Marin Krmpotic, BSc

Implementation and Evaluation of Low-Latency Key-Exchange Protocols

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Christian Rechberger

Dipl.-Ing. Dr.techn. David Derler, BSc
Dipl.-Ing. Dr.techn. Sebastian Ramacher, BSc BSc MSc

Institute of Applied Information Processing and Communications

Faculty of Computer Science and Biomedical Engineering

Graz, September 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Acknowledgments

Throughout the writing of this thesis I have received a great deal of support and assistance. I would first like to express my gratitude to my thesis advisor, Christian Rechberger, for the support and opportunity to work on a project of such nature.

I want to thank my two co-advisors for their wise counsel during the many different phases of this thesis. David Derler, thank you for the willing explanations of the many schemes, and the support during the Java implementation. An equal thank you to Sebastian Ramacher for the invaluable help with the C implementation and guidance in writing this thesis. Sebastian helped me cross the finish line even when he was no longer at TU Graz, this type of kindness did not go unnoticed.

In addition, I wish to thank my colleagues at BearingPoint, for consistently understanding that my studies are of high priority and for never trying to convince me otherwise.

Finally, I wish to express my gratitude to those closest to me. I am truly grateful to Lea for pushing me forward even when it felt like the end was nowhere near, and for all the fun we have had in Graz. The biggest THANK YOU goes to my family, especially my parents, for the support throughout my studies. Thank you for never questioning my decision to move to Graz, and also for sending me hundreds of emails asking me about the progress of my thesis. If the theory is correct, and Google's AI is reading all of our emails, I must certainly be recognized as some sort of a master of their universe.

Abstract

Forward secrecy is an important property of cryptographic primitives such as encryption schemes or key-exchange protocols. The new TLS 1.3 protocol mandates forward secrecy for all TLS sessions. In addition, it also defines zero round-trip time (0-RTT) handshakes. Yet, forward secrecy can not not be achieved for all data exchanged in this mode.

Derler, Jager, Slamanig, and Striecks (Eurocrypt'18) introduced the concept of Bloom Filter Encryption, yielding the first fully forward secret, 0-RTT protocol with replay protection, being efficient enough to be used in practice.

In this master's thesis an implementation and in-depth evaluation of this primitive is performed in practical settings. We collected and analysed results of test runs with different parameters to propose an efficient selection of parameters. As a result, we conclude that the preliminary implementation of Bloom Filter Encryption is already efficient enough to be deployed in practice, albeit with some performance trade-offs.

Bloom Filter Encryption is furthermore implemented as a TLS 1.3 extension providing a fully forward secret 0-RTT mode. We present the ways in which the protocol had to be adapted to minimally interfere with existing functionalities. We evaluate and compare the performance of this new TLS handshake protocol with the existing one. A slow down in TLS handshake package processing is recorded, but we argue that such implementation could still be effective in low bandwidth environments.

Contents

List of Figures

List of Tables

1 Introduction

Today's world is unimaginable without global connectivity. Almost any activity, from web browsing to storing data in the cloud or connecting to a company's internal network, is based on connections running through the public Internet. With the rapid popularization of the Internet and its applications like online banking, online shopping or the use of eGovernment services, the amount of sensitive data transferred using it is increasing. For example, in 2019 around 1.5 billion users access Facebook daily [Fac19b], and almost 100 billion messages are sent daily across all Facebook apps [Fac19a]. This private correspondence ought to be protected from eavesdroppers. Concerns about securing online payment transactions were present much before privacy became a mainstream topic. This market is still rapidly growing: 9.9 billion transactions were made via PayPal in 2018 [Pay19] totalling $578 B in volume. This is 27% more transactions than the year before. Online transactions have close connections with online sales. To put things into perspective, in just two days of Prime Day sale in 2019, Amazon sold more than 175 million items [Ama19]. These orders contain sensitive information such as credit card data, addresses, contact information, etc.

The Transport Layer Security (TLS) protocol ensures the secrecy of our communication during transport. In September 2015 around 46% of all connections made with the Chrome browser were secured with TLS, with more than 84% of connections being secured by June 2019 [Goo19]. This upward trend shows general acceptance and the need for encryption on the Internet. Chrome even went one step further and started marking all websites without TLS as insecure, and Firefox followed shortly thereafter. The latest version of this protocol, TLS 1.3, was published in August 2018 as a proposed standard.

Forward secrecy [Gün90] is an important property of cryptographic primitives such as encryption schemes or key-exchange protocols. In forward secret schemes, if a secret key used for decryption leaks, this does not compromise ciphertexts

from the past. This means the attacker that was eavesdropping on, and collecting, encrypted messages from some private conversation, will not be able to decrypt those messages even after acquiring the key later in time. The new TLS 1.3 protocol mandates forward secrecy for all TLS sessions. In addition, it also defines less exchanges between the client and the server during the TLS handshake, a process in place before the secure communication can be made. It directly allows the sending of encrypted application data within the first message from the client to the server, dramatically speeding it up. This is called a zero round-trip time (0-RTT) mode. Yet, forward secrecy can not not be achieved for all data exchanged in this mode.

With more companies moving their business online each day, network latency is a hot topic due to its direct influence on revenue. In a study by TABB Group [Rep08], the author estimates that if a broker's electronic platform is 5 milliseconds behind the competitor's, it could result with a 1% loss in revenue per millisecond. That is around 4 million dollars revenue loss per millisecond. In addition, the study claims latency of up to 10 milliseconds could result in a 10% revenue loss. On the same note, in 2010, the Mozilla Foundation [Cut10] reported a 15.4% increase in Firefox download conversions after they had improved page loading time by 2.2 seconds. This meant more than 10 million additional downloads of their browser per year.

Different factors influence load times. Work is usually done to optimize and compress the code, with often a Content Delivery Network (CDN) deployed to cache the content and decrease the latency caused by geographical distance. Still, before any data transfer is possible, a TLS handshake has to be performed in order for the data to remain secure. The solution to this problem would be the 0-RTT TLS mode, but maintaining full forward secrecy in this mode was deemed impossible.

Recently, Günther, Hale, Jager, and Lauer [GHJ+17] made a huge step forward and presented the first construction of a key-exchange protocol achieving full forward secrecy, 0-RTT handshakes and replay protection at the same time. Unfortunately, their construction is by far not efficient enough to be used in practice. To this end, Derler, Jager, Slamanig, and Striecks [DJS+18] introduced the concept of Bloom Filter Encryption (BFE), yielding the first such protocol that was efficient enough to be used in practice. This paper was later extended with efficiency improvements by Gellert [DGJ+18]. In their work,

many decryption keys are precomputed and mapping between ciphertext and decryption keys is done with a probabilistic Bloom filter [Blo70] data structure. After each decryption, all keys capable of decrypting the given ciphertext are destroyed. While the work on this thesis was ongoing, Aviram, Gellert, and Jager [AGJ19] presented their construction of the forward secure 0-RTT protocol. Their construction uses a special case of a constrained pseudorandom function (PRF) [BW13] called puncturable PRF, that enables the derivation of keys for a subset of inputs. In [DGJ+18], with each decryption the key is updated with a smaller subset making it incapable of decrypting the same message again. Looking from a higher level, all three mentioned protocols are based on similar ideas of providing forward secrecy by updating the secret key with each decryption.

1.1 Contribution

There are two types of Bloom Filter Encryption presented in [DJS+18], known as Basic BFE and Time-Based BFE. In this thesis we present implementations of both primitives, each in Java and C. We evaluate the implementations by measuring performance with different sets of parameters. The selection of parameters highly depends on intended use-case and in general comes as a trade-off between lower memory consumption and longer-lasting secret keys. We present the performance results in detail to demonstrate this correlation.

To demonstrate practical usage of BFE, we created a new TLS 1.3 extension the `GnuTLS` library, providing a fully forward secret 0-RTT mode. Here, we measure performance of this new TLS handshake protocol and we evaluate and compare it with the full TLS handshake. We discuss why such implementation incorporates nicely into TLS 1.3 while not being easily implementable with previous versions of TLS.

To support further development and the reproducibility of results, all code created as part of this research is released to the public domain.

1.2 Outline

Chapter 2 discusses the preliminaries with an overview of basic secure communication concepts. In Chapter 3, the encryption schemes used for construction of BFE are described in detail.

The implementations of Basic BFE and Time-Based BFE with justifications for the specific design decisions are presented in Chapter 4. Separate sections are dedicated to the C and Java implementations of the schemes. Expected results of performance tests and their outcomes are presented as a part of the same chapter.

The implementation of the `bfe_key` TLS 1.3 extension is presented in Chapter 5. There, the performance measurements are elaborated in detail and compared with the full TLS handshake.

2 Preliminaries

This chapter provides an overview of secure communication concepts used today. We start by discussing the problems such communication faces and continue by describing protocols used to mitigate them. Furthermore, in order to support easier comprehension of cryptographic primitives used in the implementation part of the thesis, we describe the probabilistic data structure Bloom filter.

2.1 Key Exchange

Key exchange protocols are integral parts of today's communication. With such a protocol, two parties are able to agree on a set of keys used for secure transmission of messages over insecure channels. As the key exchange takes place before the real application data can be sent, it naturally adds a performance overhead to connection itself. The time it takes can be unambiguously expressed in the number of roundtrips between the two parties before a secure connection is achieved. Protocols currently used in practice make several roundtrips prior to the secure connection. Sending application data with the very first message sent to the other party would yield the lowest transmission latency. This type of protocol is called a zero round-trip time (0-RTT) protocol. Some novel protocols offer 0-RTT but their real-life applicability is often unproven, or they have weaker security properties.

0-RTT protocols are especially vulnerable when it comes to replay attacks and forward secrecy. Replay attacks are trivial to perform: an attacker does not need any secret information from any of the involved parties. With access to the communication channel, an attacker takes a data package while in transit and then resends the same package again to the receiving party. If not protected, a receiving party will interpret the message again like it was legitimately sent. If

the protocol does not mitigate replay attack threats, they are still manageable at the application layer.

In case the decryption key leaks at some moment, encrypted messages sent previously are at risk of being decrypted. A basic solution to this problem is updating the public and secret key before each message, adding latency to the communication. In more advanced forward-secure schemes [Gün90], the public key remains unchanged. Still, if the key leaks at time t, it can't be used to decrypt messages sent at $t-1$. This is usually achieved by periodically updating the secret key at the receiver's end, and since it does not affect the public key there is no need for key renegotiation.

The first such non-trivial construction of forward-secure public key encryption was presented in [CHK03]. Their construction is based on Hierarchical Identity Based Encryption (HIBE) [GS02] where each node is a secret key of one time interval, with messages being encrypted for the specific time interval using a single public key. Except for communication protocols, forward secrecy is an important property of other cryptographic schemes as well. In digital signatures, forward secrecy can insure signatures pertaining to the past can not be forged in the case of current key leakage [BM99].

2.2 Transport Layer Security Protocol

The main purpose of the Transport Layer Security (TLS) protocol is ensuring private data exchange between two parties. It does so by using symmetric encryption protocols with different keys being used for each connection. In symmetric encryption schemes, the same key is used for encryption and decryption. To retain data confidentiality, the key must not be transferred unprotected via insecure channels. In TLS this is handled by a handshake protocol established by asymmetric key exchange prior to exchange of application data.

Being a proposed standard for communication between independently developed applications, TLS aims to achieve interoperability. It eliminates the need for potentially flawed proprietary protocols by facilitating secure data flow without having to know implementation details of the other system. Moreover, it is application layer independent so higher level protocols can build on top of it.

2.2.1 Version 1.2

As version 1.3 of the protocol was finalized only recently in 2018, TLS 1.2 [RD08] is still extensively used today. It was presented in 2008 as a successor of TLS 1.1.

When initiating communication, a client and a server have to agree on a protocol version and cryptographic algorithms to be used. Both sides can then derive shared secrets. The protocol can additionally require authentication of a client and/or a server, which is enforced by digital certificates.

The client initiates the connection by sending a `ClientHello` message to the server. This has to be the first message sent from a client to a server, and can also be sent as a response to a `HelloRequest` coming from server. `ClientHello` consists of a session identifier, the supported TLS version, random value, cipher suite list, compression methods, and optional extensions. If set, the session identifier implies the request for a session resumption. The cipher suite list sent is a list of all cryptographic options client supports in the order of preference. The server decides on a specific one from the list to be used, or fails the whole connection if none of the provided ones is acceptable. Extensions are used for additional functionality that may be supported by the server on top of the default protocol. After sending the `ClientHello` message to the server, the client waits for a server to respond with the `ServerHello` message.

The server will dispatch a `ServerHello` message if there was an acceptable cipher suite in the received `ClientHello`. `ServerHello` consists of a TLS version, random value, session identifier, selected cipher suite, selected compression method, and extensions. If a session identifier was received in the `ClientHello` message, the server uses this value for a cache lookup searching for an existing session state. If such a state exists, the server may initiate session resumption skipping the rest of the handshake process. Otherwise, it generates a new session identifier. The cipher suite and compression method are strictly one entry each from the ones proposed by the client. In case of session resumption those values are taken from the existing session state. The server is allowed to include only extensions that were a part of `ClientHello`. If an additional extension is sent to a client, the client will abort the handshake.

Right after, sometimes the server sends a `Certificate` message to authenticate

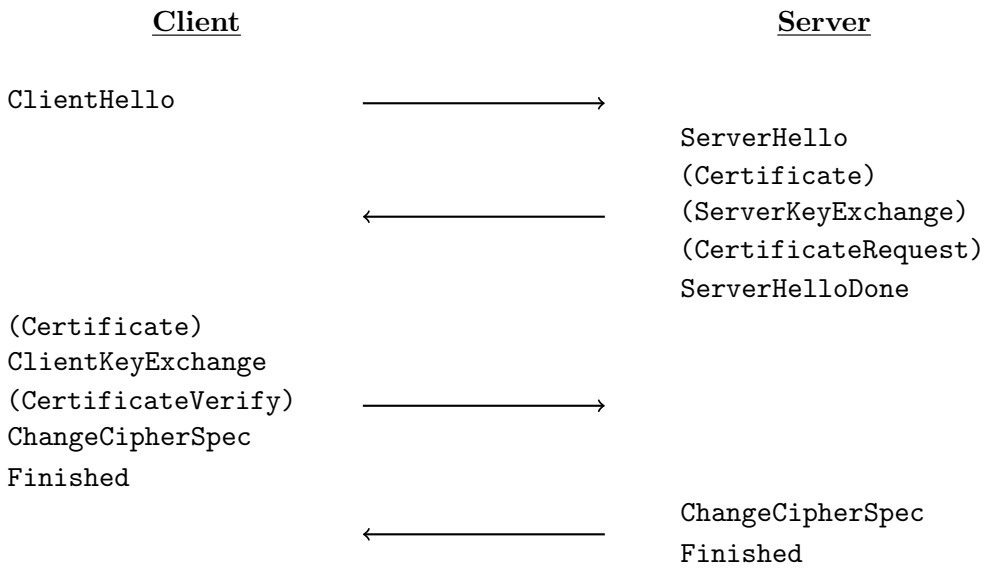


Figure 2.1: Client-server messages during a full TLS 1.2 handshake. Data shown in parentheses is optional or situation dependent.

itself to the client. This message is optional depending on the key exchanged method, since not all key exchange methods defined in TLS 1.2 use certificates. If the `Certificate` is not being sent, or if it does not contain enough data for premaster secret exchange, a `ServerKeyExchange` is sent to provide this data. In general, it contains a public key for the chosen cipher suite. If the server requires a client to be authenticated as well, it sends a `CertificateRequest` message. The very last message sent as a part of the server hello is `ServerHelloDone`, after which the server waits for the client's response.

The client sends its `Certificate` message after receiving the `ServerHelloDone` message, but only if one was requested by the server. The client may, or may not, send a certificate when requested, leaving a decision to the server if authentication is necessary. The following message is the `ClientKeyExchange`. It contains either the RSA-encrypted [RSA78] premaster secret with the server's public key, or Diffie-Hellman (DH) parameters that will be used for premaster secret derivation. If the client sent a certificate with signing capability, it

additionally sends a `CertificateVerify` message with a signature over all previously exchanged handshake messages. This proves to a server the client indeed is the owner of the provided certificate. The `Finished` message is sent straight after a `ChangeCipherSpec` message, and it contains all the messages from that handshake. This is the first message encrypted by the negotiated algorithms and it is used for verification on both sides before real data can be sent. The message flow of TLS 1.2 full handshake is illustrated in Figure 2.1.

The premaster secret acquired from the handshake is used to derive a master secret. This is done on both sides with pseudorandom function (PRF) family using premaster secret, client random, and server random, as input values. The prototype of the PRF used for TLS 1.2 master secret derivation is shown in Section 5.1.2.

