenticated port-knocking. It has the potential to be deployed in large-scale environments at real-world service providers on their own infrastructure or on leased third-party servers. The specification allows for decentralized authorization of clients without requiring an actual authentication an a per-user basis. However, there is nothing to prevent a provider to integrate the port-knocking security mechanism with their identity management environment to provide authentication before the application layer. Additionaly, the only communication overhead caused by this design is a single UDP packet per connection that has to be sent before the first application packet. Since this packet is modeled to be smaller than most MTU's (Maximum Transmission Unit) on the Internet, the probability for failing connection attempts or further delay because of fragmentation is negligible. At this point it should be mentioned, that the port-knocking request, if timed correctly, can be sent only a few milliseconds before the initial application packet, depending on the computation time required for the server to open the respective port. Through experiments we showed this to be about 11 ms for a modern desktop configured as a server running *sKnock*.

Since all of these characteristics have been achieved using standardized and widely-known technology such as X.509 certificates and UDP, it is highly unlikely that our solution would be inapplicable to any realistic target environment.

However, running in user-space and employing Elliptic Curve Cryptography as main concept for the encryption and authentication mechanisms has a noticeable impact on overall performance. This impact may be mitigated by migrating this implementation to C or Rust and optimizing the cryptographic operations to fit the respective requirements.

Conclusively, we can say that we designed a scalable solution for authenticated port-knocking, which is viable for usage in a real-world deployment. Additionally, properties of this concept have been proved by testing a prototypical implementation of the presented specification, *sKnock*, in various experiments.

The final rating of *sKnock* compared to the port-knocking implementations discussed at the beginning of this thesis can be found in Table 5.1.

Implementation	Performance	Overhead	Security	Scalability
knockdaemon	o	--	-	--
SilentKnock	++	o	o	--
fwknop	++	++	++	--
knockknock	+	++	++	--
Knock	++	-	++	--
sKnock	–	++	++	++

Table 5.1: Comparison of sKnock to other well-known port-knocking implementations

Appendix

Appendix A

Measurement Results

A.1 Per-Module Performance

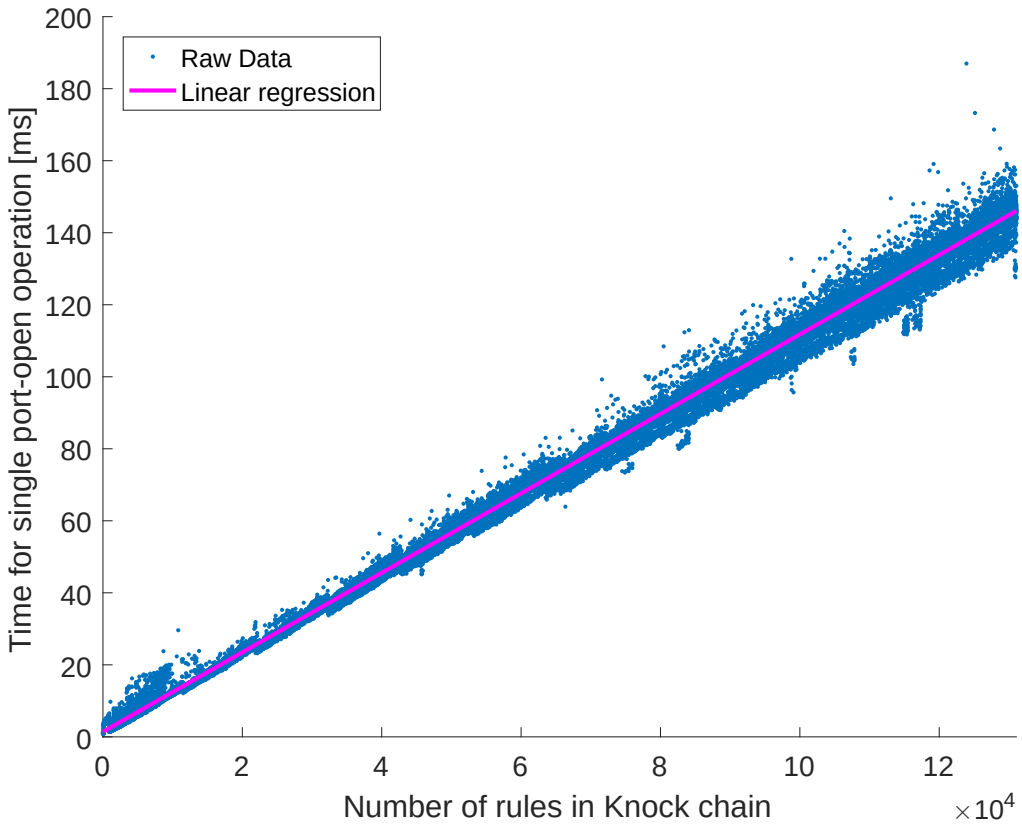

Figure A.1: Execution time for adding a rule to an iptables chain in relation to the number of active rules (IPv4)

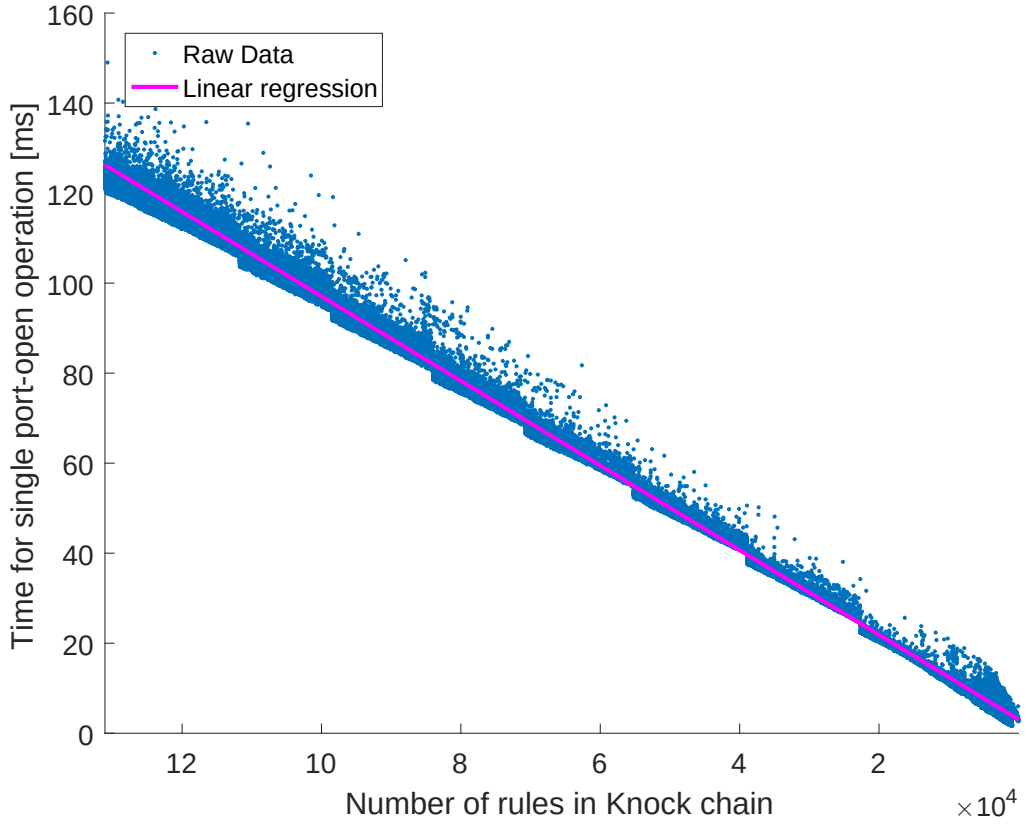

Figure A.2: Execution time for removing a rule from an iptables chain in relation to the number of active rules (IPv4)

