Exploring threshold cryptosystems

Analysing and improving performance of Thetacrypt

Master Thesis

Michael Senn

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

Prof. Christian Cachin
Orestis Alpos

June, 2023

Abstract

Threshold cryptography distributes the secret information — such as the secret key — across multiple parties. A large enough subset of these parties is then required to collaborate to perform certain operations.

Threshold cryptography goes back many decades, but has not found use outside of a select few organizations for a long time. Recently there has been an uptick of interest, with companies such as DFINITY and organizations such as NIST working on adopting and standardizing it. However existing threshold cryptosystems still remain difficult to deploy and maintain. The CRYPTO research group at the University of Bern has been working on Thetacrypt, an application aiming to be used as a middleware providing threshold services in distributed systems such as blockchains.

Multiple IND-CCA-secure threshold ciphers have been proposed in the literature, such as one by Shoup and Gennaro in 2002, and another by Baek and Zheng in 2003. Support for both of these has been added to Thetacrypt. This thesis has the goal of analysing and improving performance of these two threshold ciphers in Thetacrypt. It finds the cause of and fixes many performance issues, improving decryption latency for real-world message sizes by multiple orders of magnitude.

Contents

1 Introduction

Threshold cryptography describes cryptographic schemes where the secret information — such as the private key — is distributed across a number of parties. A certain amount of these parties is then required to cooperate to perform certain operations, such as decrypting a ciphertext or signing a message. Threshold cryptography allows one to perform cryptographic operations in scenarios where no single party is universally trusted, such as in blockchains where a number of parties performs the validation of transactions. It bases its security on the assumption that at most a certain number of parties participating in the scheme will behave maliciously.

Work on threshold cryptography goes back many decades [DF90; De +94], but has not been used outside of a select few organizations for a long time. Recently the use of threshold cryptography has seen an uptick, such as DFINITY's use of distributed randomness beacons [HMW18], projects in the blockchain space such as Osmosis supporting threshold signatures for transactions [Osm], or the work of organizations such as NIST towards standardization of threshold cryptographic schemes [BDV20] [BP23]. However, existing threshold cryptosystems are still difficult to set up and maintain, hindering adoption.

The CRYPTO research group of the Institute of Computer Science at the University of Bern has been working on *Thetacrypt*, an application implementing various threshold primitives such as threshold ciphers, signature schemes, randomness beacons or key generation protocols. Thetacrypt comes with its own protocol layer allowing independent Thetacrypt nodes to execute threshold cryptographic protocols among each other. This allows deploying Thetacrypt as child nodes alongside nodes of an existing distributed application. Each Thetacrypt node will then expose an interface towards its parent node, which allows the latter to perform threshold cryptographic operations.

Thetacrypt is designed with a modular communication layer so that it can reuse existing communication layers which the distributed system might already be providing.

Among the threshold cryptographical primitives that Thetacrypt implements are two threshold ciphers. One is an implementation of work by Shoup and Gennaro [SG02], the other an implementation of work by Baek and Zheng [BZ03]. Both these schemes provide security against chosen-ciphertext attacks, but do so by different mechanisms. The Shoup-Gennaro scheme works in subgroups of $\mathbb{Z}_p$, and utilizes Zero-Knowledge proofs to achieve CCA-security. The Baek-Zheng scheme makes use of elliptic curves permitting bilinear pairings to achieve the same.

In this thesis, performance of the two threshold ciphers is analyzed using a variety of benchmarks.

Where performance bottlenecks are found they are fixed, if possible. It also tries to quantify performance, with a focus on the decryption latency.

Chapter 2 starts with an introduction to related topics. Chapter 3 describes the ways by which performance of Thetacrypt was analyzed. Chapter 4 presents which performance bottlenecks were found and how they were fixed, along with measurements showcasing the decryption latency of the two threshold ciphers in different deployment scenarios. Chapter 5 finally concludes.

2 Background

This chapter aims to provide required background on any of the topics which are relevant to the thesis. It will start with a brief introduction into the use of bilinear pairings in cryptography. It will then discuss the topic of threshold cryptography, touching upon formal security properties and two specific threshold ciphers. Then there will be an introduction to libp2p, a peer-to-peer networking stack. Finally, there will be some information about Thetacrypt, a Rust application implementing various threshold schemes.

2.1 Pairing-based cryptography

Pairing-based cryptography refers to cryptographic schemes utilizing bilinear maps — often also called bilinear pairings — to achieve their respective design goals. The use of bilinear maps to construct rather than attack cryptographic schemes started in the early 2000s, with e.g. Boneh and Franklin using a Weil pairing in an identity-based encryption scheme [BF01], or Joux using pairings for a three-party variant of the Diffie-Hellman key exchange [Jou00].

2.1.1 Bilinear maps

Formally a bilinear map is defined as follows, adapted from a lecture by John Bethencourt [Bet15]:

Definition 1 *Let G_1, G_2, G be cyclic groups of equal order. Then, a function*

$$e : G_1 \times G_2 \to G$$

such that, for all $g_1 \in G_1, g_2 \in G_2, a, b \in \mathbb{Z}$:

$$e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$$

is called a bilinear map.

This definition also permits degenerate functions, such as the one mapping all inputs to the neutral element of G. In applied cryptography, an implicit assumption is thus often that the map is not degenerate, and that it can be computed in an efficient manner.

2.1.2 Concrete pairings used in cryptography

In order to be useful for cryptographic schemes, there are additional requirements a pairing must fulfil. Among others the DLP in G_2 must be hard, as else the one in G_1 becomes easy [Bet15]. Further, it must be possible to efficiently compute the pairing as well as efficiently work with elements of the groups over which the pairing operates.

This rules out many easy constructions of bilinear pairings. Two pairings which have been shown to work are called the Weil and Tate pairings, both of which are defined over elliptical curves. For details on their construction see the survey paper by Joux [Jou02].

2.2 Threshold cryptography

Threshold cryptography is a subfield of public-key cryptography where the secret key is split across multiple parties, using some form of secret-sharing technique such as Shamir's [Sha79]. Then, multiple parties must collaborate to perform secret-key operations such as decryption (in the case of threshold ciphers) or signing (in the case of threshold signature schemes).

For notation, we will assume a system with a total of n participants, $f + 1$ of which are required to cooperate to perform the secret-key operation. The resultant system is then called an '$(f + 1)$-out-of-n' or $(f + 1, n)$ cryptosystem.

As the focus of the thesis was on threshold ciphers, threshold signature schemes will be skipped over

2.2.1 IND-CCA security

Cryptosystems aim to formalize the security properties they provide, given assumptions on the capabilities of the adversary. One such combination of adversarial capabilities and security guarantees is termed 'IND-CCA'. Informally — in the non-threshold case — it ensures that an adversary is exceedingly unlikely to learn useful amounts of information about an observed ciphertext even if they can get access to encryptions of arbitrary plaintexts, as well as to decryptions of ciphertexts other than the one they are interested in.

IND-CCA in the threshold case The IND-CCA setting can be generalized to the threshold case in a meaningful way. In addition to the capabilities the adversary has in the non-threshold case, the adversary also gets to choose f out of the n parties to corrupt, immediately gaining access to their key shares. When later on submitting a query to the decryption oracle, the adversary will also gain access to the intermediate decryption shares produced by the corrupted parties. This can make it harder to achieve provable CCA security for threshold ciphers [SG02]. A formal definition of IND-CCA in the threshold case can be found in the same paper.

Both threshold ciphers which will be discussed achieve IND-CCA. The SG02 scheme does so using zero-knowledge proofs, the BZ03 scheme using bilinear pairings.

2.2.2 SG02 scheme

In 2002, Shoup and Gennaro proposed two versions (called 'TDH1' and 'TDH2') of a CCA-secure threshold cryptosystem based on the CDH and DDH problem respectively [SG02]. They achieve provable CCA-security in the random oracle model by using various non-interactive ZKPs. While the version based on the DDH problem uses a stronger assumption, it is more efficient due to being able to reuse a group element across all encryptions with a given key pair.

According to the authors, their system is the first CCA-secure threshold cryptosystem based on the Diffie-Hellman problem which can be efficiently implemented. Specifically, they point out that their

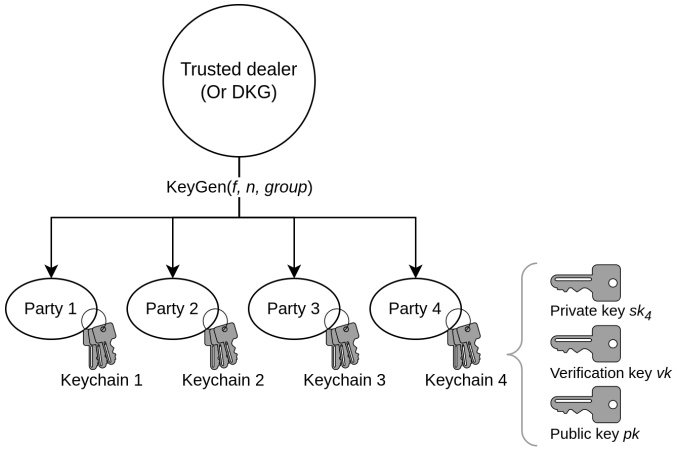


Figure 2.1: Key generation

system does not rely on any guarantees on network latency, making it suitable for use with a public communication network.

2.2.3 BZ03 scheme

In 2003, Baek and Zheng proposed an alternative CCA-secure threshold cryptosystem. They describe their work as a novel application of the techniques introduced by Shoup and Gennaro, but without the need for non-interactive ZKPs to achieve CCA security. In their system they instead achieve public verifiability of ciphertexts and decryption shares by working in a gap Diffie-Hellman ('GDH') group, constructed using a bilinear pairing [BZ03].

2.2.4 Common structure to threshold ciphers

Executions of both the SG02 and BZ03 threshold ciphers follow a common structure which will be referred to throughout the thesis. A brief explanation of each step follows.

Key generation During key generation, shown in Figure 2.1 for $n = 4$, public and private keys specific to the scheme will be generated and distributed to the involved parties. The key generation routine will, as input, take the number of parties n and the threshold parameter f, as well as which group the scheme will operate on. Once done each party will be in possession of a common public key pk and verification key vk, as well as of a share sk_i of the secret key sk.

Both schemes describe their key generation under the assumption that it is handled by a trusted party. This assumption could potentially be relaxed by employing a distributed key generation (DKG) protocol.

Encryption Encryption works the same way it does in any asymmetric cryptosystem. Any party in possession of the public key pk can encrypt a message m, producing a ciphertext c as shown in Figure 2.2.

Partial decryption To initiate decryption the to-be-decrypted ciphertext c has to be sent to sufficiently many (at least $f + 1$) parties. Figure 2.3 shows this assuming these decryption requests to be issued by a

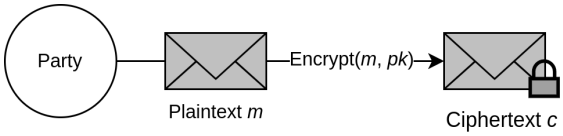


Figure 2.2: Encryption

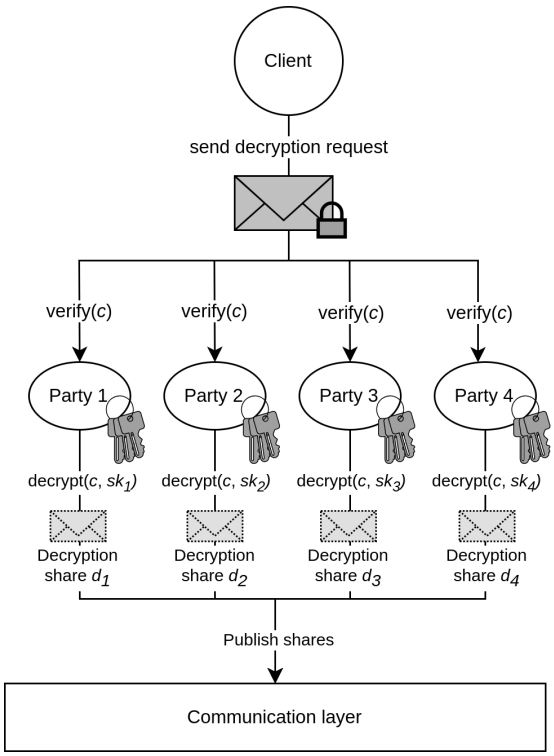


Figure 2.3: Partial decryption

central client. Depending on the specific setup this need not necessarily be the case though, as long as a decryption request ends up at a sufficiently large number of threshold parties.

Upon receiving a decryption request, a party will first verify the validity of such. While the details are specific to each scheme, this step plays an essential role in providing CCA security. If the ciphertext is deemed valid, the party will generate a decryption share using its share of the private key. These decryption shares are then sent to the other threshold parties, using whatever communication layer is available.

Share assembly Upon receiving decryption shares from the communication layer, a party will validate these shares using a scheme-specific operation. Like validation of ciphertexts this is required to provide CCA security. It is in this step that the validation key vk is required. Upon having received at least $f + 1$ valid shares the party can assemble them, yielding the original plaintext m as shown in Figure 2.4.

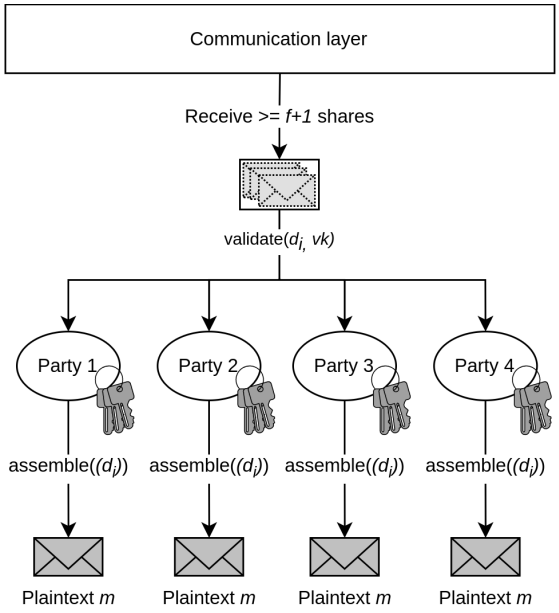


Figure 2.4: Assembly of decryption shares

2.3 Libp2p

Libp2p[1] is an implementation of a multi-platform networking stack providing various peer-to-peer protocols. One of these protocols is termed *gossipsub*. It is an implementation of a publish/subscribe system atop a gossip-based communication channel.

Publish / subscribe communication pattern Publish / subscribe ('pubsub') is a communication pattern where messages are organized around topics. Communication parties can subscribe to topics they are interested in. When a message is published, it is assigned to one (or multiple) of these topics. The system will then ensure that peers which are subscribed to any of the topics to which the message was assigned will receive it. Pubsub systems have many applications in computer science. They are used wherever there is a diverse combination of producers and consumers of messages, as they allow each consumer to choose precisely which messages they wish to receive [lib].

The pubsub pattern does not prescribe by which way messages are propagated between peers. This allows using the most appropriate form of message propagation for the given use case.