Forward secrecy in TLS 1.2 is possible, but not mandatory. Whether the protocol is forward-secure or not depends on the key exchange mode used. It is forward-secure for Diffie-Hellman Ephemeral (DHE) key exchange, a modification of DH where a new key is generated for each key exchange. Together with DHE, TLS 1.2 supports its elliptic curve variation known as Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) [BBG+06]. The two are referred together as (EC)DHE, both providing forward secrecy.

PSK Key Exchange Algorithm TLS 1.2 supports a use of pre-shared keys (PSKs) [ET05] that are shared in advance between the involved parties. The main reason for establishing a connection using the PSK is avoiding public key operations. Depending on the situation, this could be desirable in case of low computing power, or when the connection is configured manually in advance. As stated by the authors of [ET05], using PSKs is intended for only a small set of applications. All certificate related messages are excluded when a PSK key exchange algorithm is used. The client indicates the intention of using the PSK by sending PSK ciphersuites in its `ClientHello` message. If the server accepts the use of PSK, it chooses one of the given ciphersuites and sends it to the client in its `ServerHello` message. The client indicates which PSK to use by sending the identity of the key in a `ClientKeyExchange` message. The server can optionally send an identity hint in its `ServerKeyExchange` message in order to help the client in choosing the correct identity,

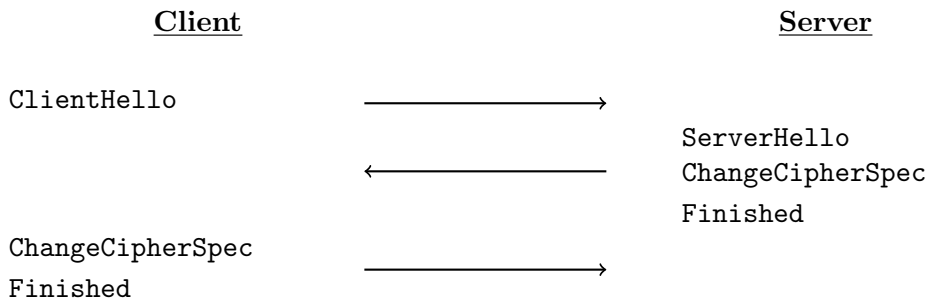


Figure 2.2: Client-server messages during a TLS 1.2 session resumption handshake

Session Resumption

Provided that a client and a server have already performed a full handshake in the past, they can skip the full handshake process and perform a resumption handshake. Such case is shown in Figure 2.2. A client sends a session identifier from the previous session in its `ClientHello` message. The server then retrieves session parameters from its cache and responds with a `ServerHello` containing the same session identifier. Both sides then send a `ChangeCipherSpec` followed with the `Finished` message. This completes a session resumption handshake and application data can be safely transmitted.

2.2.2 Version 1.3

TLS 1.3 is the latest version of the protocol. It is not directly compatible with TLS 1.2, but client and server are capable of negotiating a version they both support due to TLS versioning mechanism. TLS 1.3 uses an (EC)DHE key exchange imposing forward secrecy in all modes except one. The exception, of special interest for this thesis, is the new 0-RTT handshake mode [Res18], which speeds up the session resumption time at a cost of some security features.

Session identifier is now a legacy parameter in the `ClientHello`. The session renegotiation from TLS 1.2 is replaced by preshared key support, thus this parameter is used only for legacy and compatibility purposes. Extensions are still used for additional functionality, but some extensions are defined as mandatory. This shift to extensions was made to ensure compatibility with previous versions.

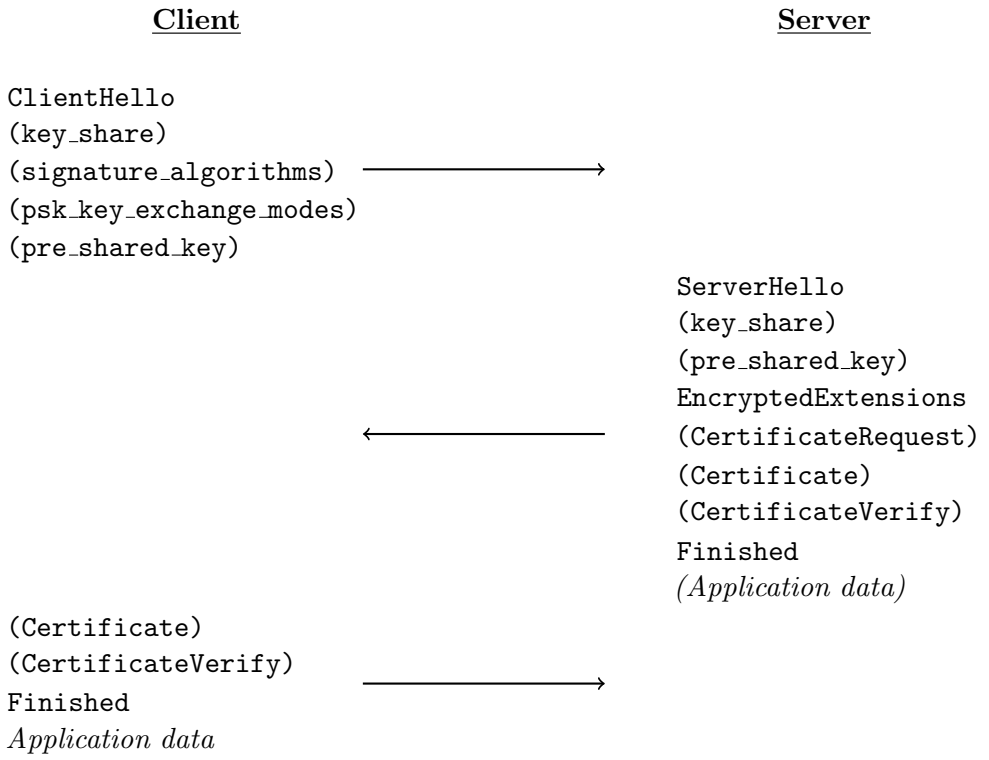


Figure 2.3: Client-server messages during a full TLS 1.3 handshake. Only chosen extensions are shown, all written in snake case style. Data shown in parentheses is optional or situation dependent.

There are a number of novel extensions defined in [Res18]. The `ClientHello` message must include at least the `supported_versions` extension, likewise for `ServerHello`, which is only allowed to include extensions important for establishing cryptographic context and protocol version. Extensions not needed for this establishment are sent in an `EncryptedExtensions` message. As is evident from the name, extensions sent inside it are encrypted by the server's handshake traffic secret.

The client can send `signature_algorithms` extensions with a list of supported signature algorithms. This extension is only mandatory if the client requests the server to authenticate itself with a certificate. If a different set of signature

algorithms is to be used for the certificates and internally in TLS, an additional `signature_algorithms_cert` extension can be used to separate the two. In the `key_share` extension a client sends multiple key shares, each for at most one supported finite field group. This list is ordered by the client's preference and strictly one or none is chosen by the server. In the former case the server returns its key share for the chosen group, while in the latter it sends a `HelloRetryRequest` with a key share of the client supported group that was not included in the `key_share` extension. The server knows the client's supported groups as they are sent by the client in a `supported_groups` extension. A simplified TLS 1.3 full handshake is illustrated in Figure 2.3.

Session Resumption and Early Data

At any time after a full handshake has been carried out, a server can send a `NewSessionTicket` message. In the message there is a unique connection of the ticket with the PSK derived from the resumption master secret. This value is used by a client in a `pre_shared_key` extension when trying to resume a connection. Each ticket has a lifetime defined and should not be used after the expiration time. The extension has to be sent in pair with a `psk_key_exchange_modes` extension notifying the server which PSK modes a client supports. This information is used by the server to send only compatible tickets in a `NewSessionTicket`. Possible key exchange modes are a PSK-only mode and a PSK with (EC)DHE. With (EC)DHE key establishment in place, the session keys are forward secret. This is not the case with PSK-only mode, as keys are derived exclusively from the PSK.

If a PSK with early data support is used, an `early_data` extension can be sent. The extension indicates application data is being sent with the very first set of messages from the client. This is referred to as a 0-RTT session resumption. The extension itself does not carry any application data, but instead the application data is sent after all extensions. The client is permitted to send early application data only until it receives a server's `Finished` message. To avoid deadlocks in a handshake, the server will not wait until the end of early data, but will instead send its `ServerHello` as soon as possible. In addition, the `pre_shared_key` extension has to be sent together with the `pre_shared_key` extension as 0-RTT

is not supported in the full handshake mode. The early application data is encrypted with the first PSK listed in the `pre_shared_key` extension.

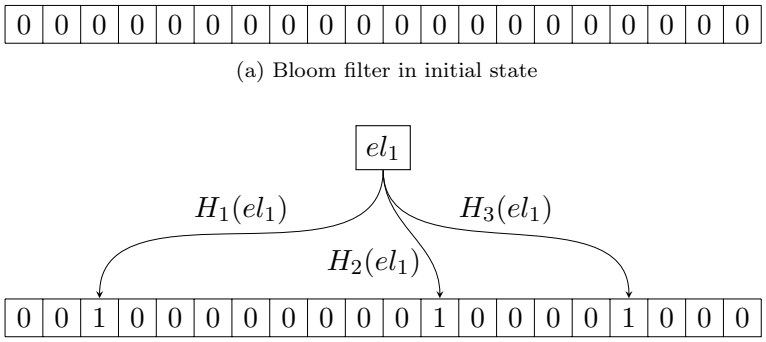
Using PSK to encrypt early data makes the data less secure compared to data transmitted after a handshake has been completed. Similar to the PSK-only key exchange mode, early data is encrypted exclusively with PSK-derived keys. This leaves data without forward secrecy and replay protection. Some mitigation techniques are offered in [Res18], the simplest one of which is using the single-use tickets. In that case, a server would maintain a database of valid tickets, destroying individual tickets upon usage. Still, the correct usage of the 0-RTT mode would be not to relay data not safe to be replayed. One such example is Cloudflare's selective 0-RTT usage [Sul17] where they use it exclusively for HTTP `GET` requests with no query parameters. By definition, `GET` requests should not change the server's state, and avoiding the ones with parameters circumvents security vulnerabilities due to their potential misusage.

2.3 Bloom Filter

A Bloom filter [Blo70] is a data structure used for checking whether an element is a part of the set or not. While its main functionalities are querying and inserting into the set, the Bloom filter does not store the elements. This is achieved with a number of hash functions and a series of bits stored in the memory. Not having to store the elements comes in quite useful with large data sets where storing would be infeasible. That aside, this also comes with a price - the Bloom filter is a probabilistic data structure. The error rate is manipulable with a set of parameters that are subject to assessment depending on the application. A Bloom filter is defined by algorithm set BF = (BF.Setup, BF.Insert, BF.Query).

2.3.1 Insertion

Elements are not inserted in the Bloom filter. Instead, a number of bits inside the filter is set which are used for querying later in time. Each Bloom filter has a predefined number of hash functions h, and a number of bits n inside it. As Figure 2.4a indicates, at first, all bits are set to 0. The Bloom filter is formally a pair (H, T) where $H = H_1, \ldots, H_h$ and $T = 0^n$ initially. When an element is



(a) Bloom filter in initial state

(b) Insertion of an element to a Bloom filter with three hash functions

Figure 2.4: Insertion of an element to a Bloom filter

to be inserted, it first gets hashed h times, once by each hash function. These digest values are mapped to bit array indices indicating which bits in the Bloom filter should be set. Evidently, the number of affected bits equals n, as shown in Figure 2.4b. Removing elements from the filter is not permitted considering it is impossible to know if a certain bit was set by another element in addition to the one being removed. This could easily cause the unintentional removal of multiple elements.

2.3.2 Query

Querying a Bloom filter is a probabilistic operation. False positives are possible with no chance of false negatives. The element goes through the same hashing procedure as for insertion, yielding bit array indices. Bits with these indices are then checked whether all of them are set or not. If not, the element is definitely not in the set, otherwise it possibly is. False positive results occur if all bits corresponding to an element have already been set by other elements at the time of querying. Figure 2.5 depicts the mentioned scenarios.

2.3.3 Parametrization

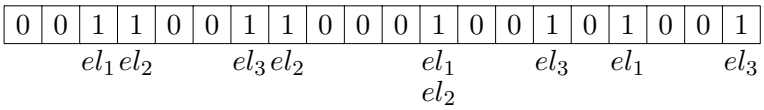
A Bloom filter is defined by three parameters: the number of bits m, the number of hash functions h, and the number of elements, or capacity, n. These together

influence the filter's false positive probability.

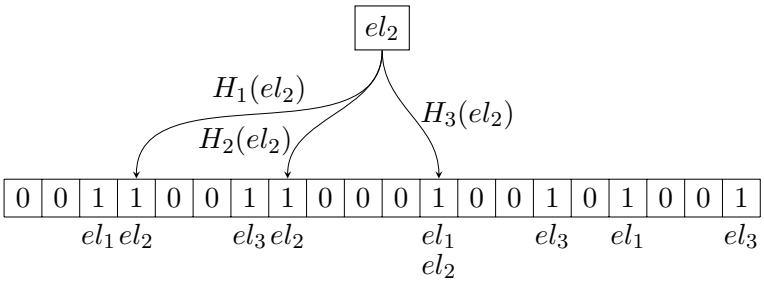
Assuming the hash function yields uniform results, the probability a certain bit is addressed is $1/m$. The probability that this bit remains unset is therefore $1 - 1/m$, being $(1 - 1/m)^h$ when having h hash functions. By adding new elements to the filter, the probability of a bit remaining unset decreases. After all elements are inserted into the filter, the probability of a certain bit to be 0 is $(1 - 1/m)^{hn}$. Inversely, the probability a bit is set is $1 - (1 - 1/m)^{hn}$. From here, the probability of a false positive is given by [FCA+98]:

$$\Pr_{\text{false}} = \left(1 - \left(1 - \frac{1}{m}\right)^{hn}\right)^n \approx \left(1 - e^{-\frac{hn}{m}}\right)^h.$$

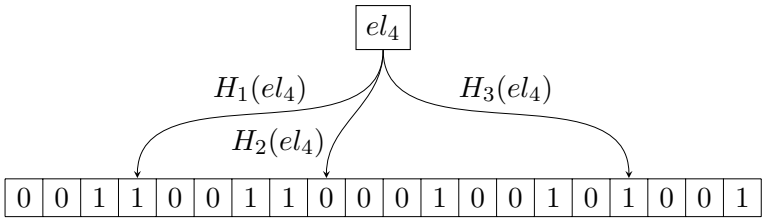
This is minimized for $h = \ln 2 \cdot \frac{m}{n}$, where h is an integer.



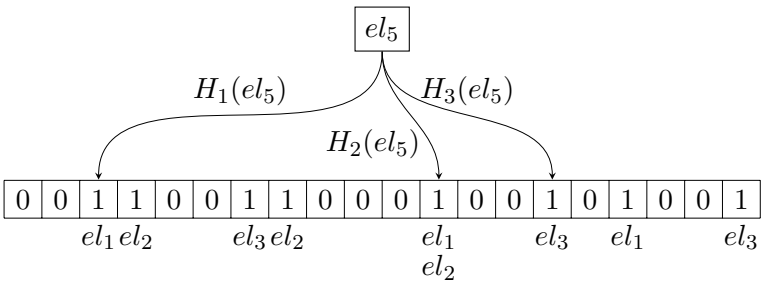
(a) Bloom filter after insertion of three elements, $h = 3$



(b) When checking whether an element is inside a filter or not, the element is hashed h times and corresponding bits are checked if all are 1



(c) If not all corresponding bits are 1, the element is definitely not in the filter



(d) False positive results occur when all bits for a certain element are set by other elements

Figure 2.5: Different results of Bloom filter queries

3 Encryption Schemes

The encryption schemes we implement in this thesis rely on bilinear pairings, Identity Based Encryption (IBE), and a concept of puncturable encryption. In this chapter we discuss these topics and additionally describe two IBE schemes that are used for the construction of a Bloom Filter Encryption (BFE) scheme. This leads us to the last remaining preliminary that is the construction of BFE itself.

3.1 Bilinear Pairings on Elliptic Curves

In 1986, Miller [Mil86] came up with the polynomial time algorithm for calculating Weil pairings [Wei40] on elliptic curves. This fast-tracked the usage of pairings in cryptography that started in early 1990's. Weil pairings were originally used for attacks on elliptic curve discrete logarithm problem. In 1992, Menezes, Vanstone and Okamoto [MVO91] demonstrated a reduction of the elliptic curve discrete logarithm problem to the logarithm problem in a multiplicative group of an extension of a finite field.

Nowadays, pairings are recognized as a constructive mechanism for cryptographic protocols. First to recognize this potential was Joux [Jou00], who in 2000 presented a one round DH tripartite key exchange protocol. The year after, Boneh and Franklin [BF03] constructed an efficient IBE scheme and in the same year Boneh, Lynn and Shacham [BLS01] presented a short signature scheme based on pairings. To name a few more, pairings have been used for the construction of self-blindable credentials [Ver01], unique signatures [Lys02] used for constructing verifiable random functions (VRFs) [MVR99], group signatures [BBS04; BS04; CL04], and non-interactive zero-knowledge proofs [GS08].

