UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Implementat and evaluate cryptographic reverse firewalls with regard to their practicality

Implementieren und Evaluieren kryptographischer Reverse-Firewalls mit Hinblick auf ihre Praxistauglichkeit

Bachelorarbeit

Im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Adrian Lukas Billen

ausgegeben und betreut von
Dr. Sebastian Berndt und Thore Tiemann M. Sc.

Lübeck, den 17. August 2022

Abstract

As modern cryptography continues to evolve, we are also dealing with increasingly powerful attacks. One example of this is the algorithm substitution attack, in which a person's secret information can be revealed to third parties via regular network traffic after the implementations of the algorithms involved have been subverted. Cryptographic reverse firewalls have been introduced as a countermeasure to this attack. These pursue the purpose of cleansing outgoing messages of potentially embedded secrets. This firewall is considered part of the public and insecure channel, protecting a person's secrets without requiring them to trust it. The reverse firewall is deployed within a person's network and cleans up outgoing messages before they become visible to potential observers on the Internet.

The goal of this work is to answer the question of whether the concept of a reverse firewall is practical in terms of the additional time required to use it in everyday life. For this purpose, we implement the Diffie-Hellman key exchange and the El-Gamal encryption and measure the required runtime and the resulting additional time overhead on several devices when reverse firewalls are added to the protocols, starting with a CPU comparable to devices in the IoT category, continuing with a laptop and a PC, and ending with servers.

Zusammenfassung

Mit der fortschreitenden Entwicklung der modernen Kryptographie haben wir es auch mit immer stärkeren Angriffen zu tun. Ein Beispiel hierfür ist die algorithm substitution attack, bei der die geheimen Informationen einer Person über den regulären Netzverkehr an Dritte preisgegeben werden können, nachdem die Implementierungen der beteiligten Algorithmen unterwandert wurden. Als Gegenmaßnahme zu diesem Angriff wurden kryptografische reverse firewalls eingeführt. Diese verfolgen den Zweck, ausgehende Nachrichten von potenziell eingebetteten Geheimnissen zu bereinigen. Diese Firewall wird dabei als Teil des öffentlichen und unsicheren Kanals betrachtet und schützt die Geheimnisse einer Person, ohne dass diese ihr vertrauen muss. Die reverse firewall wird innerhalb des Netzwerks einer Person eingesetzt und bereinigt ausgehende Nachrichten, bevor diese für potenzielle Beobachter im Internet sichtbar werden.
Ziel dieser Arbeit ist es die Frage zu beantworten, ob das Konzept einer reverse firewall im Hinblick auf den entstehenden zeitlichen Mehraufwand für die Nutzung im Alltag praktikabel ist. Zu diesem Zweck implementieren wir den Diffie-Hellman-Schlüsselaustausch und die El-Gamal Verschlüsselung und messen auf mehreren Geräten die erforderliche Laufzeit und den entstehenden zeitlichen Mehraufwand bei Ergänzung der Protokolle um reverse firewalls, angefangen bei einer CPU vergleichbar mit Geräten der IoT-Kategorie, über einen Laptop und einen PC bis hin zu Servern.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 17. August 2022

Contents

1 Introduction 1
1.1 Contribution . . . 2
1.2 Structure . . . 2

2 Related Work 3

3 Background 5
3.1 Cryptographic Protocols . . . 5
3.2 Algorithm Substitution Attack . . . 5
3.3 Diffie–Hellman Key Exchange . . . 8
3.4 El-Gamal Public Key Encryption Scheme . . . 9
3.5 Commitment Scheme . . . 9
3.6 Rerandomizable Public Key Encryption . . . 10
3.7 Key-Malleability . . . 10
3.8 Cryptographic Reverse Firewalls . . . 11

4 Implementations 13
4.1 Diffie-Hellman Key Exchange . . . 13
4.2 El-Gamal . . . 19

5 Benchmarks 25

6 Discussion 31

7 Conclusions 33
7.1 Summary . . . 33

References 35

1 Introduction

The Internet enabled communication between two participants, overcoming limiting factors such as distance and time, as a large amount of information could be transmitted in seconds. Although- and perhaps because - the concept of the Internet was revolutionary, no further thought was given to how to sufficiently secure this new way of communicating.

This is surprising. As far back as ancient Rome, the concept of basic cryptography (particularly the Caesar cipher) was considered when transmitting sensitive information over insecure channels. Throughout history, a constant emergence and evolution of cryptography can be observed. Following this tradition, early flaws related to the security of the Internet were fixed using cryptography.

Over the next several decades, these cryptographic primitives and protocols evolved to the point where they were considered demonstrably secure. Encryption schemes such as AES and RSA, random number generators, hash functions such as SHA-256, and protocols such as TLS and Wireguard are well-known schemes that emerged from this development.

In June 2013, the trustworthiness of established algorithms had to be reconsidered. The revelations [BBG13, Gre14, Sch07] of former CIA employee Edward Snowden revealed the possibility of subverting the implementations of these secure algorithms in such a way that a most likely state attacker (i.e., intelligence agencies such as the NSA) could extract secret information from the target's computer via a steganographic channel. Following this example, Snowden also revealed that the NSA introduced a bias into the standardization of the `DUAL_EC_DRBG` pseudo-random number generator that allows the attacker to predict the following random numbers after observing the generation of a single one. This in turn leads to a loss of trust in these standardized algorithms.

The above incidents, as well as subsequent work on ASA, lead us to question whether we can trust the implementations of these secure algorithms on our computers. One countermeasure, preventing the exfiltration of our secret information and proposed in 2014, is the cryptographic reverse firewall. As part of our network, it sanitizes outgoing messages from potential embedded information. In this thesis, we implement a variety of reverse firewalls to test them for practicality on various systems, starting with slow devices representing IoT and smartphones, moving to laptops and PCs, and ending with servers.

1.1 Contribution

The contribution of this work includes the following points:

- We implement reverse firewalls with the goal of achieving exfiltration resistance for a variaty of cryptographic protocols in Sage [DSJ+20] and a chosen example in `C++` for comparison.
- We test our implementations for practicability in the context of runtime for a variety of security parameters and CPUs
- We discuss our results and give an outlook on possible future developments

1.2 Structure

This work is structured as follows:
In chapter 2 we discuss related work. Following this, we lay the foundation for this thesis by introducing the background of the corresponding fundamentals in chapter 3. In chapter 4 we take a closer look at the implementations of our reverse firewalls and present the benchmark results in chapter 5. We then discuss these results in chapter 6. The conclusion of this thesis is presented in chapter 7, followed by an outlook on possible future work.

2 Related Work

This thesis is based on a number of works, especially the following:
When talking about algorithm substiution attacks (ASA) Young and Yung have to be mentioned as the first authors wworking on this topic and establishing the idea and the term of *kleptography* [YY96, YY97]. Here Young and Yung show an attack called SETUP, which targets black-box cryptosystems and resists reverse engineering, resulting in the leakage of secret key information.
This idea was further analyzed and developed by Bellare et al.[BPR14, BJK15] to the concept of the algorithm substitution attack (ASA). With the more and more realistic threat of mass surveillance in mind, the authors described this attack as the substitution of a symmetric encryption algorithm with a maliciously subverted one. The goal of this subverted algorithm would be to exfiltrate secret key information enabling a third party to break confidentiality of communictation in which the original party takes a part in.
In 2017 Berndt and Liśkiewicz have generalized the algorithm substitution attack from the symmetric setting to every randomized algorithm with high enough min-entropy and established the term of the *universal* algorithm substitution attack [BL17], which was then later utilized by Berndt et al. [BWP+20] to target the commonly used protocols TLS [Res18], Wireguard [Don17] and Signal [Sig].
The countermeasures considered in this thesis are watchdogs, as presented by Russel et al. [RTYZ16] and cryptographic reverse firewalls as established by Mironov and Davidowitz [MS15].
The idea of the watchdog is to detect ongoing exfiltration of secrets or, more generally, unusual behavior of the algorithms in place. This concept was then further developed by Bemmann, Chen and Jager [BCJ21] to the approach of splitting algorithms in less complex parts and then have their outputs recombined with the use of a trusted amalgamation. For the complete algorithm to work as intended a watchdog in place now only has to verify the correct behaviour of the trusted amalgamation. For less complex tasks like the generation of a random number a watchdog with constant runtime was shown.
The cryptographic reverse firewall on the other hand has the purpose of preventing the exfiltration of secrets for a party P. For this purpose the reverse firewall sits in party $P's$ network and sanitizes outgoing messages before they are sent to the other party. In this process the reverse firewall is seen as a part of the public and untrusted channel and as such P does not need to trust the reverse firewall. The sanitization is done by using

rerandomization and key-malleability. With the help of these tools Mironov and Davidowitz proposed concepts for cryptographic reverse firewalls for a variety of protocols, like Diffie-Hellman key exchange, El-Gamal and more.