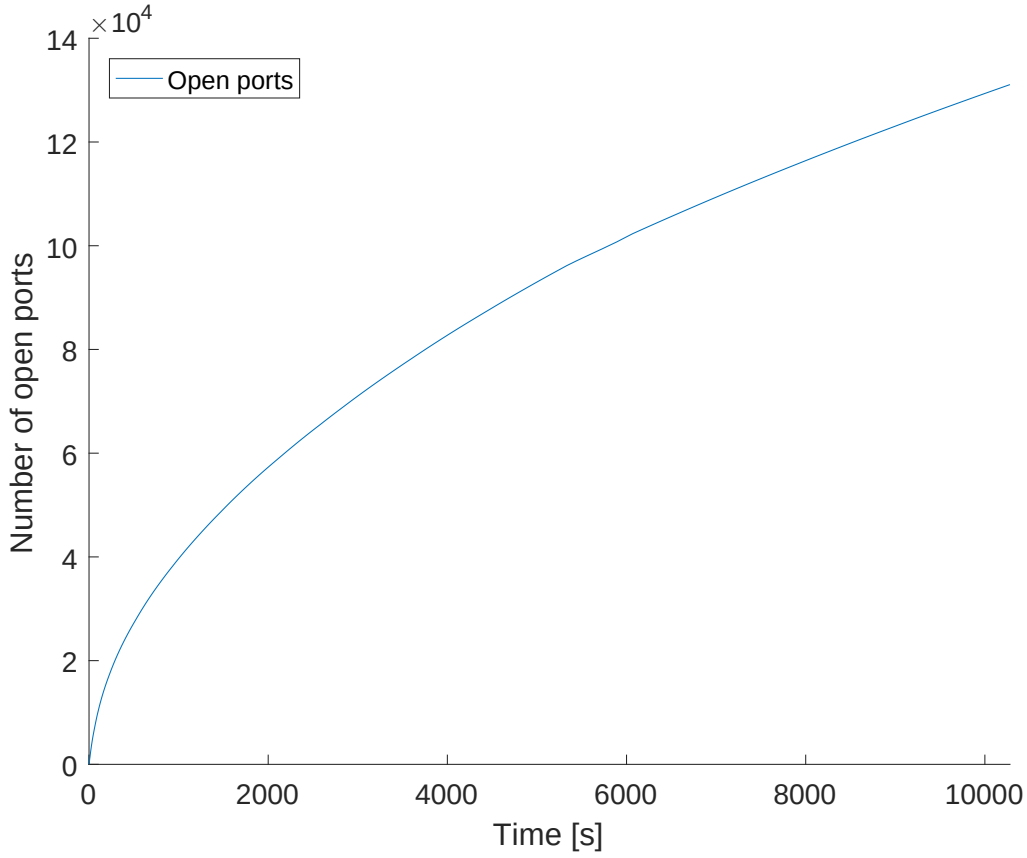

Figure A.3: Time for opening 131072 ports (65536 TCP + 65536 UDP) for an IPv6 client

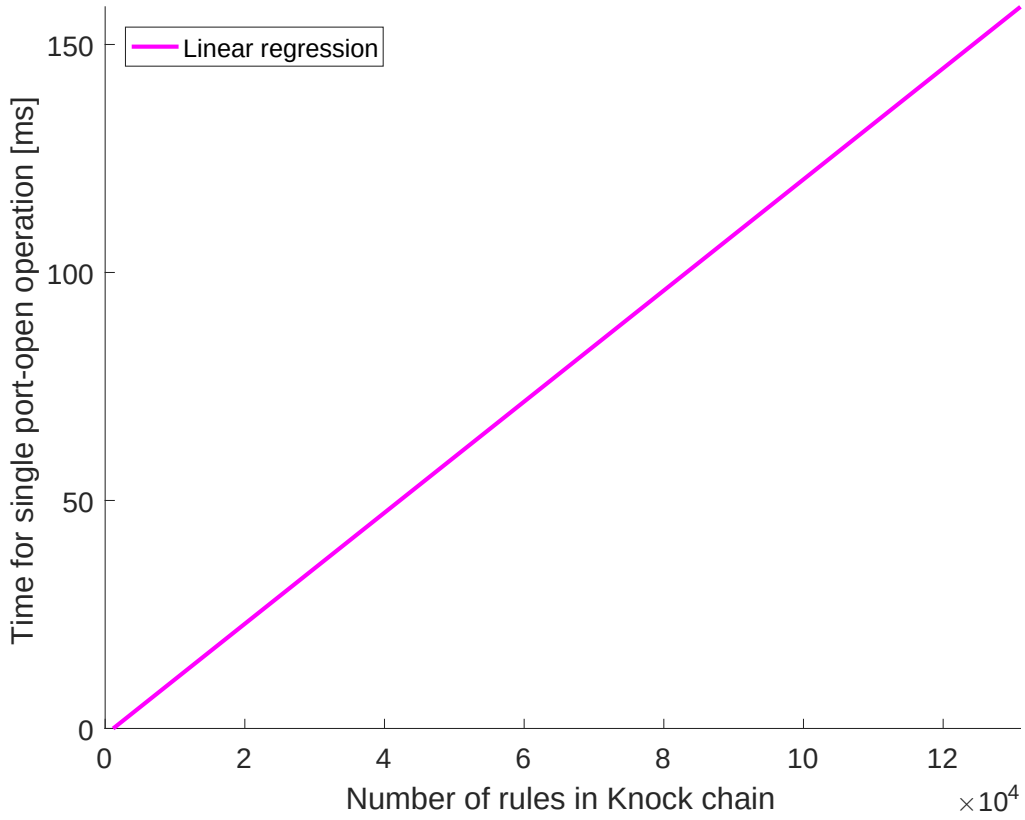

Figure A.4: Execution time for adding a rule to an iptables chain in relation to the number of active rules (IPv6)

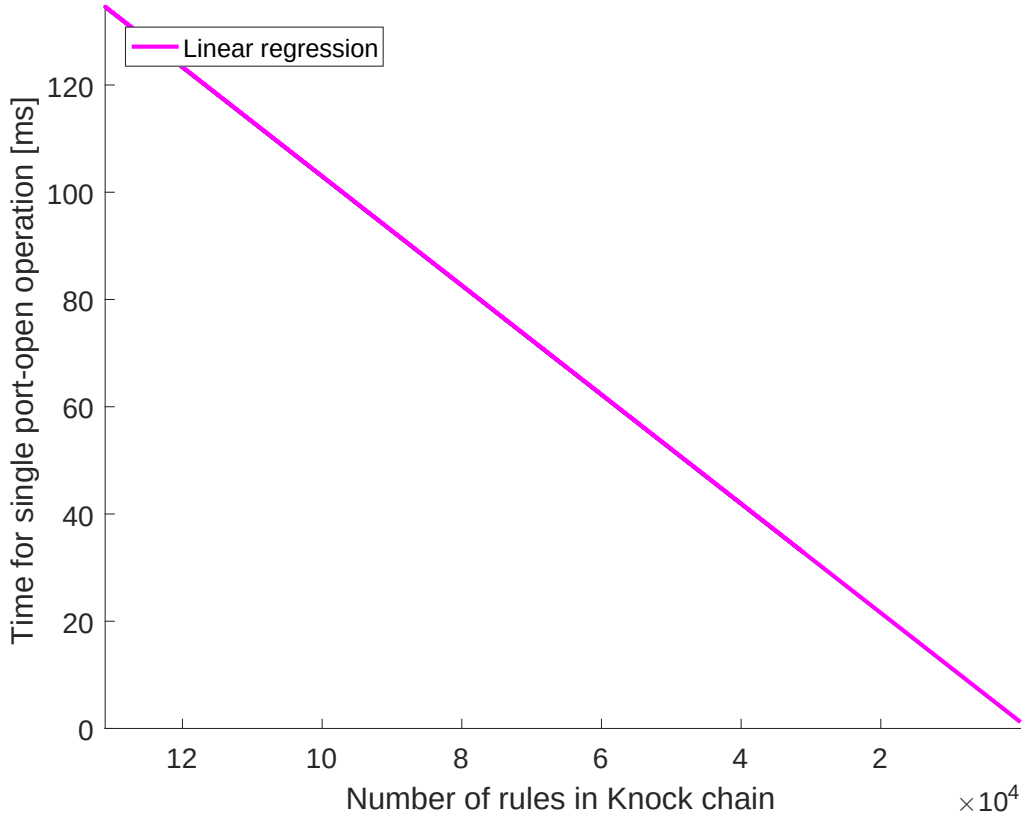

Figure A.5: Execution time for removing a rule from an iptables chain in relation to the number of active rules (IPv6)