Definition 1 (Bilinear Pairing)**.** With groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_t$ of prime order p, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$, map e is a pairing if:

- e is bilinear, meaning $e(u^a, v^b) = e(u, v)^{ab}$; $\forall u \in \mathbb{G}_1$, $\forall v \in \mathbb{G}_2$, $\forall a, b \in \mathbb{Z}$
- e is non-degenerate, implying $e(g_1, g_2) \neq 1$; $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$
- e is computable, indicating there is an efficient algorithm for computing e.

There are three types of bilinear pairings.

Definition 2. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ be a bilinear pairing, then e is

- Type-1, if $\mathbb{G}_1 = \mathbb{G}_1$
- Type-2, if $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is an efficiently computable isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$
- Type-3, if $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is no efficiently computable isomorphism ψ.

The evaluation of security levels achieved for different field sizes is an ongoing topic. The latest influential work in this area is by Barbulescu and Duquesne [BD18] where they show the security levels of pairings over popular elliptic curves are lower than previously considered. In Chapter 4, we test our implementation with different curve types and field sizes. From [BD18], pairings over Barreto-Lynn-Scott (BLS) curves [BLS03] of a 381 bit prime order provide 120 bit security[1], and pairings over BLS and Barreto-Naehrig (BN) curves [BN06] of a 461 bit order provide 128 bit security.

3.1.1 Bilinear Diffie-Hellman Assumption

In order to prove the security of cryptographic schemes based on bilinear pairings, the security of those schemes is based on the difficulty of some well known problems. One of such problems is a Bilinear Diffie-Hellman (BDH) problem.

Definition 3 (BDH Problem)**.** Let $\mathbb{G}$ and $\mathbb{G}_t$ be groups of prime order p, $\hat{e} : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$ a bilinear map, and g a generator of $\mathbb{G}$. The BDH problem in $(\mathbb{G}, \mathbb{G}_t, \hat{e})$ is the following: given (g, g^a, g^b, g^c) for some $a, b, c \in \mathbb{Z}^*$, compute $v = \hat{e}(g, g)^{abc} \in \mathbb{G}_t$.

[1] According to Razvan Barbulescu from personal communication

In order to define a BDH assumption, first we have to introduce a notion of the BDH parameter generator.

Definition 4 (BDH Parameter Generator)**.** A randomized algorithm $\mathcal{G}$ that takes security parameter λ as an input is a BDH parameter generator if it runs in time polynomial in λ and outputs descriptions of groups $\mathbb{G}$ and $\mathbb{G}_t$ of prime order p, and a map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$.

Definition 5 (BDH Assumption)**.** We define the advantage of an algorithm $\mathcal{A}$ solving BDH as:

$$\mathsf{Adv}^{\mathsf{BDH}}_{\mathcal{A},\mathcal{G}} = \Pr\left[\hat{e}(g,g)^{abc} \leftarrow \mathcal{A}(\mathbb{G}, \mathbb{G}_t, \hat{e}, g, g^a, g^b, g^c)\right] \geq \epsilon.$$

A parameter generator $\mathcal{G}$ satisfies the BDH assumption, if for any randomized polynomial time algorithm in the security parameter ϵ, the advantage $\mathsf{Adv}^{\mathsf{BDH}}_{\mathcal{A},\mathcal{G}}$ is negligibly small.

3.2 Identity Based Encryption

Public key cryptography is a cryptographic scheme that uses two keys of different kind for carrying out private communication. One key is publicly available, while the other is being held private by the specific owner. A pair of entities has to exchange public keys before they are able to engage in secure communication. After a sender has the receiver's public key, it uses it to encrypt the message that can only be decrypted by the receiver's private key. Hence, this scheme is often called *asymmetric cryptography*.

Shamir [Sha85] introduced a cryptographic scheme for private communication without a need to exchange the keys beforehand. It uses unique strings as public keys for communication with certain entities. As a result of these strings being intuitive identity-based values such as email address or username, the scheme is called Identity Based Encryption. By knowing the identity of the receiver, a sender can relay a message without knowing any other information about the receiver. The receiver has to contact the key generation service once at registration to acquire the private key, after which he may receive messages from anyone regardless of their identity. Identities being no secret are the reason why a key generation service is needed. If anyone could derive a private key for

identity, a scheme would be insecure. Complementary to this, the key generation service must be well trusted considering it has the ability to generate private keys for all possible identities in the system.

Definition 6 (IBE)**.** An IBE is defined by the following randomized algorithms: IBE.Setup, IBE.Extract, IBE.Encrypt, and IBE.Decrypt.

$\mathsf{IBE.Setup}(1^\lambda)$**:** takes a security parameter λ and returns system parameters and master key. System parameters are publicly available, while the master key is known only to the key generation service. System parameters include descriptions of finite message space $\mathcal{M}$ and a finite ciphertext space $\mathcal{C}$.

$\mathsf{IBE.Extract}(\text{params}, \text{mk}, \text{id})$**:** takes system parameters, master key, and a string $\text{id} \in \{0,1\}^*$, returns a private key sk for the given id. The id is used as a public key with sk as corresponding private key.

$\mathsf{IBE.Encrypt}(\text{params}, M, \text{id})$**:** takes system parameters, id, and $M \in \mathcal{M}$, returns ciphertext $C \in \mathcal{C}$.

$\mathsf{IBE.Decrypt}(\text{params}, C, \text{sk})$**:** takes system parameters, sk, and $C \in \mathcal{C}$, returns $M \in \mathcal{M}$.

The algorithms must satisfy a consistency constraint, with sk being a private key for identity id:

$$\forall M \in \mathcal{M} : \mathsf{IBE.Decrypt}(\text{params}, C, \text{sk}) = M \\ \text{where } C = \mathsf{IBE.Encrypt}(\text{params}, M, \text{id}).$$

IBE Correctness We require that a ciphertext can always be decrypted with the secret key of the identity for which the message was initially encrypted.

Definition 7 (IBE Correctness)**.** For all $M \in \mathcal{M}$, $\text{id} \in \{0,1\}^*$ and a pair (mk, params) generated by $\mathsf{IBE.Setup}(1^\lambda)$, for any $\text{sk}_{\text{id}} \leftarrow_\$ \mathsf{IBE.Extract}(\text{params}, \text{mk}, \text{id})$ and $C \leftarrow_\$ \mathsf{IBE.Encrypt}(\text{params}, M, \text{id})$, then $\mathsf{IBE.Decrypt}(\text{params}, C, \text{sk}_{\text{id}}) = M$.

IBE Security The scheme is chosen plaintext attack secure (IND-CPA) assuming BDH assumption is hard in the used groups generated by $\mathcal{G}$. The IND-CPA experiment is presented in Experiment 1.

$\mathsf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{IBE}}(1^\lambda)$:
$(\mathrm{mk}, \mathrm{params}) \leftarrow_\$ \mathsf{IBE.Setup}(1^\lambda)$
$(M_0, M_1, \mathrm{id}^*, \mathrm{st}) \leftarrow_\$ \mathcal{A}^{\mathsf{IBE.Extract}(\mathrm{mk},\cdot)}(\mathrm{params})$
$b \leftarrow_\$ \{0,1\}$
$C^* \leftarrow_\$ \mathsf{IBE.Encrypt}(\mathrm{params}, M_b, \mathrm{id}^*)$
$b^* \leftarrow_\$ \mathcal{A}^{\mathsf{IBE.Extract}(\mathrm{mk},\cdot)}(\mathrm{st}, C^*)$
return 1, if $b^* = b$
return 0

Experiment 1: IND-CPA experiment for IBE

Definition 8 (IBE Security)**.** The advantage of adversary $\mathcal{A}$ in the IND-CPA experiment $\mathsf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{IBE}}(\lambda)$ is defined as

$$\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{IBE}}(\lambda) = \left| \Pr\left[\mathsf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{IBE}}(\lambda) = 1\right] - \frac{1}{2} \right|.$$

An IBE is IND-CPA secure if $\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{IBE}}(\lambda)$ in λ is negligible for all adversaries $\mathcal{A}$.

Shamir was incapable of proposing a feasible implementation of IBE, but only provided concrete implementation details of the analogue signature scheme. In 2003, Boneh and Franklin [BF03] came up with the construction of a usable IBE scheme.

3.2.1 Boneh-Franklin Identity-Based Encryption

Boneh and Franklin [BF03] realized the first practical IBE scheme based on bilinear pairings. Moreover they also presented a security definition of IBE. The scheme described here and used later in the work is named *BasicIdent*.

Analogous to the definition of IBE, Boneh-Franklin IBE (BFIBE) scheme is defined by four algorithms: BFIBE.Setup, BFIBE.Extract, BFIBE.Encrypt, and BFIBE.Decrypt.

BFIBE.Setup(1^λ)**:** Security parameter $\lambda \in \mathbb{Z}^+$ is given as input to the BDH generator. The public key is computed as $\mathrm{pk} = g_1^{\mathrm{mk}}$, $g_1 \in \mathbb{G}$ being a random generator, and a random integer $\mathrm{mk} \in \mathbb{Z}_p^*$ as a master key. Bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$ is defined, with p being the order of groups $\mathbb{G}$ and $\mathbb{G}_t$. Two cryptographic hash functions are defined, G_1 mapping an arbitrary number of bytes to a point in $\mathbb{G}$, and G_2 mapping a point from $\mathbb{G}_t$ to a digest of length n. The system parameters are $\mathrm{params} = (p, \mathbb{G}, \mathbb{G}_t, e, n, g_1, \mathrm{pk}, G_1, G_2)$. It is assumed all algorithms described below implicitly receive these parameters.

BFIBE.Extract$(\mathrm{mk}, \mathrm{id})$**:** The provided string id is hashed: $Q_{\mathrm{id}} = G_1(\mathrm{id}) \in \mathbb{G}$, and the private key is returned as $\mathrm{sk}_{\mathrm{id}} = Q_{\mathrm{id}}^{\mathrm{mk}}$. Individual private keys for each identity can be reproduced as long as the master key exists.

BFIBE.Encrypt(M, id)**:** Identity string id is again hashed, producing Q_{id} that is paired with the public key as $g_{\mathrm{id}} = e(Q_{\mathrm{id}}, \mathrm{pk}) \in \mathbb{G}_t$. After choosing a random $r \in \mathbb{Z}_p^*$, the ciphertext of message M is a pair $C = (g_1^r, M \oplus G_2(g_{\mathrm{id}}^r))$.

BFIBE.Decrypt$(\mathrm{sk}_{\mathrm{id}}, C)$**:** With a private key $\mathrm{sk}_{\mathrm{id}}$, a message M is retrieved from the ciphertext of form $C = (u, v)$ as $M = v \oplus G_2(e(sk_{\mathrm{id}}, u))$. This is where the bilinearity of a map is imperative, as then g_{id}^r from the **BFIBE.Encrypt** algorithm can be reformulated as $g_{\mathrm{id}}^r = e(Q_{\mathrm{id}}, \mathrm{pk})^r = e(Q_{id}^{\mathrm{mk}}, g_1^r) = e(\mathrm{sk}_{\mathrm{id}}, g_1^r)$. Thus, retrieval of original message is possible due to same values being used for encryption and decryption.

Theorem 1 ([BF03])**.** *Let hash functions G_1 and G_2 be random oracles. The Boneh-Franklin IBE scheme is then IND-CPA secure assuming BDH assumption is hard in the groups generated by $\mathcal{G}$. Suppose there is an IND-CPA adversary $\mathcal{A}$ with advantage $\epsilon(\lambda)$. Assuming $\mathcal{A}$ makes at most q_E private key extraction queries and at most q_{G_2} queries to random oracle G_2. There is then an adversary $\mathcal{B}$ that solves BDH problem in groups generated by $\mathcal{G}$ with advantage:*

$$\mathsf{Adv}_{\mathcal{B},\mathcal{G}}^{\mathsf{BDH}}(\lambda) \geq \frac{2\epsilon(\lambda)}{\epsilon(1+q_E)\cdot q_{G_2}}.$$

3.2.2 Hierarchical Identity Based Encryption

Organizational structures are rarely horizontal. Generally, one group of entities has authority over another group, that group has authority over some other

group, et cetera. This implies that not all identites should be trusted with same level of power. Hierarchical Identity Based Encryption (HIBE) is best portrayed as a full binary tree. In a full binary tree of depth l, every node up to level $l-1$ has exactly two child nodes. Each node has a capacity to hold one private key that can recursively generate private keys of its child nodes. A node is incapable of generating private keys for its parent or sibling nodes.

Definition 9 (HIBE [HL02])**.** A HIBE is defined by four algorithms: $\mathsf{HIBE.Setup}$, $\mathsf{HIBE.KeyGen}$, $\mathsf{HIBE.Encrypt}$, and $\mathsf{HIBE.Decrypt}$.

$\mathsf{HIBE.Setup}(1^\lambda)$**:** takes a security parameter λ and returns system parameters params and a level-0 secret key sk_ϵ as a master key.

$\mathsf{HIBE.KeyGen}(\mathrm{params}, \mathrm{sk}_{\mathrm{id}'}, \mathrm{id})$**:** takes params, an identity $\mathrm{id} = (I_1, \ldots, I_k)$ at level k, and a private key $\mathrm{sk}_{\mathrm{id}'}$ where $\mathrm{id}' = (I_1, \ldots, I_{k-1})$. Returns a private key $\mathrm{sk}_{\mathrm{id}}$ for identity id.

$\mathsf{HIBE.Encrypt}(\mathrm{params}, M, \mathrm{id})$**:** takes params, a message M, and an id. Returns a ciphertext C.

$\mathsf{HIBE.Decrypt}(\mathrm{params}, \mathrm{sk}_{\mathrm{id}}, \mathrm{id}, C)$**:** takes params, a ciphertext C, an id, and a private key $\mathrm{sk}_{\mathrm{id}}$. Returns a message M.

HIBE Correctness Identical to IBE, we require that a ciphertext can always be decrypted with the secret key of the identity for which the message was initially encrypted.

Definition 10 (HIBE Correctness)**.** For all $l \in \mathbb{N}$, $M \in \mathcal{M}$, $\mathrm{id} \in \{0,1\}^*$ and a pair (mk, params) generated by the $\mathsf{HIBE.Setup}(1^\lambda, l)$ algorithm, for any private key $\mathrm{sk}_{\mathrm{id}} \leftarrow\!\!\$\ \mathsf{HIBE.KeyGen}(\mathrm{sk}_{\mathrm{id}'}, \mathrm{id})$ and ciphertext $C \leftarrow\!\!\$\ \mathsf{HIBE.Encrypt}(M, \mathrm{id})$, it holds that $\mathsf{HIBE.Decrypt}(\mathrm{sk}_{\mathrm{id}}, C) = M$.

HIBE Security Chosen plaintext security for HIBE is defined under a chosen identity attack where the adversary is allowed to adaptively choose the identity on which it will be challenged. The IND-CPA experiment is presented in Experiment 2.

$\mathsf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{HIBE}}(1^\lambda, l)$:
$(\mathrm{mk}, \mathrm{params}) \leftarrow \mathsf{HIBE.Setup}(1^\lambda, l)$
$(M_0, M_1, \mathrm{id}^*, \mathrm{st}) \leftarrow \mathcal{A}^{\mathsf{HIBE.KeyGen}(\mathrm{mk},\cdot)}(\mathrm{params})$
$b \leftarrow\!\!\$\ \{0,1\}$
$C^* \leftarrow \mathsf{HIBE.Encrypt}(\mathrm{params}, M_b, \mathrm{id}^*)$
$b^* \leftarrow \mathcal{A}^{\mathsf{HIBE.KeyGen}(\mathrm{mk},\cdot)}(\mathrm{st}, C^*)$
return 1, if $b^* = b$ and $\mathcal{A}$ is valid
return 0

Experiment 2: IND-CPA experiment for HIBE

Definition 11 (HIBE Security)**.** The advantage of adversary $\mathcal{A}$ in the IND-CPA experiment $\mathsf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{HIBE}}(1^\lambda, l)$ is defined as

$$\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{HIBE}}(1^\lambda, l) = \left| \Pr\left[\mathsf{Exp}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{HIBE}}(1^\lambda, l) = 1\right] - \frac{1}{2} \right|.$$

A HIBE is IND-CPA secure if for any $l > 0$, $\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathcal{A},\mathsf{HIBE}}(1^\lambda, l)$ in λ is negligible for all adversaries $\mathcal{A}$.

Boneh, Boyen, and Goh [BBG05] presented a HIBE scheme with constant ciphertext size and decryption cost. The private key size of the highest authority is proportional to l, but the size of keys decreases as the depth increases.