3 Background

3.1 Cryptographic Protocols

In this work we adapt the notation as given by Berndt et al. [BWP+20], where we consider two stateful randomized algorithms A, B as our parties, participating in the run of a protocol $\Pi_{A,B}$. Both participants start with their own private input X_P as can be seen in Figure 3.1.
Between the parties $P \in \{A, B\}$ a set of messages $m_1, m_2, ..., m_n$ is exchanged, while at any moment both parties are given a history $h_i = m_1, ..., m_i$ of all i messages exchanged so far and their current state $st_{P,i}$.
The state $st_{P,i}$ and history h_i are both initialized with the empty string. With these two and X_P as inputs, a new message m_i is produced and sent, while the state is updated to a new state $st_{P,i+1}$.
For the sake of simplicity we assume, that both parties A, B swap the active part with every sent message. So for every even i A takes the sending part, while for every uneven i B takes the sending part.

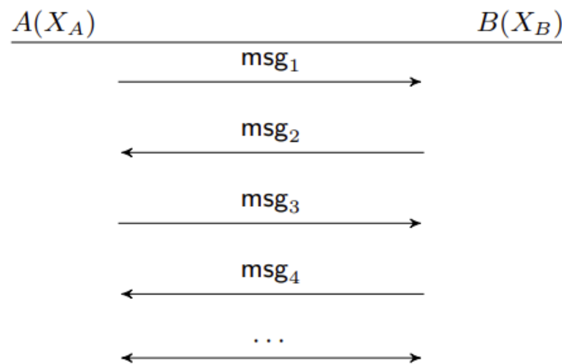


Figure 3.1: Depiction of a run of the protocol $\Pi_{A,B}$ as given by Berndt et al. [BWP+20].

3.2 Algorithm Substitution Attack

In this section we define the algorithm substitution attack (ASA). For this we refer to our definition of the previous section, in which we assume parties A, B to be stateful

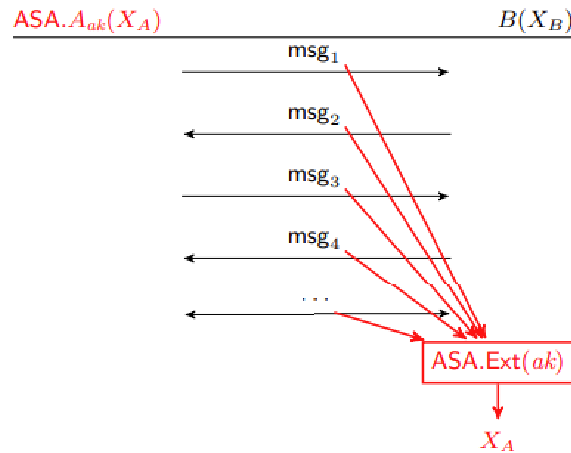


Figure 3.2: Depiction of a run of the protocol with an ongoing ASA as given by Berndt et al. [BWP+20].

randomized algorithms. But what happens, if for example algorithm A ist substituted with a tampered algorithm $ASA.A$?

For this scenario we assume, that the maliciously tampered Algorithms have knowledge over some attacker key ak. With the use of this key, the algorithm is now able to embed parts of the private input X_A into the messages m_i of the protocol.

The attacker can now, with the use of some extraction function $ASA.Ext(ak)$ and the attacker key, extract and reconstruct the private input X_A from the messages m_i of the protocol and as such break confidentiality, as can be seen in Figure 3.2.

Here the attacker has to be active only for the time in which the algorithms of a party are being substituted. After this the attacker stays passive and only has to observe runs of protocols in which A participates.

The embedding and exfiltration of the secret from the messages m_i works in two variants, the *non-universal* algorithm substitution attack and the *universal* algorithm substitution attack.

3.2.1 Non-Universal Algorithm Substitution Attack

The idea used in this attack was first desribed by Bellare et al. [BPR14] under the concept of an IV-replacement attack and was later by Berndt et al. [BWP+20] supplemented to the more general approach of a coin-replacement attack.

The IV-replacement attack targets the IV used in symmetric encryption schemes like AES in counter mode, with encryption and decryption functions $(Enc_k^{CTR}(m), Dec_k^{CTR}(m))$ over some message m and key k, where a randomly generated initialization vector is used.

As an example we take some message m

$$m = \boxed{m_1 \mid m_2 \mid ... \mid m_n}$$

evenly split in $m_1, m_2, ..., m_n$ with each part consisting of l bits equal to the block length of the symmetric cipher with encryption function Enc_k over some key k. The encryption $Enc_k^{CTR}(m)$ of the message m results in a ciphertext

$$c = \boxed{\text{IV} \mid c_1 \mid c_2 \mid ... \mid c_n}$$

where the IV is a randomly generated initialization vector.
In the next step we assume, that Enc was maliciously tampered by an attacker in a way, that the IV is no longer generated randomly, but is instead derived from the secret X_A. For this purpose we no assume, that the extraction function $ASA.Ext$ is chosen as AES in the electronic codebook mode, with encryption and decryption functions $(Enc_k^{ECB}(m), Dec_k^{ECB}(m))$ over some message m and some key k but without the use of an initialization vector.
The malicious set of Algorithms $ASA.A$ now encrypts the secret X_A with use of the attacker key ak and uses the resulting ciphertext as the IV for the encryption of the message m.

$$Enc_{ak}^{ECB}(X_A) = s \tag{3.1}$$

$$Enc_k^{CTR}(m) = \boxed{s \mid c_1 \mid c_2 \mid ... \mid c_n} = c \tag{3.2}$$

Since a randomly generated number and a ciphertext are both indistinguishable from honest randomness, they also cannot be distinguished from each other, which is why the attack here cannot be detected just by looking at the IV without knowledge of the attacker key ak.
Additionally to only IVs this attack is also applicable to nonces in messages of various protocols and publically transfered random coins in general.

3.2.2 Universal Algorithm Substitution Attack

The universal ASA targets all protocols that use the output of a randomized algorithm in their messages. In order to embed secret information bits in this output the attacker utilizes what Bellare et al. referred to as *rejection sampling* [BJK15]. This idea was then later shown to be applicable to all randomized algorithms by Berndt and Liśkiewicz [BL17].
The idea behind rejection sampling is the following, as given by Berndt et al.[BWP+20]:
As soon as a randomized algorithm R uses freshly sampled randomness to generate a

pseudo random output, a subverted instance of this algorithm can re-sample this randomness until the generated output contains information about the secret to be exfiltrated. For this exfiltration to be undetectable we again utilize our extraction function $ASA.Ext_{ak}$ and define it as a pseudorandom function $F_{ak}(\delta)$, where δ is the output of the randomized algorithm $R(x;r)$ over some input x and randomness r.
In our setting the secret to be exfiltrated s can be split into L-many blocks of length λ, so that

$$s = \boxed{s_1} \boxed{s_2} \boxed{...} \boxed{s_L} \quad ,$$

where $L = \frac{|s|}{\lambda}$.
Our extraction function $F_{ak}(\delta)$ now returns a tuple (b_i, i) consisting of a bit string b_i of length λ and an index i of length $\log(L)$.
The subverted instance of $R(x;r)$ now samples randomness r until it finds some value r^*, so that $R(x;r^*) = \delta^*$ and $F_{ak}(\delta^*) = (s_i, i)$, meaning the extraction function used on the output of the randomized algorithm utilized in the protocol, returns a tuple of a block of secret information bits and their corresponding index in the secret to be leaked s. This procedure can now be repeated until the complete secret s is exfiltrated to the Attacker.

3.3 Diffie–Hellman Key Exchange

An example for a two party protocol with the goal of exchanging a common secret k between both parties A, B in a protocol $\Pi_{A,B}$ is the Diffie–Hellman key exchange as presented by Whitfield Diffie and Martin E. Hellman [DH76] and adapted to our notation.
Here we consider the private inputs X_A, X_B the private keys of both parties. Before the run of the procotol starts we assume, that both parties A, B have already publically aggreed on a finite cyclic group $\mathbb{F}_p^*$ with a prime module p and a generator $g \in \mathbb{F}_p^*$.
As the initializing party, A uses the private input $X_A \in \mathbb{F}_p^*$ and calculates her public parameter $Pub_A = g^{X_A} \bmod p$. A then sends Pub_A to B over the insecure channel who responds with $Pub_B = g^{X_B} \bmod p$.
Both parties can now compute the shared secret k as follows:

$$Pub_B^{X_A} \bmod p = k = Pub_A^{X_B} \bmod p$$

Definition 3.1 (Decisional Diffie Hellman Problem). Given a finite cyclic group $\mathbb{F}_p^*$ with a prime module p and a generator $g \in \mathbb{F}_p^*$, a challenger $\mathcal{C}$ is given values $A = g^a$ and

$B = g^b$, with $a, b \in \mathbb{Z}_p$ and has to decide, whether for a third value $C = g^c$ the equation

$$c = a \cdot b$$

holds.