Gossipsub implementation in libp2p In Libp2p's gossipsub protocol, peers establish two types of mesh networks. A densely connected network is used to exchange metadata about which topics and messages exist as well as for network maintenance. A sparsely connected network is used to exchange the actual messages [lib].

The degree of connectedness of the sparsely connected mesh network can be tuned to find the appropriate tradeoff between fast message delivery (when the degree of meshing is high) and low network bandwidth usage (when the degree of meshing is low).

[1] `https://docs.libp2p.io/`

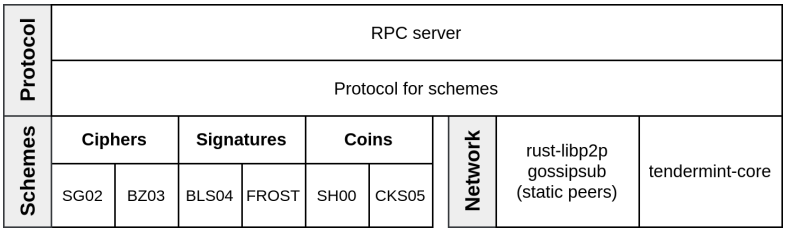


Figure 2.5: High-level architecture of Thetacrypt

Peer discovery Establishing a mesh network requires being aware of other peers. This can be either achieved by statically providing a list of peers to every party in the system, or by use of a discovery protocol based on mechanisms such as mDNS, distributed hash tables, or a set of bootstrapping nodes. Libp2p supports multiple different discovery mechanisms which can be integrated into applications using libp2p.

2.4 Thetacrypt

Thetacrypt is a Rust[2] application, aiming to implement threshold primitive such as threshold ciphers and signatures, distributed key generation, threshold coin schemes and randomness beacons. It is worked on by the Cryptology and Data Security ('CRYPTO') research group at the University of Bern, Switzerland.

2.4.1 Architecture

To the user, Thetacrypt exposes a layered architecture as shown in Figure 2.5. At the bottom layer, it functions as a Rust library ('crate'), providing direct access to various schemes, shown in Table 2.1. The API of these schemes generally mirrors the respective interface as described in their respective whitepaper. As an example, the API of a threshold cipher will provide the functionality to encrypt, to generate decryption shares, to validate ciphertexts and decryption shares, and to assemble shares into a plaintext.

Being threshold systems, these schemes generally require communication between the threshold parties. To facilitate this, Thetacrypt provides a custom protocol layer with an RPC server atop. The RPC server allows a client to submit requests, such as a ciphertext which to decrypt. The server will then handle calling the required functions of the underlying scheme, and exchange messages with other threshold nodes. Finally, the RPC server will provide the result of the operation to the calling client. Not all schemes have been integrated into the protocol layer yet, with the focus having been on integrating the threshold ciphers.

This protocol layer requires access to a network layer with which it can communicate with the other threshold nodes. Thetacrypt is designed so that this layer is exchangeable, to accommodate different usage scenarios. Thetacrypt currently supports two network modules, one based on the gossipsub implementation of the Rust version of libp2p, the other piggybacking on the communication layer of Tendermint core. The thesis relies solely on the former as it was more stable at that time.

Integration of libp2p gossipsub into Thetacrypt Thetacrypt provides a wrapper around libp2p's gossipsub implementation, allowing one to use it to exchange messages between Thetacrypt threshold nodes. It does not implement any kind of peer discovery, requiring one to provide a full set of peers — or at least overlapping sets of peers — to every Thetacrypt node.

[2] `https://www.rust-lang.org/`

Identifier	Description	RPC integration
SG02	CCA-secure threshold cipher by Shoup and Gennaro [SG02]	Yes
BZ03	CCA-secure threshold cipher by Baek and Zheng [BZ03]	Yes
BLS04	Threshold signature by Boneh, Lynn and Shacham [BLS04]	No
FROST	Threshold signature by Komlo and Goldberg [KG21]	No
SH00	Threshold signature by Shoup [Sho00]	No
CKS05	Threshold coin by Cachin, Kursawe and Shoup [CKS05]	No

Table 2.1: Threshold schemes implemented in Thetacrypt

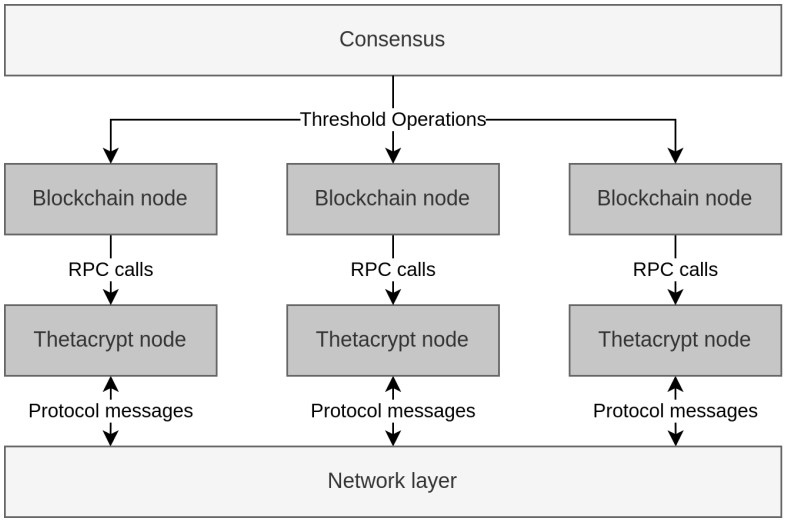


Figure 2.6: Possible deployment of Thetacrypt

Envisioned deployment scenario A deployment of Thetacrypt as part of a distributed system such as a blockchain might look as in Figure 2.6. In this scenario, each node of the distributed system would be running its own Thetacrypt RPC server. When the operation of the distributed system — such as the consensus layer of a blockchain — would require threshold operations to be performed, each node of the distributed system would then interact with its own instance of Thetacrypt via its RPC layer. The Thetacrypt nodes would execute the threshold operation among each other, and then return the result to their caller.

This also hints at the main motivation behind allowing to swap out Thetacrypt's network layer — doing so allows Thetacrypt to reuse whatever communication layer the distributed system already happens to provide.

2.4.2 Thetacrypt internals

Large parts of Thetacrypt's protocol and scheme layers can be considered as a black box for the purpose of this thesis. Some insight into its internal workings is required, however, which will be provided here.

Supported algebraic groups for discrete-logarithm based cryptosystems Thetacrypt currently provides support for three elliptical curves over which to operate. These are referred to as *Bls12381*, *Bn254*, and

Ed25519. Bls12381 refers to a pair of 381-bit curves permitting for a bilinear pairing[3], based on a construction by Barreto et al. [BLS03]. Bn254 refers to a 254-bit Weierstrass curve, also permitting for a bilinear pairing[4], based on a construction of Barreto and Naehrig [BN06]. Ed25519 finally refers to a 256-bit twisted Edwards curve, without support for bilinear pairings[5], based on work by Bernstein et al. [Ber+12].

Due to its requirement for pairings, Bls12381 and Bn254 can be used for the BZ03 scheme. All three curves can be used for SG02.

All three curves target a 128-bit security level. However recent attacks have limited Bn254 to around 100 bits of security [BD19]. Bls12381, while also falling slightly short of its goal, is still assumed to provide around 126 bits of security [GMT20]. No attacks are known against Ed25519.

Concurrent instances of threshold protocols A Thetacrypt server is able to handle multiple concurrent instances of a threshold protocol. As an example, it can handle any number of concurrent threshold decryptions using any of the supported schemes. To do so it must firstly keep track of the individual state of each protocol instance, and secondly be able to determine, upon receiving a message, which protocol instance it belongs to.

To handle the first task, Thetacrypt introduces what it calls a 'state manager'. Upon receiving a client request the state manager will instantiate a new instance of the protocol, and assign it a unique ID. The state manager keeps track of each instance's state, such as the submitted ciphertext and the set of all received decryption shares.

To handle the second task, Thetacrypt will tag any outgoing message belonging to a protocol instance with the instance's ID. When it later receives a message it can forward it to the appropriate protocol instance for handling.

The choice of how to determine such an instance ID depends on the assumptions of how Thetacrypt will be deployed, respectively on the trust assumptions put on each server's clients. If each server trusts its clients (that is if there is no trust boundary between a Thetacrypt server and whatever is calling it), then it is sufficient to use the 'label' field defined as part of the threshold cipher. It is then up to the client to ensure that these labels are unique for each request. If a Thetacrypt node did not trust its clients then it would have to include e.g. a hash of the decryption request in the protocol instance ID, to prevent trivial DOS attacks.

Support for multiple schemes and groups Thetacrypt is able to support not only concurrent instances of one scheme and group but indeed concurrent instances of multiple combinations of scheme and group. To do so, each Thetacrypt node can be in possession of a keychain containing any number of keypairs for any of the supported schemes and groups. Each decryption request specifies which keypair is to be used, which implicitly specifies which scheme and group to use.

This keychain is provided to the Thetacrypt server binary as an input parameter. A separate binary exists to generate keychains if a trusted dealer exists, with distributed key-generation protocols still being under development.

Thetacrypt further provides an endpoint for clients to query for any of the public keys on its keychain, simplifying key distribution — assuming the connection between a Thetacrypt server and its clients is secure and the client trusts its Thetacrypt server.

Blocking and non-blocking API of RPC server To interact with Thetacrypt's RPC server, a client can choose to use either the blocking or the non-blocking endpoints. In the blocking case, the client will

[3] `https://neuromancer.sk/std/bls/BLS12-381`
[4] `https://neuromancer.sk/std/bn/bn254`
[5] `https://neuromancer.sk/std/other/Ed25519`

submit its query — such as a decryption request — which will make Thetacrypt initiate and start the underlying threshold protocol. The connection to the client will be kept open until the threshold protocol terminates. When it terminates, the response is returned to the calling client. For the client, this has the benefit that it receives the response the moment it is available, but it must be able to handle a connection which will be blocking for a — potentially — indefinite time.

In the non-blocking case, as in the blocking case, the client will first submit its query. Thetacrypt will then initiate the underlying threshold protocol and will return the protocol's instance ID to the client, closing the connection. At any point in the future, the client can then query Thetacrypt for the status of a protocol instance by providing its instance ID. If the protocol instance has terminated since, it will receive the result, otherwise a message indicating that it is in progress still. To the client, this requires implementing logic to periodically poll for status updates, which also means it will likely receive the result later than if it had used the blocking API. It does however free the client from having to keep a potentially long-lived connection alive.

On the side of Thetacrypt, the non-blocking interface requires keeping track of results after the protocol has terminated so that it can send the result to the client at a later point. Currently, Thetacrypt keeps results in memory forever, leading to unbounded memory usage over time. Fixing this would require implementing a cache-eviction policy, to remove old decryption results based on some heuristic.

Hybrid cryptosystem Both described threshold cryptosystems, due to operating over structures on finite fields, will only support messages of a small size. Specifically, the elliptic curves supported by Thetacrypt will limit messages to at most $256\,\mathrm{bit}$. To handle messages of arbitrary size, Thetacrypt implements a hybrid cryptosystem.

Informally it will first derive a key sk_{sym} for an (IND-CCA-secure) symmetric cipher. It then encrypts the message m with the symmetric cipher, yielding ciphertext c_{sym}. The secret key sk_{sym} is then encrypted with the threshold cipher, yielding ciphertext c_{thresh}. The tuple $(c_{\text{sym}}, c_{\text{thresh}})$ then forms the full ciphertext.

To decrypt, c_{thresh} is first decrypted using the threshold scheme, yielding the secret key sk_{sym} of the symmetric cipher. Then c_{sym} can be decrypted, yielding the message m.

Specifically, Thetacrypt implements ECIES as described in the SEC 1 standard [Bro09, section 5.1]. It uses ChaCha20-Poly1305 as an authenticated-encryption primitive, combining the ChaCha20 stream cipher with the Poly1305 MAC.

3 Methods

This chapter aims to describe the methods employed to analyze and improve performance of Thetacrypt. Two techniques are used - microbenchmarks to analyze performance of specific functions, and macrobenchmarks to analyze performance of a full deployment of Thetacrypt, attempting to emulate real-world workloads. A brief performance evaluation of Thetacrypt's gossipsub network layer is also discussed.

There are some properties a good benchmark should strive to achieve. It should be repeatable, allowing to re-run it — as automatically as possible — when the underlying system changes. It should measure and expose metrics are meaningful. And the results of the analysis should be presented in a way such that they are easily understood [Gre20]. The described benchmarks attempt to stick to these best practices.

3.1 Microbenchmarks

Microbenchmarks measure performance of a narrow code path — such as an invocation of a single function — given a set of input parameters. Since they generally exercise small amounts of code, they allow to easily find the cause of a performance issue when one is observed. Their main disadvantage is that the measured workload is not necessarily representative of the workload in a real-life deployment [Gre20].

Thetacrypt's API of the scheme layer closely mirrors the structure common to threshold ciphers as described in Section 2.2.4. This allowed to separately benchmark each of those steps.

Choosing benchmarking metrics Before running a benchmark one must decide on a set of metrics to measure during the benchmark. By necessity, they must be measurable and should be indicative — in some way — of what one wants to achieve with the benchmark.

As the goal was to measure Thetacrypt's performance, the focus was put on three metrics: Execution time, output size for functions which produce an output, and heap memory usage. Some of these can be measured directly when calling the code, others require the use of third-party tools.

Choice of microbenchmark parameters One of the choices to be made is which parameters to vary, as well as which values to pick for the chosen parameters. This choice can be based on prior knowledge of

Parameter	Expected affected stages
Scheme	All
Group	All
Number of parties n	Key generation
Threshold parameter f	Key generation, decryption share assembly
Message size	Encryption, decryption

Table 3.1: Parameters supported in microbenchmark experiment

Scheme	Group	$(n, f+1)$	Message size [B]
SG02	Bls12381	$\text{keygen-params}(\{10, 50, 100, 200, 500, 1000, 2000, 3000, 4000\})$	-
	Bn254	$\text{keygen-params}(\{10, 50, 100, 200, 500, 1000, 2000, 3000, 4000\})$	-
	Ed25519	$\text{keygen-params}(\{10, 50, 100, 200, 500, 1000, 2000, 3000, 4000\})$	-
BZ03	Bls12381	$\text{keygen-params}(\{10, 50, 100, 200, 500, 1000, 2000, 3000, 4000\})$	-
	Bn254	$\text{keygen-params}(\{10, 50, 100, 200, 500, 1000, 2000, 3000, 4000\})$	-

Table 3.2: Choice of microbenchmark parameters for key generation. The choice of message size is irrelevant for key generation.

the components which are to be benchmarked. Taking an encryption function as an example one expects its execution time to depend on the message size, but not on the total number of participants in the system. Nonetheless it can be advisable to briefly verify that a component's behaviour does indeed not depend on such parameters. Table 3.1 shows which input parameters were varied during the microbenchmark, as well as which stages of the threshold cipher they are expected to affect in some way.