Interface of the Boneh-Boyen-Goh HIBE (BBGHIBE) is similar to the one of BFIBE, having BBGHIBE.Setup, BBGHIBE.KeyGen, BBGHIBE.Encrypt, and BBGHIBE.Decrypt algorithms. From a black box perspective the two schemes look similar, with some restrictions and additions. The most evident differences are that identities have a strict naming convention and a maximum length of l, and that a master key is not the only key with key generation capacity.

BBGHIBE.Setup$(1^\lambda, l)$**:** $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$ is a bilinear map. System parameters are $\mathrm{params} = (g, g_1, g_2, g_3, h_1, \ldots, h_l)$, where $g, g_2, g_3, h_1, \ldots, h_l \in \mathbb{G}$ are random elements, and $g_1 = g^r$ with random $r \in \mathbb{Z}_p$. The master key is stored as $\mathrm{mk} = g_2^r$. It is assumed all algorithms described below implicitly receive system parameters.

BBGHIBE.KeyGen($\mathrm{sk}_{\mathrm{id}'}, \mathrm{id}$): A private key for identity $\mathrm{id} = (I_1, \ldots, I_k) \in (\mathbb{Z}_p^*)^k$ is generated as

$$\mathrm{sk}_{\mathrm{id}} = \left(g_2^r \cdot \left(g_3 \cdot h_1^{I_1} \cdots h_k^{I_k}\right)^s, g^s, h_{k+1}^s, \ldots, h_l^s\right), k \leq l,$$

where $s \in \mathbb{Z}_p$ is a random value. With a private key of form $\mathrm{sk}_{\mathrm{id}' \in (\mathbb{Z}_p^*)^{k-1}} = (a, b, c_k, \ldots, c_l)$, the private key for $\mathrm{id} \in (\mathbb{Z}_p^*)^k$ is derived as

$$\mathrm{sk}_{\mathrm{id} \in (\mathbb{Z}_p^*)^k} = \left(a \cdot c_k^{I_k} \cdot \left(g_3 \cdot h_1^{I_1} \cdots h_k^{I_k}\right)^u, b \cdot g^u, c_{k+1} \cdot h_{k+1}^u, \ldots, c_l \cdot h_l^u\right),$$

where $u \in \mathbb{Z}_p$ is random. Set $(c_{k+1}, \ldots, c_l)$ gets smaller as k increases, proving the keys are shorter deeper in the tree.

BBGHIBE.Encrypt(M, id): A message $M \in \mathbb{G}_t$ is encrypted for the identity $\mathrm{id} = (I_1, \ldots, I_k)$ as

$$C = \left(e(g_1, g_2)^t \cdot M, g^t, \left(g_3 \cdot h_1^{I_1} \cdots h_k^{I_k}\right)^t\right) = \left(a', b', c'\right)$$

with random $t \in \mathbb{Z}_p$.

BBGHIBE.Decrypt($\mathrm{sk}_{\mathrm{id}}, C$): To decrypt a ciphertext C with the private key $\mathrm{sk}_{\mathrm{id}}$ of form $(a, b, c_{k+1}, \ldots, c_l)$, the algorithm outputs

$$M = a' \cdot \frac{e\left(b, c'\right)}{e\left(b', a\right)}.$$

This again holds due to map bilinearity.

Definition 12 (BDHI Problem [BB04])**.** Let g be a bilinear group of prime order p, $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_t$ a bilinear map, g a generator of $\mathbb{G}$, and $r \in \mathbb{Z}_p^*$. The q-th Bilinear Diffie-Hellman Inversion problem denoted as q-BDHI, is as follows: given the tuple $(g, g^r, g^{(r^2)}, \ldots, g^{(r^q)})$, compute $e(g, g)^{1/r}$.

Definition 13 (BDHI Assumption)**.** We define the advantage of an algorithm $\mathcal{A}$ solving q-BDHI as:

$$\mathsf{Adv}_{\mathcal{A},\mathcal{G}}^{\mathsf{q-BDHI}} = \Pr\left[e(g, g)^{1/r} \leftarrow \mathcal{A}\left(g, g^r, g^{(r^2)}, \ldots, g^{(r^q)}\right)\right] \geq \epsilon.$$

We say that the (t, q, ϵ)-BDHI assumption holds in $\mathbb{G}$ if no t-time algorithm has advantage at least ϵ in solving the q-BDHI problem in $\mathbb{G}$.

Definition 14 (Weak BDHI Problem [BBG05])**.** Let g and h be two random generators of $\mathbb{G}$, and $s \in \mathbb{Z}_p^*$. The q-wBDHI problem is as follows: given the tuple $(g, h, g^s, g^{(s^2)}, \ldots, g^{(s^q)})$, compute $e(g,h)^{(s^{q+1})}$.

Definition 15 (Weak BDHI Assumption)**.** We define the advantage of an algorithm $\mathcal{A}$ solving q-wBDHI as:

$$\mathsf{Adv}_{\mathcal{A},\mathcal{G}}^{\mathsf{q-wBDHI}} = \Pr\left[e(g,h)^{(s^{q+1})} \leftarrow \mathcal{A}\left(g, h, g^s, g^{(s^2)}, \ldots, g^{(s^q)}\right)\right] \geq \epsilon.$$

We define the advantage of an algorithm $\mathcal{B}$, that outputs $b \in \{0,1\}$, in solving decisional q-wBDHI as:

$$\mathsf{Adv}_{\mathcal{A},\mathcal{G}}^{\mathsf{Dec.q-wBDHI}} = \Big|\Pr\left[\mathcal{B}\left(g, h, g^s, g^{(s^2)}, \ldots, g^{(s^q)}, e(g,h)^{(s^{q+1})}\right) = 0\right] - \Pr\left[\mathcal{B}\left(g, h, g^s, g^{(s^2)}, \ldots, g^{(s^q)}, T\right) = 0\right]\Big| \geq \epsilon.$$

We say that the (Decision) (t, q, ϵ)-wBDHI assumption holds in $\mathbb{G}$ if no t-time algorithm has advantage at least ϵ in solving the (Decision) q-wBDHI problem in $\mathbb{G}$.

Theorem 2 ([BBG05])**.** *Let $\mathbb{G}$ be a bilinear group of prime order p. Suppose the Decision (t, q, ϵ)-wBDHI assumption holds in $\mathbb{G}$. Then the q-BBGHIBE system is (t', p_S, ϵ)-selective identity, chosen plaintext (IND-sID-CPA) secure for arbitrary q, p_S, and $t' < t - \theta(\tau q p_S)$, where τ is the maximum time for an exponentiation in $\mathbb{G}$.*

Hierarchical Identity Based Key Encapsulation

Key encapsulation methods are used for the construction of hybrid encryption schemes. For a secure exchange of a symmetric key via an insecure channel, the key is first encrypted using asymmetric encryption. A Hierarchical identity based key encapsulation mechanism (HIBKEM) is such a scheme built with HIBE.

Definition 16 (HIBKEM)**.** A Hierarchical identity based key encapsulation mechanism is defined as a tuple of algorithms HIBKEM = (HIBKEM.KGen, HIBKEM.Del, HIBKEM.Enc, HIBKEM.Dec) with the following syntax:

$\mathsf{HIBKEM.KGen}(1^\lambda)$**:** takes a security parameter λ and returns a public key pk and a level-0 secret key sk_ϵ.

$\mathsf{HIBKEM.Del}(\mathrm{sk}_{\mathrm{id}'}, \mathrm{id})$**:** takes a secret key for identity id' at level $k-1$, and an identity at level k. It returns a secret key $\mathrm{sk}_{\mathrm{id}}$ for identity id.

$\mathsf{HIBKEM.Enc}(\mathrm{pk}, \mathrm{id})$**:** takes a public key pk and an identity id, and returns a pair (C, K) where C is encapsulated K.

$\mathsf{HIBKEM.Dec}(\mathrm{sk}_{\mathrm{id}}, C)$**:** takes a private key $\mathrm{sk}_{\mathrm{id}}$ and an encapsulated key C, and returns a key K or a notification of error.

HIBKEM Security The construction of scheme from [DJS+18] requires only one-wayness under selective-ID and chosen-plaintext attacks (OW-sID-CPA) security notion. The OW-sID-CPA experiment is presented in Experiment 3.

Definition 17 (HIBKEM Security)**.** The advantage of adversary $\mathcal{A}$ in the OW-sID-CPA experiment $\mathsf{Exp}^{\mathsf{OW-sID-CPA}}_{\mathcal{A},\mathsf{HIBKEM}}(\lambda)$ is defined as

$$\mathsf{Adv}^{\mathsf{OW-sID-CPA}}_{\mathcal{A},\mathsf{HIBKEM}}(\lambda) = \Pr\left[\mathsf{Exp}^{\mathsf{OW-sID-CPA}}_{\mathcal{A},\mathsf{HIBKEM}}(\lambda) = 1\right].$$

A HIBKEM is OW-sID-CPA secure if $\mathsf{Adv}^{\mathsf{OW-sID-CPA}}_{\mathcal{A},\mathsf{HIBKEM}}(\lambda)$ in λ is negligible for all adversaries $\mathcal{A}$.

3.3 Puncturable Encryption

Green and Miers [GM15] came up with a new form of forward secure encryption named puncturable encryption. The problem they tried to solve is providing forward secrecy in asynchronous messaging systems. In such systems, there is no requirement for the receiver and the sender to be online simultaneously and messages can be delayed for longer periods of time. Their approach was creating a fine-grained revocation of decrypting capability for specific messages.

As a form of tag-based encryption [MRY04], each message in their scheme is marked with a tag uniquely identifying the message. The receiver can modify the decryption key to make it unusable for decrypting messages with specific tag whilst still being able to decrypt other messages. The public key does not change, nor does the other party have to be notified. Key modification of such a type is called *key puncturing*, therefore the term *puncturable encryption*.

$\mathsf{Exp}^{\mathsf{OW-sID-CPA}}_{\mathcal{A},\mathsf{HIBKEM}}(\lambda)$:
$(\mathrm{id}^*, \mathrm{st}) \leftarrow\!\!\$\ \mathcal{A}(1^\lambda)$
$(\mathrm{pk}, \mathrm{sk}_\epsilon) \leftarrow\!\!\$\ \mathsf{HIBKEM.KGen}(1^\lambda)$
$(C, K) \leftarrow\!\!\$\ \mathsf{HIBKEM.Enc}(\mathrm{pk}, \mathrm{id}^*)$
$K^* \leftarrow\!\!\$\ \mathcal{A}(\mathrm{pk}, C, \mathrm{st})$
return 1, if $K^* = K$
return 0

Experiment 3: OW-sID-CPA experiment for HIBKEM

Definition 18 (Puncturable Encryption)**.** The puncturable encryption scheme is a tuple of algorithms PPKE = (PPKE.KeyGen, PPKE.Puncture, PPKE.Encrypt, PPKE.Decrypt) with the following syntax:

PPKE.KeyGen$(1^\lambda, d)$**:** takes security parameter λ and a maximum number of tags per ciphertext d. Returns a public key pk and initial secret key sk_0.

PPKE.Puncture$(\mathrm{pk}, \mathrm{sk}_{i-1}, t)$**:** takes a public key pk, secret key sk_{i-1} at time $i-1$, and a tag t. Outputs an updated secret key sk_i capable of decrypting all the same ciphertexts as sk_{i-1}, except the ones encrypted with t.

PPKE.Encrypt$(\mathrm{pk}, M, t_1, \ldots, t_d)$**:** takes a public key pk, a message M, and a list of tags $t_1, \ldots, t_d$. Outputs a ciphertext C.

PPKE.Decrypt$(\mathrm{pk}, \mathrm{sk}_i, C, t_1, \ldots, t_d)$**:** takes a public key pk, a secret key sk_i, a ciphertext C, and a list of tags $t_1, \ldots, t_d$. Outputs a message M or $\perp$ if decryption fails.

In recent years, puncturable encryption has found its use in different applications. To mention some, these include forward secure key exchange protocols [GHJ+17; DJS+18], where in both constructions puncturing is not based on identification by tags, but instead on a specific ciphertext. In [BMO17] they use it to ensure their searchable symmetric encryption scheme is backward secret by using puncturable encrpytion as a tool for secure deletion. Puncturable encryption is also used for devising a chosen-ciphertext secure fully homomorphic encryption [CRR+17] and a forward secret proxy re-encryption scheme [DKL+18]. In the latter, the authors present a concept of *fully puncturable encryption* extending the puncturing from [GM15] in a way that it is also possible to puncture a secret key with a tag making the secret key only capable of decrypting

ciphertexts with this tag.

3.4 Bloom Filter Encryption

Bloom Filter Encryption (BFE) [DJS+18] combines IBE schemes with Bloom filters introducing a replay attack resistant and forward secure encryption scheme usable in real-life scenarios. Decryption keys in this scheme are pregenerated and stored in an indexed collection (e.g. array). The IBE identities for which the keys are derived correspond to the index at which the key is stored inside a collection. Each key is used once before it gets destroyed, strictly enforcing the scheme's security properties. Alongside with keys, a Bloom filter of the same size is used for mapping data to particular keys.

3.4.1 Basic Bloom Filter Encryption

To generate n keys, a Bloom filter of size m with h hash functions is first initialized in an all-zero state. The BFIBE.Setup is then triggered producing a master key followed by m BFIBE.Extract calls filling up all slots of BFE key collection. The master key is destroyed immediately after collection has been finalized, preventing regeneration of the keys later in time.

As for 0-RTT application only key encapsulation mechanism (KEM) is needed for symmetric key exchange, BFE is formulated as a puncturable key encapsulation mechanism (PKEM).

Definition 19 (PKEM)**.** A PKEM scheme with key space $\mathcal{K}$ is a tuple of algorithms PKEM = (PKEM.KGen, PKEM.Punc, PKEM.Enc, PKEM.Dec). The algorithms are as follows:

PKEM.KGen$(1^\lambda, m, h)$**:** Given security parameter λ and parameters m and h, it outputs a secret key sk and public key pk.

PKEM.Punc(sk, C)**:** Given the secret key sk and a ciphertext C, it outputs updated secret key sk'.

PKEM.Enc(pk)**:** Given a public key pk, it outputs a ciphertext C and a key K.

PKEM.Dec(sk, C)**:** Given a secret key sk and a ciphertext C, it outputs the key K or $\perp$ if decapsulation fails.

$\mathsf{Exp}^{\mathsf{T}}_{\mathcal{A},\mathsf{PKEM}}(\lambda, m, h)$:
  $(\mathrm{sk}, \mathrm{pk}) \leftarrow_{\$} \mathsf{PKEM.KGen}(1^{\lambda}, m, h)$
  $(C^*, K_0) \leftarrow_{\$} \mathsf{PKEM.Enc}(\mathrm{pk})$
  $\mathcal{Q} \leftarrow \emptyset$
  $K_1 \leftarrow_{\$} \mathcal{K}$
  $b \leftarrow_{\$} \{0,1\}$
  $b^* \leftarrow_{\$} \mathcal{A}^{\mathcal{O},\mathsf{PKEM.Punc}(\mathrm{sk},\cdot),\mathsf{Corr}}(\mathrm{pk}, C^*, K_b)$
    where $\mathcal{O} \leftarrow \{\mathsf{PKEM.Dec}'(\mathrm{sk},\cdot)\}$ if $T = \text{IND-CCA}$ and $\mathcal{O} \leftarrow \emptyset$ otherwise.
    $\mathsf{PKEM.Dec}'(\mathrm{sk}, C)$ behaves as $\mathsf{PKEM.Dec}$ but returns $\perp$ if $C = C^*$
    $\mathsf{PKEM.Punc}(\mathrm{sk}, C)$ runs $\mathrm{sk} \leftarrow_{\$} \mathsf{PKEM.Punc}(\mathrm{sk}, C)$ and $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{C\}$
    Corr returns sk if $C^* \in \mathcal{Q}$ and $\perp$ otherwise
  return 1, if $b^* = b$
  return 0

Experiment 4: IND-CPA and IND-CCA experiments for PKEM, $T \in \{\text{IND-CPA}, \text{IND-CCA}\}$

PKEM Correctness We require that a ciphertext can always be decapsulated with unpunctured secret keys. In addition, when decapsulating a ciphertext with punctured secret keys, the probability that decapsulation fails is bounded by a non-negligible function of scheme's parameters m and h.

Definition 20 (PKEM Correctness)**.** For all $\lambda, m, h, \in \mathbb{N}$, together with any $(\mathrm{sk}, \mathrm{pk}) \leftarrow_{\$} \mathsf{PKEM.KGen}(1^{\lambda}, m, h)$ and $(C, K) \leftarrow_{\$} \mathsf{PKEM.Enc}(\mathrm{pk})$, then we have $\mathsf{PKEM.Dec}(\mathrm{sk}, C) = K$. Additionally, for any number of invocations of function $\mathrm{sk}' \leftarrow_{\$} \mathsf{PKEM.Punc}(\mathrm{sk}, C')$ determined by m, h, for any $C' \neq C$ it holds that $\Pr[\mathsf{PKEM.Dec}(\mathrm{sk}', C) = \perp] \leq \mu(m, h)$.