We assume the Diffie-Hellman key exchange is secure for a finite cyclic group $\mathbb{F}_p^*$, when a probabilistic polynomial time-attacker $\mathcal{A}$ has a success rate for the decisional Diffie Hellman problem bounded by $\frac{1}{2} + \epsilon$, where ϵ is negligibly small.
Alternatively, if we want to utilize a discrete elliptic curve E mod prime q with some generator point $h \in E$, the key exchange protocol changes marginally:
Here we now compute the public parameters Pub_A, Pub_B as $Pub_A = X_A \cdot h, Pub_B = X_B \cdot h$ and the shared secret k correspondingly:

$$X_A \cdot Pub_B = k = X_B \cdot Pub_A$$

3.4 El-Gamal Public Key Encryption Scheme

With the El-Gamal public key encryption scheme we have a two party protocol $\Pi_{A,B}$ with parties $P \in \{A, B\}$ and algorithms `(Keygen, Encrypt, Decrypt)`, where `Keygen` takes as input a security parameter 1^λ returning a key pair (k_{sec}, k_{pub}), where k_{sec} is a uniformly random chosen element $a \in \mathbb{Z}_p^*$ of a multiplicative group with prime module p and k_{pub} is a tuple $(g, h) = (g, g^a)$ with $g \in \mathbb{Z}_p^*$ being a generator of the group.
For the encryption of a message $m \in \mathbb{Z}_p^*$ with public key $k_{pub} = (g, h)$ we use the function `Encrypt` over the public key with the message m as input and returning a ciphertext

$$c = (c_0, c_1) = (g^r, h^r \cdot m),$$

where $r \in \mathbb{Z}_p^*$ is chosen uniformly random.
For the decryption we use the function `Decrypt` over the private key $k_{sec} = a$ with the ciphertext $c = (c_0, c_1)$ as our input and returning the message m as

$$m = c_0^{-a} \cdot c_1$$

Formal definitions for this protocol can be found in the work of Katz and Lindell [KL14].

3.5 Commitment Scheme

We give the definition of a commitment scheme as presented by Dodis et al. [DMS16].

A commitment scheme is a two party protocol $\Pi_{A,B}$ with parties $P \in \{A, B\}$ and a set of algorithms `(Keygen, Commit, Open, Verify)`, where `Keygen` takes as input 1^λ with security parameter λ returning a public parameter ρ. The algorithm `Commit` takes as input ρ, a plaintext m and freshly generated randomness r, returning a commitment `comm` C. The algorithm `Open` takes as inputs ρ, a commitment C and randomness r returning an opening `opn` x. `Verify` takes as inputs a commitment C and opening x and outputs either a plaintext m or an error symbol $\perp$. The commitment scheme is correct, if $\texttt{Verify}(C, x) = m$ holds, whenever the conditions $C = \text{Commit}(m)$ and $x = \texttt{Open}(C)$ are met.
Additionally we require to scheme to be perfectly hiding, meaning for any two plaintexts m_1, m_2 the distribution of $\texttt{Commit}(m_1)$ is identical to the distribution of $\texttt{Commit}(m_2)$, and computationally binding, meaning no probabilistic polynomial-time attacker $\mathcal{A}$ can generate a commitment C and to openings x_1, x_2, such that $\texttt{Verify}(C, x_1)$ and $\texttt{Verify}(C, x_2)$ both return different plaintexts.

3.6 Rerandomizable Public Key Encryption

We present the defintion of a rerandomizable public key encryption scheme (PKES) as given by Dodis et al. [DMS16].
For this we define an algorithm `Rerand` with knowledge of the public key. The PKES is rerandomizable, if for `Rerand` the decryption function `Decrypt` and a ciphertext c holds

$$\texttt{Rerand(Decrypt}(c) = \texttt{Decrypt}(c)$$

while the pairs $(c, \texttt{Rerand}(c))$ and $(c, \texttt{Rerand(Encrypt}(0)))$ with the encryption function `Encrypt` are indistinguishable.
The El-Gamal PKES is an example for a rerandomizable PKES.

3.7 Key-Malleability

We present the definition of a key-malleable PKES as given by Dodis et al. [DMS16]
A PKES is key-malleable, if the following conditions apply:

- The algorithm `Keygen`'s output is uniformly distributed over the public key space $\mathcal{K}$.
- Each public key pub is associated with a unique private key sec
- We have a randomized algorithm `KeyMaul` with a public key pub as input, returning a new public key pub' uniformly randomly distributed over $\mathcal{K}$ and an algorithm

`CipherMaul` meeting the following requirements:
If we have a key pair (sec, pub), randomness r, and a key pair (sec', pub') with $pub' = \texttt{KeyMaul}(pub; r)$, then the algorithm `CipherMaul`, with a ciphertext $cand$ randomness r as inputs, returns a ciphertext c' such that $\texttt{Decrypt}_{sec'}(c) = \texttt{Decrypt}_{sec}(c')$.

The El-Gama PKES is an example for a key-malleable public key encryption scheme.

3.8 Cryptographic Reverse Firewalls

In this work we use the notation as introduced by Mironov et al. [MS15], adapted to the established defintion of cryptographic protocols.
In compliance with our definition of a cryptographic protocol we interprete a cryptographic reverse firewall (RF) as a stateful algorithm $\mathcal{W}$, that takes it's state and a message as inputs, returning an updated state and a new message as outputs.
Additionally we denote a composed party $R \circ \mathcal{W}$ as the conjunction of a party P with a RF $\mathcal{W}$.
A reverse firewall has to satisfy the following conditions:

Functionality maintaining For the composition of any party P with any RF $\mathcal{W}$ we denote the stacking of reverse firewalls as:

$$\mathcal{W}^1 \circ P = \mathcal{W} \circ P$$
$$\mathcal{W}^k \circ P = \mathcal{W} \circ (\mathcal{W}^{k-1} \circ P), \text{ for } k \geq 2$$

For a protocol $\Pi_{A,B}$ with participating parties $P \in \{A, B\}$ we can formalize functionality requirements $\mathcal{F}$, where we can define which properties must apply after a run of the protocol. So for example in the case of a Diffie-Hellman key exchange the corresponding functionality requirement would be, that both parties A, B have aggreed on the same shared key k.
Generally speaking we want reverse firewalls to be stackable, so we say that the composition $\mathcal{W} \circ P$ of a reverse firewall $\mathcal{W}$ and a party P maintains $\mathcal{F}$ for any polynomial bounded $k \geq 1$ if the stacked composition $\mathcal{W}^k \circ P$ maintains $\mathcal{F}$.

Security-preservation Crucial for the applicability of a cryptographic reverse firewall is the security-preservation. Here we demand, that if the underlying protocol is secure,

and regardless of the machines behaviour, the reverse firewall must not compromise the protocol's security.

Exfiltration-resistance With this property we define the reverse firewall's actual task: Sanitizing outgoing messages from potential embedded secrets and thus preventing the exfiltration of secret information.
To formalize this, we consider a probabilistic polynomial-time attacker $\mathcal{A}$ and two protocols $\Pi_{W \circ A,B}$ with history h_1 and $\Pi_{W \circ \tilde{A},B}$ with history h_2 for parties $P \in \{A, B\}$ and RF W, where $\tilde{A}$ are maliciously subverted algorithms of party A embedding information in the messages of the protocol using a key $ak_{\mathcal{A}}$ known to the attacker. After the run of both protocols the attacker $\mathcal{A}$ is given a history h_i and has to decide, by which protocol it was generated.
We say the RF W is resisting exfiltration if the attacker's probability of success is bound by $\frac{1}{2} + \epsilon$, where ϵ is negligibly small.

4 Implementations

This section will cover the implementation the given protocols in their various forms, as proposed by Dodis et al.[DMS16], with regard to current security standards and parameters and with the goal of setting up a basis usable for benchmarking the reverse firewall's performance on a variety of CPUs.
For both protocols covered in this chapter, algorithms over multiplicative groups modulo a prime p and over elliptic curves are implemented. Here for the prime p we generated a cryptographically secure prime of 3072 bit, as security standards by the NIST [Bar16] suggest. Cryptographically secure in this context means, that for our prime number p holds:

$$p = 2 \cdot q + 1, \text{ where } q \text{ is prime.}$$

For this we wrote a program in `C++`, that generates a random number n consisting of 3072 bit, with the most significant bit set to 1 and the least significant to 0. For every odd prime candidate $m > n$, we use the Rabin-Miller primality test as originally introduced by Miller [Mil76] and implemented in the boost library [Boo22].
Once a candidate m is tested to be prime, the number $\frac{m-1}{2}$ is also checked for primality. Once this requirement is met, our prime module p of 3072 bits is found.
As this bit size corresponds to a security strength of 128 bit, the elliptic curve is chosen accordingly, in this case we use `CURVE25519` [LHT16], as this curve is intedend for key exchanges and asymmetric cryptography.
With the goal of measureing the resulting overhead when adding a cryptographic reverse firewall to an already established protocoll, without the need for the code to be as performant as possible, we chose Sage [DSJ+20] as our programming language, since here most of the necessary structures and operations are already intrinsically present.