A.2 Firewall Filtering

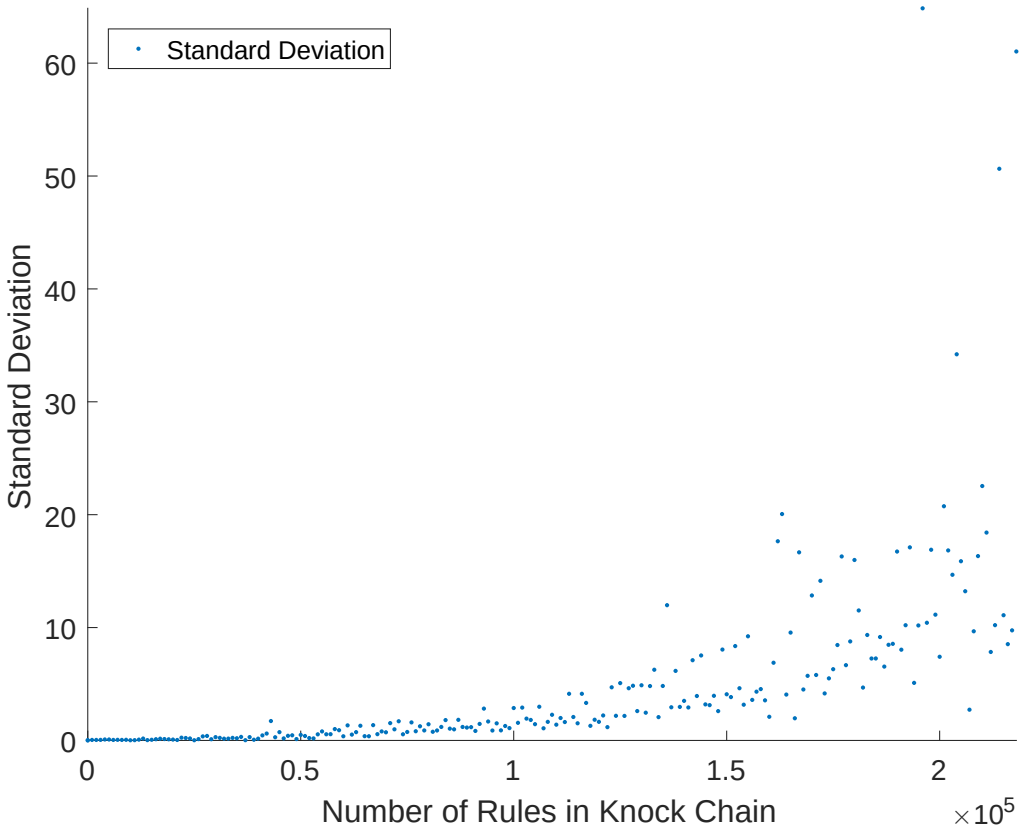

Figure A.6: Standard deviation for measurements concerning the firewall filtering performance

A.3 Reliability Under Packet Loss

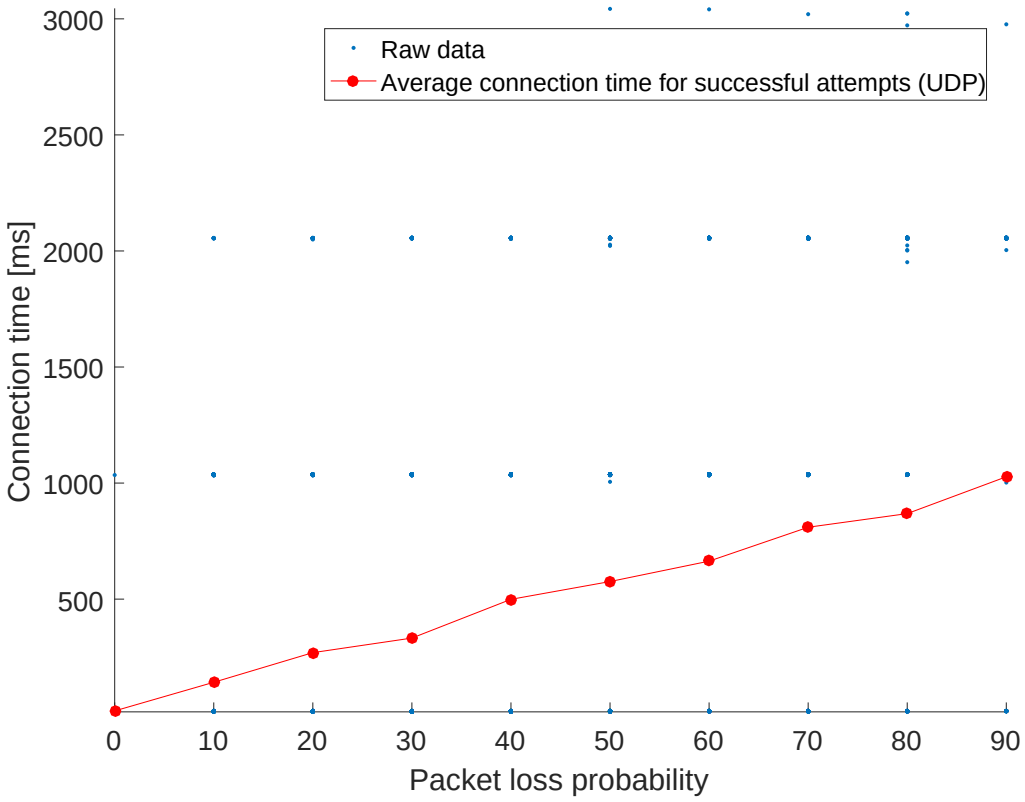

Figure A.7: Average time needed to establish a connection (for successful attempts) using UDP and a REJECT policy

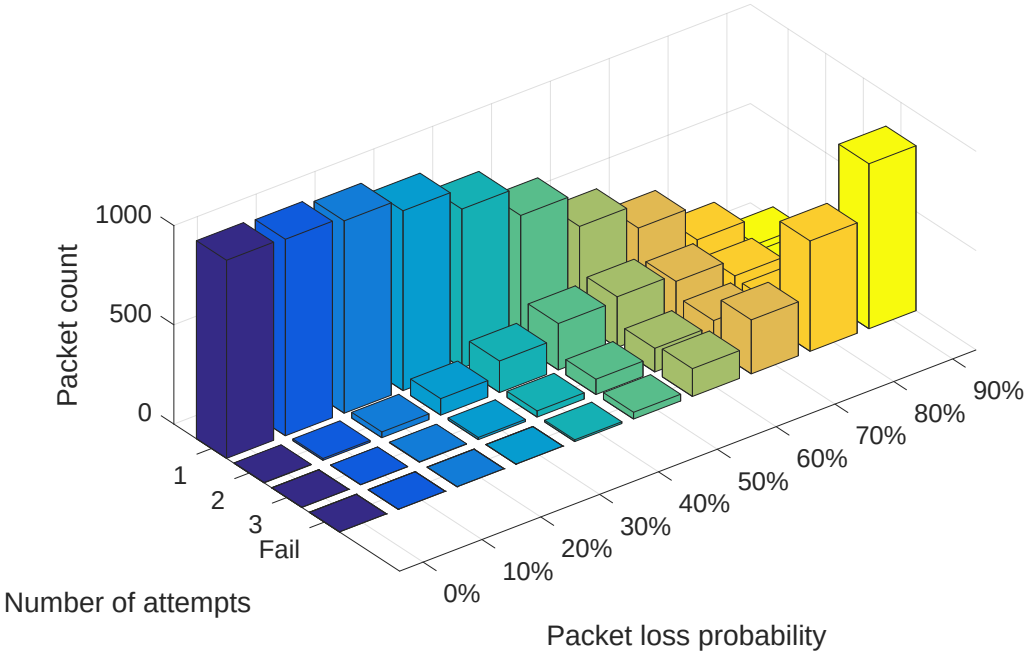

Figure A.8: Number of attempts for the TCP test application using port-knocking in relation to the packet loss probability with firewall policy set to REJECT.
Timeout: 1s, number of retries per iteration: 3, number of iterations: 1000