To abbreviate notation we introduce a shorthand for the list of valid $(n, f+1)$ combinations. Let $X \subset \mathbb{N}$ be a finite set of natural numbers. Define $\text{keygen-params}(X) := \{(a, b) : a, b \in X \wedge a \geq b\}$. As an example, $\text{keygen-params}(\{10, 50, 100\}) = \{(10, 10), (50, 10), (50, 50), (100, 10), (100, 50), (100, 100)\}$. Each such tuple (a, b) is then a valid choice for key generation with $n = a$, $f + 1 = b$.

As not all parameters were expected to influence all stages of the threshold cipher, not every stage was tested with every combination of values for all parameters. Instead, specific choices of parameters are shown in Table 3.2 for the key generation stage, in Table 3.3 for encryption and assembly of decryption shares, and in Table 3.4 for ciphertext validation, partial decryption, and validation of decryption shares.

Scheme	Group	n	f	Message size [B]
SG02	Bls12381	1000	333	$\{2^0, 2^4, 2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}\}$
	Bn254	1000	333	$\{2^0, 2^4, 2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}\}$
	Ed25519	1000	333	$\{2^0, 2^4, 2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}\}$
BZ03	Bls12381	1000	333	$\{2^0, 2^4, 2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}\}$
	Bn254	1000	333	$\{2^0, 2^4, 2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}, 2^{28}\}$

Table 3.3: Choice of microbenchmark parameters for encryption and assembly of decryption shares. Message size is provided as a list, every choice of which was tried in combination with any of the other parameters on the same line. The choice of n and $f + 1$ does not affect performance so is arbitrary.

Scheme	Group	n	f	Message size [B]
SG02	Bls12381	1000	333	1024
	Bn254	1000	333	1024
	Ed25519	1000	333	1024
BZ03	Bls12381	1000	333	1024
	Bn254	1000	333	1024

Table 3.4: Choice of microbenchmark parameters for ciphertext validation, partial decryption, and validation of a single decryption share. The choice of n, $f+1$ and the message size does not affect performance so is arbitrary.

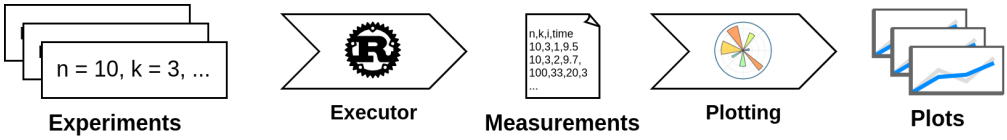


Figure 3.1: Microbenchmark pipeline

3.1.1 Microbenchmarking Thetacrypt

While it would be possible to test every stage of the threshold decryption process with every possible combination of input parameters, this would involve a lot of time spent executing code with no observable effect. Instead a smarter procedure is required which can selectively evaluate the various components using subsets of input parameters.

To do this, a custom microbenchmarking pipeline was defined, shown in Figure 3.1. As input, it takes a set of 'experiments', which define which component of the threshold cipher to benchmark, which metrics to collect, and which input parameters to vary. It further allows to specify the RNG seed to use — for repeatability — as well as an iteration count which governs how often each specific measurement is repeated. Parameters can be specified as a set of values to try, such as defining to try values of $n \in \{3, 7, 22, 100\}$, or can be configured dependant on one another, such as defining the threshold parameter f to be dynamically chosen such that $N = 3F + 1$.

These experiments are passed to the executor, a Rust application which interacts with Thetacrypt's schemes layer. It is aware of how to translate the requirements formalized in the experiment definitions into calls to Thetacrypt. When executing Thetacrypt code it is also able to gather all requested metrics and log them as a set of CSV files.

These CSV files are then passed to a suite of Python tools. First, the data is aggregated and analyzed with the Pandas[1] library. Then, results are visualized using Matplotlib[2]. The resulting plots are saved to disk as images for further analysis.

Measuring memory consumption While some metrics such as execution time and output size can be measured from within Rust directly, this is not possible for memory consumption. Instead, for experiments where memory consumption is to be measured, the pipeline is wrapped in Valgrind's *massif* tool[3] — a tool to profile heap memory consumption of applications. To automate the profiling of memory usage, utility scripts were created to act as glue between the experimental definitions, Valgrind, and the

[1] `https://pandas.pydata.org/`
[2] `https://matplotlib.org`
[3] `https://valgrind.org/docs/manual/ms-manual.html`

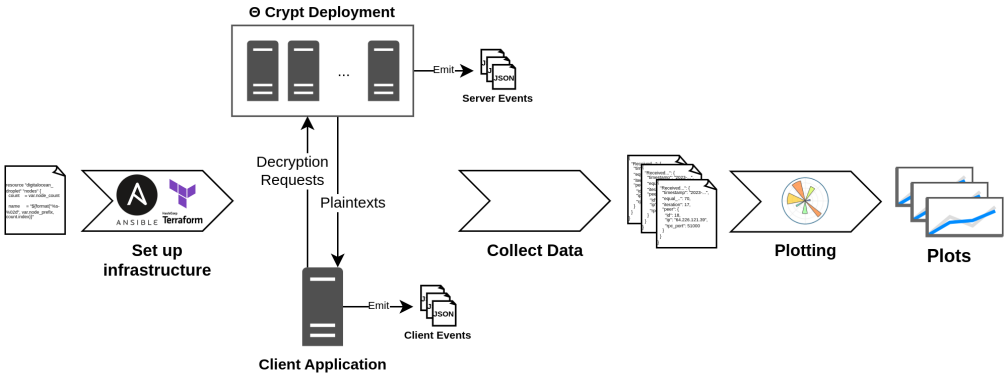


Figure 3.2: Macrobenchmark pipeline

Rust-based executor.

3.2 Macrobenchmarks

Macrobenchmarks, sometimes also called simulation benchmarks, exercise a whole system with an approximation of real-life workloads [Gre20]. They can be used when recordings of real-life workloads are not available, as in our case since there is no real-life deployment of Thetacrypt yet, or when other factors such as confidentiality prevent using such.

Their advantage is that they provide a middle-ground between the artificial workloads of microbenchmarks, and the lack of flexibility of replaying real-world workloads. They can easily be tuned to accommodate new requirements while still exercising large parts of the system.

3.2.1 Macrobenchmarking Thetacrypt

In order to macrobenchmark Thetacrypt a custom client application was written. This client application interfaces with Thetacrypt's RPC and scheme layers. It can thus encrypt a plaintext, submit decryption requests for the resultant ciphertext to a deployment of Thetacrypt, and receive the plaintexts which the Thetacrypt nodes return.

Automating the deployment of Thetacrypt will be discussed in more detail in Section 3.3. Until then a multi-server deployment of Thetacrypt is assumed to exist.

Similarly to microbenchmarks a custom pipeline was defined, shown in Figure 3.2. Its centerpiece is the client application which interacts with the Thetacrypt deployment. Both Thetacrypt as well as the client application emit events allowing to track how much time they spent on various stages. This data is collected, analyzed, and visualized using the same tools as are used for microbenchmarks.

3.2.2 Choosing a network layer

As mentioned earlier Thetacrypt supports different network layers. For macrobenchmarks the libp2p-based gossipsub layer was chosen for two reasons. Firstly its integration into Thetacrypt was deemed stable enough to be usable for a benchmark. Secondly, it was assumed to be performant enough so that the macrobenchmark could capture performance of Thetacrypt, rather than performance of its networking layer.

Number of parties n	Threshold parameter f	Geographical distribution
7	2	Regional
7	2	Global
22	7	Regional
22	7	Global
100	33	Regional
100	33	Global

Table 3.5: Deployment parameters for macrobenchmark

Thetacrypt's integration of this network layer does not support any form of peer discovery. As such each Thetacrypt server must be provided with a list of all other Thetacrypt servers so that it can establish a mesh network. As the set of all nodes is well-known in the case of a macrobenchmark this was no real limitation.

To ensure performance of the network layer was not be the limiting factor of the benchmark, it too was evaluated in isolation. The network layer's benchmark is described in more detail in Section 3.4.

3.2.3 Macrobenchmark parameters

For macrobenchmarks, multiple parameters are considered:

- The number of threshold parties n, with the threshold parameter f chosen such that $n = 3f + 1$
- The geographical distribution of the servers
- The combinations of threshold cipher and groups
- The size of ciphertexts which to decrypt

Number of servers and geographical distribution Two geographical distributions are differentiated: *Regional* refers to a deployment where all servers are in the same geographical region. *Global* refers to a deployment where servers are distributed across the world.

The choice of the number of servers n is based on NIST's first call for multi-party threshold schemes which specifies five ranges [BP23]: *Two* for $n = 2$, *Three* for $n = 3$, *Small* for $4 \leq n \leq 8$, *Medium* for $9 \leq n \leq 64$, *Large* for $65 \leq n \leq 1024$ and *Enormous* for $n \geq 1025$.

Of these, $n = 7$ for the small profile, $n = 22$ for the medium one, and $n = 100$ for the large one are chosen. In all cases a 2/3 honest majority is assumed, that is the threshold parameter f is chosen such that $n = 3f + 1$. The full set of deployment parameters used for macrobenchmarks is shown in table 3.5. The sole exception is the first benchmark, where $n = 21, f = 7$ is used.

Choice of threshold ciphers and groups Both threshold ciphers supported by Thetacrypt (SG02 and BZ03) were evaluated. For either scheme, all groups supported by Thetacrypt (Bls12381 and Bn254 for both schemes, Ed25519 for SG02 only) were evaluated.

Choice of message size For the size of messages, logarithmically spaced values between $1\,\mathrm{B}$ and $16\,\mathrm{MB}$ were chosen. This range spans values of messages which one might feasibly encounter in a threshold setting, such as an AES-256 key ($32\,\mathrm{B}$), a block on a blockchain such as Ethereum (lately $32\,\mathrm{kB}$ to $512\,\mathrm{kB}$), or a small PDF document.

3.2.4 Inner workings of benchmarking client

Having provisioned a Thetacrypt deployment, the next step is to start the benchmarking client. It takes multiple parameters as input:

- A list of Thetacrypt servers, as IPs or DNS records
- A set of combinations of scheme and group which to benchmark
- A set of message sizes which to benchmark
- A PRNG seed

In the first step, the client application will seeds its PRNG with the provided seed, to ensure repeatable experiments. Having done that the client will iterate over all supplied combinations of a scheme and group. It will query all Thetacrypt nodes for the appropriate public key pk for this combination of scheme and group.

Then the client will iterate over all supplied message sizes. For each it will generate a random plaintext m of appropriate size and encrypt it using the retrieved public key, generating ciphertext $c = Enc(m, pk)$.

In the next step, it will send a decryption request for ciphertext c to all servers. Calls are issued to Thetacrypt's blocking API, using an asynchronous runtime on the client's side to parallelize requests. As network requests are heavily IO-bound, this speeds it up significantly over sending requests to Thetacrypt servers in sequence. However, it also implies the client is unaware when a decryption request arrived at any server, as the connection will be kept open until the protocol has terminated.

Having queued up all decryption requests for sending the client will begin to wait until it has received a decryption response from all Thetacrypt servers. Whenever it receives one, it emits an event allowing to keep track at what time decryptions were returned.

Each of these measurements will be repeated a total of 10 times to accommodate for variance. Having done so it will move on to the next message size and, once all of those have been benchmarked, on to the next combination of scheme and group.

Centralized client versus one per Thetacrypt server In the chosen setup there exists only a single client sending decryption requests to all servers of the Thetacrypt deployment. This vastly simplifies the logic on the client side, as no synchronisation of clients is required, but it comes with certain limitations.

As long as ciphertexts are reasonably small this matters little. Sending 100 decryption requests for a $1\,\mathrm{kB}$ ciphertexts using a $1\,\mathrm{Gbit}$ network link will take $0.8\,\mathrm{ms}$. As the network latency to any of the Thetacrypt servers will be far larger than this, it will be no different than if there had been one client per server, all of which had triggered the decryption request at the same time. However, with larger ciphertexts the bandwidth available to the single client will suddenly become a bottleneck. Sending 100 $16\,\mathrm{MB}$ ciphertexts over a $1\,\mathrm{Gbit}$ network link will already take $8\,\mathrm{s}$. At this point some Thetacrypt servers may receive their decryption request noticeably later than others, artificially increasing the decryption times.

In practice, this means that, for large message sizes and Thetacrypt deployments, the chosen setup is not suitable. Anything less than a few dozen kilobytes should be unaffected, however. Indeed no adverse effect could be observed for messages of $1\,\mathrm{MB}$ and below, as will be seen when results are discussed.

Estimating decryption latency vs decryption throughput In this benchmark only one threshold decryption is performed at a time. This corresponds to a best-case scenario, where every Thetacrypt server can dedicate all its resources toward one request. This setup does thus not allow to reliably estimate the decryption throughput — that is the number of decryptions a Thetacrypt can handle. It is however

Event	Event trigger
ProfilingParameters	On client startup, contains parameters of experiment such as n, f, ...
StartSendingDecryptionRequests	Starting to send decryption requests
SentDecryptionRequest	Queueing up a single decryption request for sending.
FinishedSendingDecryptionRequest	Having finished sending decryption requests.
ReceivedPublicKey	Receiving a public key from a threshold node.
ReceivedSuccessfulDecryption	Receiving a decryption result from a threshold node. Count indicates how many equal plaintexts were received.
ReceivedFailedDecryption	Receiving a failed decryption result from a threshold node.
DecryptionTimedOut	Not receiving a response in time from a threshold node.
ReceivedSufficientDecryptions	Having received $f + 1$ equal decryptions
ReceivedAllDecryptions	Having received n decryptions

Table 3.6: Client events

sufficient to estimate decryption latency — that is the time which elapses between starting to decrypt a ciphertext, and having finished doing so.

Two types of decryption latency are differentiated. Client-sided decryption latency measures the time between when the client starts sending out a ciphertext, and when it has received the corresponding plaintext. Server-sided decryption latency measures the time between when a Thetacrypt server has fully received the ciphertext, and when it has finished decrypting it. As such server-sided decryption latency does not include any time spent transferring plain- and ciphertext between the client and server.

3.2.5 Client and server events

At key points of the process, such as when it has received a decryption result, the client will emit various types of events. Each event will at the very least contain a timestamp indicating when it was emitted, with most events also containing additional metadata such as the Thetacrypt node from which a decryption result was received. Table 3.6 lists all events emitted by the client.

Observing the threshold decryption process from the point of view of only the client comes with some limitations. From the moment the client sends a decryption request to the moment it receives a decryption response it is unaware of what the Thetacrypt nodes spend time on. To fill in this gap, Thetacrypt was also made to emit server-sided events during the process. These events are shown in Table 3.7. Similarly to the client events, these contain metadata to allow correlation of server-sided events with client-issued decryption requests. The choice of how many events to emit is a tradeoff between keeping analysis simple while still capturing all parts where a significant amount of time is spent. The current choice of events has emerged iteratively based on observations. It provides observability into all the high-level steps of the threshold cipher as well as into some implementation details of Thetacrypt's protocol layer which turned out to be relevant for performance.