4.1 Diffie-Hellman Key Exchange

For a run of a protocol $\Pi_{A,B}$ of a Diffie-Hellman key exchange for two parties $P \in \{A, B\}$, in the following only referred to as `DH`, we instantiate both our Clients *Alice* and *Bob* with the already established prime module p for the multiplicative group $\mathbb{Z}_p^*$ as well as a generator $g \in \mathbb{Z}_p^*$, as we can assume that these parameters are publically aggreed on. Each client has access to the functions `keygen`, where the parameters necessary for the exchange are generated and a function `exchange`, where the other partie's public and the own private

parameter are taken as inputs an the shared secret k is returned.
As shown in Listing 4.1both parties now generate their private and public parameters a, g^a and b, g^b respectively, while in the next step the pairs (a, g^b) and (b, g^a) are used to calculate the shared key candidates ka and kb. In case these candidates do not match, an error is returned.
This procedure is repeated 1000 times and the time needed for each run is measured in order to get an average run time needed for the operations.

Listing 4.1: DH

```
1  elapsed_time = 0
2  for i in range(1000):
3          start = time.process_time()
4
5          a, ga = alice.keygen()
6          b, gb = bob.keygen()
7
8          ka = alice.exchange(a, gb)
9          kb = bob.exchange(b, ga)
10
11         elapsed_time += time.process_time() - start
12         if not ka == kb:
13                 print("Error")
```

In this run of the protocol the private parameters generated by Alice and Bob are of size 256 bit, as suggested by the NIST [Bar16].

Adding the reverse firewall In the next step the reverse firewall W is added to the protocol. As it is part of Alice's network and not visible from the internet, Listing 4.1 is still representative for the publically visible operations, the functioniality of the algorithms `keygen` and `exchange` is adjusted to the composition of Alice and the reverse firewall to $A \circ W$. The basic functionalities of `keygen` and `exchange` are inherited from the previous implementations and referenced via the call `super()`. In Listing 4.2 we can now see, that the key generation references the algorithm `keygen` from before, but now before the generated parameters are returned, the reverse firewall rerandomizes the public parameter. For this the firewall samples a secret value $r \in \mathbb{Z}_p^*$, where

$$\texttt{Rerand}(g^a) = (g^a)^r = g^{ar}$$

With this rerandomization any embedded secrets in Alice's public parameter g^a are no longer extractable from the new public parameter g^{ar}.
The function `exchange` is also slightly changed, as now, before calculating $(g^b)^a = g^{ab}$

Bob's public parameter is rerandomized with the same randomness r as before. So now we get $(g^b)^r = g^{br}$ as Bob's new public parameter as seen by Alice.
Consequently Alice now exchanges Bos's rerandomized public parameter with her private parameter, leading to

$$ka = (g^{br})^a = g^{abr}$$

while Bob exchanges Alice's rerandomized public parameter with his own private parameter, resulting in

$$kb = (g^{ar})^b = g^{abr}$$

and thus both parties agree on their shared secret

$$ka = g^{abr} = kb\ .$$

Listing 4.2: DH_FW: Algorithms for $A \circ W$

```
1 def keygen(self):
2         sec, pub = super().keygen()
3         return sec, self.reverseFirewall.rerandomize(pub)
4
5 def exchange(self, sec, pub):
6         pub = self.reverseFirewall.rerandomize(pub)
7         return super().exchange(sec, pub)
```

Extend the key exchange with a commitment scheme In our described scenario Alice initiates the run of `DH`. Here we have to take into account, that with knowledge over Alice's public parameter, Bob still has control over the final key k, as a subverted algorithm on his machine can compromise the security of the key by rejection sampling of his private parameter until for example the shared secret's 20 least significant bits follow a predefined pattern.
To prevent this from happening we supplement our protocol `DH` to `DH_Comm` by adding a variation of the El-Gamal commitment as shown by Dodis et al. [DMS16], that deviates from the definition in the implementation, as here the algorithm `Verify` only checks if the message `opn` was used for the commitment `comm` and `Open` opens the commit.
Bob initiates the key exchange by generating a private parameter and sends a commitment of the coresponding public parameter to Alice. After receiving Alice's public parameter, a subverted alogorithm on Bob's machine can no longer use this information for compromising the security of the shared secret k, as his public parameter is already set. The same is true for Alice, since she learns the value Bob's public parameter only after she has decided on her own public parameter and sent it to him.

In this setting Alice has access to the algorithms `keygen, exchange, verify` and `open` and Bob has access to the algorithms `keygen, exchange` and `commit`. In the run of this protocol, and with a second generator $h \in \mathbb{Z}_p^*$, Bob now choses a private parameter b and two random coins r, s, with $b, r, s \in \mathbb{Z}_p^*$ uniformly randomly and sends a commit $\texttt{comm} = (g^r \cdot h^s, h^s \cdot g^b)$ to Alice. In the next step Alice choses a secret parameter $a \in \mathbb{Z}_p^*$ uniformly randomly and sends her public parameter g^a to Bob. Bob now has all values necessary for the key exchange and sends the random coins used to open commitment $open = (r, s)$ back to Alice. With `verify` she can now check, whether the sent random coin were actually used for the commitment, by verifing that the first part of the commitment equals $g^r \cdot h^s$ and if so, open the commitment with use of the algorithm `open` by calculating

$$h^s \cdot g^b \cdot h^{-s} = h^{s-s} \cdot g^b = g^b.$$

Alice can now exchange Bob's public and her own private parameter to the shared secret k.

This procedure, as can be seen in Listing 4.3, is again repeated a 1000 times and the average time necessary for the operations `keygen, commit, verify, open` and `exchange` is measured.

Listing 4.3: DH_Comm

```
1  elapsed_time= 0
2  for i in range(1000):
3          start = time.process_time()
4
5          b, comm, open = bob.keygen()
6          a, ga = alice.keygen()
7
8          if not alice.verify(comm, open):
9                  print("No␣valid␣open")
10                 exit()
11
12         gb = alice.open(comm, open)
13         ka = alice.exchange(a, gb)
14         kb = bob.exchange(b, ga)
15
16         elapsed_time += time.process_time() - start
17
18         if not ka == kb:
19                 print("Error")
```

Adding the reverse firewall Since the commitment scheme, used in `DH_Comm`, is a randomized algorithm it is again targetable by an universal algorithm substitution attack. In the next protocol `DH_Comm_FW`, we add a reverse firewall W, this time as a part of Bob's network and with the purpose of rerandomizing the messages `comm` and `opn`. For this purpose the firewall W choses two random coins $u, v \in \mathbb{Z}_p^*$ uniformly random and rerandomizes these messages as follows:

$$\begin{aligned} \texttt{Rerand}_{u,v}(\texttt{comm}) &= \texttt{Rerand}_{u,v}(g^r h^s, h^s g^b) && (4.1)\\ &= (g^u h^v \cdot g^r h^s, h^s g^b \cdot h^v) && (4.2)\\ &= (g^{r+u} h^{s+v}, h^{s+v} g^b) && (4.3)\\ \texttt{Rerand}_{u,v}(\texttt{opn}) &= \texttt{Rerand}_{u,v}(r, s) && (4.4)\\ &= (r+u, s+v) && (4.5) \end{aligned}$$

Again as before, Alice can now use the sent randomness $(r+u, s+v)$ to verify the first part of `comm` by comparing the value with $g^{r+u} \cdot h^{s+v}$ and then open the commitment with use of `opn` by calculating

$$h^{s+v} \cdot g^b \cdot h^{-(s+v)} = h^{s+v-s-v} \cdot g^b = g^b \; .$$

Here it is important to note that in this protocol Alice's public parameter is again vulnerable to an ASA. To address this problem, the reverse firewall would additionally need to generate another random value α, rerandomize Alice's parameter g^a to $g^{a\alpha}$, and maul the messages `Comm` and `open` from (4.3) and (4.5) as follows:

$$\begin{aligned} \texttt{MaulCommit}_\alpha(g^{r+u} h^{s+v}, h^{s+v} g^b) &= ((g^{r+u} h^{s+v})^\alpha, (h^{s+v} g^b)^\alpha) && (4.6)\\ &= (g^{r\alpha+u\alpha} h^{s\alpha+v\alpha}, h^{s\alpha+v\alpha} g^{b\alpha}) && (4.7)\\ \texttt{MaulOpen}_\alpha(r+u, s+v) &= ((r+u)\alpha, (s+v)\alpha) && (4.8)\\ &= (r\alpha+u\alpha, s\alpha+v\alpha) && (4.9) \end{aligned}$$

With these two messages Alice now retrieves Bob's rerandomized public parameter $g^{b\alpha}$, with Bob having Alice's rerandomized public parameter $g^{a\alpha}$ yielding the shared secret $k = g^{ab\alpha}$.

Calculations over elliptic curve `CURVE25519` Since cryptography over elliptic curves is becoming more and more popular, we implement the algorithms from before over a discrete elliptic curve E mod $2^{255} - 19$ standardized under the name `CURVE25519`[LHT16].

E is described by the equation

$$y^2 = x^3 + 486662x^2 + x \pmod{2^{255} - 19}$$

Here with the generation of E we also generate two publically known generator points $g, h \in E$ and use them for every protocol.