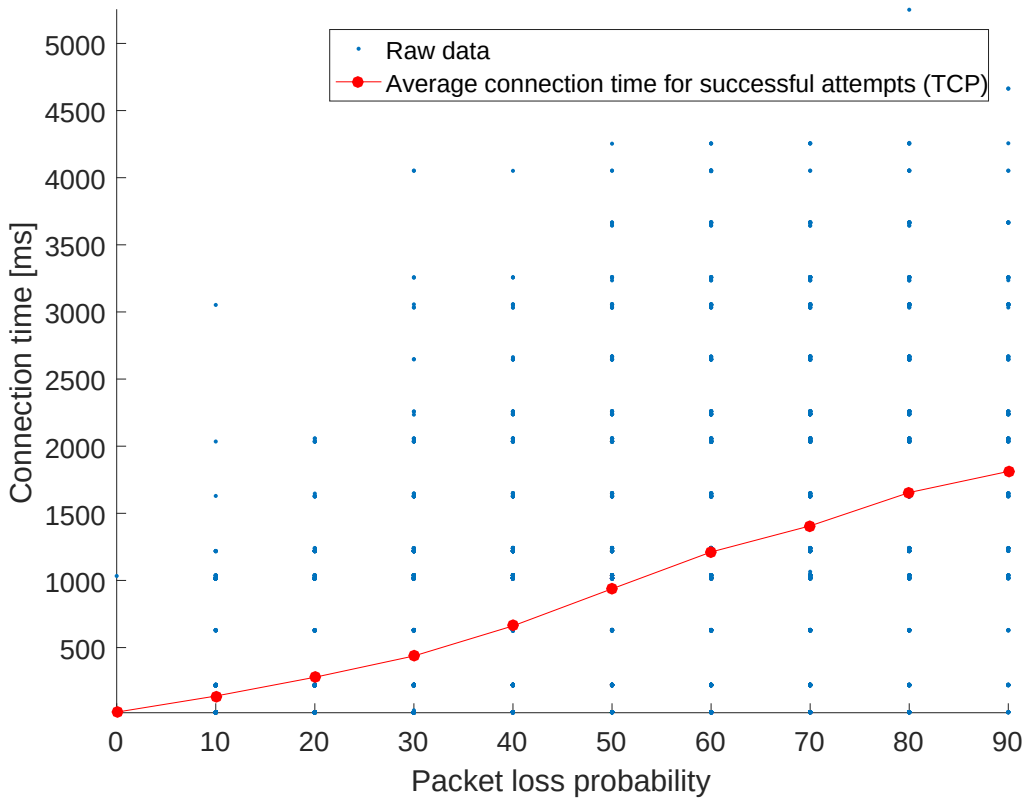

Figure A.9: Average time needed to establish a connection (for successful attempts) using TCP and a DROP policy

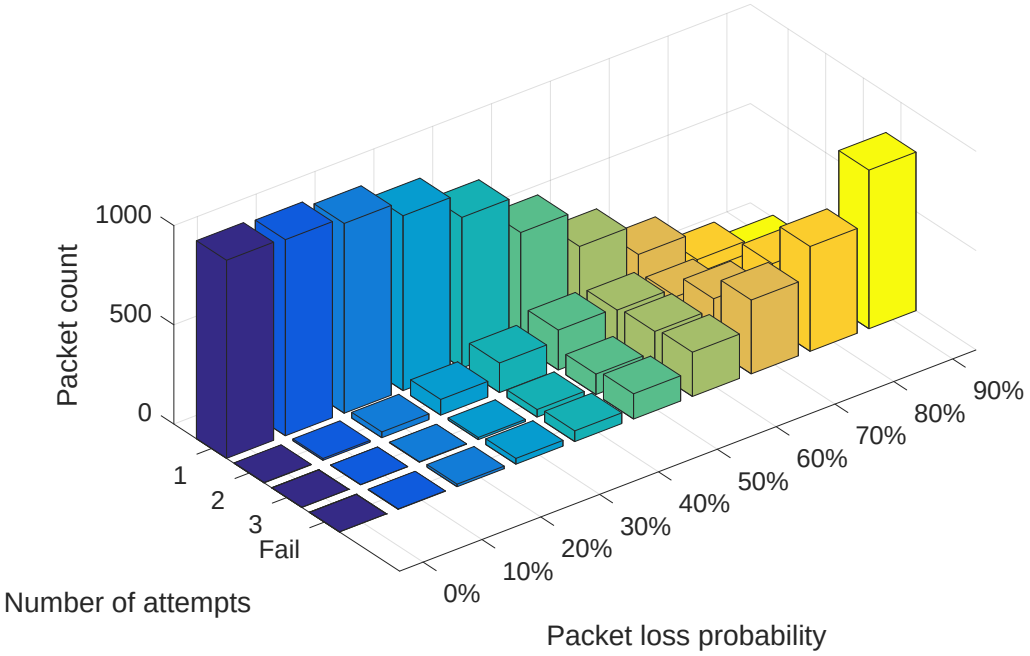

Figure A.10: Number of attempts for the UDP test application using port-knocking in relation to the packet loss probability with firewall policy set to REJECT.
Timeout: 1s, number of retries per iteration: 3, number of iterations: 1000

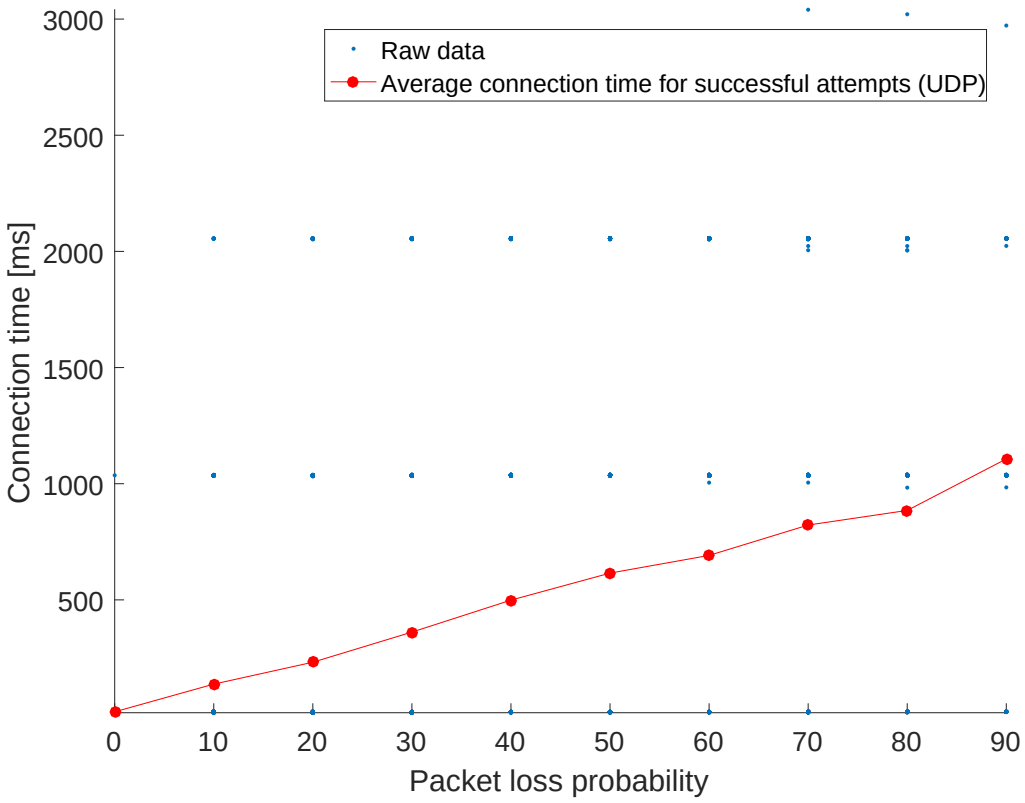

Figure A.11: Average time needed to establish a connection (for successful attempts) using UDP and a DROP policy