PKEM Security Two security notions defined for PKEM are IND-CPA and IND-CCA, experiments for both are defined in Experiment 4.

Definition 21 (PKEM Security)**.** Let $T \in \{\text{IND-CPA}, \text{IND-CCA}\}$, where the advantage of adversary $\mathcal{A}$ in the T experiment $\mathsf{Exp}^{\mathsf{T}}_{\mathcal{A},\mathsf{PKEM}}(\lambda, m, h)$ is defined as

$$\mathsf{Adv}^{\mathsf{T}}_{\mathcal{A},\mathsf{PKEM}}(\lambda, m, h) = \left| \Pr\left[\mathsf{Exp}^{\mathsf{T}}_{\mathcal{A},\mathsf{PKEM}}(\lambda, m, h) = 1\right] - \frac{1}{2} \right|.$$

A PKEM is T secure if for any $m, h > 0$, $\mathsf{Adv}^{\mathsf{T}}_{\mathcal{A},\mathsf{PKEM}}(\lambda, m, h)$ in λ is negligible for all adversaries $\mathcal{A}$.

Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ be a bilinear map of prime groups $\mathbb{G}_1$ and $\mathbb{G}_2$, with generators g_1 and g_2 of each group respectively. BFE is defined as tuple of algorithms BFE = (BFE.KGen, BFE.Enc, BFE.Punc, BFE.Dec) defined as:

BFE.KGen$(1^\lambda, m, h)$: A Bloom filter $(H, T) \leftarrow \mathsf{BF.Setup}(m, h)$ is initialized and random $r \in \mathbb{Z}_p$ is chosen. Let $G_1 : \mathbb{N} \to \mathbb{G}_2$ and $G_2 : \mathbb{G}_t \to \{0,1\}^\lambda$ be cryptographic hash functions. Secret key is defined as $\mathrm{sk} = (T, (G_1(i)^r)_{i\in[m]})$ and public key as $\mathrm{pk} = (g_1^r, H)$.

BFE.Enc(pk): A random key $K \leftarrow \{0,1\}^\lambda$ and $s \in \mathbb{Z}_p$ are generated. It computes indices $i_j = H_j(g_1^s)_{j\in[h]}; H_j \in H$. Ciphertext is defined as

$$C = (g_1^s, (G_2(e(g_1^r, G_1(i_j))^s \oplus K)_{j\in[h]})).$$

Finally, the algorithm outputs a pair (C, K).

BFE.Punc(sk, C): Value g_1^s from ciphertext C is used as a tag when inserting to T, $T' \leftarrow \mathsf{BF.Insert}(H, T, g_1^s)$. With a secret key of form $\mathrm{sk} = (T, sk[i])_{i\in[m]}$, the updated secret key is computed as

$$sk'[n] = \begin{cases} sk[n] & \text{if } T'[i] = 0, \\ \bot & \text{if } T'[i] = 1. \end{cases}$$

A pair $\mathrm{sk}' = (T', (\mathrm{sk}'[i])_{i\in[m]})$ is returned to replace the existing secret key. Figure 3.1 illustrates the state of sk after a single key puncturing.

BFE.Dec(sk, C): Given a secret key $\mathrm{sk} = (T, \mathrm{sk}[i])_{i\in[m]}$ and ciphertext $C = (t, c[j])_{j\in[h]}$, if $\mathsf{BF.Query}(H, T, t) = \mathsf{true}$, the ciphertext cannot be decrypted and $\bot$ is returned. This happens if a ciphertext with the same tag t had prevously been decrypted, or in the case of a false positive query. If a query returns false, it means there is at least one key in the key collection capable of decrypting the ciphertext. The first index $i' = i[j] \in [i]$ is taken for which $\mathrm{sk}[i'] \neq \bot$ and as a result of successful decryption the key of form

$$K = c[j] \oplus G_2(e(t, \mathrm{sk}[i']))$$

is returned.

0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
s_0	s_1	s_2		s_4	s_5		s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}		s_{17}	s_{18}	s_{19}

Figure 3.1: BFE's Bloom filter and accompanying key collection after single key puncturing, $h = 3$

Basic BFE Security IND-CPA security for Basic BFE is based on Bilinear Computational Diffie-Hellman (BCDH) in groups generated with $\mathcal{G}$.

Definition 22 (BCDH assumption)**.** The advantage of adversary $\mathcal{A}$ in solving the BCDH problem is defined as

$$\mathsf{Adv}^{\mathsf{BCDH}}_{\mathcal{A},\mathcal{G}}(\lambda) = \Pr\left[e(g_1, h_2)^{rs} = \mathcal{A}(\text{params}, g_1^r, g_1^s, g_2^s, h_2)\right],$$

with $\text{params} = (p, e, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2) \leftarrow\!\!\$\, \mathcal{G}(1^\lambda)$, $(\text{params}, g_1^r, g_1^s, g_2^s, h_2) \leftarrow\!\!\$\, \mathbb{G}_1^2 \times \mathbb{G}_2$.

Theorem 3 ([DJS+18])**.** *Assuming that the BCDH assumptions holds, BFE is a IND-CPA secure PKEM. More precisely, from each efficient adversary $\mathcal{B}$ against IND-CPA of BFE that sends q queries to random oracle G_2, an efficient adversary $\mathcal{A}$ against BCDH is defined as*

$$\mathsf{Adv}^{\mathsf{BCDH}}_{\mathcal{A},\mathcal{G}}(\lambda) \geq \frac{\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathcal{B},\mathsf{PKEM}}(\lambda, m, h)}{hq}.$$

The Fujisaki-Okamoto (FO) transformation [FO99] is used to acquire an adaptive chosen ciphertext attack secure (IND-CCA) scheme. We shortly sketch how the FO can be applied to BFE:

IND-CCA-secure Construction If PKEM has separable randomness, this means $(C, K) \leftarrow \mathsf{BFE.Enc}(\text{pk}) = \mathsf{BFE.Enc}(\text{pk}; (r, K))$ for uniformly random $(r, K) \leftarrow \{0,1\}^{p+\lambda}$. Let $R : \{0,1\}^* \to \{0,1\}^{p+\lambda}$ be a hash function, and the following are algorithms that differ from the ones of IND-CPA scheme.

$\mathsf{BFE.Enc}'(\text{pk})$**:** Random key $K \leftarrow \{0,1\}^\lambda$ is generated. Then it computes $(r, K') \leftarrow R(K)$, runs $(C, K) \leftarrow \mathsf{BFE.Enc}(\text{pk}; (r, K))$, and returns (C, K').

$\mathsf{BFE.Dec}'(\mathrm{sk}, C)$**:** Original key K is retrieved by $K \leftarrow \mathsf{BFE.Dec}(\mathrm{sk}, C)$. If $K = \perp$ then $\perp$ is returned, otherwise it computes $(r, K') \leftarrow R(K)$. If $(C, K) = \mathsf{BFE.Enc}(\mathrm{pk}; (r, K))$ it returns K', otherwise it returns $\perp$.

Theorem 4 ([DJS+18, Theorem 3])**.** *After applying the FO transformation to BFE, BFE is a IND-CCA secure PKEM.*

Correctness Error The correctness error of the scheme corresponds to the correctness of the Bloom filter, together with the statistically small probability that two different ciphertexts share the same randomness r. As noted in Section 2.3.3, the false positive probability of the Bloom filter is $\Pr_{\text{false}} \approx (1 - e^{-\frac{hn}{m}})^h \leq 2^{-h}$. If two independent ciphertexts would share the same r, this would yield the same tag t, leading to the collision of the two ciphertexts in the Bloom filter. As r is uniformly random, the probability for this to happen is upper bounded by n/p, where n is the number of ciphertexts. From this, a correctness error of Basic BFE is given as approximately $2^{-h} + n/p$.

3.4.2 Time-Based Bloom Filter Encryption

After using up all the keys from the BFE key collection, a new instance of the BFE scheme has to be initialized. This results with a new public key that has to be shared with other parties. Having a longer-lasting public key increases the scheme's setup time resulting with a longer secret key. Time-Based Bloom Filter Encryption (TBBFE) is an extension of the Basic BFE scheme in which the lifetime of one BFE instance is divided into multiple time slots. The key collection of each time slot can be calculated independently at different times, keeping the same public key for all intervals.

Similar to formulating a Basic BFE scheme as PKEM, a Time-Based BFE is formulated as a puncturable forward secret key encapsulation (PFSKEM) [GHJ+17] defined below. Different from the PKEM defined in Definition 19, algorithms are defined with a specific time interval context, and additional algorithm $\mathsf{PFSKEM.PuncInt}$ is introduced for puncturing a secret key with respect to an entire time interval.

Definition 23 (PFSKEM)**.** A puncturable forward secret key encapsulation scheme is defined as a tuple consisting of five probabilistic polynomial

time algorithms PFSKEM = (PFSKEM.KGen, PFSKEM.Enc, PFSKEM.PuncCtx, PFSKEM.Dec, PFSKEM.PuncInt).

PFSKEM.KGen$(1^\lambda, m, h, t)$**:** Given a security parameter λ, parameters m and h, and a number of time intervals t, it outputs a secret key sk and a public key pk.

PFSKEM.Enc(pk, τ)**:** Given a public key pk and time interval τ, it outputs a ciphertext C and a key K.

PFSKEM.PuncCtx(sk, τ, C)**:** Given a secret key sk, time interval τ, and a ciphertext C, it outputs updated secret key sk'.

PFSKEM.Dec(sk, τ, C)**:** Given a secret key sk, time interval τ, and a ciphertext C, it outputs the key K or $\perp$ if decapsulation fails.

PFSKEM.PuncInt(sk, τ)**:** Given a secret key sk and time interval τ, it outputs an updated secret key sk' for time interval $\tau + 1$.

PFSKEM Correctness The correctness definition of PKEM is extended to take into account time intervals.

Definition 24 (PFSKEM Correctness)**.** For all $\lambda, m, h, t, \in \mathbb{N}$, together with any $(\text{sk}, \text{pk}) \leftarrow\!\!\$\ \mathsf{PFSKEM.KGen}(1^\lambda, m, h, t)$, τ^*, and $(C^*, K) \leftarrow\!\!\$\ \mathsf{PFSKEM.Enc}(\text{pk}, \tau^*)$, for any number of invocations of $\text{sk}' \leftarrow\!\!\$\ \mathsf{PFSKEM.PuncCtx}(\text{sk}, \tau, C')$ determined by m, h for any $(C', \tau) \neq (C^*, \tau^*)$, or $\text{sk}' \leftarrow\!\!\$\ \mathsf{PFSKEM.PuncInt}(\text{sk}, \tau)$ for any $\tau \neq \tau^*$, it holds that $\Pr[\mathsf{PFSKEM.Dec}(\text{sk}', \tau^*, C^*) = \perp] \leq \mu(m, h)$.

PFSKEM Security PFSKEM security is defined as a selective time experiment, meaning an adversary has to choose a specific time interval τ^* to attack before seeing the public key. The IND-CPA and IND-CCA experiments are presented in Experiment 5.

Definition 25 (PFSKEM Security)**.** Let $T \in \{\text{IND-CPA}, \text{IND-CCA}\}$, the advantage of adversary $\mathcal{A}$ in the s-T experiment $\mathsf{Exp}^{\mathsf{s-T}}_{\mathcal{A},\mathsf{PFSKEM}}(\lambda, m, h, t)$ is defined as

$$\mathsf{Adv}^{\mathsf{s-T}}_{\mathcal{A},\mathsf{PFSKEM}}(\lambda, m, h, t) = \left| \Pr\left[\mathsf{Exp}^{\mathsf{s-T}}_{\mathcal{A},\mathsf{PFSKEM}}(\lambda, m, h, t) = 1 \right] - \frac{1}{2} \right|.$$

A PFSKEM is s-T secure if for any $m, h, t > 0$, $\mathsf{Adv}^{\mathsf{s-T}}_{\mathcal{A},\mathsf{PKEM}}(\lambda, m, h, t)$ in λ is negligible for all adversaries $\mathcal{A}$.

$\mathsf{Exp}^{\mathsf{s-T}}_{\mathcal{A},\mathsf{PFSKEM}}(\lambda, m, h, t)$:
$\tau^* \leftarrow\!\!\$\ \mathcal{A}(1^\lambda)$
$(\mathrm{sk}, \mathrm{pk}) \leftarrow\!\!\$\ \mathsf{PFSKEM.KGen}(1^\lambda, m, h, t)$
$(C^*, K_0) \leftarrow\!\!\$\ \mathsf{PFSKEM.Enc}(\mathrm{pk}, \tau^*)$
$K_1 \leftarrow\!\!\$\ \mathcal{K}$
$b \leftarrow\!\!\$\ \{0,1\}$
$\mathcal{Q}_C \leftarrow \emptyset$
$\mathcal{Q}_\tau \leftarrow \emptyset$
$b^* \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O},\mathsf{PFSKEM.PuncCtx}(\mathrm{sk},\cdot,\cdot),\mathsf{PFSKEM.PuncInt}(\mathrm{sk},\cdot),\mathsf{Corr}}(\mathrm{pk}, C^*, K_b)$
where $\mathcal{O} \leftarrow \{\mathsf{PFSKEM.Dec}'(\mathrm{sk},\cdot)\}$ if $T = \text{IND-CCA}$ and $\mathcal{O} \leftarrow \emptyset$ otherwise.
$\mathsf{PFSKEM.Dec}'(\mathrm{sk}, \tau, C)$ behaves as $\mathsf{PFSKEM.Dec}$ but returns $\perp$ if $C = C^*$ and $\tau = \tau^*$
$\mathsf{PFSKEM.PuncCtx}(\mathrm{sk}, \tau, C)$ runs $\mathrm{sk} \leftarrow\!\!\$\ \mathsf{PFSKEM.PuncCtx}(\mathrm{sk}, \tau, C)$ and $\mathcal{Q}_C \leftarrow \mathcal{Q}_C \cup \{(C, \tau)\}$
$\mathsf{PFSKEM.PuncInt}(\mathrm{sk}, \tau)$ runs $\mathrm{sk} \leftarrow\!\!\$\ \mathsf{PFSKEM.PuncInt}(\mathrm{sk}, \tau)$ and $\mathcal{Q}_\tau \leftarrow \mathcal{Q}_\tau \cup \{\tau\}$
Corr returns sk if $(C^*, \tau^*) \in \mathcal{Q}$ or $\tau^* \in \mathcal{Q}_\tau$ and $\perp$ otherwise
return 1, if $b^* = b$
return 0

Experiment 5: IND-CPA and IND-CCA experiments for PFSKEM

TBBFE is based on HIBE whose nodes are binary-coded. 0-th level leaf has an empty string identity ϵ, while the identites of i-th level nodes are of length i. Each identity consists of its parent's identity string with additional 0 in case of a left node, or 1 if it is a right node. Thus, a node at level i is an ancestor of a node at level $i + j$, if the former's identity is a prefix of the latter's. Similarly, two nodes at level i are siblings if the first $i - 1$ characters of their identities are equal.

TBBFE capable of 2^t time intervals is formed by a TBBFE tree of depth $t + \log_2 m$, where m is a number of bits in the Bloom filter of a single interval. This number is calculated the same way as in Basic BFE. Keys at level t are time interval keys. Under each of the time interval keys an independent binary tree is generated. Leaves of each independent tree form a BFE key collection.

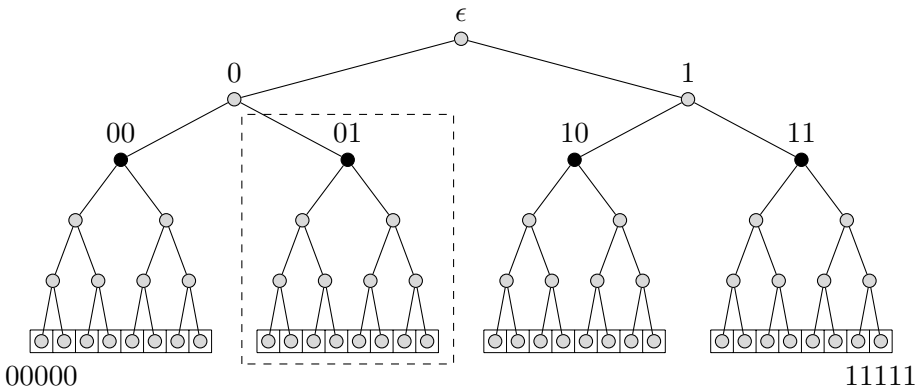


Figure 3.2: Full TBBFE private key tree with 2^2 time intervals. Black nodes are time interval keys, making the subtree above them a time tree with leaves 00, 01, 10, 11. Each of these leaves is a starting node of independent Bloom filter tree whose leaves are stored in a key collection. Dashed line outlines a Bloom filter tree of time interval 01.