Diffie-Hellman key exchange We refer to this protocol over the elliptic curve E as `EC_DH`. The overall procedure in this implementation is the same as in Listing 4.1. What changes are the algorithms `keygen` and `exchange` as can be seen in Listing 4.4

Listing 4.4: EC_DH: Algorithms of both clients

```
def keygen(self):
        sec = randint(1,self.E.order())
        pub = self.g*sec
        return sec, pub

def exchange(self, sec, pub):
        return pub*sec
```

In the algorithm `keygen` a client choses a secret value sec uniformly random out of an interval $[1, order(E)]$, where $order(E)$ represents the amount of group elements as points on the curve.
The corresponding public parameter pub is then calculated as $g \cdot sec$.
The secret is then aggreed on in the algorithm `exchange` where again a public parameter pub and a secret parameter sec are evaluated to $k = pub \cdot sec$, where k is the shared secret between both clients Alice and Bob.

Adding the reverse firewall Just as before, to obtain a protocol `EC_DH_FW`, we add a reverse firewall, that rerandomizes Alice's public parameter before exiting her network and Bob's public parameter before it enters Alice's network. The implementation of this firewall is similar to what is shown in Listing 4.2.

Adding the commitment scheme We again add the commitment scheme to the protocol `EC_DH` so that we get an instance of `EC_DH_Comm`. The code for this protocol is similar to what we can see in Listing 4.3. To adapt the operations in the algorithms `keygen`, `exchange`, `commit`, `open` and `verify` to our elliptic curve E, every exponentiation ˆ is substituted with a multiplication $*$ and every multiplication from before is substituted with an addition $+$.

The same method is applied in order to add the reverse firewall rerandomizing the messages `commit` and `open`, resulting in our protocol `EC_DH_Comm_FW`.
Now for the rerandomization we again have our generator points $g, h \in E$ and two random coins $u, v \in [1, order(E)]$ leading us to:

$$\begin{aligned} \texttt{Rerand(Comm)} &= \texttt{Rerand}(g \cdot r + h \cdot s, h \cdot s + g \cdot b) && (4.10)\\ &= (g \cdot u + h \cdot v + g \cdot r + h \cdot s, h \cdot s + g \cdot b + h \cdot v) && (4.11)\\ &= (g \cdot (r + u) + h \cdot (s + v), h \cdot (s + v) g \cdot b) && (4.12)\\ \texttt{Rerand(Open)} &= \texttt{Rerand}(r, s) && (4.13)\\ &= (r + u, s + v) && (4.14) \end{aligned}$$

4.2 El-Gamal

The El-Gamal public key cryptosystem in a protocol $\Pi_{A,B}$ with parties $P \in \{A, B\}$ is implemented over the multiplicative group $\mathbb{Z}_p^*$ modulo prime p and a generator $g \in \mathbb{Z}_p^*$.
Here Alice has access to the algorithms `keygen`, and `decrypt`, while Bob has a algorithm `encrypt`.
In a run of this protocol Alice generates a key pair (sec, pub) with `keygen`, where sec is a uniformly random chosen value $a \in \mathbb{Z}_p^*$ and pub is a tuple (g, h) with g being a generator of the group and $h = g^a$.
In the next step a random message $m \in \mathbb{Z}_p^*$ is chosen and encrypted by Bob using `encrypt` with a uniformly random chosen value $b \in \mathbb{Z}_p^*$.
`Encrypt` takes as input a message m and returns the ciphertext using the public key pub and b as follows:

$$\texttt{Encrypt}_{b,pub}(m) = (g^b, h^b \cdot m) = (c_0, c_1) = c$$

Here we get a ciphertext $c = (c_0, c_1)$ that Alice then decrypts to a message candidate m' using the algorithm `decrypt`. If m and m' are not equal, the program returns an error.
This procedure is again repeated 1000 times and the time needed for the execution of the given algorithms is measured, as can be seen in Listing 4.5.

Listing 4.5: EG

```
1  elapsed_time = 0
2  for i in range(1000):
3          start = time.process_time()
4
5          g, h = alice.keygen()
6
7          m = randint(1, p)
```

```
8
9         c0, c1 = bob.encrypt(m, g, h)
10        m_dash = alice.decrypt(c0, c1)
11
12        if not m == m_dash:
13                print("Fail")
14
15        elapsed_time += time.process_time() - start
```

Adding the firewall For this protocol both rerandomization and key-malleability can be used to sanitize the ciphertext sent from Bob to Alice. Consequently the reverse firewall is part of Bob's network and they form the composed party $W \circ B$.
In each scenario the overall procedure, as depicted in Listing 4.5, stays the same.

Rerandomization We start with using rerandomization as the reverse firewall's tool and get a protocol `EG_Rerand`:
As shown in Listing 4.6, the reverse firewall only takes the ciphertext resulting from use of the inherited algorithm `encrypt` as input and rerandomizes the message by chosing a uniformly random value $r \in \mathbb{Z}_p^*$ and returning

$$c' = (c'_0, c'_1) = (g^r \cdot c_0, (g^a)^r \cdot c_1)$$

as a new ciphertext.
With Bob chosing a random secret value $b \in \mathbb{Z}_p^*$ and calculating c as

$$c = (c_0, c_1) = \texttt{encrypt}(m, g, h) = (g^b, g^{ab} \cdot m)$$

leading us to the rerandomization of c as follows:

$$\begin{aligned} c' = (c'_0, c'_1) = \texttt{Rerand}(c_0, c_1) &= (g^r \cdot c_0, g^{ar} \cdot c_1) && (4.15) \\ &= (g^r \cdot g^b, g^{ar} \cdot g^{ab} \cdot m) && (4.16) \\ &= (g^{r+b}, g^{ar+ab} \cdot m) && (4.17) \end{aligned}$$

Alice can now decrypt this new ciphertext using her private key a and algorithm `decrypt`:

$$m = \texttt{decrypt}(c') = \texttt{decrypt}(c'_0, c'_1) = c_0'^{-a} \cdot c_1 \tag{4.18}$$
$$= (g^{r+b})^{-a} \cdot g^{ar+ab} \cdot m \tag{4.19}$$
$$= g^{-ar-ab} \cdot g^{ar+ab} \cdot m \tag{4.20}$$
$$= g^{ar+ab-ar-ab} \cdot m \tag{4.21}$$
$$= g^0 \cdot m \tag{4.22}$$
$$= m \tag{4.23}$$

Listing 4.6: EG_Rerand: Algorithms for party $W \circ B$

```
1  def encrypt(self, m, g, h):
2          r = randint(1, self.p)
3          c_0, c_1 = super().encrypt(m, g, h)
4          return rerandomize(c_0,c_1)
5
6  def rerandomize(self, c0, c1, g, h):
7          return (g^self.r)*c0, (h^self.r)*c1
```

Key-malleability Using two functions `keyMaul` and `cipherMaul` instead of rerandomization leads us to the protocol `EG_Maul`. As with `EG_Rerand` the overall procedure shown in Listing 4.4 still applies.
Again only Bob's algorithm `encrypt` is supplemented with the reverse firewall's algorithms as visualized in Listing 4.7.

Listing 4.7: EG_Maul: Algorithms for party $W \circ B$

```
1  def encrypt(self, m, g, h):
2          g',␣h' = keyMaul(g, h)
3          c0, c1 = super().encrypt(m, g',␣h')
4          return cipherMaul(c0, c1)
5
6  def keyMaul(self, g, h):
7          return g^self.alpha, h^self.beta
8
9  def CipherMaul(self, c0, c1):
10         return c0^(self.beta / self.alpha), c1
```

Here we see, that the reverse firewall applies it's function `keyMaul` to Alice's public key, before Bob uses it to encrypt the message.

For this the reverse firewall samples two random coins $\alpha, \beta \in \mathbb{Z}_p^*$ and calculates

$$(g', h') = \texttt{keyMaul}(g, h) = (g^\alpha, h^\beta) = (g^\alpha, g^{a\beta})$$

After Bob now encrypts the message m with use of the key (g', h') to get a ciphertext

$$c = (c_0, c_1) = \texttt{encrypt}(m, g', h') = (g'^b, h'^b \cdot m) \tag{4.24}$$
$$= (g^{\alpha b}, g^{ab\beta} \cdot m) \tag{4.25}$$

the reverse firewall applies it's function `cipherMaul` to the ciphtertext c. This now gives us a new ciphertext c' as follows:

$$\begin{aligned} c' = (c_0', c_1') = \texttt{cipherMaul}(c_0, c_1) &= (c_0^{\beta/\alpha}, c_1) \\ &= (g^{\alpha b\beta/\alpha}, g^{ab\beta} \cdot m) \\ &= (g^{b\beta}, g^{ab\beta} \cdot m) \end{aligned}$$

Now, as before, Alice can decrypt the ciphertext c with her algorithm `decrypt` and her private value a:

$$m = \texttt{decrypt}(c') = \texttt{decrypt}(c_0', c_1', a) = c_0'^{-a} \cdot c_1 \tag{4.26}$$
$$= g^{-ab\beta} \cdot g^{ab\beta} \cdot m \tag{4.27}$$
$$= g^{ab\beta - ab\beta} \cdot m \tag{4.28}$$
$$= g^0 \cdot m \tag{4.29}$$
$$= m \tag{4.30}$$