Correlating events Having events emitted on both the client- as well as the server-side requires being able to correlate those. As an example each time the client sends a decryption request to a specific

Event	Event trigger
ReceivedDecryptionRequest	Receiving a decryption request
StartDeserializingDecryptionRequest, *FinishedDeserializingDecryptionRequest*	Before and after deserializing decryption request
StartAssigningInstanceID, *FinishedAssigningInstanceID*	Before and after calculating instance ID in protocol layer
StartKeypairLookup, *FinishedKeypairLookup*	Before and after looking up keypair to use.
StartThresholdDecryption	Starting with steps of threshold decryption.
StartCiphertextValidation, *FinishedCiphertextValidation*	Before and after validating ciphertext.
StartPartialDecryption, *FinishedPartialDecryption*	Before and after generating decryption share.
ReceivedDecryptionShare	Having received and validated a decryption share from a threshold node.
StartAssemblingShares, *FinishedAssemblingShares*	Before and after assembling decryption shares into plaintext.
StartProtocolShutdown, *FinishedProtocolShutdown*	Before and after shutting down protocol.
FinishedThresholdDecryption	Having finished with steps of threshold decryption.
FinishedStateCleanup	Having finished with cleanup and bookkeeping on protocol layer.
ReturningDecryptionResponse	Immediately before returning decryption result to client.

Table 3.7: Server events

threshold node, there will be one *SentDecryptionRequest* event on the client-side, and a corresponding *ReceivedDecryptionRequest* event in the output of the specific threshold node. As the client will send multiple decryption requests to any single node — for various combinations of parameters and iterations — the naïve approach of a 1:1 mapping does not work.

This problem can be solved by utilizing the 'label' field common to threshold ciphers. It is a field of metadata, attached to the ciphertext, which the client can choose at will. The client picks a unique identifier for each decryption request, which the server will attach to the metadata of all events it emits when handling this request.

3.2.6 From events to durations

While events form the basis of evaluating performance, what is of particular interest is the duration of each of the steps. Some of these durations are within one server, e.g. the duration between the server starting and finishing deserialization of the decryption request. Others cross network boundaries, such as the duration between the client starting to send, and the server having received, a decryption request.

Limitations of the system clock on Linux Ideally all involved systems would have access to clocks which are perfectly synchronized, monotonic, and of constant speed. In a real-life deployment, this is hard to achieve. When using cloud-based virtual machines, as in this case, one has to work with Linux's system clock synchronized by means of NTP. NTP is able to provide sub-millisecond accuracy in local deployments and accuracy of a few milliseconds in global deployments [Net]. This does not allow one to directly compare timestamps between different threshold nodes, as many of the involved operations (even across network boundaries) will take less than a few milliseconds. One example was trying to estimate the time it took for a decryption request to travel from the client to the server. The estimate was as likely to produce plausible latencies as it was to produce latencies of less than 0 seconds.

Another theoretical limitation of a system clock synchronized by NTP is that there is neither a guarantee that all clock ticks are of equal length, nor that time moves monotonically. When the reference and system clocks differ, NTP will choose to either *step* or *slew* the system clock [Net]. If the difference is large it will step the system clock, changing it to the new value at once — including potentially backwards in time. If the difference is small it will slew it, changing the duration of the system clock's ticks until the system and reference clock are synchronized.

In practice, this did not turn out to be an observable limitation. Starting NTP well before conducting the measurements allows it to perform its corrections ahead of time. Assuming the system clock to have a reasonably low drift this then prevents any noticeable stepping or slewing while benchmarks are running.

Calculating durations In summary, this means that system clocks are not sufficiently synchronized, but are sufficiently monotonic and constant-speed. This prevents comparing timestamps across server boundaries, but it does allow comparing timestamps within one server. Thus one can calculate the duration it takes to handle the full request within one server — from having received the decryption request to sending the decryption result — as well as the duration of each intermediate step such as deserialization, validation of ciphertexts, etc.

Estimating network transmission time Now, network transmission time can be estimated as the difference between how long the duration took from the point of view of the client, and how long it took from the point of view of the server. The first can be calculated as the time between the client sending a decryption request and receiving a decryption response. The second can be calculated as the time between the server having received a decryption request and sending a decryption response. Figure 3.3 illustrates this process for a single client-server interaction.

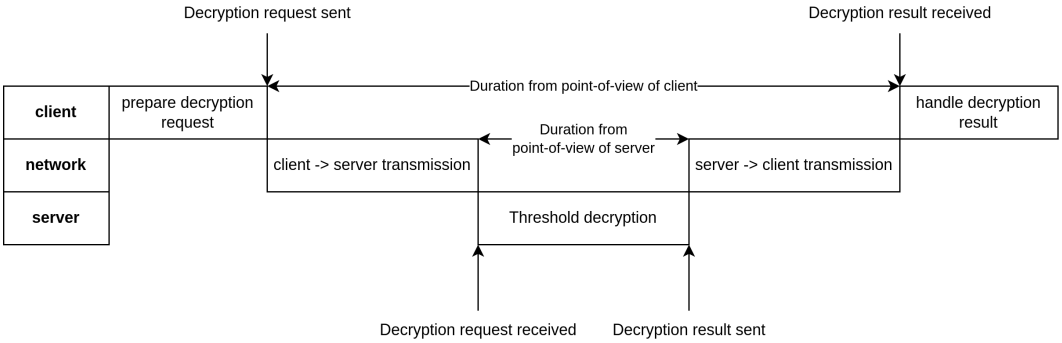


Figure 3.3: Estimating network latency

One obvious downside of this approximation is that it only provides an estimate of the sum of both transmission times, rather than separate ones for either direction. Given that the plaintext and ciphertext are of comparable size it is plausible that this is a good enough estimate, however. Since the time spent on network communication was not the focus of this work, no further effort was made to estimate the two durations any more accurately.

3.3 Automating Thetacrypt deployment

The macrobenchmark requires access to a deployment of Thetacrypt which it can query. This poses a few challenges, as some parameters of the deployment — such as the number of total nodes and their distribution — are parameters one wants to vary as part of the benchmark. This requires the ability to quickly provision and tear down a deployment of Thetacrypt. To achieve this, the full process of provisioning a Thetacrypt deployment and benchmarking it was automated. On a high level, it consists of five steps which will be discussed in more detail:

Compilation & packaging In the first step the Thetacrypt application must be compiled, and packaged into a format which allows easy installation and execution on servers

Infrastructure automation Infrastructure such as servers and DNS records must be provisioned

Configuration management The provisioned systems must be configured such that benchmarks can be ran on there.

Benchmarking Thetacrypt must be started, the benchmark run, and the resulting data collected.

Cleanup The provisioned infrastructure must be torn down.

While microbenchmarks were done on internal hardware of the university, macrobenchmarks require the ability to dynamically provision hundreds of servers, and tear them down again when they are no longer needed. For this reason, macrobenchmarks took place on DigitalOcean, a cloud provider offering infrastructure services across the globe. With its hourly pricing, it was feasible to run large-scale experiments without incurring the heavy cost of having a large number of machines running constantly.

Distribution	Thetacrypt servers	Benchmarking client
Regional	Frankfurt 1	Frankfurt 1
Global	New York 1, New York 3, Singapore 1, London 1, Amsterdam 3, Frankfurt 1, Toronto 1, San Francisco 3, Sydney 1	Frankfurt 1

Table 3.8: Choice of DigitalOcean data centers for macrobenchmark

Choice of regions on DigitalOcean DigitalOcean offers the ability to provision infrastructure in 14 data centers across 9 geographical regions. Not all of these data centers still had capacity though, so some had to be excluded. Further, the Bangalore region turned out to be unusable as it was impossible to establish reliable network connections from servers hosted there to any other server. Table 3.8 shows which data centers were chosen to host the Thetacrypt server and benchmarking client for both the regional and global deployments.

Size of servers Another choice to be made is the size of virtual machines to use. Cloud providers such as DigitalOcean offer different size classes, usually differentiated by the amount of RAM and the number of CPU cores available to the machine. Cost tends to be linear in both the amount of RAM as well as CPU cores, so there is an incentive to stay with servers as small as one can get away with without compromising benchmark results.

The proposed benchmarking mechanism did not involve more than one concurrent decryption, and Thetacrypt's gossipsub networking layer was shown to require very little in terms of computational resources. Thus it was deemed sufficient to provision Thetacrypt servers with a single CPU. In terms of RAM, each machine was assigned $2\,\mathrm{GB}$ of memory, which was sufficient for operation.

The single exception was the one server on which both the benchmarking client as well as the monitoring server (see Section 3.3 for more details on that) was running. As it had to perform more work — some of which was also concurrent — it was provisioned with two CPUs and $16\,\mathrm{GB}$ of RAM. This choice of deployment ensured that the hourly cost of even large deployments with $n = 100$ servers was well below two USD.

The provisioned virtual machines were running on hypervisors using AMD server-grade CPUs, with single-core frequencies of around $2\,\mathrm{GHz}$.

Compilation & packaging Theoretically it would be feasible to compile the benchmarking client and Thetacrypt server locally and copy them onto all involved servers, as Rust supports cross-compilation and produces mostly statically-linked binaries. However, it does link dynamically to glibc, so care would have to be taken to have binaries be compatible.

Rather than handle this, the application is compiled in and distributed as a Docker[4] container. This ensures that the compilation and execution environment are well-defined. This also allows to easily automate the process as part of a CI job on Gitlab. Whenever a commit is pushed to the institute's self-hosted Gitlab instance, a worker process will compile the application, run tests, package the application into a Docker container and publish the container to Gitlab's built-in container registry.

Each server can then simply pull the container from Gitlab's container registry and start it.

[4] `https://www.docker.com/`

Infrastructure automation To automate provisioning of the infrastructure, Terraform [5] was used. In the first stage, one can specify the desired infrastructure — such as servers, DNS records, and databases — as a set of YAML documents. This specification can be parameterized, to easily support changing e.g. the number of servers without needing to change the bulk of the specification. In the second stage, Terraform will then execute a sequence of commands against the cloud provider's API, provisioning the specified infrastructure. This allows one to specify the desired infrastructure once, and then easily provision it anew whenever benchmarks are to be run.

Configuration management While Terraform ensures that servers and DNS records are present, involved systems must still be configured. This includes installation of required software such as Docker, creation of configuration files for each Thetacrypt node, or the opening of required ports in the server's software firewall. This functionality is provided by so-called configuration-management tools, one of which is Ansible[6]. As with the automation of infrastructure one first specifies the desired configuration of a system as a set of YAML documents. Then, Ansible connects to every managed system per SSH, issuing a sequence of commands to configure them as described. This ensures that any amount of servers can be configured quickly and identically, which not only helps keep the setup effort low but also ensures that experiments are repeatable.

Benchmarking At this point all required infrastructure has been provisioned and configured appropriately. As a next step, the benchmark has to be run. This involves first starting the Thetacrypt server binary and the benchmarking client. Once the benchmark has been completed the benchmarking events produced by both the benchmarking client as well as every Thetacrypt server have to be collected and aggregated. This process was automated with Ansible where applicable, and with basic shell scripts otherwise.

Cleanup Cleanup of provisioned infrastructure is supported by Terraform out of the box. This allows to quickly and efficiently shut down all servers and other resources once the benchmark has been completed, ensuring that there is no unused infrastructure accumulating cost.

System monitoring To monitor system usage metrics, such as CPU and memory usage, a standard monitoring stack was deployed. An instance of node_exporter [7] was run on every virtual machine, which collects various system metrics. These metrics were then collected and aggregated by Prometheus [8] running on a virtual machine dedicated to monitoring. Finally, Grafana [9] was used as a dashboard solution to visualize collected data.

This data allows gaining a high-level insight into the load the benchmark generated. It allows one to ensure that benchmark results are not limited by any hardware component being at or above capacity.

3.4 Benchmarking libp2p's gossipsub network

To ensure that the bottleneck of the Thetacrypt macrobenchmark was not the networking layer, its performance was also evaluated in isolation. To do this a basic ping server was written. It consists of a Rust application which utilizes Thetacrypt's wrapper around libp2p's gossipsup network. The ping server can be started in either *emit* or *listen* mode. In emit mode it will emit a sequence of PING messages via the

[5] https://www.terraform.io/
[6] https://www.ansible.com/
[7] https://github.com/prometheus/node_exporter
[8] https://prometheus.io/
[9] https://grafana.com/

PING
ID *id*
Payload

PONG
ID *id*
Sender *s*
Payload

Figure 3.4: Structure of PING and PONG messages

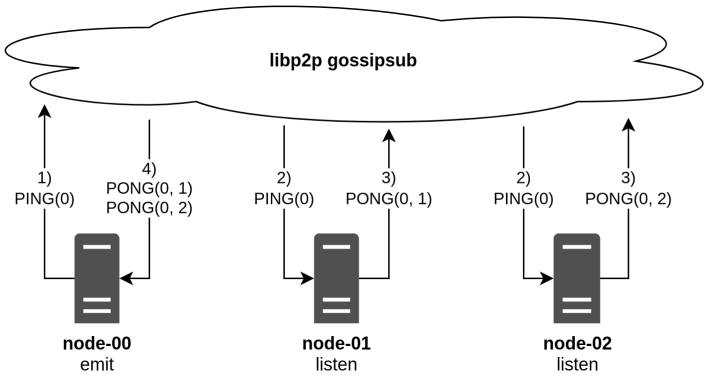


Figure 3.5: Benchmarking of libp2p Gossipsub network

gossipsub network. In 'listen' mode it will listen for PING messages from the network. On receiving one, it will respond with a PONG message.

The structure of both types of messages is shown in Figure 3.4. A PING message contains a unique identifier id which, for the current implementation, is simply incremented for every new message. It also supports an arbitrary payload which defaults to zero bytes. A PONG message will contain the identifier id of the PING message to which it is an answer to, a unique identifier of the node s which sent the PONG message, and a copy of the PING message's payload if applicable.

This allows for a deployment of the ping servers as shown in Figure 3.5. They can be deployed the same way Thetacrypt is, for varying choices of the number of parties n and regional distributions of involved servers. One server is started in emitting mode, all others in listening mode.

The emitting server will then emit a sequence of PING messages, and keep track if, and after how much time, it receives the corresponding PONG message from each node in the network. This data is logged and can then be analyzed.

Comparison with pure network latency To be able to estimate the overhead — if any — imposed by the gossipsub network, pure network latencies between servers are also estimated. This is done by pinging each server repeatedly using the ICMP protocol, and recording the latencies.

4 Results

This chapter will present acquired results. They will be discussed in mostly chronological order, as the optimization process was heavily iterative. Results will often be presented as an initial observation based on benchmark results, followed by a description of how the root cause was found and fixed, and then a second benchmark to verify the effectiveness of the fix.

The first results which will be discussed are microbenchmarks of each separate step of the threshold ciphers. Then the focus will shift towards macrobenchmarks of Thetacrypt in various types of deployments. Near the end data from the two types of benchmarks is combined, to serve as a plausibility check of achieved results. Lastly, there is a brief attempt to estimate the throughput of decryptions which might be achievable with Thetacrypt.

4.1 Thetacrypt microbenchmarks

4.1.1 Performance of supported curves

In all obtained results there was a noticeable difference in the performance between the supported curves. Of the three, Bls12381 was the slowest, followed by Bn254, followed by Ed25519. The exact impact it had on specific results depends on how much of the time was spent on group operations, but differences of up to a factor of two between the slowest and the fastest curve can be observed in some cases. No attempt was made to benchmark group operations in isolation. Nonetheless this immediately provides the SG02 scheme with an advantage, as it can use the more-performant Ed25519 curve, whereas the BZ03 scheme has to stick to Bn254.