Appendix B

Profiling Results

Listing B.1: Profiler log for analysis of cryptographic engine (server side)

```
2016-03-14 20:41:00,684 - __main__ - INFO - Initializing...
2016-03-14 20:41:00,691 - __main__ - INFO - Computing CryptoEngine performance based on
    data set of 300000 packets
2016-03-14 20:49:50,094 - __main__ - INFO - Benchmark finished!
2016-03-14 20:49:50,094 - __main__ - INFO - Result: 529.401249s overall compuation time,
     1.764671ms time per packet, approx. 566 pps (packets per second)
Wrote profile results to ap_cryptoengine.py.lprof
Timer unit: 1e-06 s

Total time: 243.478 s
File: /home/daniel/knock/common/modules/CertUtil.py
Function: verifyCertificateAndSignature at line 43

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
43                                           @profile
44                                           def verifyCertificateAndSignature(self,
     rawCert, payloadSignature, payload):
45    300000       154453      0.5      0.1          try:
46    300000     11973616     39.9      4.9              cert = crypto.load_certificate(
    crypto.FILETYPE_ASN1, rawCert)
47                                               except:
48                                                   LOG.error("Invalid Certificate
    data!")
49                                                   return False
50
51    300000    231349733    771.2     95.0          return self.verifyCertificate(cert)
     and self.verifySignature(cert, payloadSignature, payload)

Total time: 121.945 s
File: /home/daniel/knock/common/modules/CertUtil.py
Function: verifyCertificate at line 54

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
54                                           @profile
55                                           def verifyCertificate(self, cert):
56    300000       224460      0.7      0.2          if(self.platform == PlatformUtils.
    LINUX):
```

```
57    300000      2149086      7.2      1.8              CAContext = crypto.
    X509StoreContext(self.CA, cert)
58    300000       149754      0.5      0.1              try:
59    300000    113100916    377.0     92.7                  CAContext.
    verify_certificate()
60    300000      1472158      4.9      1.2                  LOG.debug("Certificate␣OK!"
    )
61    300000      3701444     12.3      3.0                  if not (self.
    revokedCertificateSerials is None or format(cert.get_serial_number(), 'x').upper()
    in self.revokedCertificateSerials):
62    300000       991170      3.3      0.8                      LOG.debug("Certificate␣
    Revocation␣Status␣OK")
63    300000       156459      0.5      0.1                      return True
64                                                           else:
65                                                               LOG.warning("
    Certificate␣with␣Serial␣Number:␣%s␣is␣revoked!", cert.get_serial_number())
66                                                               return False
67
68                                                       except:
69                                                           LOG.debug("Certificate␣
    check␣failed!")
70                                                           return False

Total time: 104.62 s
File: /home/daniel/knock/common/modules/CertUtil.py
Function: verifySignature at line 73

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
73                                           @profile
74                                           def verifySignature(self, cert,
    signature, message):
75    300000       215710      0.7      0.2          if(self.platform == PlatformUtils.
    LINUX):
76    300000       121024      0.4      0.1              try:
77    300000    102997547    343.3     98.4                  crypto.verify(cert,
    signature, message, self.hashAlgorithm)
78    300000      1156619      3.9      1.1                  LOG.debug("Signature␣OK!")
79    300000       129464      0.4      0.1                  return True
80                                                       except:
81                                                           LOG.debug("Invalid␣
    Signature!")
82                                                           return False

Total time: 219.581 s
File: /home/daniel/knock/common/modules/CryptoEngine.py
Function: decryptWithECIES at line 37

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
37                                           @profile
38                                           def decryptWithECIES(self,
    encryptedMessage):
39    300000      1359335      4.5      0.6          LOG.debug("Calculating␣decryption␣
    key...")
40    300000       292716      1.0      0.1          ephPubKey = encryptedMessage[-91:]
41    300000       339900      1.1      0.2          encryptedMessage = encryptedMessage
    [0:-91]
42    300000    193225102    644.1     88.0          ecdhSecret = self.privKey.
    compute_dh_key(EC.load_pub_key_bio(BIO.MemoryBuffer(convertDERtoPEM(ephPubKey))))
43    300000     18605125     62.0      8.5          aesKey = self._hkdf(ecdhSecret)
```

```
44
45    300000      1405815      4.7      0.6          LOG.debug("Decrypting␣AES␣encrypted
      ␣request...")
46    300000      2721220      9.1      1.2          decrypt = EVP.Cipher(alg='
      aes_128_cbc', key=aesKey, iv = '\0' * 16, padding=1, op=0)
47    300000       828510      2.8      0.4          decryptedMessage = decrypt.update(
      encryptedMessage)
48    300000       652110      2.2      0.3          decryptedMessage += decrypt.final()
49
50    300000       151144      0.5      0.1          return decryptedMessage
```

Appendix C

Log Files

Listing C.1: Result log file for CryptoEngine single-threaded performance test

```
2016-02-29 16:01:00,726 - __main__ - INFO - Initializing...
2016-02-29 16:01:00,867 - __main__ - INFO - Computing CryptoEngine performance based on
    data set of 300000 packets
2016-02-29 16:09:04,584 - __main__ - INFO - Benchmark finished!
2016-02-29 16:09:04,584 - __main__ - INFO - Result: 483.715815s overall compuation time,
     1.612386ms time per packet, approx. 620 pps (packets per second)
```

Listing C.2: Result log file for CryptoEngine multi-threaded performance test

```
2016-02-29 10:06:05,577 - __main__ - WARNING - Initializing...
2016-02-29 10:06:05,798 - __main__ - WARNING - Computing CryptoEngine performance based
    on data set of 300000 packets
2016-02-29 10:25:08,140 - __main__ - WARNING - Benchmark finished!
2016-02-29 10:25:08,140 - __main__ - WARNING - Result: 1142.341816s overall compuation
    time, 3.807806ms time per packet, approx. 262 pps (packets per second)
```

Listing C.3: Result Log for Packet Processing Benchmark with 1% port-knocking requests, 5% large random packets, and 94% small random packets

```
2016-02-29 17:58:15,366 - __main__ - INFO - Initializing...
2016-02-29 17:58:15,366 - __main__ - INFO - Computing PacketProcessor performance based
    on data set of 5000000 packets
2016-02-29 18:11:51,894 - __main__ - INFO - Compute Time: 802.669472
2016-02-29 18:11:51,894 - __main__ - INFO - Thread wait time: 13.858088
2016-02-29 18:11:51,894 - __main__ - INFO - Benchmark finished!
2016-02-29 18:11:51,894 - __main__ - INFO - Result: 816.527560s overall compuation time,
     0.163306ms time per packet, approx. 6123 pps (packets per second)
```

Listing C.4: Result Log for Packet Processing Benchmark with 100% port-knocking requests, 0% large random packets, and 0% small random packets

```
2016-02-29 15:21:55,821 - __main__ - INFO - Initializing...
2016-02-29 15:21:55,821 - __main__ - INFO - Computing PacketProcessor performance based
    on data set of 3000000 packets
2016-02-29 15:31:12,323 - __main__ - INFO - Compute Time: 548.267900
2016-02-29 15:31:12,324 - __main__ - INFO - Thread wait time: 8.234358
2016-02-29 15:31:12,324 - __main__ - INFO - Benchmark finished!
```

```
2016-02-29 15:31:12,324 - __main__ - INFO - Result: 556.502258s overall compuation time,
      0.185501ms time per packet, approx. 5390 pps (packets per second)
```