Rerandomization and key-malleability In the last variant of El-Gamal utilizing both, rerandomization and key malleablitiy, leads us to the procotol `EG_ReMaul` in which the composed party $W \circ B$ has access to the algorithms shown in Listing 4.8

Listing 4.8: EG_ReMaul: Algorithms for party $W \circ B$

```
1  def encrypt(self, m, g, h):
2          g', h' = keyMaul(g, h)
3          c0, c1 = super().encrypt(m, g, h)
4          c0', c1' = rerandomize(c0, c1, g, h)
5          return  cipherMaul(c0, c1)
6
7  def keyMaul(self, g, h):
8          return g^self.alpha, h^self.beta
9
10 def CipherMaul(self, c0, c1):
```

```
11         return c0^(self.beta / self.alpha), c1
12
13 def rerandomize(self, c0, c1, g, h):
14         return (g^self.r)*c0, (h^self.r)*c1
```

Here we can see, that in the first step of the encryption, Alice's public key (g, h) is mauled to a new key (g', h') with use of the random coins $\alpha, \beta \in \mathbb{Z}_p^*$, followed by the encryption of message m to ciphertext $c = (c_0, c_1)$.
This ciphertext is then rerandomized with use of randomness $r \in \mathbb{Z}_p^*$ to a new ciphertext c'. Finally, in the last step, this ciphertext c' is then again mauled with use of the algorithm `cipherMaul`.
As the overall procedure shown in Listing 4.5 still apllies here, Alice can now decrypt the mauled and rerandomized ciphertext and retrieves the message m, without further calculations necessary.

Calculations over elliptic curve CURVE25519 As with the Diffie-Hellman key exchange, the protocols `EG` and `EG_Rerand` are translated to our discrete elliptic curve E modulo $2^{255} - 19$ standardized under the name `CURVE25519`, resulting in the protocols `EC_EG` and `EC_EG_Rerand`.
For this to work, the same adaptions as with the key exchange protocol are necessary and sufficient, meaning every exponentiation ˆ is replaced with a multiplication $*$ and each multiplication $*$ is substituted with an addition $+$.

5 Benchmarks

Go over findings / results

In the following section, we look at the results of the benchmarks performed with the code elaborated in the previous chapter. These results were calculated from the average run times of 1000 repitions of the core algorithms in each protocol.
To match the security strength of 128 bit[Bar16], we use a prime integer p of 3072 bits for our calculations over a multiplicative group $\mathbb{Z}_p^*$ modulo p, although the private key sizes vary in the different protocols, to get an idea of a *worst case scenario* regarding the resulting overhead from utilizing a reverse firewall.

The testing systems In Table 5.1 we see the different systems with their CPUs used for the benchmarking. We chose these systems with the intention of quantifying the resulting run times over slower devices like smart home and internet of things devices (IOT) or smartphones, up the scale of CPUs used for servers. The implementations using Sage were tested on all systems except IOT and Smartphone. To get an idea of the resulting overhead on these devices, we implement the procotols `DH` and `DH_FW` in C++, With these results we then evaluate the usability of reverse firewalls on these systems.

Table 5.1: List of testing systems with corresponding CPU

System	CPU
IOT	ARM ARM1176@1 GHz
Smartphone	ARM Cortex-A53@1.4 GHz
Laptop	Intel i3 10110U@2.10 GHz
PC	AMD Ryzen 5 2600@3.4 GHz
Server 1	Intel Xeon Silver 4114 CPU@2.2 GHz
Server 2	Intel Xeon Gold 6342 CPU@ 2.8 GHz

Diffie-Hellman key exchange We start with evaluating the run times for the various implementations of the Diffie-Hellman key exchange. In Figure 5.1 we see the visual representationn of our results for our protocols `DH`, `DH_FW`, `DH_Comm` and `DH_Comm_FW` as given in Table 5.2 implemented in Sage and tested on our systems Laptop, PC, Server 1 and Server 2.

Table 5.2: Benchmark results for various Diffie-Hellman key exchange protocols on all systems except IOT and Smartphone

System	DH	DH_FW	DH_Comm	DH_Comm_FW
PC	2.8ms	31.8ms	76.2ms	122.7ms
Laptop	3.9ms	43ms	83.8ms	134.4ms
Server 1	5.1ms	59.2ms	141.6ms	227.8ms
Server 2	3.9ms	45.6ms	109.9ms	174.6ms

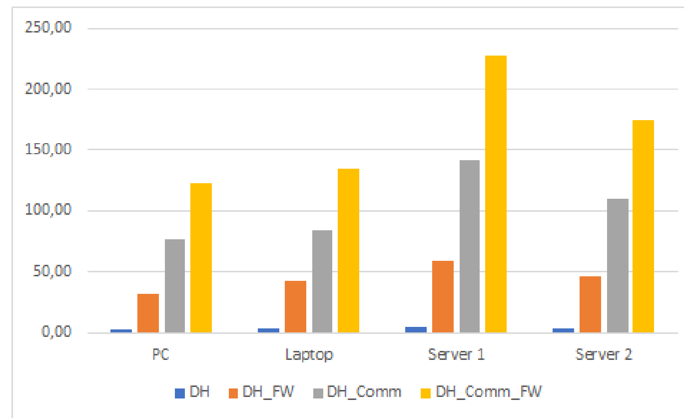


Figure 5.1: Visualization of benchmark results given in Table 5.2

Here we see the benchmarks for our implementation of DH in Sage ranging from 2.8ms on PC to 5.1ms on Server 1, averaging at 3.93ms across all platforms. The bar right next to it, in orange, shows us the execution time needed for DH_FW, which ranges from 31.8ms on the PC to 59.2ms on Server 1, with an average runtime of 44.9ms across all systems. With these results we can see, that DH_FW takes about 11.4 times longer than DH on PC, 11.6 times longer on Server 1 and about 11.4 times longer on average across all systems. For these two protcols we use private private keys of 256 bit for the Diffie-Hellman parameters and a private key of 3072 bit for the reverse firewall.

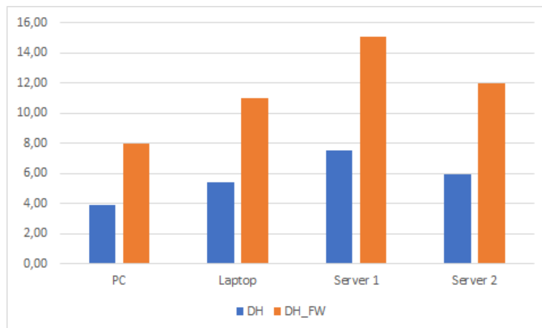
With the bars colored in gray and yellow we can now compare the resulting overhead of adding a reverse firewall to our protocol DH_Comm, where we use consistent private key sizes of 3072 bit.
For DH_Comm we see benchmarks of 76.2ms on a PC up to 141.6ms for Server 1, followed by benchmarks for DH_Comm_FW of 122.7ms on PC ranging to 227.8ms on Server 1. This gives a resulting overhead of about 61% on PC, 60.9% on Server 1 and averaging at about 60.3%.

The discrepancy in the overheads from adding a firewall to DH and DH_Comm is reason to implement DH and DH_FW in C++, this time with all private key sizes being set to 256 bit.

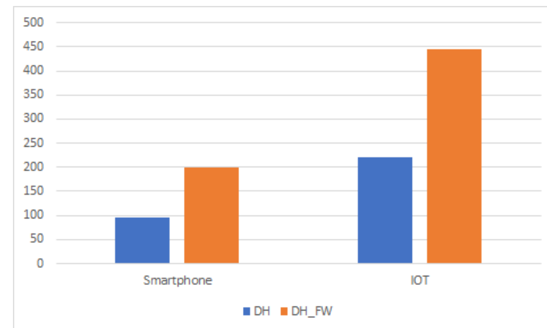
This leads us to the benchmarks visualized in Figure 5.2a and Figure 5.2b, with absolute values shown in Table 5.3
Here, again in blue, we see the runtime of `DH` ranging from 3.9ms on PC to 7.5ms on Server 1 in Figure 5.2a, and from 96.7ms on Smartphone to 221.4ms on IOT in Figure 5.2b.
The corresponding benchmarks for `DH_FW` in this setting, in orange, range from 8ms on PC to 15m.1s on Server 1, and from 199.5ms on a Smartphone to 445.5ms on a IOT device. With these benchmarks we observe a consistent overhead of about 100% throughout all systems.

Table 5.3: Benchmark results for `DH` and `DH_FW` written in `C++` on all present systems.