It is unclear if these performance differences are inherent to the curves themselves, are due to their implementation in MIRACL Core[1] — the library used by Thetacrypt to provide cryptographic operations over elliptical curves — or are due to Thetacrypt's wrapper around MIRACL Core.

[1] `https://github.com/miracl/core`

4.1.2 Key generation

Performance of Thetacrypt's key generation was evaluated briefly. As Thetacrypt does not yet have any support for distributed key generation, existing routines require a trusted party to generate and distribute keys.

Key generation was found not to be significant in terms of performance at all. In a $(f + 1)$-out-of-n system it was found to be linear in both $f + 1$ as well as n, although with such small coefficients that keys for huge deployments can be generated within a few milliseconds. As an example for $f = 1000$, $n = 3001$, key generation took approximately $4\,\mathrm{ms}$. No significant difference was observed between the two schemes.

One caveat is that memory usage of the key generation procedure is currently quadratic in n, with e.g. $n = 3000$ consuming around $7\,\mathrm{GB}$ of heap memory. This is due to the size of the threshold ciphers' verification keys being linear in n, and Thetacrypt keeping n individual copies of the verification key in memory. This could easily be fixed but was deemed to be a low priority as the intention is to eventually support DKG protocols.

4.1.3 Encryption and ciphertext validation

The performance of the encryption operation is expected to be linearly dependent on the size of the message due to the underlying hybrid cryptosystem. It is further expected to vary — by a constant amount — based on the choice of scheme and group. The result shown in Figure 4.1 confirms this. For small messages, the constant-time overhead of the threshold cipher dominates, while for larger messages the execution time of the hybrid cryptosystem takes over. Results also show the previously-mentioned performance of the three available groups, which can lead to a sizeable reduction of the time the encryption operation takes from e.g. $2\,\mathrm{ms}$ to $0.7\,\mathrm{ms}$

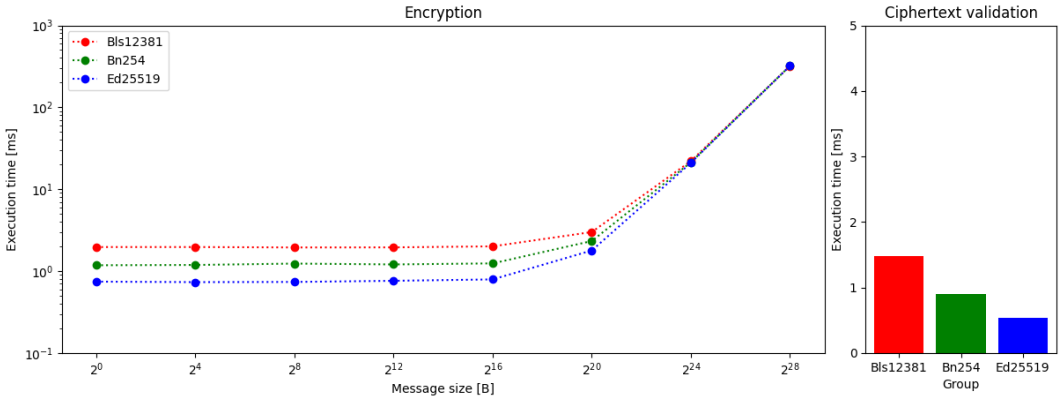
Of interest is also the execution time of the ciphertext validation step, also shown in Figure 4.1. There is a significant difference between the two schemes, with ciphertext validation of BZ03 being roughly three times as expensive as ciphertext validation of SG02. While ciphertext validation has to only be performed once, by each participating node, during the threshold decryption process, a difference of a few milliseconds of CPU time can still add up when many decryption requests are handled at once. Earlier results also showed that ciphertext validation of the BZ03 took longer the larger the message was. This turned out to be a bug in Thetacrypt, which was subsequently fixed.

4.1.4 Decryption and decryption share validation

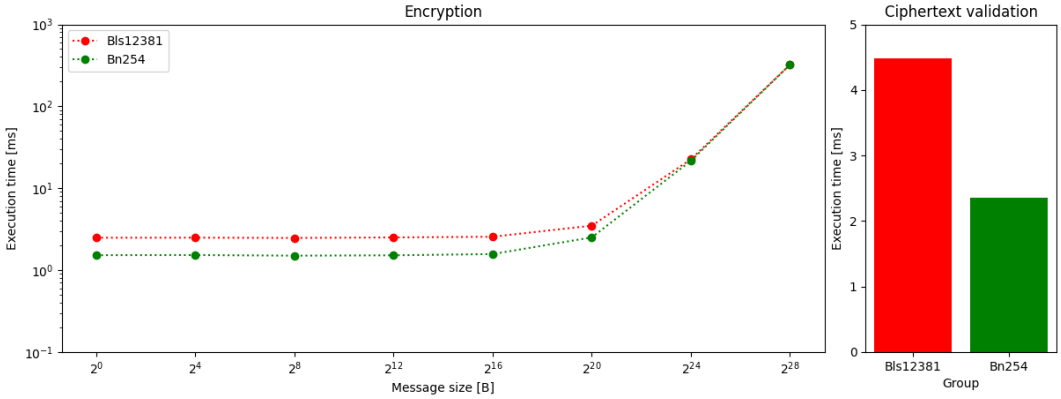
Figure 4.2 shows the performance of the partial decryption, decryption share validation, and share assembly steps of the two threshold ciphers. For all steps the usual performance difference of the available groups can be seen.

When it comes to partial decryption, a small difference can be seen between the two schemes. Interestingly BZ03 actually outperforms SG02 when they both use the same group. However, this small difference is more than compensated for by SG02's ability to use the more performant Ed25519 curve. In their most performant configuration, partial decryption takes less than $1\,\mathrm{ms}$ for either scheme. Given this step only happens once during threshold decryption this is unlikely to be the limiting factor in real-world usage.

Validation of decryption shares is more interesting. While a single execution thereof is a constant-time operation, independent of any input parameters, it has to be performed once for every incoming share. When one thus requires $F + 1$ shares to assemble the plaintext, ciphertext validation will have to be performed $F + 1$ times. In this step, the two schemes differ hugely — while SG02 can perform it in approximately $0.5\,\mathrm{ms}$, BZ03 will take more than $2\,\mathrm{ms}$. Assuming a reasonably-sized deployment with $N = 100, F = 33$, this will amount to a sum of time spent validating decryption shares of $17\,\mathrm{ms}$ for

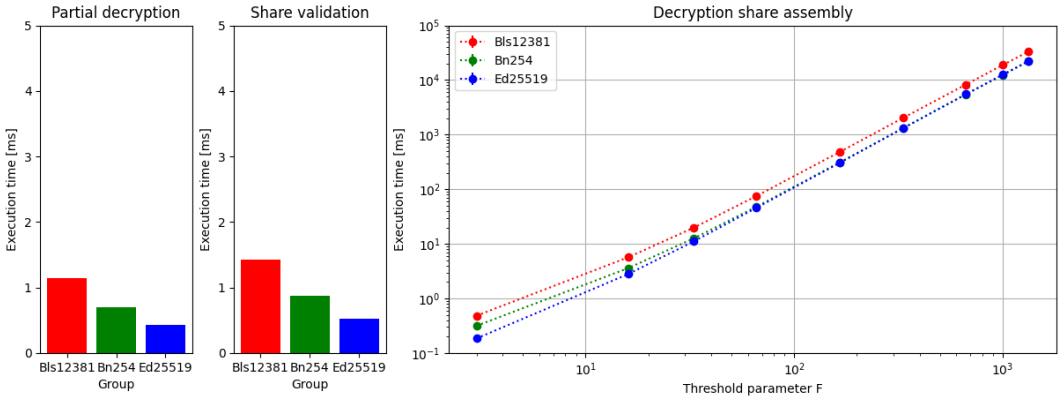


(a) SG02 threshold cipher

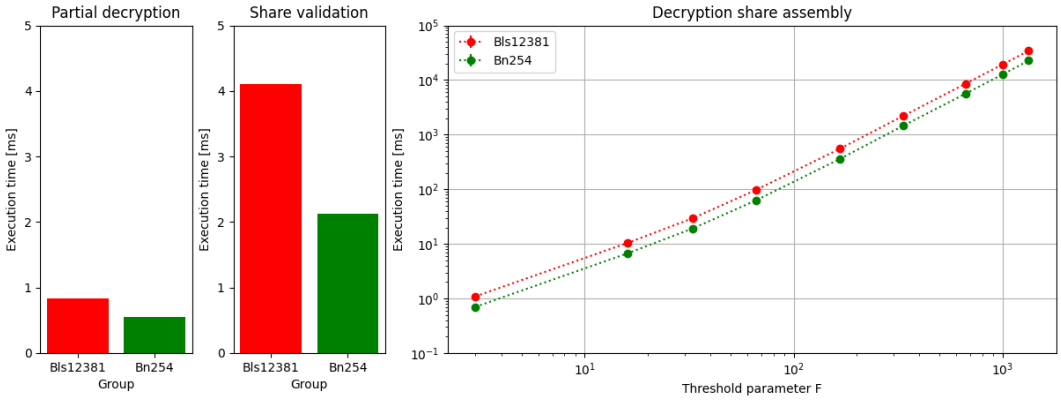


(b) BZ03 threshold cipher

Figure 4.1: Execution time of encryption and ciphertext validation steps



(a) SG02 threshold cipher



(b) BZ03 threshold cipher

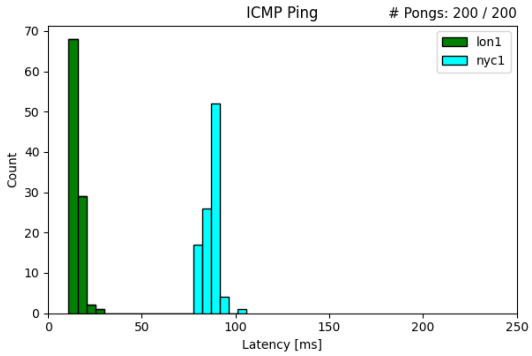
Figure 4.2: Execution time of partial decryption, decryption share validation, and share assembly steps

SG02 versus 68 ms for BZ03 for each decryption. This is likely to be a significant bottleneck when it comes both to decryption latency as well as throughput in larger deployments.

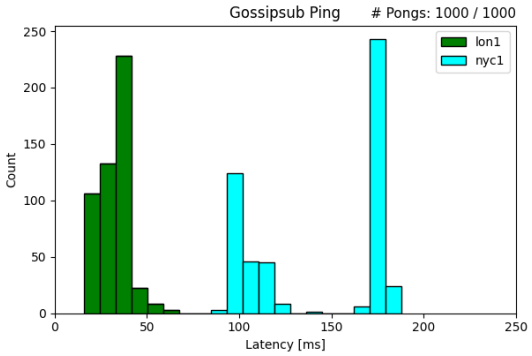
The performance of share assembly lastly is, as expected, linear in the threshold parameter F. This is mostly due to the reconstruction of the Shamir-secret-sharing-shared value. There is no noticeable difference between the two schemes, as the steps they perform to reconstruct the secret are virtually identical. There is a second dependence of decryption share assembly on the size of the message — not shown in any plot — as, during this step, Thetacrypt also performs the symmetric decryption operation of the hybrid cryptosystem. The utilized symmetric cipher will perform on the order of a few hundred MB per second, depending on the hardware, which will have to be added on top.

4.1.5 Differences in output size

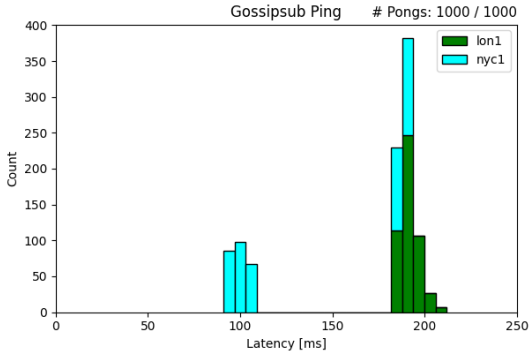
There are very few differences in the output size of operations. The size of the ciphertext, irrespective of scheme, is dominated by the size of the plaintext. The size of the private key depends on the group only, and is between 50 B and 70 B. The size of the public key is linear in n, between 700 B and 1 kB per n depending on the group. The size of a decryption share depends on the group and scheme and is between 160 B and 240 B.



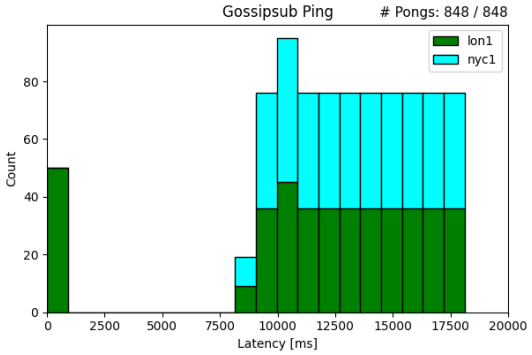
(a) ICMP latencies



(b) Error-free execution of gossipsub latency measurement



(c) Significant latency overhead due to gossipsub network



(d) Complete breakdown of gossipsub network

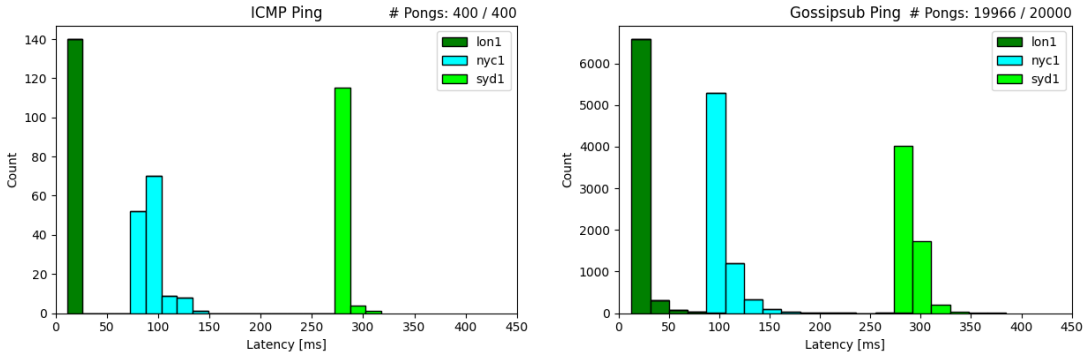
Figure 4.3: Evaluation of initial gossipsub implementation by measuring round-trip latencies between servers of a Thetacrypt deployment. Latencies are measured between a dedicated (emitting) server and all other (listening) servers. Figure a) shows a histogram of ICMP latencies, Figures b) through d) latencies via the gossipsub network. Overlapping data is stacked.