Such a tree is depicted in Figure 3.2. It is important to delete intermediate keys as soon as they are not needed anymore to ensure the scheme's security properties. There is no need for all subtrees to be generated at one point in time. It suffices to have a single fully generated subtree with key material for generating other time intervals. Generation of the next time interval subtree can be started before the existing one is fully used, ensuring it is ready when needed without down-time.

$\mathsf{TBBFE.KGen}(1^\lambda, m, h, t)$**:** The first step is the initialization of Bloom filter $(H, T) \leftarrow \mathsf{BF.Setup}(m, h)$ followed by $(\mathrm{pk}, \mathrm{sk}_\epsilon) \leftarrow \mathsf{HIBKEM.KGen}(1^\lambda)$. The private key is divided on time interval keys and Bloom filter keys. Time interval keys are recursively generated for the first time interval with identity 0^t as

$$\mathrm{sk}_{0^d} \leftarrow \mathsf{HIBKEM.Del}(\mathrm{sk}_{0^{d-1}}, 0), \forall d \in [t].$$

Bloom filter keys for the same interval are then recursively generated as

$$\mathrm{sk}_{\mathrm{bloom}} = \left\{\mathrm{sk}_{0^t|d} \leftarrow \mathsf{HIBKEM.Del}(\mathrm{sk}_{0^t}, d)\right\}, \forall d \in [m].$$

The updated time interval keys do not include the key that could be used to generate keys on the same path again. This time interval puncturing is

expressed as

$$\mathrm{sk}_{\mathrm{time}} = \Big\{\mathrm{sk}_{0^{d-1}|1} \leftarrow \mathsf{HIBKEM.Del}(\mathrm{sk}_{0^{d-1}}, 1)\Big\}, \forall d \in [t].$$

The algorithm returns a secret key $\mathrm{sk} = (T, \mathrm{sk}_{\mathrm{bloom}}, \mathrm{sk}_{\mathrm{time}})$, and a public key $\mathrm{PK} = (\mathrm{pk}, (H_j)_{j\in[h]})$.

TBBFE.Enc(pk, τ): Given HIBKEM public key pk and a time interval identifier $\tau \in \{0,1\}^t$ as input, it generates a random tag $c \leftarrow \{0,1\}^\lambda$ and a key $K \leftarrow \{0,1\}^\lambda$. Then it generates h HIBKEM key encapsulations as $(C_j, K_j) \leftarrow \mathsf{HIBKEM.Enc}(\mathrm{pk}, d_j)$ where $d_j = (\tau, H_j(c)), j \in [h]$ is a HIBKEM identity. The algorithm returns a ciphertext

$$C = (c, (C_j, G_2(K_j) \oplus K)_{j\in[h]}),$$

G_2 being a cryptographic hash function.

TBBFE.PuncCtx(sk, τ, C): Puncturing a secret key for a specific ciphertext is similar to the one of Basic BFE. Given a secret key $\mathrm{sk} = (T, \mathrm{sk}_{\mathrm{bloom}}, \mathrm{sk}_{\mathrm{time}})$ and a ciphertext of form $C = (c, (a)_{j\in[h]})$, the Bloom filter is first updated as $T' = \mathsf{BF.Insert}(H, T, c)$. With the Bloom filter key of form $\mathrm{sk}_{\mathrm{bloom}} = (\mathrm{sk}_{\tau|d})_{d\in[m]}$, the updated secret key is computed as

$$\mathrm{sk}'_{\tau|d} = \begin{cases} \mathrm{sk}_{\tau|i} & \text{if } T'[i] = 0, \\ \perp & \text{if } T'[i] = 1, \end{cases}$$

for each $i \in [m]$. A triple $\mathrm{sk}' = (T', \mathrm{sk}'_{\mathrm{bloom}}, \mathrm{sk}_{\mathrm{time}})$ is returned as an updated secret key.

TBBFE.Dec(sk, τ, C): With a secret key of form $\mathrm{sk} = (T, \mathrm{sk}_{\mathrm{bloom}}, \mathrm{sk}_{\mathrm{time}})$ and a ciphertext of form $C = (c, (a)_{j\in[h]})$ where $a = (C_j, D_j)$, the algorithm checks if there is any key left capable of decrypting the given ciphertext. The returned value is defined as

$$K = \begin{cases} \perp & \text{if } \mathrm{sk}_{\tau|H_j(c)} = \perp, \forall j \in [h], \\ D_j \oplus G_2(K_j) & \text{if } \mathrm{sk}_{\tau|H_j(c)} \neq \perp, \exists j \in [h]. \end{cases}$$

TBBFE.PuncInt(sk, τ): After it has been used up, an interval is punctured. It starts by resetting all Bloom filter bits. Having a secret key of form $\mathrm{sk} = (T, \mathrm{sk}_{\mathrm{bloom}}, \mathrm{sk}_{\mathrm{time}})$ for time interval $\tau' < \tau$, it uses $\mathrm{sk}_{\mathrm{time}}$ key set to derive a time key for interval τ, destroying all intermediary keys laying

on the traversal path. Bloom filter keys for the new interval are then recursively generated as

$$\mathrm{sk}_{\mathrm{bloom}} = \left\{\mathrm{sk}_{\tau|d} \leftarrow \mathsf{HIBKEM.Del}(\mathrm{sk}_{\tau}, d)\right\}, \forall d \in [m].$$

If the τ interval key is a left child node, its sibling node key is generated before both the τ and its parent node are destroyed. This new key is stored in $\mathrm{sk}_{\mathrm{time}}$ set and will be used as the key for the next time interval.

Theorem 5 ([DJS+18]). *From each efficient adversary $\mathcal{B}$ against IND-s-CPA of Time-Based BFE that sends q queries to random oracle G_2, an efficient adversary $\mathcal{A}$ can be constructed as*

$$\mathsf{Adv}^{\mathsf{OW-sID-CPA}}_{\mathcal{A},\mathsf{HIBKEM}}(\lambda) \geq \frac{\mathsf{Adv}^{\mathsf{IND-s-CPA}}_{\mathcal{B},\mathsf{PFSKEM}}(\lambda, m, h)}{hq}.$$

Correctness Error Much the same as with Basic BFE, the correctness error of TBBFE is defined by the false positibe probability of the Bloom filter, and by the probability of two ciphertexts having the same tag c. This yields the correctness error of the scheme as approximately $2^{-h} + n \cdot 2^{-\lambda}$.

4 Scheme Implementation

Implementations of BFE and TBBFE were carried out in two phases. We implemented both encryption schemes first in Java. We made this decision to first create an easily understandable implementation due to Java's higher level of abstraction. Moreover, implementation errors in such high-level languages are in most cases logic errors, as every basic data structure can be imported from one of existing packages. Implementation of same schemes later in C was then accelerated by the experience collected from the Java implementation. Following both implementations, we performed a series of tests yielding interesting performance comparisons.

Both Java and C implementations of BFE are released under CC0 license. Third-party libraries and code used or included in these implementations are licensed separately by their copyright owners. Their respective licences will be noted with library's first mention in the further text.

Bloom filter is common for both schemes. We used `MurmurHash3` as its internal hash function. `MurmurHash3` code we use in the BFE was written by Austin Appleby and released to the public domain. It is a non-cryptographic hash function, chosen due to its high speed hash calculation intended for hash-based indexing [Ram15]. We use two `MurmurHash3` hash functions to simulate h hash functions with no influence on false positive probability [KM06]. This is achieved with the following formula:

$$G'_i(x) = G_1(x) + iG_2(x) + i^2 \mod m, i \in [h].$$

Such solution conveniently simplifies implementation of use cases with non-constant number of hash functions.

4.1 Basic Bloom Filter Encryption

Together with the Bloom filter, BFIBE is one of the main building blocks of Basic BFE. This IBE scheme was originally constructed for Type-1 pairings. Nevertheless, BFIBE is proven secure in asymmetric setting of Type-2 [SV07].

Protocols in the Type-2 setting are transformable to the Type-3 setting [CM11] that we use in this implementation. For this to work, a generator g_1 and consequent public key pk are elements of $\mathbb{G}_1$. Hash function G_1 maps to $\mathbb{G}_2$. The rest remains as originally described.

Furthermore, the scheme requires a cryptographic hash function $G_2 : \mathbb{G}_t \rightarrow \{0,1\}^\lambda$. As its digest is not of fixed length, this called for an extendable output function (XOF). We use the `SHAKE256` function, being the first XOF approved by the National Institute of Standards and Technology (NIST) [Nat15]. It is a part of the SHA-3 family offering 256 bits of security for a sufficiently long digest. In general, to achieve the highest security level a digest has to be at least 512 bits in size.

4.1.1 Java Implementation

The minimum requirement for Java implementation of the scheme is Java 7. We use libraries `IAIK-JCE 5.51` [Ins18b] and `ECCelerate 5.0` [Ins18a] for cryptographic functionality and elliptic curve cryptography protocols respectively. Both libraries are commercial, but offering free licenses to educational, research, and open source projects.

The scheme is implemented as a single-threaded application, therefore no performance gains from concurrency are possible. In a real-world scenario this would be a major oversight since in the key generation phase each key is generated independently. Concurrent execution of key generation would noticeably boost the performance, especially considering servers have a higher number of central processing unit (CPU) cores than personal computers. As Java implementation was only a step before the real implementation in C, we placed multithreading out of scope.

Pairings are Type-3 over Barreto-Naehrig (BN) curves [BN06]. The base field size depends on the intended level of security. The `ECCelerate` library supports

a large set of field sizes that are configurable on runtime. We used different field sizes during testing and then made the comparison.

All introduced data structures are used in an abstract way to separate specific implementations and libraries from the main class. The `Bfe` was designed as a stateless class with all exposed methods being static. Following the original algorithm description in [DJS+18], the methods exposed are:

Bfe.generateKeys(keySize, n, fPosProbability)**:** Initializes the Bloom filter and an array of private keys of size equivalent to the filter's capacity. It iterates over the elements of the array and extracts the private keys that are then stored in the array. System parameters and the filter with the key array are returned separately.

Bfe.encapsulate(sysParams)**:** Generates the random symmetric key and calculates positions in the Bloom filter for which the key will be encrypted. After encrypting the key for all defined positions, it returns the ciphertext object which is fundamentally an array of ciphertexts, together with the key K', explained in more detail as part of IND-CCA-secure construction in Section 3.4.1.

Bfe.decapsulate(sysParams, sk, C)**:** Queries the Bloom filter for the presence of the ciphertext in the set. If there is no such element already in the set, it calculates positions in the Bloom filter for which the key was encrypted. From the key array it takes over the first available key from the calculated positions and decrypts the ciphertext. The retrieved key is then encrypted again as part of the consistency check described in more detail as part of IND-CCA-secure construction in Section 3.4.1. If the check holds it returns the key K', otherwise no value is returned.

Bfe.puncture(sk, C)**:** It inserts the given ciphertext to the Bloom filter followed by the deletion of the keys from the key array at the positions equal to the affected ones of the Bloom filter.

Due to the class being stateless, its consumers are responsible for the storage and handling of system parameters. System parameters including the public key are returned as publicly known results of the key generation.

4.1.2 C Implementation

Basic BFE is implemented as a library for C99 standard. For cryptographic and elliptic curve operations we use the `RELIC` library [AG], dual-licensed under Apache 2.0 and LGPL 2.1. The last `RELIC` stable release 0.4.0 was rolled out in 2014, and for this reason we use the unreleased 0.5.0 that is still under active development. `RELIC` supports different types of curves configurable separately from the `BFE` library at build time. Base field sizes depend on the chosen curve and can not be chosen separately.

Additionally, we use `SimpleFIPS202` from the eXtended Keccak Code Package (XKCP) for its `SHAKE256` implementation. Being released under CC0 license, this code is included as part of the `BFE` library, therefore no separate installation is necessary.

C implementation was carried out mostly by rewriting Java code to C, with differences coming out predominantly from the additional complexity of memory handling. The bloomfilter_enc.h header file exposes following functions:

bloomfilter_enc_init_setup_pair(keySize, n, fPosProbability)**:** Allocates the memory for the used data structures, initializes the Bloom filter, and sets the system parameters excluding the public key. Returns a setup pair structure.

bloomfilter_enc_keygen(setupPair)**:** Iterates over the key array inside the setup pair and extracts the private key storing them in the array. Only the status code is returned since storing is done in place affecting the existing array inside the setup pair.

bloomfilter_enc_init_ciphertext(sysParams)**:** Allocates the memory for the ciphertext. Returns a ciphertext pair structure.

bloomfilter_enc_encapsulate(ciphertextPair, sysParams)**:** Identical to a matching method from Section 4.1.1. The exception is, the ciphertext is stored in the passed ciphertext pair structure with the function itself returning only a status code.

bloomfilter_enc_decapsulate(K, sysParams, sk, C)**:** Identical to the method from Section 4.1.1. The exceptions is, the byte array K in which the decapsulated key will be stored has to be initialized beforehand and passed to the function as a parameter. After successful decapsulation the key is stored in it, while the function itself returns only a status code.

bloomfilter_enc_puncture(sk, C)**:** Identical to the matching method from Section 4.1.1.

There are additional helper utility functions exposed in the header file together with additional memory de/allocation functions for introduced data structures. In addition to the single-threaded mode this implementation of Basic BFE supports multithreaded key generation.

Multithreaded Key Generation

Following the general idea of the protocol, key generation will be executed on a server. Such computers are equipped with multi-core CPU units, enabling applications to benefit from internal task concurrency. While not likely used for a standard web application, cloud computing platforms like Microsoft Azure already offer server instances with more than a hundred CPU cores.

The highest CPU load is during key generation. Moreover, it is also the only phase where concurrency makes sense due to a high number of repetitive independent function calls. The implementation itself is intuitive as the keys are stored inside an indexed data structure. We extended the key generation function bloomfilter_enc_keygen(setupPair, t) with additional argument t for the number of threads. After Bloom filter size calculation and key collection initialization, the collection is logically divided on t equal parts. Each part is assigned to a different thread to generate and fill in the keys. With a large enough Bloom filter this could in theory increase the key generation speed close to t times.

The number of threads is left to be defined by the integrator as it depends on the system's CPU. As a rule of thumb, on a n-core processor, the number of threads should be somewhere between n and $2n$, depending on whether the CPU supports hyper-threading or not. Creating too many threads on a smaller target key collection could negatively influence the performance due to thread's lifecycle overhead.

4.1.3 Performance Expectations and Results

All performance tests were conducted on a standard `F4s` Microsoft Azure virtual machine with 4 virtual CPU cores (2.4 GHz Intel® Xeon® E5-2673 v3 Haswell)

and 8 GB of RAM. We measured full key generation, encryption, decryption, and puncturing times. We used different elliptic curves allowing different levels of security.

Testing Process

Constant values defined the same for all test scenarios are:

- false positive probability $\Pr_{\text{false}} = 10^{-3}$
- single key size $\lambda = 128$ bit
- number of encryption operations executed per test $n_{\text{enc}} = 500$
- number of decryption operations executed per test $n_{\text{dec}} = 500$
- number of puncturing operations executed per test $n_{\text{punc}} = 500$.

Values that changed over different test scenarios are elliptic curve type and a Bloom filter size defined by the number of possible puncturings for given $\Pr_{\text{false}}$ $n \in \{5 \cdot 10^3, 10 \cdot 10^3, 20 \cdot 10^3, 30 \cdot 10^3, 40 \cdot 10^3, 50 \cdot 10^3\}$.

For time measurement in Java we used `StopWatch` from `Apache Commons Lang 3.8.1 API`. In C we measured time with a Unix system call gettimeofday from `sys/time.h`.

We measured full key generation time as a run time of the BFE.KGen function. Encryption times were measured in a loop over n_{enc}. Right after a single encryption was performed, a decryption of the produced ciphertext was triggered followed by key puncturing. Each of the three operations had a dedicated time counter aggregating total times. From the measured values we derive average times for a single encryption, decryption, and puncturing operation.

The testing setup was able to identify false positive hits and includes the number of such cases in the test report. This was done for completeness' sake of the tests, but such situations were not expected as defined n_{punc} is not high enough to pose a threat of this happening.

We tested each scenario twice and derived a unique result as an average of the two.

Table 4.1: Average run times of basic elliptic curve functions in Java and C; BN-382; $n = 500$

Operation	t_{Java} [ms]	t_{C} [ms]	$t_{\text{Java}}/t_{\text{C}}$
Multiplication in $\mathbb{G}_1$	2.726	0.466	5.85
Multiplication in $\mathbb{G}_2$	7.204	1.556	4.63
Addition in $\mathbb{G}_1$	0.024	0.002	9.48
Addition in $\mathbb{G}_2$	0.034	0.006	5.33
Pairing	16.087	6.758	2.38
Multiplication in $\mathbb{G}_t$	0.054	0.019	2.76
Exponentiation in $\mathbb{G}_t$	20.414	5.058	4.04

Expected Performance Results

The most noteworthy difference between Java and C is the layers of abstraction. Java is compiled to Java Bytecode that is interpreted on a Java Virtual Machine (JVM), which then runs the code on the native environment. This makes Java highly portable. C is more straightforward than Java as it is compiled directly to binary code. This makes it platform dependent, but it runs faster than Java.