System	`DH`	`DH_FW`
PC	3.9ms	8ms
Laptop	5.4ms	11ms
Server 1	7.5ms	15.1ms
Server 2	5.9ms	12ms
IOT	96.7ms	199.5ms
Smartphone	221.4ms	445.5ms



(a) For all Systems except IOT and Smartphone



(b) For IOT and Smartphone

Figure 5.2: Visualization of benchmark results given in Table 5.3

El-Gamal The protocols `EG`, `EG_Rerand`, `EG_Maul` and `EG_Rerand` over the multiplicative group $\mathbb{Z}_p^*$ modulo prime p of size 3072bit and with private keys of size 3072 bit, are tested on the devices PC, Laptop, Server 1 and Server 2 with the resulting benchmarks visualized in Figure 5.3 and with absolute values given in Table 5.4.
Here we can see the benchmark results for `EG` ranging from 30.2ms on PC up to 56.7ms, when executed on Server 1.
When adding a reverse firewall with the functionality of rerandomizing the ciphertext, we get benchmark results for `EG_Rerand` of 53.1ms, when executed on a PC and 99.4ms

on our Server 1. With these measured values we can now calculate the overhead of about 53% on the PC and 51% on Server 1, with an average of 51% across all systems. Here

Table 5.4: Benchmark results for `EG`, `EG_Rerand`, `EG_Maul` and `EG_Rerand` on the given systems

System	EG	EG_Rerand	EG_Maul	EG_ReMaul
PC	30.2ms	53.1ms	46.3ms	69.1ms
Laptop	33.6ms	58.3ms	50ms	75ms
Server 1	56.7ms	99.4ms	85.8ms	127.8ms
Server 2	43.6ms	76.6ms	65.7ms	98.4ms

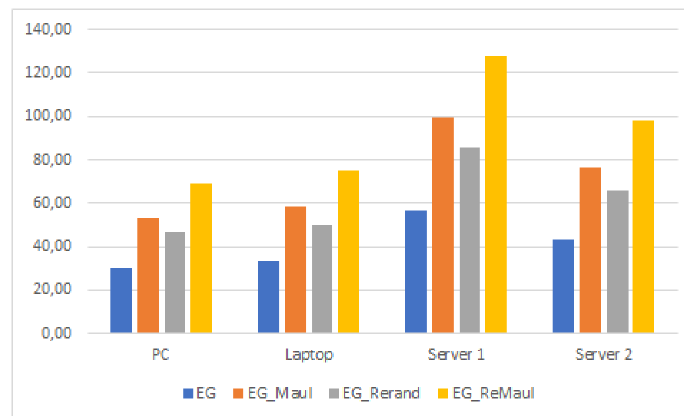


Figure 5.3: Visualization of benchmark results given in Table 5.4

we can see the benchmark results for `EG` ranging from 30.2ms on PC up to 56.7ms, when executed on Server 1.
When adding a reverse firewall with the functionality of rerandomizing the ciphertext, we get benchmark results for `EG_Rerand` of 53.1ms, when executed on a PC and 99.4ms on our Server 1. With these measured values we can now calculate the overhead of about 53% on the PC and 51% on Server 1, with an average of 51% across all systems.
If instead mauling the public key and the ciphertext we get benchmark results for `EG_Maul` starting at 53.1ms on the PC up to 99.4ms on our Server 1, leading to us to a relative overhead of 75.7% on PC and 75.3% on Server 1 with an average of 75.08% across all platforms.
In the last variant `EG_ReMaul` we utilze both, rerandomization and key mauling and get benchmark results ranging from 69.1ms on our PC to 127.8ms on Server 1 giving us an overhead of 128.8% on PC and of 125.3% on Server 1, averaging at 125.7% over all systems.
Additionally here we can observe the linear dependency of the execution time over-

head incurred, when stacking the reverse firewall's functionalities, as the resulting from overheads from EG_Rerand and EG_Maul roughly add up to the overhead incurring in EG_ReMaul.

Benchmarks over elliptic curve CURVE25519 In this paragraph we look at the results when benchmarking our protocols over our elliptic curve CURVE25519.
We start by visualizing the benchmark results of the protocols EC_DH, EC_DH_FW, EC_DH_Comm and EC_DH_Comm_FW shown in Table 5.5, in the graphs depicted in Figure 5.4.

Table 5.5: Benchmark results for the variations of EC_DH on the given systems

System	EC_DH	EC_DH_FW	EC_DH_Comm	EC_DH_Comm_FW
PC	35.5ms	52.6ms	84.2ms	137.1ms
Laptop	29.9ms	42.7ms	72.8ms	114.2ms
Server 1	42.7ms	58.9ms	100.6ms	156.1ms
Server 2	28ms	37.9ms	65.4ms	100.4ms

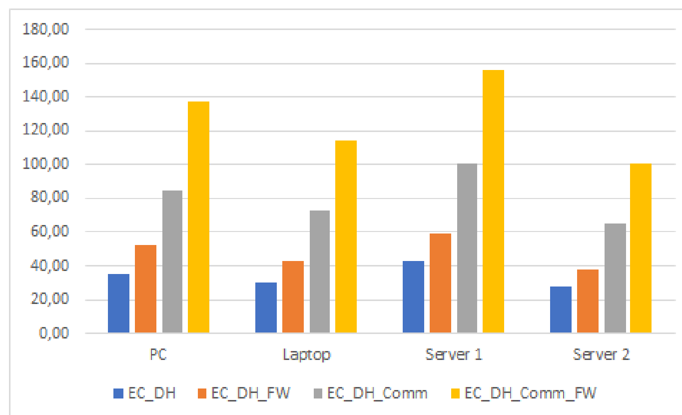


Figure 5.4: Visualization of benchmark results given in Table 5.5

The execution of EC_DH on server 2 is the fastest with 28ms, followed by the Laptop with 29.9ms, while server 1 is the slowest with 42.7ms.
Supplementing this protocol with a reverse firewall to EC_DH_FW we can see, that the execution time needed by server 2 increases to 37.9ms, while server 1 now takes 58.9ms. This leads to a time overhead of 35.3% on server 2 and of 37.9% on server 1. The discrepancy of the overhead throughout the different systems is quite large in this setting, as the PC has a time overhead incurred of 48.2% when looking at the execution time of 35.5ms needed for EC_DH and comparing with the benchmark of 52.6ms for EC_DH_FW. Overall the in this reverse firewall adds a time overhead of 41% in this scenario.

Looking at the benchmark results of protocol EC_DH_Comm we can see, that the execution on server 2 time increased to 65.4ms, followed by the laptop with 72.8ms, the PC with 84.2ms and concluded with 100.6ms on server 1. Adding the reverse firewall yielding in EC_DH_Comm_FW increases the execution time needed additionally to 100.4ms on server 2, to 114.2ms on the laptop, to 137.1ms on the PC and to 156.1ms on server 1. This leads to incurring time overheads of 53.5% for server up to 62.8% for the PC with an average time overhead of 57.1% across all systems.

El-Gamal We conclude this section with the benchmarks of our protocols EC_EG and EC_EG_Rerand. For this we show the absolute values of our results in Table 5.6 and visualize them as graphs in Figure 5.5.

Table 5.6: Benchmark results for EC_EG and EC_EG_Rerand on the given systems

System	EC_EG	EC_EG_Rerand
PC	35.5ms	52.6ms
Laptop	29.9ms	42.7ms
Server 1	42.7ms	58.9ms
Server 2	28ms	37.9ms

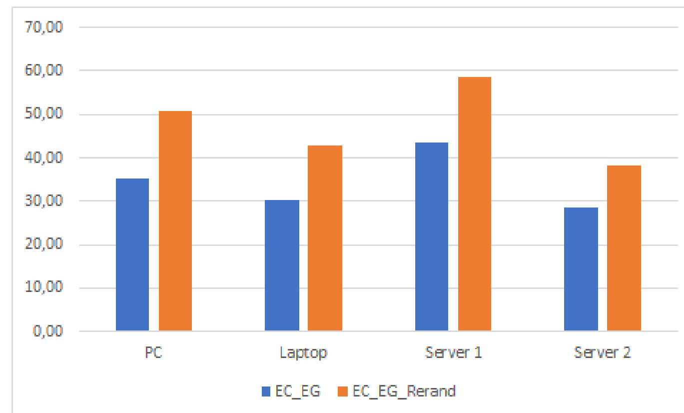


Figure 5.5: Visualization of benchmark results given in Table 5.6

The protocol EC_EG needs an average of 28.5ms on server 2, followed by 30.2ms on the laptop, 35.3ms on the PC and 43.4ms on server 1.
Supplementing this protocol with the reverse firewall to EC_EG_Rerand gives us benchmark results of 38.1ms when executed on server 2, 42.7ms on the laptop, 50.8ms on the PC, concluded with 58.6ms on server 1. These results lead us to an incurring time overhead ranging from 33.6% on server 2 to 43.9% on the PC and averaging at 38.5% across all systems.

6 Discussion

Based on the results of our benchmarks presented in chapter 5 of our various protocols elaborated in chapter 4, we can observe a fairly constant time overhead in the range of about 35 to 100%. Only the implementation of `DH` in sage provides us with an outlier of about 1040%, which leads us to conclude that this result can be neglected. We implement this protocol in `C++` and measure an approxiamte time overhead of 100% after adding the reverse firewall.