Listing C.5: Excerpt from Logfile for PTP Time-Synchronization

```
2016-03-08 23:16:15:284863, slv, 64006afffe521cba/01, 0.000000000, 0.000003369,
     0.000000000, 0.000351000, 711
2016-03-08 23:16:16:284753, slv, 64006afffe521cba/01, 0.000000000, 0.000007996,
     0.000000000, 0.000354000, 718
2016-03-08 23:16:17:284845, slv, 64006afffe521cba/01, 0.000000000, 0.000007077,
     0.000000000, 0.000349000, 725
2016-03-08 23:16:18:284612, slv, 64006afffe521cba/01, 0.000000000, -0.000010842,
     0.000000000, 0.000318000, 715
2016-03-08 23:16:19:284846, slv, 64006afffe521cba/01, 0.000000000, -0.000010321,
     0.000000000, 0.000350000, 705
2016-03-08 23:16:20:284747, slv, 64006afffe521cba/01, 0.000000000, 0.000006200,
     0.000000000, 0.000351000, 711
2016-03-08 23:16:21:284849, slv, 64006afffe521cba/01, 0.000000000, 0.000005638,
     0.000000000, 0.000349000, 716
2016-03-08 23:16:22:284730, slv, 64006afffe521cba/01, 0.000000000, 0.000004076,
     0.000000000, 0.000348000, 720
2016-03-08 23:16:23:284796, slv, 64006afffe521cba/01, 0.000000000, 0.000002486,
     0.000000000, 0.000346000, 722
2016-03-08 23:16:24:284633, slv, 64006afffe521cba/01, 0.000000000, -0.000001603,
     0.000000000, 0.000340000, 721
2016-03-08 23:16:25:284858, slv, 64006afffe521cba/01, 0.000000000, 0.000001289,
     0.000000000, 0.000352000, 722
2016-03-08 23:16:26:284742, slv, 64006afffe521cba/01, 0.000000000, 0.000005680,
     0.000000000, 0.000349000, 727
2016-03-08 23:16:27:284858, slv, 64006afffe521cba/01, 0.000000000, 0.000008091,
     0.000000000, 0.000357000, 735
2016-03-08 23:16:28:284740, slv, 64006afffe521cba/01, 0.000000000, 0.000007502,
     0.000000000, 0.000348000, 742
2016-03-08 23:16:29:284779, slv, 64006afffe521cba/01, 0.000000000, -0.000031473,
     0.000000000, 0.000279000, 711
2016-03-08 23:16:30:284749, slv, 64006afffe521cba/01, 0.000000000, -0.000030949,
     0.000000000, 0.000349000, 681
2016-03-08 23:16:31:284859, slv, 64006afffe521cba/01, 0.000000000, 0.000006585,
     0.000000000, 0.000354000, 687
2016-03-08 23:16:32:284805, slv, 64006afffe521cba/01, 0.000000000, 0.000039120,
     0.000000000, 0.000414000, 726
2016-03-08 23:16:33:284657, slv, 64006afffe521cba/01, 0.000000000, -0.000006882,
     0.000000000, 0.000262000, 720
2016-03-08 23:16:34:284744, slv, 64006afffe521cba/01, 0.000000000, -0.000036884,
     0.000000000, 0.000354000, 684
2016-03-08 23:16:35:284870, slv, 64006afffe521cba/01, 0.000000000, 0.000009687,
     0.000000000, 0.000355000, 693
2016-03-08 23:16:36:284750, slv, 64006afffe521cba/01, 0.000000000, 0.000008758,
     0.000000000, 0.000352000, 701
2016-03-08 23:16:37:284787, slv, 64006afffe521cba/01, 0.000000000, -0.000016147,
     0.000000000, 0.000305000, 685
2016-03-08 23:16:38:284670, slv, 64006afffe521cba/01, 0.000000000, -0.000015553,
     0.000000000, 0.000353000, 670
2016-03-08 23:16:39:284865, slv, 64006afffe521cba/01, 0.000000000, 0.000008933,
     0.000000000, 0.000354000, 678
2016-03-08 23:16:40:284753, slv, 64006afffe521cba/01, 0.000000000, 0.000008920,
     0.000000000, 0.000353000, 686
2016-03-08 23:16:41:284837, slv, 64006afffe521cba/01, 0.000000000, 0.000008918,
     0.000000000, 0.000354000, 694
2016-03-08 23:16:42:284743, slv, 64006afffe521cba/01, 0.000000000, 0.000007916,
     0.000000000, 0.000351000, 701
```

```
2016-03-08 23:16:43:284767, slv, 64006afffe521cba/01, 0.000000000, -0.000033061,
    0.000000000, 0.000272000, 668
2016-03-08 23:16:44:284662, slv, 64006afffe521cba/01, 0.000000000, -0.000034538,
    0.000000000, 0.000348000, 634
2016-03-08 23:16:45:284872, slv, 64006afffe521cba/01, 0.000000000, 0.000008998,
    0.000000000, 0.000359000, 642
2016-03-08 23:16:46:284772, slv, 64006afffe521cba/01, 0.000000000, 0.000015034,
    0.000000000, 0.000360000, 657
2016-03-08 23:16:47:284749, slv, 64006afffe521cba/01, 0.000000000, -0.000003922,
    0.000000000, 0.000321000, 654
2016-03-08 23:16:48:284676, slv, 64006afffe521cba/01, 0.000000000, -0.000041378,
    0.000000000, 0.000285000, 613
2016-03-08 23:16:49:284872, slv, 64006afffe521cba/01, 0.000000000, 0.000013189,
    0.000000000, 0.000430000, 626
2016-03-08 23:16:50:284794, slv, 64006afffe521cba/01, 0.000000000, 0.000051256,
    0.000000000, 0.000361000, 677
2016-03-08 23:16:51:284876, slv, 64006afffe521cba/01, 0.000000000, 0.000010272,
    0.000000000, 0.000348000, 687
2016-03-08 23:16:52:284760, slv, 64006afffe521cba/01, 0.000000000, 0.000009288,
    0.000000000, 0.000359000, 696
2016-03-08 23:16:53:284854, slv, 64006afffe521cba/01, 0.000000000, 0.000012745,
    0.000000000, 0.000355000, 708
```

Listing C.6: Logfile from calibration to 12ms wait time

```
2016-03-06 04:46:56,850 - __main__ - INFO - Calibrating wait time...
2016-03-06 04:46:56,850 - __main__ - DEBUG - Iteration 1, Run 1
2016-03-06 04:46:58,067 - __main__ - DEBUG - Iteration 1, Run 2
2016-03-06 04:46:59,287 - __main__ - DEBUG - Iteration 1, Run 3
2016-03-06 04:47:00,507 - __main__ - DEBUG - Iteration 1, Run 4
2016-03-06 04:47:01,727 - __main__ - DEBUG - Iteration 1, Run 5
2016-03-06 04:47:02,946 - __main__ - DEBUG - Iteration 1, Run 6
2016-03-06 04:47:04,167 - __main__ - DEBUG - Iteration 1, Run 7
2016-03-06 04:47:05,387 - __main__ - DEBUG - Iteration 1, Run 8
2016-03-06 04:47:06,606 - __main__ - DEBUG - Iteration 1, Run 9
2016-03-06 04:47:07,827 - __main__ - DEBUG - Iteration 1, Run 10
2016-03-06 04:47:09,047 - __main__ - DEBUG - Iteration 1, Run 11
2016-03-06 04:47:10,266 - __main__ - DEBUG - Iteration 1, Run 12
2016-03-06 04:47:11,487 - __main__ - DEBUG - Iteration 1, Run 13
2016-03-06 04:47:13,506 - __main__ - INFO - Request timed out.
2016-03-06 04:47:13,506 - __main__ - DEBUG - Iteration 1, Run 14
2016-03-06 04:47:14,726 - __main__ - DEBUG - Iteration 1, Run 15
2016-03-06 04:47:15,946 - __main__ - DEBUG - Iteration 1, Run 16
2016-03-06 04:47:17,166 - __main__ - DEBUG - Iteration 1, Run 17
2016-03-06 04:47:18,386 - __main__ - DEBUG - Iteration 1, Run 18
2016-03-06 04:47:19,606 - __main__ - DEBUG - Iteration 1, Run 19
2016-03-06 04:47:20,826 - __main__ - DEBUG - Iteration 1, Run 20
2016-03-06 04:47:22,046 - __main__ - DEBUG - Iteration 1, Run 21
2016-03-06 04:47:23,266 - __main__ - DEBUG - Iteration 1, Run 22
2016-03-06 04:47:24,486 - __main__ - DEBUG - Iteration 1, Run 23
2016-03-06 04:47:25,706 - __main__ - DEBUG - Iteration 1, Run 24
2016-03-06 04:47:26,926 - __main__ - DEBUG - Iteration 1, Run 25
2016-03-06 04:47:28,146 - __main__ - DEBUG - Iteration 1, Run 26
2016-03-06 04:47:29,366 - __main__ - DEBUG - Iteration 1, Run 27
2016-03-06 04:47:30,586 - __main__ - DEBUG - Iteration 1, Run 28
2016-03-06 04:47:31,806 - __main__ - DEBUG - Iteration 1, Run 29
2016-03-06 04:47:33,026 - __main__ - DEBUG - Iteration 1, Run 30
2016-03-06 04:47:34,246 - __main__ - DEBUG - Iteration 1, Run 31
2016-03-06 04:47:35,466 - __main__ - DEBUG - Iteration 1, Run 32
2016-03-06 04:47:36,686 - __main__ - DEBUG - Iteration 1, Run 33
2016-03-06 04:47:37,906 - __main__ - DEBUG - Iteration 1, Run 34
```

```
2016-03-06 04:47:39,126 - __main__ - DEBUG - Iteration 1, Run 35
2016-03-06 04:47:40,346 - __main__ - DEBUG - Iteration 1, Run 36
2016-03-06 04:47:41,566 - __main__ - DEBUG - Iteration 1, Run 37
2016-03-06 04:47:42,786 - __main__ - DEBUG - Iteration 1, Run 38
2016-03-06 04:47:44,006 - __main__ - DEBUG - Iteration 1, Run 39
2016-03-06 04:47:45,226 - __main__ - DEBUG - Iteration 1, Run 40
2016-03-06 04:47:46,446 - __main__ - DEBUG - Iteration 1, Run 41
2016-03-06 04:47:47,666 - __main__ - DEBUG - Iteration 1, Run 42
2016-03-06 04:47:48,886 - __main__ - DEBUG - Iteration 1, Run 43
2016-03-06 04:47:50,106 - __main__ - DEBUG - Iteration 1, Run 44
2016-03-06 04:47:51,326 - __main__ - DEBUG - Iteration 1, Run 45
2016-03-06 04:47:52,546 - __main__ - DEBUG - Iteration 1, Run 46
2016-03-06 04:47:53,766 - __main__ - DEBUG - Iteration 1, Run 47
2016-03-06 04:47:54,986 - __main__ - DEBUG - Iteration 1, Run 48
2016-03-06 04:47:56,206 - __main__ - DEBUG - Iteration 1, Run 49
2016-03-06 04:47:57,426 - __main__ - DEBUG - Iteration 1, Run 50
2016-03-06 04:47:58,646 - __main__ - DEBUG - Iteration 1, Run 51
2016-03-06 04:47:59,866 - __main__ - DEBUG - Iteration 1, Run 52
2016-03-06 04:48:01,086 - __main__ - DEBUG - Iteration 1, Run 53
2016-03-06 04:48:02,306 - __main__ - DEBUG - Iteration 1, Run 54
2016-03-06 04:48:03,526 - __main__ - DEBUG - Iteration 1, Run 55
2016-03-06 04:48:04,746 - __main__ - DEBUG - Iteration 1, Run 56
2016-03-06 04:48:05,967 - __main__ - DEBUG - Iteration 1, Run 57
2016-03-06 04:48:07,187 - __main__ - DEBUG - Iteration 1, Run 58
2016-03-06 04:48:08,406 - __main__ - DEBUG - Iteration 1, Run 59
2016-03-06 04:48:09,626 - __main__ - DEBUG - Iteration 1, Run 60
2016-03-06 04:48:10,846 - __main__ - DEBUG - Iteration 1, Run 61
2016-03-06 04:48:12,066 - __main__ - DEBUG - Iteration 1, Run 62
2016-03-06 04:48:13,286 - __main__ - DEBUG - Iteration 1, Run 63
2016-03-06 04:48:14,506 - __main__ - DEBUG - Iteration 1, Run 64
2016-03-06 04:48:15,726 - __main__ - DEBUG - Iteration 1, Run 65
2016-03-06 04:48:16,946 - __main__ - DEBUG - Iteration 1, Run 66
2016-03-06 04:48:18,166 - __main__ - DEBUG - Iteration 1, Run 67
2016-03-06 04:48:19,386 - __main__ - DEBUG - Iteration 1, Run 68
2016-03-06 04:48:20,606 - __main__ - DEBUG - Iteration 1, Run 69
2016-03-06 04:48:21,826 - __main__ - DEBUG - Iteration 1, Run 70
2016-03-06 04:48:23,046 - __main__ - DEBUG - Iteration 1, Run 71
2016-03-06 04:48:24,266 - __main__ - DEBUG - Iteration 1, Run 72
2016-03-06 04:48:25,486 - __main__ - DEBUG - Iteration 1, Run 73
2016-03-06 04:48:26,706 - __main__ - DEBUG - Iteration 1, Run 74
2016-03-06 04:48:27,926 - __main__ - DEBUG - Iteration 1, Run 75
2016-03-06 04:48:29,146 - __main__ - DEBUG - Iteration 1, Run 76
2016-03-06 04:48:30,366 - __main__ - DEBUG - Iteration 1, Run 77
2016-03-06 04:48:31,586 - __main__ - DEBUG - Iteration 1, Run 78
2016-03-06 04:48:32,806 - __main__ - DEBUG - Iteration 1, Run 79
2016-03-06 04:48:34,026 - __main__ - DEBUG - Iteration 1, Run 80
2016-03-06 04:48:35,246 - __main__ - DEBUG - Iteration 1, Run 81
2016-03-06 04:48:36,466 - __main__ - DEBUG - Iteration 1, Run 82
2016-03-06 04:48:37,686 - __main__ - DEBUG - Iteration 1, Run 83
2016-03-06 04:48:38,906 - __main__ - DEBUG - Iteration 1, Run 84
2016-03-06 04:48:40,126 - __main__ - DEBUG - Iteration 1, Run 85
2016-03-06 04:48:41,346 - __main__ - DEBUG - Iteration 1, Run 86
2016-03-06 04:48:42,566 - __main__ - DEBUG - Iteration 1, Run 87
2016-03-06 04:48:43,786 - __main__ - DEBUG - Iteration 1, Run 88
2016-03-06 04:48:45,006 - __main__ - DEBUG - Iteration 1, Run 89
2016-03-06 04:48:46,226 - __main__ - DEBUG - Iteration 1, Run 90
2016-03-06 04:48:47,446 - __main__ - DEBUG - Iteration 1, Run 91
2016-03-06 04:48:49,466 - __main__ - INFO - Request timed out.
2016-03-06 04:48:49,466 - __main__ - DEBUG - Iteration 1, Run 92
2016-03-06 04:48:50,686 - __main__ - DEBUG - Iteration 1, Run 93
2016-03-06 04:48:51,906 - __main__ - DEBUG - Iteration 1, Run 94
```

```
2016-03-06 04:48:53,126 - __main__ - DEBUG - Iteration 1, Run 95
2016-03-06 04:48:54,346 - __main__ - DEBUG - Iteration 1, Run 96
2016-03-06 04:48:55,566 - __main__ - DEBUG - Iteration 1, Run 97
2016-03-06 04:48:56,786 - __main__ - DEBUG - Iteration 1, Run 98
2016-03-06 04:48:58,006 - __main__ - DEBUG - Iteration 1, Run 99
2016-03-06 04:48:59,226 - __main__ - DEBUG - Iteration 1, Run 100
2016-03-06 04:49:00,446 - __main__ - DEBUG - Calibration - number of failures: 2
2016-03-06 04:49:00,446 - __main__ - DEBUG - Calibration - SMALLER & < 0.1
```