Performance expectations were made prior to implementation. To do that, the most computationally expensive elliptic curve operations used were identified and individually measured in Java and C. The results that are shown in Table 4.1 present significant differences in run times between the two.

As the number of function calls is known, expected run times of BFE functions were estimated. Together with BFE functions there are other functions as well that collectively impact the performance. Since this impact is not taken into account for the estimation, estimated values present the lower boundary of the run times. Elliptic curve operations used by each of BFE functions are presented in Table 4.2. This data was used to calculate the expected times of C implementation with the formulae:

$$t_{\mathsf{BFE.KGen}} \geq t_{\mathbb{G}_1\text{mul}} + m \cdot t_{\mathbb{G}_2\text{mul}} \tag{4.1}$$

$$t^{\text{C}}_{\mathsf{BFE.Enc}} \geq t_{\mathbb{G}_1\text{mul}} \cdot (2h + 1) + t_{\text{pair}} \cdot h \tag{4.2}$$

$$t^{\text{Java}}_{\mathsf{BFE.Enc}} \geq t_{\mathbb{G}_1\text{mul}} \cdot (h + 1) + t_{\text{pair}} \cdot h + t_{\mathbb{G}_t\text{exp}} \tag{4.3}$$

Table 4.2: Elliptic curve operations per each BFE function. Values in brackets are specific to Java implementation, values in parentheses are specific to C implementation.

	BFE.KGen	BFE.Enc	BFE.Dec	BFE.Punc
Mult. in $\mathbb{G}_1$	1	$h+1$ $(+\ h)$	$h+1$ $(+\ h)$	0
Mult. in $\mathbb{G}_2$	m	0	0	0
Pairing	0	h	$h+1$	0
Exp. in $\mathbb{G}_t$	0	$[h\ ]$	$[h\ ]$	0

$$t^{\mathrm{C}}_{\mathsf{BFE.Dec}} \geq t_{\mathbb{G}_1\mathrm{mul}} \cdot (2h+1) + t_{\mathrm{pair}} \cdot (h+1) \tag{4.4}$$

$$t^{\mathrm{Java}}_{\mathsf{BFE.Dec}} \geq t_{\mathbb{G}_1\mathrm{mul}} \cdot (h+1) + t_{\mathrm{pair}} \cdot (h+1) + t_{\mathbb{G}_t\mathrm{exp}} \tag{4.5}$$

BFE.Enc functions were implemented differently in Java than in C. In C, exponentiation was replaced with multiplication in $\mathbb{G}_1$ for performance benefits. This is why Formulae 4.2 and 4.3 are not the same. The same holds for Formulae 4.4 and 4.5 since encapsulation is a part of decapsulation's consistency check. There are no computationally expensive operations in BFE.Punc, and for this reason no estimations were made.

Key Generation Considering an elliptic curve and a prime field size is fixed for all test-runs of one test scenario, the only factor influencing the total key generation time is the number of keys generated. In other words, key generation time is proportional to Bloom filter size. As described in Section 2.3.3, this is defined by the number of possible key puncturings and a defined false positive probability.

From Formula 4.1, we expect the run times of the BFE.KGen function to be linear over the different number of possible key puncturings. This would make it possible to define the scheme's parameters before having to deploy it on a larger scale. The function for the estimation of the key generation run time is $t = \frac{n_{\mathrm{target}}}{x}$, where x is a constant defined for a specific environment where the scheme is to be deployed. To define this constant, a scheme would have to be deployed with a small key set of size n from where the constant is acquired as $x = \frac{n}{t'}$.

Encryption The single variable affecting the encryption run time is a number of hash functions inside a Bloom filter. This is evident from Table 4.2. Since false positive probability is constant over all test scenarios, it means the number of hash functions is also constant. Therefore we do not expect encryption run times to be influenced by the change in the key set size.

Decryption Focusing solely on elliptic curve operations, the single difference between encryption and decryption is the latter having an additional pairing operation. This yields the same conclusion as for encryption, that we do not expect run time to be influenced by the change in the key set size.

Results

We will start presenting performance results by comparing the results of implementations in Java and C. Further on, all results will be focused on the C implementation. As libraries used in the two different implementations do not support all the same elliptic curve types, a one to one comparison of the two is very limited. For this purpose, we use a Barreto-Naehrig curve of 382 bit prime order since it is supported by both libraries.

Barreto-Naehrig 382 The comparison of Java and C run time performance is shown in Figure 4.1. It is evident from the plot that the BFE.KGen function indeed is linear. The figure also supports the expectation for encryption run times, showing them to be nearly constant. The expectation for decryption also proved to be correct, with a noticeable difference from encryption due to the additional pairing operation. The difference in run times of encryption and decryption from Table 4.3 can be paired almost perfectly with the run time of a single pairing operation from Table 4.1.

Other Elliptic Curves The following elliptic curves were used for the testing of C implementation:

- Barreto-Naehrig (BN) of a 446 bit (BN-446) and 638 bit (BN-638) prime order

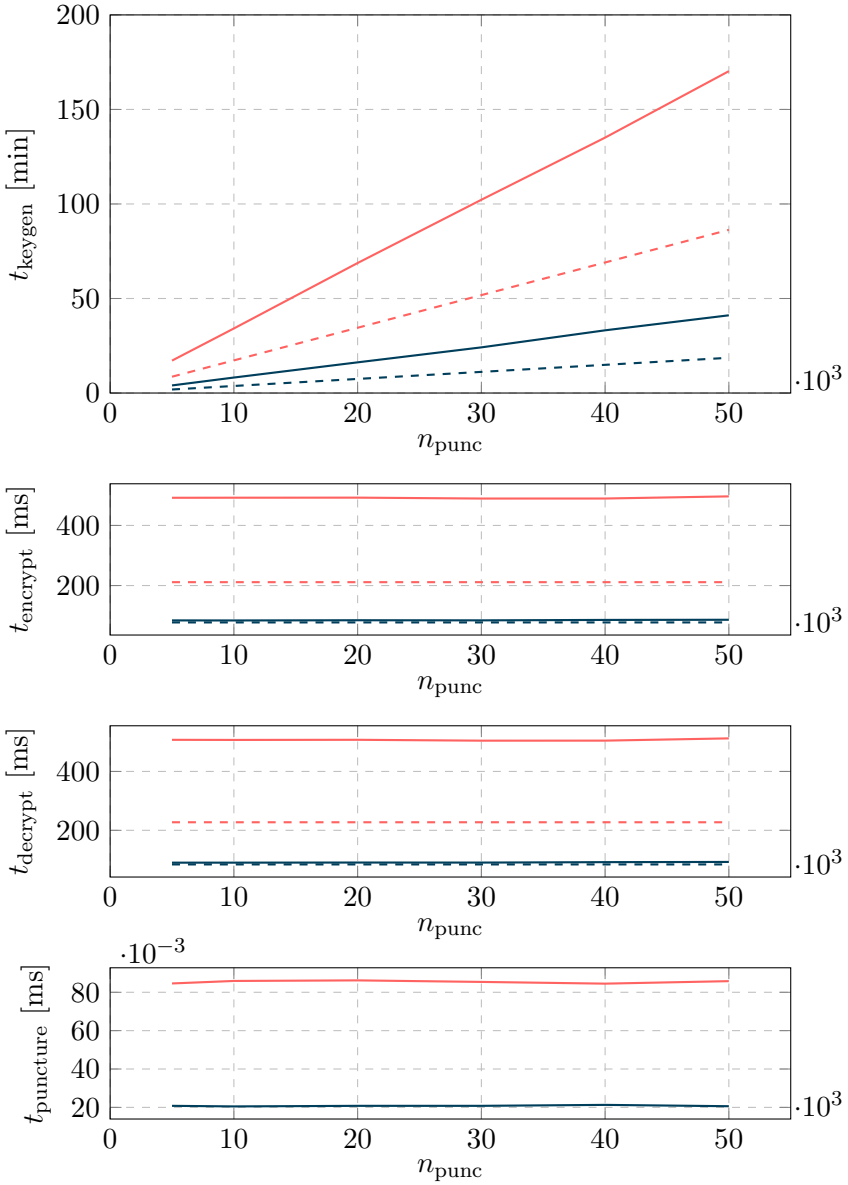


Figure 4.1: Comparison of Java (——) and C (——) run times of BFE operations in relation to the number of possible puncturings; BN-382. Expected run time lower boundary is indicated by - - - for C, and by - - - for Java.

Table 4.3: Run times of BFE functions in Java and C; BN-382; $n = 5 \cdot 10^4 \Rightarrow m = 718882$

Function	t_{Java}	t_{C}	$t_{\text{Java}}^{\text{est}}$	$t_{\text{C}}^{\text{est}}$	$t_{\text{Java}}/t_{\text{C}}$
BFE.KGen [min]	170.25	41.11	86.31	18.64	4.14
BFE.Enc [ms]	496.62	86.19	211.27	77.37	5.76
BFE.Dec [ms]	512.28	92.55	227.36	84.12	5.54
BFE.Punc [ms]	0.086	0.022	∅	∅	3.96

Table 4.4: Run times of BFE functions in C over different elliptic curves; $n = 5 \cdot 10^4 \Rightarrow m = 718882$

Function	$t_{\text{BN}-382}$	$t_{\text{BN}-446}$	$t_{\text{BN}-638}$	$t_{\text{BLS}-381}$	$t_{\text{BLS}-446}$
BFE.KGen [min]	41.11	54.15	153.15	34.05	47.90
BFE.Enc [ms]	86.19	123.79	268.31	74.08	104.59
BFE.Dec [ms]	92.55	132.99	285.10	79.23	112.01
BFE.Punc [ms]	0.022	0.024	0.031	0.020	0.024

- Barreto-Lynn-Scott (BLS) of a 381 bit (BLS-381) and 446 bit (BLS-446) prime order

Run times of BFE functions with different curves are presented in Table 4.4, with data plotted in Figure 4.2.

In [DJS+18] the authors defined $n_{\text{punc}} = 2^{20}$ as a choice of parameter reasoning that amounts to adding 2^{12} elements to the Bloom filter each day for a year. We fitted our measurements to linear curves allowing us to estimate how much time such key generation would take on our test system. We present the results in Table 4.5.

Multithreaded Key Generation Particularly noticeable performance improvements are the result of the concurrent key generation. Key extractions are completely independent of each other and therefore a speed up is expected when the number of threads is up to the number of cores. Such a setup was tested with a different number of threads with fixed $n = 5 \cdot 10^4$. The results are presented in Table 4.6. We can see in Figure 4.3 there is a steep drop in key generation time as soon as one additional thread is mobilized, and then run

Table 4.5: Estimated run times of BFE key generation in C; $n = 2^{20} \Rightarrow m \approx 15.076 \cdot 10^6$

Elliptic Curve	t_{keygen} [h]
BN-382	14.43
BN-446	19.25
BN-638	52.78
BLS-381	11.86
BLS-446	16.95

Table 4.6: Run times of BFE operations relative to the number of threads; $n = 5 \cdot 10^4 \Rightarrow m = 718882$. Unit of time is a minute.

n_{threads}	$t_{\text{BN}-382}$	$t_{\text{BN}-446}$	$t_{\text{BN}-638}$	$t_{\text{BLS}-381}$	$t_{\text{BLS}-446}$
1	41.11	54.15	153.15	34.05	47.90
2	23.16	29.89	79.28	19.06	25.86
3	15.62	20.30	55.58	13.78	17.60
4	12.27	16.03	45.17	10.85	13.61
5	12.36	16.49	45.55	10.72	13.67
6	12.39	16.54	46.83	11.33	14.66
7	12.05	16.34	46.05	11.01	14.64
8	12.63	15.97	45.94	11.15	14.92

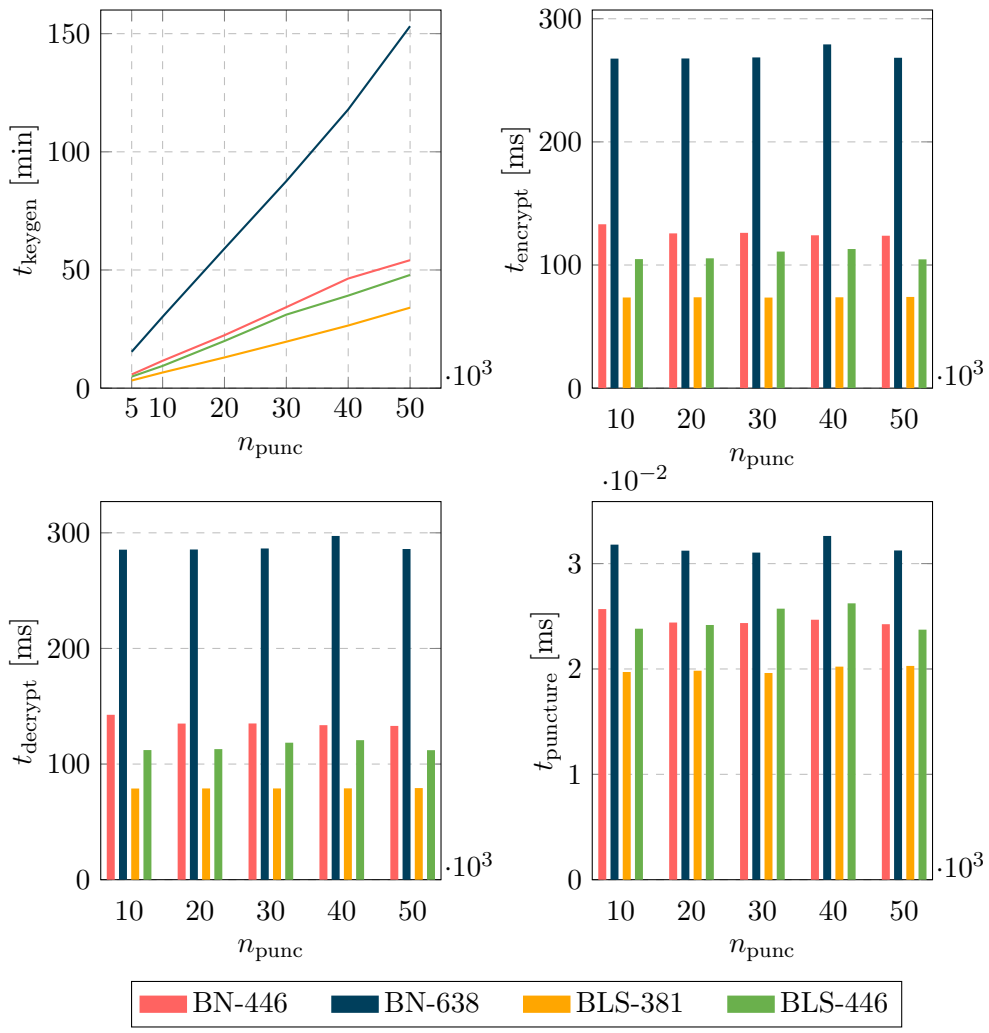


Figure 4.2: Run times of BFE operations in relation to the number of possible puncturings, C implementation

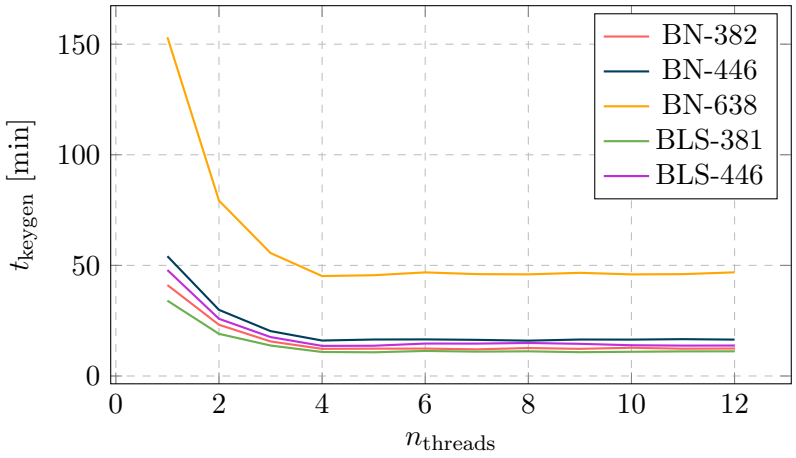


Figure 4.3: Run times of multithreaded BFE key generation over the number of threads; $n = 5 \cdot 10^4 \Rightarrow m = 718882$

time becomes nearly constant after 4 threads. The CPU of the test environment has 4 cores so this fits the expectations.