The Diffie-Hellman key exchange has a comparably low computational complexity. We neglect the generation of the private keys and only consider the asymmetric operations. Here we have four asymmetric operations within one execution, including the calculation of the public parameters and the exchange of the corresponding pairs of private and public values. When introducing the reverse firewall, we add two more asymmetric operations for rerandomizing the public parameters, yielding an estimated computational overhead of 50%. Our measurements for `DH_FW` written in `C++` show a time overhead that exceeds the calculated one by a factor of two. Possible reasons for this include high CPU utilization, missing optimization of the procedures and more.

Another observation, we can make is that for all implementations over a multiplicative group $\mathbb{Z}_p^*$ modulo a prime p, the PC is the fastest, followed by the laptop, the server 2, and finally the server 1. Once we use cryptographic operations over a discrete elliptic curve E, Server 2 and the laptop, the systems that use the latest CPUs among those available, outperform the other platforms. Here, server 2 undercuts the calculated overhead by about 25% when adding a reverse firewall to the Diffie-Hellman key exchange. From these observations, we conclude that our measurement results estimate an upper bound for a realistic scenario and that we may assume more efficient implementations in optimized environments.
Measurements of the performance of TLS on FPGAs,performed by Bellemou et al. [BGC$^+$19] result in a runtime of 1.7ms for an implementation resembling our protocol `EC_DH` in an optimized environment. If we take our determined additional time of about 40% into account here, the runtime would increase to about 2.38ms. This difference of 0.68ms is negligibly small in the context of the Internet, where latencies of about 30ms are not uncommon. This result emphasizes the practicality of reverse firewalls.

Following these findings, we discuss the deployment location of the reverse firewall. Since our attack scenario involves subverted algorithms on a machine, we assume that the attacker had access to this machine at some point in time, rendering all implementations on the system vulnerable. Consequently, and with the idea of performance in mind, we outsource the reverse firewall to a separate physical appliance. Here, we follow the concept of a hardware security module (HSM) [Fox09] or a trusted platform module (TPM). As such, the reverse firewall might utilize hardware acceleration and ASICs for the specific cryptographic operations, further reducing our estimated time overhead. Since the reverse firewall is supposed to act as part of Alice's network and not be consciously noticed by her, integration into a router or the deployment at a provider for web proxies is realistic.

In this context, we now assess whether we need to consider a stateful or a stateless implementation of the reverse firewall. Here, it all depends on the underlying protocol and the specific functionality of the reverse firewall. Examples, where a stateless implementation suffices, are protocols vulnerable to non-universal ASA, such as TLS 1.2 and Wireguard, as shown by Berndt et al. [BWP$^+$20]. To rerandomize a nonce used as a steganographic channel, the reverse firewall needs fresh randomness, but this is also only needed for this one operation, after which it can be discarded. Similarly, the implementation of the reverse firewall in `EG_Rerand` and `EC_EG_Rerand` may be stateless. Here each ciphertext is rerandomized with freshly generated randomness, which is not needed afterward and thus discarded. In contrast, protocols whose reverse firewall generated private keys are used for several messages of the run. In this case, the keys must be kept and assigned to a session, so we need a stateful implementation.

Even if the reverse firewall proves to be a viable tool to prevent exfiltration of secret information, we still face the problem here that with an ASA, state-level attackers, among others, are eligible. Glenn Greenwald [Gre14] has pointed out that organizations like the NSA are capable of intercepting and manipulating routers at delivery. We can therefore assume that this can also happen with our reverse firewall appliance. Consequently, we also have to think about other measures. Watchdogs, as introduced by Russel et al.[RTYZ16] and later developed further by Bemmann et al.[BCJ21], come into question here. These are deployed on the system to be protected and check the executed code for correct behavior. In the scenario of an ASA, we must assume that any system on site can be compromised, this includes the reverse firewall. Therefore, it is essential that for sufficient protection in security-critical areas, regular checks of the code and the deployed systems are also carried out.

7 Conclusions

7.1 Summary

Following the developments in modern cryptography, we are also dealing with increasingly powerful attacks. An example of this is the algorithm substitution attack, in which a person's secret information is leaked to a third party via regular network traffic after the algorithms involved have been subverted.
Cryptographic reverse firewalls were introduced as a countermeasure and to sanitize outgoing messages from potentially embedded secrets. This is seen as part of the public channel and protects a person's secrets without requiring them to trust it in any way. The reverse firewall is deployed within a person's own network and modifies outgoing messages before they become visible to potential observers on the Internet.
Since most of the work on this concept has been theoretical, we devote this thesis to testing the practicality of reverse firewalls in terms of runtime behavior. We found that on average we can expect an additional time overhead of about 35 to 100%, although these results are an upward estimate of a realistic scenario due to missing optimizations, among other things. For this purpose, we implement the Diffie-Hellman key exchange and the ElGamal encryption with the addition of various reverse firewalls and measure the required runtime and the resulting additional time overhead of utilizing the RF on various devices, starting with the IoT category, continuing with a laptop and PC, and ending with servers. We discovered that, on average, an additional time overhead of about 35 to 100% can be expected, although these results are an upward estimate for a realistic setting, due in part to a lack of optimizations.
To implement the reverse firewall as performantly as possible, the idea of using a dedicated physical appliance makes sense, where hardware acceleration and other optimizations concerning cryptographic operations can be used. However, against the background that we can expect state-level attackers here, we must consider that the reverse firewall may also be compromised in the event of such an attack. To counteract this, additional checks of the implementations of the algorithms used should therefore also take place regularly.

Future work

References

[Bar16] Elaine Barker. Recommendation for Key Management, Part 1: General, 2016-01-28 2016.

[BBG13] James Ball, Julian Borger, and Glenn Greenwald. Revealed: How US and UK Spy Agencies Defeat Internet Privacy and Security. *The guardian*, 2013.

[BCJ21] Pascal Bemmann, Rongmao Chen, and Tibor Jager. Subversion-Resilient Public Key Encryption with Practical Watchdogs. In *Public Key Cryptography (1)*, volume 12710 of *Lecture Notes in Computer Science*, pages 627–658. Springer, 2021.

[BGC+19] Mohamed Bellemou, Antonio Garcia, E. Castillo, Nadjia Benblidia, Anane Mohamed, J. Álvarez Bermejo, and Luis Parrilla. Efficient Implementation on Low-Cost SoC-FPGAs of TLSv1.2 Protocol with ECC_AES Support for Secure IoT Coordinators. *Electronics*, 8, 10 2019.

[BJK15] Mihir Bellare, Joseph Jaeger, and Daniel Kane. Mass-surveillance without the State: Strongly Undetectable Algorithm-Substitution Attacks. In *CCS*, pages 1431–1440. ACM, 2015.

[BL17] Sebastian Berndt and Maciej Liśkiewicz. Algorithm Substitution Attacks from a Steganographic Perspective. In *CCS*, pages 1649–1660. ACM, 2017.

[Boo22] Boost. Boost C++ Libraries. `http://www.boost.org/`, 2022.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of Symmetric Encryption against Mass Surveillance. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014.

[BWP+20] Sebastian Berndt, Jan Wichelmann, Claudius Pott, Tim-Henrik Traving, and Thomas Eisenbarth. ASAP: Algorithm Substitution Attacks on Cryptographic Protocols. *IACR Cryptol. ePrint Arch.*, page 1452, 2020.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[DMS16] Yevgeniy Dodis, Ilya Mironov, and Noah Stephens-Davidowitz. Message Transmission with Reverse Firewalls - Secure Communication on Corrupted Machines. In *CRYPTO (1)*, volume 9814 of *Lecture Notes in Computer Science*, pages 341–372. Springer, 2016.

[Don17] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *NDSS*. The Internet Society, 2017.

[DSJ+20] The Sage Developers, William Stein, David Joyner, David Kohel, John Cremona, and Burçin Eröcal. *SageMath, version 9.0*, 2020. https://www.sagemath.org/.

[Fox09] Dirk Fox. Hardware Security Module (HSM). *Datenschutz und Datensicherheit - DuD*, 33, 09 2009.

[Gre14] Glenn Greenwald. No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State. In *Metropolitan Books*, 2014.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. *RFC*, 7748:1–22, 2016.

[Mil76] Gary L. Miller. Riemann's Hypothesis and Tests for Primality. *J. Comput. Syst. Sci.*, 13(3):300–317, 1976.

[MS15] Ilya Mironov and Noah Stephens-Davidowitz. Cryptographic Reverse Firewalls. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 657–686. Springer, 2015.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[RTYZ16] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Cliptography: Clipping the Power of Kleptographic Attacks. In *ASIACRYPT (2)*, volume 10032 of *Lecture Notes in Computer Science*, pages 34–64, 2016.

[Sch07] Bruce Schneier. Did NSA Put a Secret Backdoor in New Encryption Standard?, 2007.

[Sig] Signal Foundation, Signal Messenger LLC and contributors. Signal.

[YY96] Adam L. Young and Moti Yung. The Dark Side of "Black-Box" Cryptography, or: Should We Trust Capstone? In *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 1996.

[YY97] Adam L. Young and Moti Yung. Kleptography: Using Cryptography Against Cryptography. In *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1997.