4.2 Performance of Thetacrypt gossipsub networking

Before doing proper macrobenchmarks, Thetacrypt's network layer was evaluated as described in Section 3.4. The first measurement was done on a network of $n = 21$ servers. The emitting one was located in Frankfurt, the remaining 20 were split evenly between New York and London. Results are visualized in Figure 4.3. Figure 4.3a presents the ICMP latencies which are as expected, with latencies from Frankfurt to London around $20\,\mathrm{ms}$ and latencies from Frankfurt to New York around $80\,\mathrm{ms}$. The other three plots show three independent measurements of gossipsub latency, where the Frankfurt node sent out 50 ping messages each. Figure 4.3b shows a scenario where the behaviour of the network was mostly as expected, with the gossipsub layer adding little additional latency on top. Figure 4.3c shows unexpected behaviour where servers in the London region took around twice as long to respond as those in the New York region, in contrast to their network latencies as established by ICMP. Figure 4.3d finally shows a complete network breakdown where latencies to all servers skyrocketed, hitting tens of seconds each.

Further investigation found that how Thetacrypt initialized the Gossipsub network was improper. Rather than initializing a mesh network, each node came up with only a single connection to a random other node. While this was usually sufficient to establish a fully connected network, it was prone to establish a network topology unsuitable for low-latency delivery of messages.

Having fixed Thetacrypt's network layer, the benchmark was run a second time on a deployment



(a) ICMP latencies

(b) Gossipsub latencies

Figure 4.4: Evaluation of gossipsub implementation after having fixed establishment of mesh network. Figure a) shows the histogram of ICMP latencies, Figure b) of latencies via the gossipsub network.

of $n = 21$ servers. This time the emitting server was in Frankfurt, with the remaining 20 being split across the London, New York, and Sydney data centers. Results of this are shown in Figure 4.4. Figure 4.4a shows ICMP latencies which again are as expected. Figure 4.4b shows gossipsub latencies which confirm that, with the fix in place, the gossipsub layer adds little observable overhead. These results were reproducible with different independent deployments.

4.3 Thetacrypt message backlogging mechanism

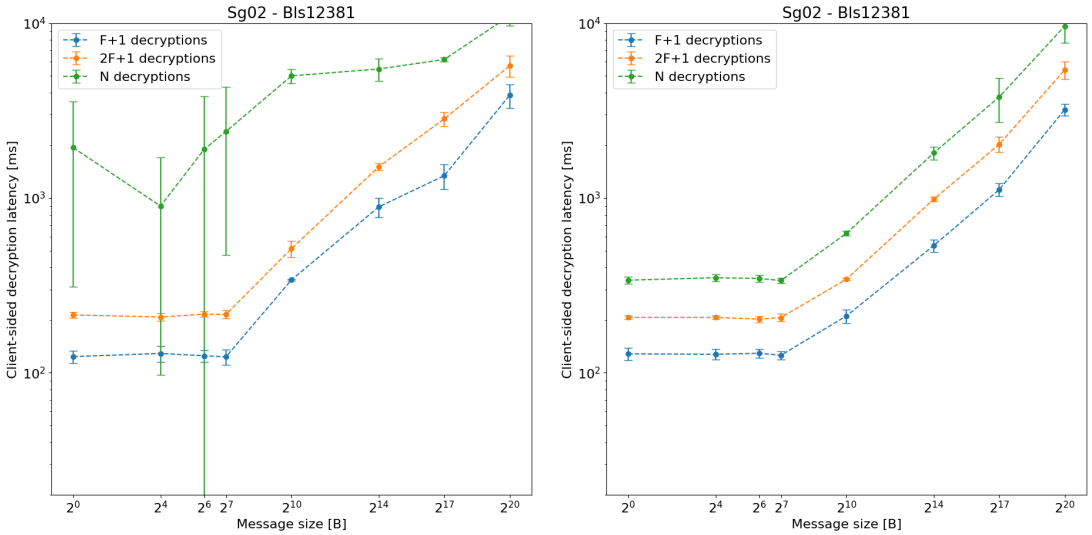
Having ensured that the gossipsub layer was reliable, the next step was to perform a macrobenchmark of Thetacrypt's threshold ciphers. Figure 4.5a shows the results of the very first benchmark, using a global deployment of $N = 21$ servers.

Ignoring the interesting pattern of decryption time versus message size for now, the most striking observation to be made in Figure 4.5a is the huge variance of when the last decryption arrived at the client. The choice of using the SG02 scheme with the Bls12381 curve is arbitrary, similar results were seen for other combinations too.

This implies that some of the threshold nodes, on occasion, took much longer to respond than they normally did. One typical explanation of this would be high load, where some requests end up queued behind many others. However this was ruled out as firstly there was only one active decryption request at a time by design, and secondly as system monitoring tools confirmed that server load was low throughout.

The cause was found by evaluating Thetacrypt's log files. As described in Section 2.4.2, Thetacrypt handles concurrent decryption sessions by forwarding incoming messages to the appropriate protocol instance. However, these protocol instances are only instantiated when a Thetacrypt server receives a decryption request. Thus it may receive another server's decryption shares before it has received the client's decryption request. When this happened it put those decryption shares into a queue and attempted to forward them to the appropriate protocol instance once more after some time had elapsed. This caused the decryption response to be delayed, explaining the observed behaviour.

The fix then was to improve the handling of backlogged messages. Whenever a new protocol instance is spawned, Thetacrypt now immediately checks its backlog for messages belonging to that protocol instance. If there are any they are processed right away. To verify the fix, a second — otherwise identical — benchmark was run, shown in Figure 4.5b. With the fix in place, the variance was much smaller, indicating that this issue was fixed.



(a) Before changes to backlogging mechanism

(b) After changes to backlogging mechanism

Figure 4.5: Client decryption latencies for a global deployment of $N = 21$ servers. The data points indicate the time it took for the client to receive $F + 1$, $2F + 1$ and N decryptions respectively. Each data point is the mean of 20 measurements. Error whiskers extend one sample standard deviation in each direction.

The time it takes for N decryptions to arrive will not be shown anymore in subsequent plots, as it is of little interest to a client.

4.4 Ciphertext (de)serialization performance

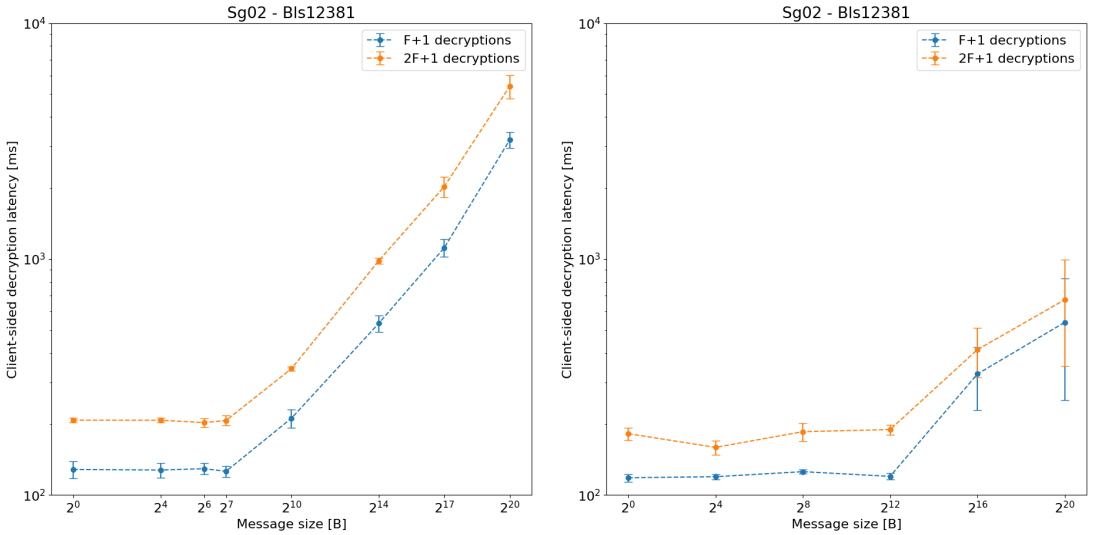
The next goal was to find out more about the cause of the seemingly linear relation between decryption latency and message size shown in Figure 4.6a. While a linear relation is expected to surface asymptotically, due to the performance of components such as the underlying hybrid cryptosystem, it seemed implausible that it would already dominate execution time for messages as small as 1 KiB.

From log output of the client, it became apparent that quite some time was spent on the serialization of decryption requests before sending them out via the network. To further investigate this, a microbenchmark of the serialization and deserialization functionality was performed, shown in Figure 4.7. It compares performance of the previously used library *rasn*[2] with *asn1*[3], which it was replaced with.

Both libraries show the same behaviour where the (de)serialization rate first increases with the size of the object being (de)serialized. This is expected as, with larger objects, more time is spent on actual (de)serialization rather than on constant-time overhead. The two libraries differ significantly, however, in that *rasn* achieves a throughput of at most $2\,\mathrm{MB/s}$, whereas *asn1* achieves a throughput of up to $1\,\mathrm{GB/s}$ — a full three orders of magnitude faster. While there is a small difference between the speed of serialization and deserialization this is of little practical significance.

Having replaced *rasn* with *asn1* in Thetacrypt, the macrobenchmark was run again with results shown in Figure 4.6b. The improvement it had on decryption latencies is apparent. As an example, it took around

[2] `https://docs.rs/rasn/latest/rasn/`
[3] `https://docs.rs/asn1/latest/asn1/`



(a) With *rasn* library

(b) With *asn1* library

Figure 4.6: Impact of changes to serialization on decryption latencies. Plot details are as in Figure 4.5

$7000\,\mathrm{ms}$ to receive $2F + 1$ decryptions of a $1\,\mathrm{MB}$ ciphertext before, and only around $700\,\mathrm{ms}$ after — a speedup of an order of magnitude for large messages.

4.5 Verbose logging of plaintexts

Checking the log output of the Thetacrypt server binary during a benchmark showcased another issue. Upon a successful assembly of decryption shares, the server logged the hex-encoded plaintext to stdout which, by default, will end up in a terminal device. Not only is this undesired from an operational point of view, but it is also comparably slow. Writing multiple megabytes of string data to a mostly unbuffered device such as a terminal will quickly take seconds to complete. Having fully removed the printing of plaintext, the benchmark was run again with results shown in Figure 4.8a for performance before it was removed, and in Figure 4.8b after it was removed. Unlike with previous improvements, the speedup is not immediately visible, partially due to the inherent noise of these measurements, partially due to the logarithmic scale. However, for e.g. $16\,\mathrm{MB}$ messages the average time has decreased from about $8000\,\mathrm{ms}$ to $5000\,\mathrm{ms}$.

4.6 Calculation of protocol instance IDs

At this point relying on client-sided durations and log output to detect what Thetacrypt spent its time on became insufficient. To counteract this, the Thetacrypt server was made to emit profiling events as well. These allow one to analyze in depth how much time each step of the decryption process costs. Figure 4.9a shows an example of how this data can be visualized.

Importantly these plots no longer include network transmission time between benchmarking client and Thetacrypt servers. Instead, they show the execution time of a threshold decryption from the point of view of the Thetacrypt server. This is beneficial for analysis as the performance of the network between a Thetacrypt client and server is not under the control of Thetacrypt anyway.

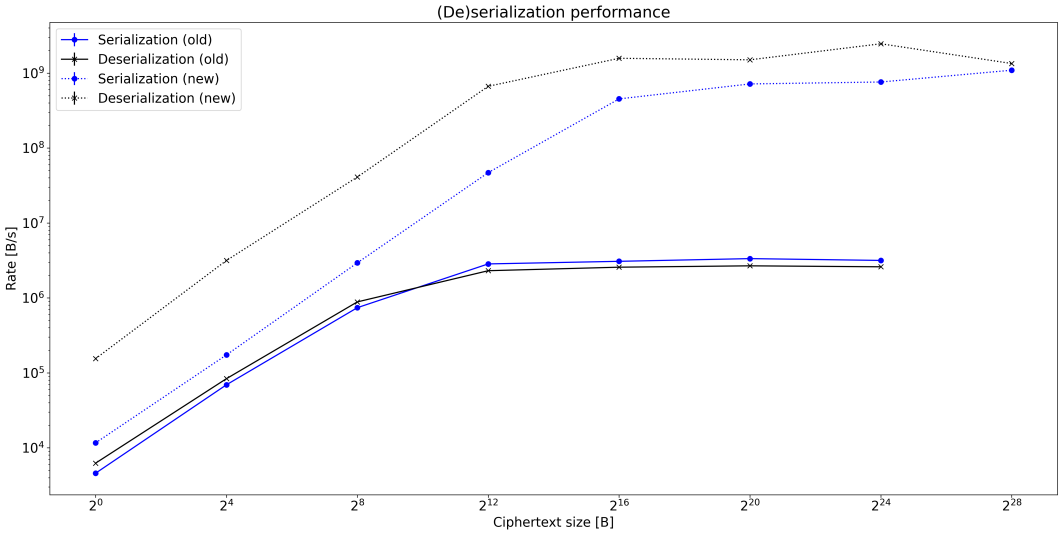
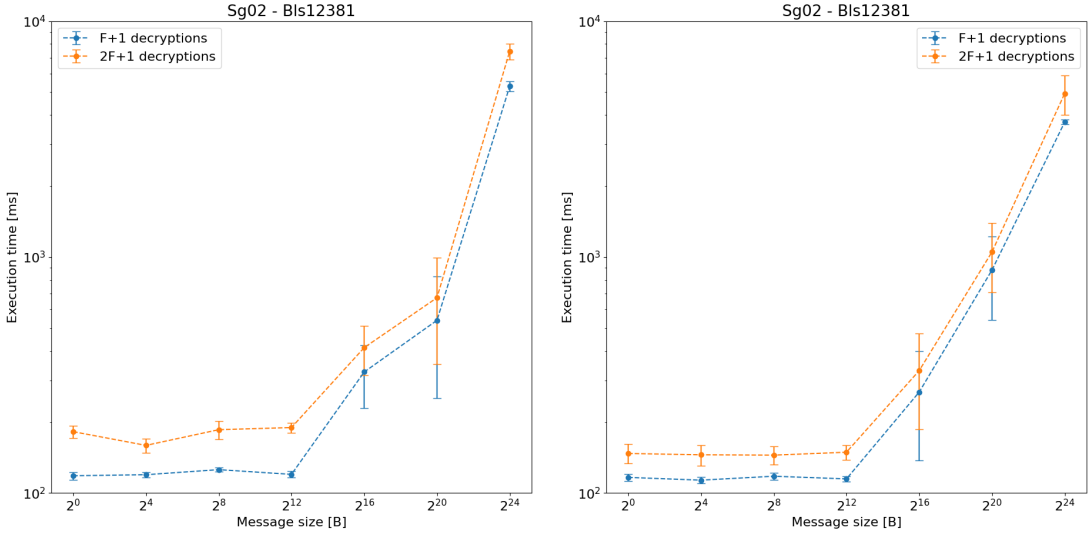


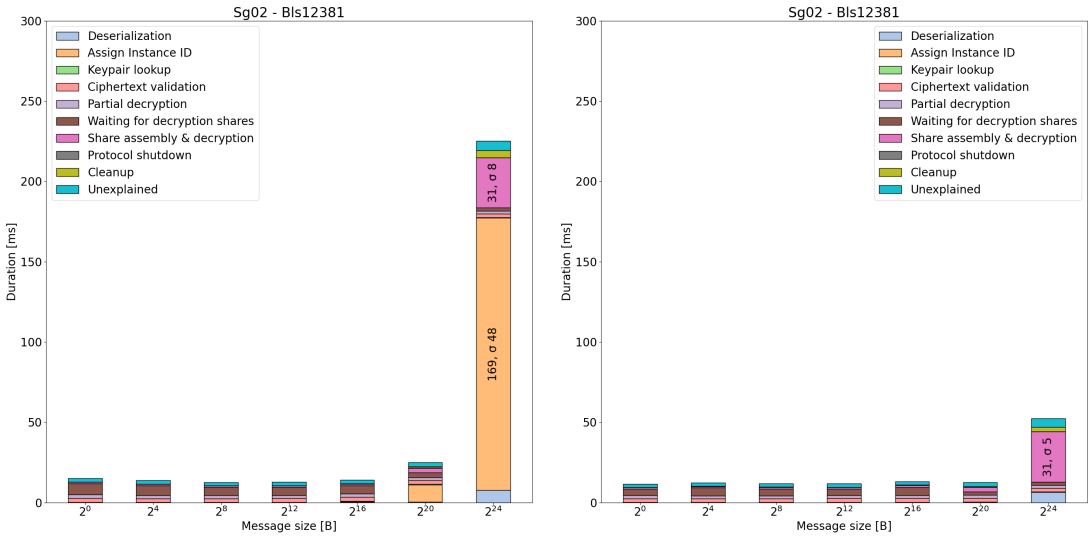
Figure 4.7: Comparison of (de)serialization rates of two ASN.1 libraries in relation to the size of the ciphertext to be (de)serialized. The full lines show performance of the previously-used ASN.1 library, the dotted of the newly-used one.