Listing C.7: Client Logfile from Overhead Measurement for UDP at 12ms wait time

```
2016-03-06 05:24:58,242 - __main__ - INFO - Request timed out.
2016-03-06 06:35:42,966 - __main__ - INFO - Request timed out.
2016-03-06 06:43:34,596 - __main__ - INFO - Request timed out.
2016-03-06 07:11:05,351 - __main__ - INFO - Request timed out.
2016-03-06 07:22:52,817 - __main__ - INFO - Request timed out.
2016-03-06 08:17:54,280 - __main__ - INFO - Request timed out.
2016-03-06 08:29:41,730 - __main__ - INFO - Request timed out.
2016-03-06 08:49:20,841 - __main__ - INFO - Request timed out.
2016-03-06 09:08:59,934 - __main__ - INFO - Request timed out.
2016-03-06 09:54:51,175 - __main__ - INFO - Request timed out.
2016-03-06 10:17:46,796 - __main__ - INFO - Request timed out.
2016-03-06 11:27:12,983 - __main__ - INFO - Request timed out.
2016-03-06 13:48:42,499 - __main__ - INFO - Request timed out.
2016-03-06 14:00:29,958 - __main__ - INFO - Request timed out.
2016-03-06 15:26:57,949 - __main__ - INFO - Request timed out.
2016-03-06 15:50:32,874 - __main__ - INFO - Request timed out.
2016-03-06 16:18:03,621 - __main__ - INFO - Request timed out.
2016-03-06 16:25:55,254 - __main__ - INFO - Request timed out.
2016-03-06 16:45:34,338 - __main__ - INFO - Request timed out.
2016-03-06 16:47:53,086 - __main__ - DEBUG - Signal 2 received
2016-03-06 16:47:53,086 - __main__ - INFO - Stopping client...
2016-03-06 16:47:53,086 - __main__ - INFO - Total number of sent port-knocking requests:
     14129
2016-03-06 16:47:53,086 - __main__ - INFO - Total number of failed port-knocking
    requests: 19
```

Bibliography

[1] M. Griffiths-Harvey, B. Neill, K. Smith, T. Rosati, W. Davis, A. Walters, R. Tsang, D. Brown, and S. Vanstone, "Authenticated radio frequency identification and key distribution system therefor," Mar. 13 2008, wO Patent App. PCT/CA2007/001,567. [Online]. Available: http://www.google.com/patents/WO2008028291A1?cl=en

[2] T. Oder, T. Pöppelmann, and T. Güneysu, "Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[3] B. Daya, "Network security: History, importance, and future," *University of Florida Department of Electrical and Computer Engineering*, 2013.

[4] D. Bandyopadhyay and J. Sen, "Internet of things: Applications and challenges in technology and standardization," *Wireless Personal Communications*, vol. 58, no. 1, pp. 49–69, 2011.

[5] G. Pallis, "Cloud computing: the new frontier of internet computing," *IEEE Internet Computing*, vol. 14, no. 5, p. 70, 2010.

[6] A. Behl, "Emerging security challenges in cloud computing: An insight to cloud security challenges and their mitigation," in *Information and Communication Technologies (WICT), 2011 World Congress on*. IEEE, 2011, pp. 217–222.

[7] Z. Whittaker. (2015) Amazon force-resets some account passwords, citing password leak. (Retrieved: 02/06/2016). [Online]. Available: http://www.zdnet.com/article/amazon-is-resetting-account-passwords-for-some-accounts/

[8] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 1980.

[9] P. Lindstrom and R. Director, "Intrusion prevention systems (ips): Next generation firewalls," *Spire Security*, 2004.

[10] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[11] D. Gollmann, "Computer security," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 5, pp. 544–554, 2010.

[12] M. Krzywinski, "Port knocking from the inside out," *SysAdmin Magazine*, vol. 12, no. 6, pp. 12–17, 2003.