(a) With printing to terminal

(b) Without printing to terminal

Figure 4.8: Impact of printing plaintext to terminal on decryption latencies. Plot details are as in Figure 4.5



(a) Calculating instance ID by hashing ciphertext (b) Calculating instance ID by relying on ciphertext label

Figure 4.9: Server-sided decryption latencies for a global deployment of $N = 7$ servers. The components of each stacked bar show the time the Thetacrypt servers spent on separate parts of the threshold decryption process. Each component is the median of all measurements. The order of components, top-to-bottom on the legend, bottom-to-top in the bar chart, correspond to the order of steps during the decryption process. The significant components are annotated with their value and standard deviation.

From Figure 4.9a it is apparent that the server, for large enough messages, spends a significant amount of time on assigning the protocol instance ID. This instance ID is required to keep track of which protocol message belongs to which instantiation of the threshold decryption protocol, as was described in Section 2.4.2.

Investigating how instance IDs were assigned showed that Thetacrypt calculated them as the concatenation of the 'ciphertext label' as defined by the threshold cipher, and a hex-encoded SHA256-hash of the full ciphertext. This mechanism is functional insofar as it ensures that equal decryption requests get assigned an equal instance ID. However, having to hash the full ciphertext is inherently linear in the size of the ciphertext.

To further investigate this, a microbenchmark of the function which assigns an instance ID, as well as of the SHA-256 implementation it uses and of another SHA-256 implementation was done. Results are shown in Figure 4.10. The microbenchmark confirmed that firstly instance ID generation was dominated by the runtime of the utilized hash function. It also showed that the SHA-256 implementation which was used, provided by MIRACL Core, was only able to offer a hash rate of approximately $150\,\mathrm{MB/s}$. Alternative implementations such as the one provided by Ring[4] were able to handle up to $400\,\mathrm{MB/s}$, which is also what OpenSSL, without hardware acceleration, was able to provide on the same platform.

A straightforward fix would thus have been to swap out the implementation of SHA-256. However, it was determined that there was no need to rely on including a hash of the ciphertext in the protocol instance ID. Instead, it was deemed sufficient to rely on the ciphertext label supplied by the threshold cipher. As there is no trust boundary between a Thetacrypt server and its clients in the way Thetacrypt is assumed to be used — that is either both a Thetacrypt server and all its clients are honest, or all of them are malicious — honest clients of honest servers can be assumed to provide non-conflicting ciphertext labels. If this assumption did not hold, a trivial denial of service would be possible by submitting different decryption requests with ciphertext labels conflicting with third-party decryption requests.

Having removed the hashing of ciphertexts the macrobenchmark was run again, with results shown in Figure 4.9b. It confirmed that assignment of instance IDs now takes no time at all, and is a constant-time operation irrespective of the message size.

4.7 Unexplained component of server-sided decryption latency

Noticeably there is a component referring to 'unexplained' time, coloured in cyan, which seems to be proportional to the size of messages. It is likely to be the result of the handling of messages within Thetacrypt's protocol layer. There are many occasions in the protocol layer where full plain- or ciphertexts are duplicated, such as when copying decryption results into the state manager's long-term storage. While operations in RAM are usually fast, their execution time will still be inherently linear in the size of handled data. As the unexplained time was on the order of a single ms for usual message sizes it was not deemed pressing to investigate it any further for now. It might become a target for optimization if larger messages are desired, although then the performance of the symmetric cipher is likely to be more important by roughly an order of magnitude.

4.8 Comparing schemes and regions

At this point, it is now possible to attempt to quantify performance of Thetacrypt with all prior fixes in place. Figure 4.11 shows the server-sided decryption latency of Thetacrypt for different deployment scenarios. For each of the schemes, the most performant group was chosen. Messages were $1\,\mathrm{MiB}$ each. A few observations can be made based on the figure.

[4] `https://docs.rs/ring/latest/ring/index.html`

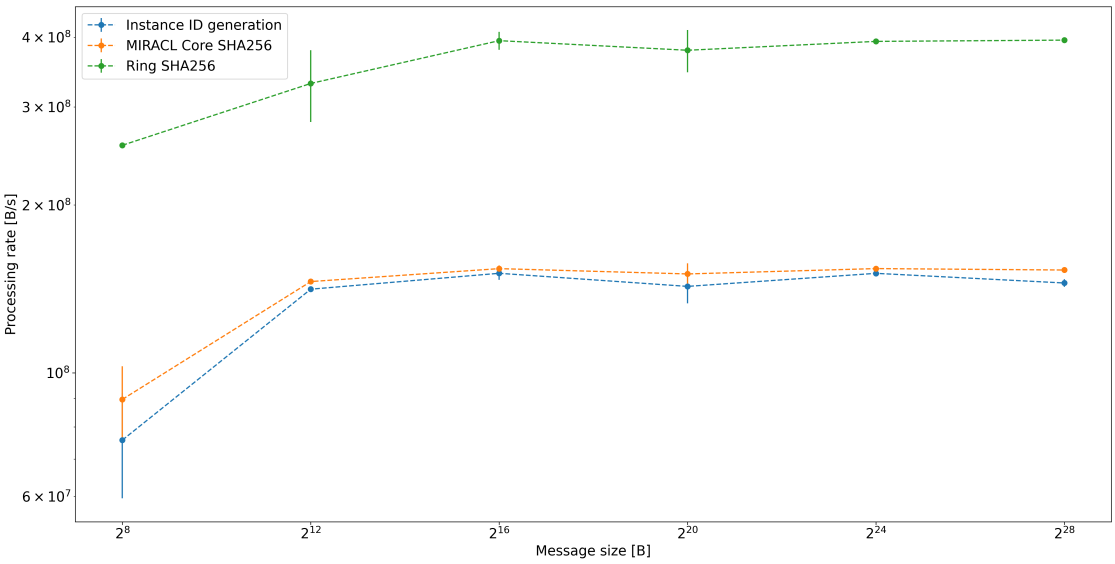


Figure 4.10: Performance of instance ID generation and of two hash functions. The blue line is the processing rate of the instance-ID generation function of Thetacrypt. The orange line is the hash rate of the utilized implementation of SHA-256. The green line is the hash rate of an alternative implementation of SHA-256. Error whiskers extend one sample standard deviation in each direction.

Firstly, as seen in many benchmarks before, the SG02 scheme consistently outperforms the BZ03 scheme, being about twice as fast. This is significant not only as it decreases decryption latency, but also as all the operations in which they differ are CPU bound. This implies that decryption throughput would be twice as high if using SG02 over BZ03.

Secondly, for a small deployment, Thetacrypt is, in most cases, able to provide a threshold decryption in less than $30\,\mathrm{ms}$. For larger deployments, threshold decryption will usually take less than $500\,\mathrm{ms}$ still, and — even in the worst case — barely ever more than $1\,\mathrm{s}$.

Thirdly there is a noticeable difference between regional deployments where inter-server latency is low, and global deployments where inter-server latency is high. Generally, the distribution of decryption latencies of a global deployment is wider with a longer tail than the corresponding distribution of a regional deployment. However, the median of the distribution is unaffected, as the mesh network usually allows servers to find a sufficient amount of nearby peers to cooperate with.

4.9 Describing decryption latency as a sum of its components

By combining all the collected data one can also attempt to explain the observed server-sided decryption latency as the sum of its contributing factors. On the one hand, this will allow to guide future performance optimization, on the other hand, it will serve as a kind of double bookkeeping to ensure that results are internally consistent. As an example, we will focus on the BZ03 scheme in a $N = 100$ deployment, in both the global as well as regional cases.

From Figure 4.11 we see that the median decryption latency for both the global and regional deployment is at $200\,\mathrm{ms}$. The two deployments mostly differ in the tail end of the distribution as discussed earlier. Looking at the corresponding server timings in Figure 4.12 we find that the server-sided latency is dominated by two components: Around $160\,\mathrm{ms}$ on ‘Waiting for decryption shares’ and approximately

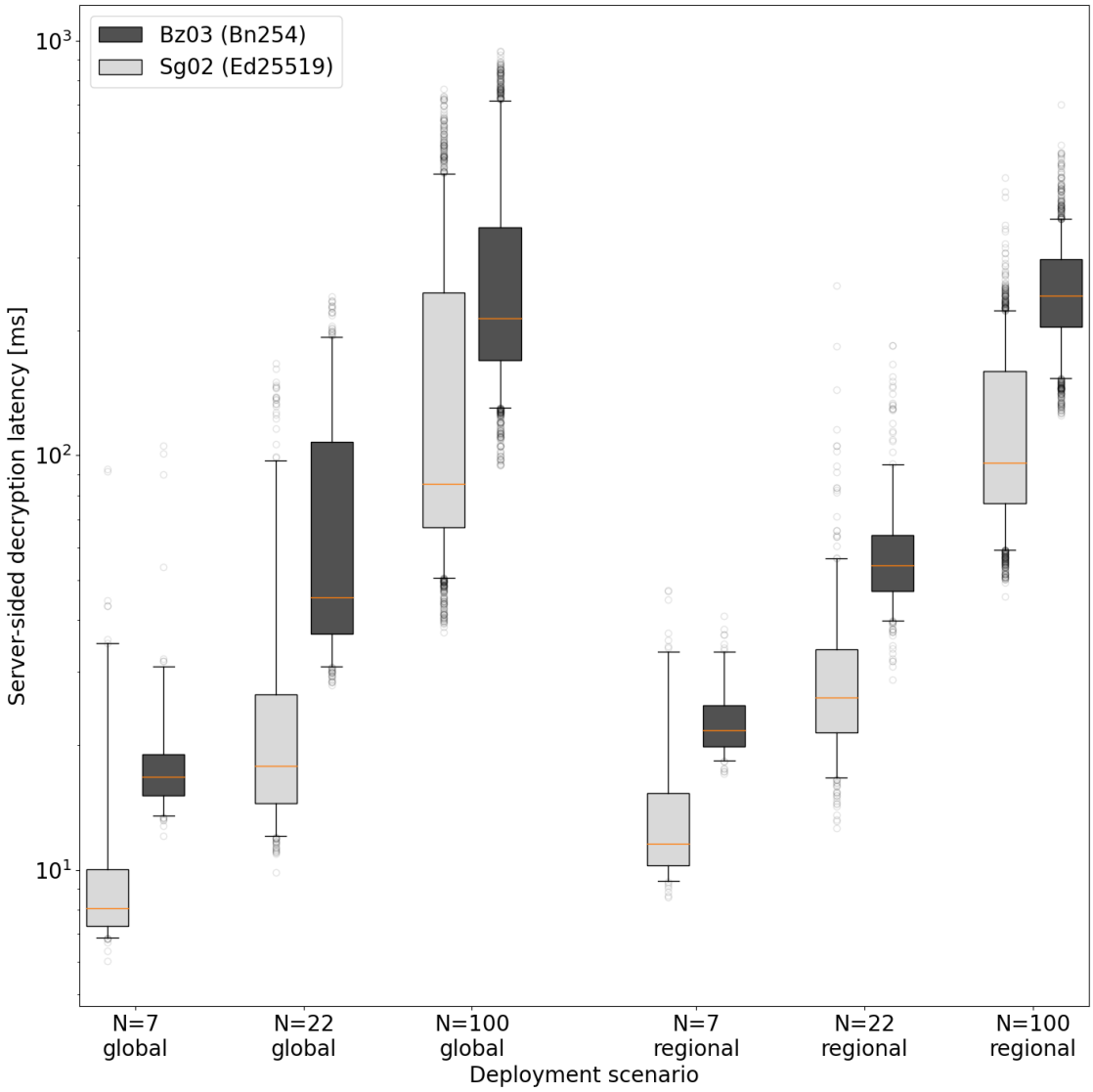
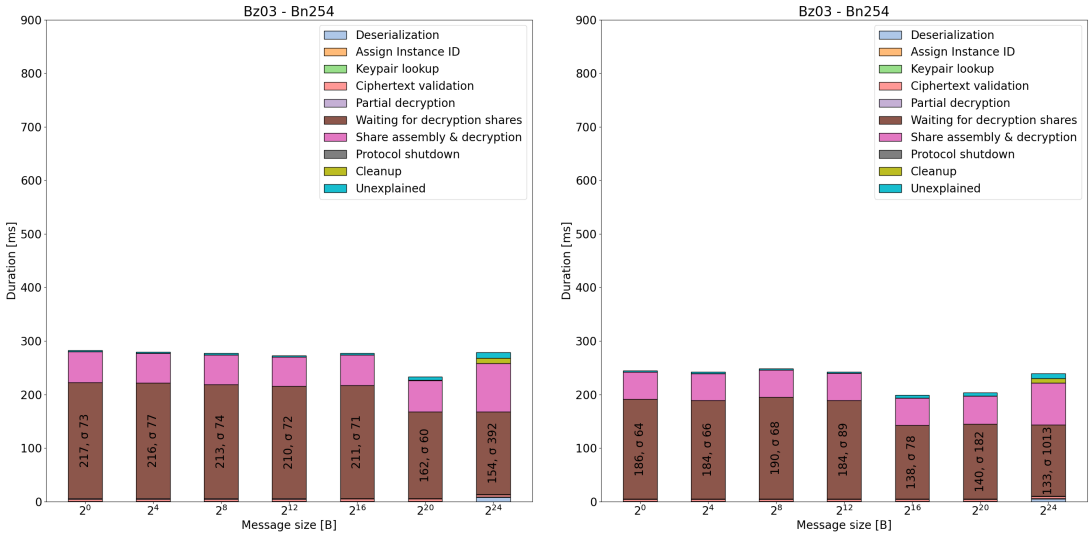


Figure 4.11: Server-sided decryption latencies of threshold ciphers for different deployment scenarios of Thetacrypt. Ciphertexts are $1\,\mathrm{MB}$ in size. The horizontal orange line represents the median of observed decryption latencies. The box extends from the first to the third quartile. The whiskers extend from the fifth to the 95th percentile, so cover $90\,\%$ of the data. Observations outside of whisker range are plotted as little circles.



(a) Regional deployment

(b) Global deployment

Figure 4.12: Server timings of threshold decryption for different message sizes. BZ03 scheme with $N = 100$. Each component is the median of all measurements. The largest component is annotated with value and standard deviation.

40 ms on 'Share assembly and decryption'.

It should be noted that in this figure, median decryption latencies of the global deployment would seem to be slightly lower than those of the regional deployment. Indeed this can already be seen in Figure 4.11, where the median of the global 'Bz03 (Bn254)' deployment with $n = 100$ is slightly lower than the median of the corresponding regional deployment. The spread of the data is as expected however, with the global distribution having a significantly longer tail. It is likely that the slightly paradoxical appearance of the median is the result of noise on either the system or network level.

Time spent on waiting for and validating decryption shares The former step encompasses two main tasks: Servers wait until they have accumulated $F + 1$ decryption shares, as well as validation of all incoming shares. Looking back to the microbenchmark of BZ03's decryption share validation routine in Figure 4.2b we see that each validation will take around 2 ms. For our case of $F + 1 = 34$ the server will thus have spent around 70 ms — nearly half the total time spent on this step — on validating them. It would be promising to attempt to optimize this step further, as it's responsible for a fairly large percentage of the decryption latency.

Time spent on assembling decryption shares and symmetric decryption The later step consists of two tasks as well: First the symmetric key of Thetacrypt's hybrid cryptosystem is assembled from the decryption shares, then the bulk of the ciphertext is decrypted. From earlier microbenchmarks of ChaCha20 we know that decryption of a 1 MiB ciphertext will have taken around 20 ms. The remaining 20 ms are the result of assembling $F + 1$ decryption shares. Looking back at the corresponding microbenchmark in Figure 4.2b we also expected the assembling of 34 shares to have taken approximately 20 ms, so those results too are internally consistent

N	Scheme	Smallest decryption latency [ms]	Decryption throughput [op. per s per core]
7	SG02	6	280
7	BZ03	11	150
22	SG02	10	170
22	BZ03	30	60
100	SG02	40	40
100	BZ03	90	20

Table 4.1: Extrapolated lower bounds on Thetacrypt decryption throughput

Consistency of micro- and macro-benchmarks Similar reasoning can be applied to other combinations of N, scheme and geographical deployment as well. In all cases results are generally consistent, showing that the results of the macrobenchmark can be roughly explained as the sum of the results of the various microbenchmarks.

4.10 Extrapolating encryption throughput

Combining the server-sided decryption latency with collected system monitoring data allows to provide a very rough lower limit for the expected decryption throughput at capacity.

First, during macrobenchmarks, CPU usage on all of the servers was never above $60\,\%$. Thus whatever decryption throughput was achieved during these benchmarks, the maximum achievable throughput per CPU core will be at least $(1/0.6)$ times as large.

Second, the lowest observed decryption latency of a given deployment provides an estimate of how much CPU time was spent in the case where a server 'got lucky', and did not have to spend a lot of time waiting on the network. This is thus an upper bound on the CPU time spent on a single decryption. For the SG02 scheme with e.g. $N = 7$ this would be around $6\,\mathrm{ms}$, and with $N = 100$ around $50\,\mathrm{ms}$.

Combining this allows the calculation of a lower bound on the expected decryption throughput of e.g. $(1/0.006\mathrm{s}) \times (1/60\%) \approx 280$ decryptions per second per core for the SG02 scheme with $N = 7$. Results for other deployment scenarios are shown in Table 4.1. These results will naturally also depend on the CPU frequency of involved hosts.

5 Conclusion

This work has analyzed and optimized the performance of two threshold ciphers in Thetacrypt. It has done so by systematically measuring the performance of both the components which make up Thetacrypt as well as of Thetacrypt as a whole. Where performance issues had been found, they were fixed. Doing so has lowered the decryption latency, in general, but especially for messages of sizes as they might be used in real-world deployments, by multiple orders of magnitude.

While doing so it has also laid a lot of groundwork — both within Thetacrypt as well as in separate tools — by developing a pipeline which assists in efficiently performing different types of benchmarks of Thetacrypt. The pipelines assist with most stages of the benchmarking process, from the provisioning of infrastructure, the collection of metrics, up to analysis and visualization of results.

Future work could broadly proceed along three main paths. Firstly this thesis has focused very much on decryption latency, while it would be just as — if not more — interesting to investigate Thetacrypt's decryption throughput in different scenarios. Secondly, Thetacrypt provides other threshold primitives such as signature schemes or randomness beacons, performance of which has not been analyzed at all yet. And lastly, it is very likely that there are still some performance issues in Thetacrypt's current implementation of the two threshold ciphers, which could be investigated more closely. For all three of these paths, this work could either serve as a staging ground, allowing to reuse some of the groundwork which has been laid, or just as a source of inspiration of what does — and what does not — work.

Bibliography

[Sha79] Adi Shamir. 'How to Share a Secret'. In: *Communications of the ACM* 22.11 (November 1979), pp. 612–613. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359168.359176. URL: https://dl.acm.org/doi/10.1145/359168.359176 (visited on 11/03/2022).

[DF90] Yvo Desmedt and Yair Frankel. 'Threshold Cryptosystems'. In: *Advances in Cryptology CRYPTO 89 Proceedings*. Ed. by Gilles Brassard. Vol. 435. New York, NY: Springer New York, 1990, pp. 307–315. ISBN: 978-0-387-97317-3. DOI: 10.1007/0-387-34805-0_28. URL: http://link.springer.com/10.1007/0-387-34805-0_28 (visited on 20/06/2023).

[De +94] Alfredo De Santis et al. 'How to Share a Function Securely'. In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing - STOC '94*. The Twenty-Sixth Annual ACM Symposium. Montreal, Quebec, Canada: ACM Press, 1994, pp. 522–533. ISBN: 978-0-89791-663-9. DOI: 10.1145/195058.195405. URL: http://portal.acm.org/citation.cfm?doid=195058.195405 (visited on 02/06/2023).

[Jou00] Antoine Joux. 'A One Round Protocol for Tripartite DiffieHellman'. In: *Algorithmic Number Theory*. Ed. by Wieb Bosma. Red. by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Vol. 1838. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 385–393. ISBN: 978-3-540-67695-9 978-3-540-44994-2. DOI: 10.1007/10722028_23. URL: http://link.springer.com/10.1007/10722028_23 (visited on 18/04/2023).

[Sho00] Victor Shoup. 'Practical Threshold Signatures'. In: *Advances in Cryptology EUROCRYPT 2000*. Ed. by Bart Preneel. Red. by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Vol. 1807. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 207–220. ISBN: 978-3-540-67517-4 978-3-540-45539-4. DOI: 10.1007/3-540-45539-6_15. URL: http://link.springer.com/10.1007/3-540-45539-6_15 (visited on 25/04/2023).

[BF01] Dan Boneh and Matt Franklin. 'Identity-Based Encryption from the Weil Pairing'. In: *Advances in Cryptology CRYPTO 2001*. Ed. by Joe Kilian. Red. by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Vol. 2139. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 213–229. ISBN: 978-3-540-42456-7 978-3-540-44647-7. DOI: 10.1007/3-540-44647-8_13. URL: http://link.springer.com/10.1007/3-540-44647-8_13 (visited on 18/04/2023).

[Jou02] Antoine Joux. 'The Weil and Tate Pairings as Building Blocks for Public Key Cryptosystems'. In: *Algorithmic Number Theory*. Ed. by Claus Fieker and David R. Kohel. Red. by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Vol. 2369. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 20–32. ISBN: 978-3-540-43863-2 978-3-540-45455-7. DOI: 10.1007/3-540-45455-1_3. URL: http://link.springer.com/10.1007/3-540-45455-1_3 (visited on 18/04/2023).

[SG02] Victor Shoup and Rosario Gennaro. 'Securing Threshold Cryptosystems against Chosen Ciphertext Attack'. In: *Journal of Cryptology* 15.2 (January 2002), pp. 75–96. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/s00145-001-0020-9. URL: http://link.springer.com/10.1007/s00145-001-0020-9 (visited on 11/03/2022).

[BZ03] Joonsang Baek and Yuliang Zheng. 'Simple and Efficient Threshold Cryptosystem from the Gap Diffie-Hellman Group'. In: *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*. GLOBECOM '03. IEEE Global Telecommunications Conference. San Francisco, CA, USA: IEEE, 2003, pp. 1491–1495. ISBN: 978-0-7803-7974-9. DOI: 10.1109/GLOCOM.2003.1258486. URL: http://ieeexplore.ieee.org/document/1258486/ (visited on 11/03/2022).

[BLS03] Paulo S. L. M. Barreto, Ben Lynn and Michael Scott. 'Constructing Elliptic Curves with Prescribed Embedding Degrees'. In: *Security in Communication Networks*. Ed. by Stelvio Cimato, Giuseppe Persiano and Clemente Galdi. Vol. 2576. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267. ISBN: 978-3-540-00420-2 978-3-540-36413-9. DOI: 10.1007/3-540-36413-7_19. URL: http://link.springer.com/10.1007/3-540-36413-7_19 (visited on 05/06/2023).

[BLS04] Dan Boneh, Ben Lynn and Hovav Shacham. 'Short Signatures from the Weil Pairing'. In: *Journal of Cryptology* 17.4 (September 2004), pp. 297–319. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/s00145-004-0314-9. URL: http://link.springer.com/10.1007/s00145-004-0314-9 (visited on 20/04/2023).

[CKS05] Christian Cachin, Klaus Kursawe and Victor Shoup. 'Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography'. In: *Journal of Cryptology* 18.3 (July 2005), pp. 219–246. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/s00145-005-0318-0. URL: http://link.springer.com/10.1007/s00145-005-0318-0 (visited on 25/04/2023).

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. 'Pairing-Friendly Elliptic Curves of Prime Order'. In: *Selected Areas in Cryptography*. Ed. by Bart Preneel and Stafford Tavares. Red. by David Hutchison et al. Vol. 3897. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 319–331. ISBN: 978-3-540-33108-7 978-3-540-33109-4. DOI: 10.1007/11693383_22. URL: http://link.springer.com/10.1007/11693383_22 (visited on 05/06/2023).

[Bro09] Daniel R. L. Brown. *SEC 1: Elliptic Curve Cryptography*. Certicom Research, 21st May 2009, p. 144. URL: https://www.secg.org/sec1-v2.pdf (visited on 04/04/2023).

[Ber+12] Daniel J. Bernstein et al. 'High-Speed High-Security Signatures'. In: *Journal of Cryptographic Engineering* 2.2 (September 2012), pp. 77–89. ISSN: 2190-8508, 2190-8516. DOI: 10.1007/s13389-012-0027-1. URL: http://link.springer.com/10.1007/s13389-012-0027-1 (visited on 05/06/2023).

[Bet15] John Bethencourt. *Intro to Bilinear Maps*. 2015. URL: https://people.csail.mit.edu/alinush/6.857-spring-2015/papers/bilinear-maps.pdf (visited on 13/04/2022).

[HMW18] Timo Hanke, Mahnush Movahedi and Dominic Williams. 'DFINITY Technology Overview Series, Consensus System'. Version 1. In: (2018). DOI: 10.48550/ARXIV.1805.04548. URL: https://arxiv.org/abs/1805.04548 (visited on 05/06/2023).

[BD19] Razvan Barbulescu and Sylvain Duquesne. 'Updating Key Size Estimations for Pairings'. In: *Journal of Cryptology* 32.4 (October 2019), pp. 1298–1336. ISSN: 0933-2790, 1432-1378. DOI: 10.1007/s00145-018-9280-5. URL: http://link.springer.com/10.1007/s00145-018-9280-5 (visited on 07/06/2023).

[BDV20] Luis T A N Brandao, Michael Davidson and Apostol Vassilev. *NIST Roadmap toward Criteria for Threshold Schemes for Cryptographic Primitives*. NIST IR 8214A. Gaithersburg, MD: National Institute of Standards and Technology, July 2020, NIST IR 8214A. DOI: 10.6028/NIST.IR.8214A. URL: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8214A.pdf (visited on 02/06/2023).

[Gre20] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Second edition. Addison-Wesley Professional Computing Series. Boston: Addison-Wesley, 2020. ISBN: 978-0-13-682015-4.

[GMT20] Aurore Guillevic, Simon Masson and Emmanuel Thomé. 'CocksPinch Curves of Embedding Degrees Five to Eight and Optimal Ate Pairing Computation'. In: *Designs, Codes and Cryptography* 88.6 (June 2020), pp. 1047–1081. ISSN: 0925-1022, 1573-7586. DOI: 10.1007/s10623-020-00727-w. URL: http://link.springer.com/10.1007/s10623-020-00727-w (visited on 07/06/2023).

[KG21] Chelsea Komlo and Ian Goldberg. 'FROST: Flexible Round-Optimized Schnorr Threshold Signatures'. In: *Selected Areas in Cryptography*. Ed. by Orr Dunkelman, Michael J. Jacobson and Colin O'Flynn. Vol. 12804. Cham: Springer International Publishing, 2021, pp. 34–65. ISBN: 978-3-030-81651-3 978-3-030-81652-0. DOI: 10.1007/978-3-030-81652-0_2. URL: https://link.springer.com/10.1007/978-3-030-81652-0_2 (visited on 20/04/2023).

[BP23] Luís T. A. N. Brandão and René Peralta. *NIST First Call for Multi-Party Threshold Schemes*. NIST IR 8214C ipd. NIST, January 2023, p. 71. URL: https://doi.org/10.6028/NIST.IR.8214C.ipd (visited on 04/04/2023).

[lib] libp2p maintainers. *What Is Publish/Subscribe*. URL: https://docs.libp2p.io/concepts/pubsub/overview/ (visited on 11/05/2023).

[Net] Network Time Foundation. *The NTP FAQ and HOWTO*. URL: http://www.ntp.org/ntpfaq/ (visited on 05/05/2023).

[Osm] Osmosis maintainers. *Osmosis Core: Multisig*. URL: https://docs.osmosis.zone/osmosis-core/keys/multisig (visited on 05/06/2023).

Declaration of consent

on the basis of Article 30 of the RSL Phil.-nat. 18

Name/First Name: Senn, Michael

Registration Number: 16-126-880

Study program: MSc Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

Title of the thesis: Exploring threshold cryptosystems

Supervisor: Prof. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of 5 September, 1996 is authorized to revoke the title awarded on the basis of this thesis.

For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with future theses submitted by others.

Bern, 20.06.2023

Place/Date

Signature