[13] A. Kumar, "Zero day exploit," *Available at SSRN 2378317*, 2014.

[14] M. Rash, "Single packet authorization with fwknop," *login: The USENIX Magazine*, vol. 31, no. 1, pp. 63–69, 2006.

[15] H. Al-Bahadili and A. H. Hadi, "Network security using hybrid port knocking," *IJCSNS*, vol. 10, no. 8, p. 8, 2010.

[16] M. Krzywinski. (2003, Jul.) Portknocking (by martin krzywinski) - documentation. (Retrieved: 03/11/2016). [Online]. Available: http://www.portknocking.org/view/documentation

[17] E. Y. Vasserman, N. Hopper, J. Laxson, and J. Tyra, "Silentknock: practical, provably undetectable authentication," in *Computer Security–ESORICS 2007*. Springer, 2007, pp. 122–138.

[18] M. Rash. (2016, Jan.) A comprehensive guide to strong service concealment with fwknop. (Retrieved: 03/11/2016). [Online]. Available: http://www.cipherdyne.org/fwknop/docs/fwknop-tutorial.html

[19] M. Marlinspike. Software » knockknock.

[20] J. Kirsch, "Improved kernel-based port-knocking in linux," *Master's Thesis (TUM)*, 2014.

[21] W. Mao, *Modern cryptography: theory and practice*. Prentice Hall Professional Technical Reference, 2003.

[22] G. J. Simmons, "Symmetric and asymmetric encryption," *ACM Computing Surveys (CSUR)*, vol. 11, no. 4, pp. 305–330, 1979.

[23] F. P. Miller, A. F. Vandome, and J. McBrewster, "Advanced encryption standard," 2009.

[24] S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu, "Internet x. 509 public key infrastructure certificate policy and certification practices framework," Tech. Rep., 2003.

[25] R. Rivest, A. Shamir, and L. Adleman, "Cryptographic communications system and method," Sep. 20 1983, uS Patent 4,405,829. [Online]. Available: https://www.google.com/patents/US4405829

[26] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[27] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Crypto*, vol. 99, no. 32. Springer, 1999, pp. 537–554.

[28] E. Rescorla, "Diffie-hellman key agreement method," 1999.

[29] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology—ASIACRYPT 2000*. Springer, 2000, pp. 531–545.

[30] H. Krawczyk, R. Canetti, and M. Bellare, "Hmac: Keyed-hashing for message authentication," 1997.

[31] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, "Rsa-oaep is secure under the rsa assumption," in *Advances in Cryptology—CRYPTO 2001*. Springer, 2001, pp. 260–274.

[32] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, "An efficient protocol for authenticated key agreement," *Designs, Codes and Cryptography*, vol. 28, no. 2, pp. 119–134, 2003.

[33] S. Kim, J. H. Cheon, M. Joye, S. Lim, M. Mambo, D. Won, and Y. Zheng, "Strong adaptive chosen-ciphertext attacks with memory dump (or: The importance of the order of decryption and validation)," in *Cryptography and Coding*. Springer, 2001, pp. 114–127.

[34] N. P. Smart, "The exact security of ecies in the generic group model," in *Cryptography and Coding*. Springer, 2001, pp. 73–84.

[35] D. Kravitz, "Digital signature algorithm," Jul. 27 1993, uS Patent 5,231,668. [Online]. Available: https://www.google.com/patents/US5231668

[36] E. DeBusschere and M. McCambridge, "Modern game console exploitation," 2012.

[37] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," in *Cryptographic Hardware and Embedded Systems–CHES 2011*. Springer, 2011, pp. 124–142.

[38] N. Jansma and B. Arrendondo, "Performance comparison of elliptic curve and rsa digital signatures," *nicj.net/files*, 2004.

[39] P. Syverson, "A taxonomy of replay attacks [cryptographic protocols]," in *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings.* IEEE, 1994, pp. 187–191.

[40] Y. Desmedt, "Man-in-the-middle attack," in *Encyclopedia of Cryptography and Security.* Springer, 2011, pp. 759–759.

[41] D. E. Denning and G. M. Sacco, "Timestamps in key distribution protocols," *Communications of the ACM*, vol. 24, no. 8, pp. 533–536, 1981.

[42] N. Haller, C. Metz, P. Nesser, and M. Straw, "A one-time password system," Tech. Rep., 1998.

[43] J. Postel, "Transmission control protocol," 1981.

[44] J. Postel, "Udp: User datagram protocol," 1980.

[45] C. Basso, J. L. Calvignac, M. C. Heddes, J. F. Logan, and F. J. Verplanken, "Data structures for efficient processing of ip fragmentation and reassembly," Aug. 30 2005, uS Patent 6,937,606.

[46] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating systems support for planetary-scale network services." in *NSDI*, vol. 4, 2004, pp. 19–19.

[47] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.

[48] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, 2004, pp. 57–72.

[49] T.-F. Fuh, S. H. Fan, and D. Qu, "Local authentication of a client at a network device," Oct. 8 2002, uS Patent 6,463,474.

[50] D. W. Chadwick, A. Otenko, and E. Ball, "Role-based access control with x. 509 attribute certificates," *Internet Computing, IEEE*, vol. 7, no. 2, pp. 62–69, 2003.

[51] A. B. Butt, P. B. Hillyard, and J. Su, "Certificate-based authentication system for heterogeneous environments," Jun. 22 2004, uS Patent 6,754,829.

[52] R. Housley, W. Polk, W. Ford, and D. Solo, "Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile," 2002.

[53] D. M. Nessett and W. P. Sherer, "Multilayer firewall system," Oct. 19 1999, uS Patent 5,968,176.

[54] G. Huston, "Tcp performance," *The Internet Protocol Journal*, vol. 3, no. 2, pp. 2–24, 2000.

[55] R. L. Cottrell, T. Barbosa, B. White, J. Abdullah, U. UMalaysia, T. White, R. Connection *et al.*, "Worldwide internet performance measurements using lightweight measurement platforms," SLAC National Accelerator Laboratory (SLAC), Tech. Rep., 2016.

[56] H. Krawczyk, "Cryptographic extraction and key derivation: The hkdf scheme," in *Advances in Cryptology–CRYPTO 2010*. Springer, 2010, pp. 631–648.

[57] S. Landau, "Highlights from making sense of snowden, part ii: What's significant in the nsa revelations," *Security & Privacy, IEEE*, vol. 12, no. 1, pp. 62–64, 2014.

[58] S. Josefsson, "Using Curve25519 and Curve448 in PKIX," Working Draft, IETF Secretariat, Internet-Draft draft-josefsson-pkix-newcurves-01, Oct. 2015.

[59] T. pyOpenSSL developers. (2016, Jan.) pyopenssl's documentation. (Retrieved: 02/21/2016). [Online]. Available: https://python-iptables.readthedocs.org/en/latest/intro.html

[60] T. pyOpenSSL developers. (2016, Feb.) pyopenssl's documentation. (Retrieved: 02/16/2016). [Online]. Available: https://pyopenssl.readthedocs.org/en/latest/index.html

[61] H. Toivonen. (2009, Aug.) M2crypto's documentation. (Retrieved: 02/16/2016). [Online]. Available: http://www.heikkitoivonen.net/m2crypto/api/

[62] S. Masini and P. Bientinesi, "High-performance parallel computations using python as high-level language," in *Euro-Par 2010 Parallel Processing Workshops*. Springer, 2010, pp. 541–548.

[63] K. RMKI and S. EK, "Netfilter performance testing."

[64] S. Suehring, *Linux Firewalls: Enhancing Security with Nftables and Beyond*. Addison-Wesley Professional, 2015.

[65] M. Zhang, M. Dusi, W. John, and C. Chen, "Analysis of udp traffic usage on internet backbone links," in *Applications and the Internet, 2009. SAINT'09. Ninth Annual International Symposium on*. IEEE, 2009, pp. 280–281.

[66] G. U. tiran. (2015, May) Segfault in x509 object from verify callback #273. (Retrieved: 03/11/2016). [Online]. Available: https://github.com/pyca/pyopenssl/issues/273

[67] R. Kern. (2015, Dec.) Github repository: line_profiler. (Retrieved: 03/14/2016). [Online]. Available: https://github.com/rkern/line_profiler

[68] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, pp. 1482–1493, 1991.

[69] H. Weibel, "High precision clock synchronization according to ieee 1588 implementation and performance issues," *Proc. Embedded World 2005*, 2005.

[70] J. Rosenberg, R. Mahy, C. Huitema, and J. Weinberger, “Stun-simple traversal of udp through network address translators,” 2003.