Memory Consumption The BFE secret key has to be created at once before using the scheme. Considering there is one Boneh-Franklin IBE secret key stored per each Bloom filter position, this key can become rather large. To measure the real memory allocation of such a key we used `Valgrind` together with `Massif Visualizer` to profile the memory on a small key set. Results are presented in Table 4.7 and plotted in Figure 4.4. Memory allocation of potentially larger keys was estimated since memory profiling is extremely time consuming and the function is anyhow linear. Puncturing keys does not affect the memory allocation since the whole memory block stays allocated up until the key is fully used. For $\Pr_{\text{false}} = 10^{-3}$, Formula 4.6 can be used for estimation of memory allocation of the secret key.

$$\text{size}_{\text{heap}} \approx \frac{x n_{\text{punc}}}{10^3} \quad [\text{MB}] \tag{4.6}$$

$$x_{\text{BN}-382,\text{BLS}-381} = 4.06, x_{\text{BN}-446,\text{BLS}-446} = 4.72, x_{\text{BN}-638} = 6.69$$

Table 4.7: Measured heap allocation of BFE secret key in MB

n_{punc}	BN-382/BLS-381	BN/BLS-446	BN-638
100	0.406	0.483	0.669
200	0.812	0.966	1.339
300	1.218	1.415	2.008

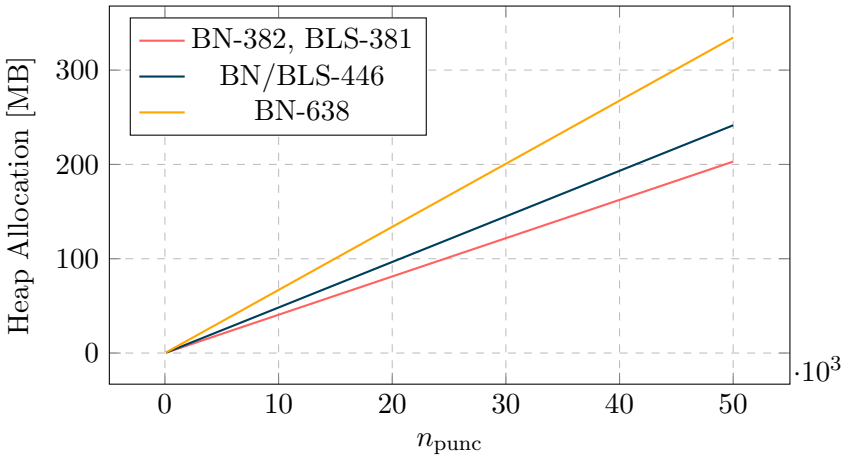


Figure 4.4: Heap allocation of BFE secret key. Values for $n_{\text{punc}} > 300$ are estimations.

In [DJS+18] the authors estimated for BLS-381 and $n_{\text{punc}} = 2^{20}$ the expected size of the secret key to be approximately 700 MB. By Formula 4.6, for a same BFE configuration, the secret key size would be approximately 4257 MB. Time performance had higher priority over memory performance in this implementation, therefore to avoid time costly decompression operations we did not store the elliptic curve points in a compressed format. If points would have been compressed, the total secret key size would be close to the one expected in [DJS+18].

4.2 Time-Based Bloom Filter Encryption

The Time-Based Bloom Filter Encryption (TBBFE) is built on top of the HIBE scheme. This IBE scheme is originally constructed for symmetric bilinear pairings. Because of the libraries used and potential performance benefits we had to adapt it to work in the Type-3 setting. We accomplished this by duplicating all system parameters into both groups [AHO16]. If $g \in \mathbb{G}_1$ and $\hat{g} \in \mathbb{G}_2$, then the HIBE system parameters are $(g, g_1, g_2, g_3, h_1, \ldots, h_l, \hat{g}, \hat{g}_1, \hat{g}_2, \hat{g}_3, \hat{h}_1, \ldots, \hat{h}_l)$, with $\mathrm{mk} = g_2^r$.

In BBGHIBE.Setup, a private key for identity $\mathrm{id} = (I_1, \ldots, I_k)$ is

$$\mathrm{sk}_{\mathrm{id}} = \left(g_2^r \cdot \left(g_3 \cdot h_1^{I_1} \cdots h_k^{I_k}\right)^s, g^s, h_{k+1}^s, \ldots, h_l^s\right) = (a, b, c_{k+1}, \ldots, c_l)\,.$$

Having a secret key in $\mathbb{G}_1$ makes it more compact, meaning the TBBFE key collection is smaller. The trade-off is having larger ciphertexts. In BBGHIBE.Encrypt, a single ciphertext is computed as

$$C = \left(e\left(g_1, \hat{g}_2\right)^t \cdot M, \hat{g}^t, \left(\hat{g}_3 \cdot \hat{h}_1^{I_1} \cdots \hat{h}_k^{I_k}\right)^t\right) = \left(a', b', c'\right),$$

with all elements ending up either in $\mathbb{G}_2$ or $\mathbb{G}_t$. BBGHIBE.Decrypt retrieves the original message as

$$M = a' \cdot \frac{e(b, c')}{e(a, b')}.$$

HIBE keys are generated in a recursive function that runs in a single thread. We did not implement TBBFE to support multithreading. This is still achievable for key generation where time performance benefits should be noticeable.

Outstanding Improvements While not implemented as part of this work, worth mentioning are two potential improvements that could have a significant performance impact. In Section 4.1.3 we have already demonstrated how much key generation can benefit from multithreading. Key generation of TBBFE is slightly more complex due to dependencies between the keys. Multithreading could be implemented for 2^t threads by generating the tree nodes recursively

down to level t, from where a new thread would be created for each node and parts of the tree continued to be generated independently in separate threads. Another improvement would be passing forward intermediary results between the nodes during key generation. For a single node on level k, this would replace current k point additions in $\mathbb{G}_1$ with a single addition. While point additions are not costly, on a larger scale this could potentially become noticeable.

4.2.1 Java Implementation

TBBFE is a part of the same codebase as the BFE implementation. Time interval keys are stored in a hash map while final Bloom filter keys are stored in a list. We did not use a tree-like data structure since intermediary keys are destroyed on the fly and the full tree is never formed in practice. Both hash map and list offer element retrieval of time complexity $O(1)$ suitable for optimal time performance.

`TimeBasedBfe` was designed as a stateless class with all exposed methods being static. The methods exposed are:

TimeBasedBfe.generateKeys(keySize, n, fPosProbability, timeExp)**:** Initializes a Bloom filter and two hash maps storing the time keys and Bloom filter keys separately. It extracts two keys for identities 0 and 1, stores them in the time interval hash map, and then punctures the first interval. System parameters and a secret key object are returned separately. The secret key includes the Bloom filter, both hash maps, and a current time interval identity.

TimeBasedBfe.encapsulate(sysParams, timeInt)**:** Generates the random symmetric key and calculates positions in the Bloom filter for which the key will be encrypted. It returns an array of encapsulated keys after encrypting the key for all defined positions.

TimeBasedBfe.decapsulate(sysParams, sk, C)**:** Queries the Bloom filter for the presence of the ciphertext in the set. If there is no such element already in the set, it calculates positions in the Bloom filter for which the key was encrypted. From the Bloom filter key hash map it takes over the first available key from the calculated positions and decrypts the ciphertext. The decrypted value is the symmetric key being returned as the output of the method.

TimeBasedBfe.punctureKey(sk, C): Inserts the given ciphertext to the Bloom filter followed by deletion of the keys from the Bloom filter key hash map at the positions equal to the affected ones of the Bloom filter.

TimeBasedBfe.punctureInterval(sysParams, sk, timeInt): It recursively extracts the keys until reaching the last level of the tree. It stores the last-level keys in the Bloom filter hash map, and the time keys needed for future time intervals in the time hash map. Intermediary keys not relevant for future time intervals are deleted on-the-fly. The time interval parameter is optional and typically not recommended since unsystematic puncturing of time intervals could lead to unreachable time intervals.

As with Basic BFE, the library consumers are responsible for the storage and handling of system parameters.

4.2.2 C Implementation

TBBFE is a part of the same library as BFE implementation. Time interval keys and Bloom filter keys are both stored inside a same list. Developing a hash map in C for a small number of time interval keys was infeasible, instead we identified a maximum possible number of time interval keys at one point of time. A TBBFE scheme with 2^t time intervals can at a certain time store a maximum of t time interval keys. This holds under the assumption that the intervals are punctured from left to right in a correct order. It is a reasonable assumption since doing it in any other order would only incur additional memory overhead.

The `tb_bloomfilter_enc.h` header file exposes the following functions:

tb_bloomfilter_enc_init_setup_pair(n, fPosProbability, timeExp): Allocates memory for the used data structures, initializes the Bloom filter, and sets a part of system parameters. Returns a setup pair structure.

tb_bloomfilter_enc_setup(setupPair, keySize): Extracts two keys for identities 0 and 1, stores them in the key array, and then punctures the first interval. Time keys and Bloom filter keys are stored inside the same array with time keys stored at lower indexes followed with Bloom filter keys at the higher ones. The generated keys are stored inside the passed setup pair and the function returns only a status code.

tb_bloomfilter_enc_init_ciphertext(sysParams): Allocates the memory needed for the ciphertext and returns the ciphertext structure.

tb_bloomfilter_enc_encapsulate(C, sysParams, timeInt): Identical to a matching method from Section 4.2.1. The exception is, the ciphertext is stored in the passed ciphertext structure with the function itself returning only a status code.

tb_bloomfilter_enc_decapsulate(K, sysParams, sk, C): Identical to the matching method from Section 4.2.1. The exception is, the byte array K in which the decapsulated key will be stored has to be initialized beforehand and passed to the function as a parameter. After successful decapsulation the key is stored in it, while the function itself returns only a status code.

tb_bloomfilter_enc_puncture_key(sysParams, sk, C): Identical to the matching method from Section 4.2.1.

tb_bloomfilter_enc_puncture_int(sysParams, sk): Recursively extracts the keys until reaching the last level of the tree. It stores the time keys needed for future time intervals and the last-level keys together in the key array. Intermediary keys not relevant for future time intervals are deleted on-the-fly. Due to the limited capacity of locations reserved for time keys inside the key array, it is not possible to select a specific time interval. Instead, time intervals are punctured in an ordered manner.

Additionally, additional utility and memory de/allocation functions for introduced data structures are exposed.

4.2.3 Performance Expectations and Results

The same environment as for Basic BFE was used with the same time measurement functions. We measured partial key generation, encryption, decryption, and puncturing of keys and time intervals. We used different elliptic curves offering different levels of security.

Testing Process

Constant values defined the same for all test scenarios are:

- false positive probability $\Pr_{\text{false}} = 10^{-3}$

- single key size $\lambda = 128\,\text{bit}$
- number of encryption operations executed per test $n_{\text{enc}} = 500$
- number of decryption operations executed per test $n_{\text{dec}} = 500$
- number of puncturing operations executed per test $n_{\text{punc}} = 500$
- total key capacity $m_{\text{total}} = n_{\text{time}} \cdot m_{\text{bloom}} = m_{\text{total}} = 291725$.

The values changed over different test scenarios are the elliptic curve type and a Bloom filter size of a single time interval n_{bloom} defined by the number of possible puncturings for given Pr_{false}. To keep m_{total} constant, n_{time} has to be changed accordingly. For each test scenario, the parameter configuration for test runs was $n_{\text{bloom}} = 2^{22} - i$ and $n_{\text{time}} = 2^i$, for $i = 2, \ldots, 15$.

We measured time tree key generation (also known as time interval puncturing) and Bloom filter key generation times separately. Bloom filter key generation was measured only for the first Bloom filter tree. Measuring the generation times of all trees would take a long time and the results should be similar for each of the trees. Run times of time interval puncturing highly depend on the interval being punctured. In the best case, it is instant as the key was generated as a part of previous interval puncturing. If not, intermediate keys are generated along the way to the final one. To be as general as possible, in all test scenarios all time intervals were punctured and we measured the time collectively. The time tree generation time was then defined as an average of all intervals.

We tested encryption and decryption times in the same way as for Basic BFE. Again, we tested each configuration twice, and derived a unique result as an average of the two.

Expected Performance Results

As with Basic BFE, we used values from Table 4.1 to estimate the expected C implementation run times.

As the TBBFE.KGen function can not be a synonym for key generation as it was in Basic BFE, parts of this function were not measured to simplify the process. Excluded from measurement are the time key derivations for two first level keys. We consider this as negligible on a larger scale.

Table 4.8: Elliptic curve operations per TBBFE function

	$n_{\text{time}} \cdot$ TBBFE.PuncInt	TBBFE.Enc	TBBFE.Dec
Mult. in $\mathbb{G}_1$	Formula 4.8	h	0
Mult. in $\mathbb{G}_2$	0	$2h$	0
Add. in $\mathbb{G}_1$	Formula 4.7	0	0
Add. in $\mathbb{G}_2$	0	$h(l+1)$	0
Pairing	0	h	2
Mult. in $\mathbb{G}_t$	0	0	1
Point doubling in $\mathbb{G}_1$	Formula 4.9	$h(l/2)$	0

Elliptic curve operations used by each of the TBBFE functions are presented in Table 4.8. To simplify the formulae, tree generation times were estimated as one full tree including both time interval tree and Bloom filter trees. The number of point additions in $\mathbb{G}_1$ in full tree generation is approximately

$$n_{\mathbb{G}_1\text{add}} = 4n + \sum_{i=1}^{l} 2^i \left(l+1\right). \tag{4.7}$$

The number of point multiplications in $\mathbb{G}_1$ in full tree generation is approximately

$$n_{\mathbb{G}_1\text{mul}} = \frac{5}{2}n + \sum_{i=1}^{l} 2^i \left(l-i+1\right), \tag{4.8}$$

where $n = 2^{l+1} - 1$ is the total number of nodes. The number of point doubling in $\mathbb{G}_1$ in full tree generation is approximately

$$n_{\mathbb{G}_1\text{dbl}} = \sum_{i=1}^{l} i \cdot 2^{i-1}. \tag{4.9}$$

There are no computationally expensive operations in TBBFE.PuncCtx. The run times were estimated as follows:

$$n_{\text{time}} \cdot t_{\text{TBBFE.PuncInt}} \geq t_{\mathbb{G}_1\text{add}} \cdot n_{\mathbb{G}_1\text{add}} + t_{\mathbb{G}_1\text{mul}} \cdot n_{\mathbb{G}_1\text{mul}} + t_{\mathbb{G}_1\text{add}} \cdot n_{\mathbb{G}_1\text{dbl}} \tag{4.10}$$

$$\begin{aligned} t_{\text{TBBFE.Enc}} \geq\ & t_{\mathbb{G}_1\text{mul}} \cdot h + t_{\mathbb{G}_2\text{mul}} \cdot 2h + t_{\mathbb{G}_2\text{add}} \cdot h(l+1) \\ & + t_{\text{pair}} \cdot h + t_{\mathbb{G}_2\text{add}} \cdot h\frac{l}{2} \end{aligned} \tag{4.11}$$

Table 4.9: Run times of TBBFE functions in Java and C; BN-382; $m_{\text{bloom}} = 2^{18} \Rightarrow n_{\text{bloom}} = 18233$, $n_{\text{time}} = 2^4$

Function	t_{Java}	t_{C}	$t_{\text{Java}}^{\text{est}}$	$t_{\text{C}}^{\text{est}}$	$t_{\text{Java}}/t_{\text{C}}$
$\mathsf{TBBFE.PuncInt}_{\text{bloom}}$ [min]	246.64	31.97	115.05	18.98	7.71
$\mathsf{TBBFE.PuncInt}_{\text{time}}$ [ms]	97.06	27.46	115.36	19.61	3.53
TBBFE.Enc [ms]	716.19	110.87	343.77	105.40	6.46
TBBFE.Dec [ms]	31.43	11.67	32.23	13.54	2.69
TBBFE.PuncCtx [ms]	0.095	0.019	∅	∅	5.00

$$t_{\mathsf{TBBFE.Dec}} \geq 2t_{\text{pair}} + t_{\mathbb{G}_t\text{mul}} \tag{4.12}$$

In Formulae 4.10 and 4.11, instead of point doubling times we use point addition times as we expect these two functions to be very close.

Results

Comparison of Java and C implementation performance results will again serve as an introduction to the results. Further on, we focus on the performance of C implementation. We use the BN-382 curve for comparison for the already mentioned reasons.

Barreto-Naehrig 382 The comparison of Java and C run time performance is shown in Figure 4.5, with values being presented in Table 4.9.

Other Elliptic Curves We use the following elliptic curves for testing of C implementation: BN-446, BN-638, BLS-381, and BLS-446. The run times of TBBFE functions with different curves are presented in Table 4.10, with data plotted in Figure 4.6. From these results it is noticeable that the encryption time does not depend on the ratio between time tree and Bloom tree size. This is as expected since the full tree is of constant size, meaning the number of point additions is constant over all test scenarios. Furthermore, we can see that the average time of single interval puncturing is decreasing as the number of time intervals is decreasing. This is explainable by the fact that there are less intermediary keys to be derived as part of the time tree generation.

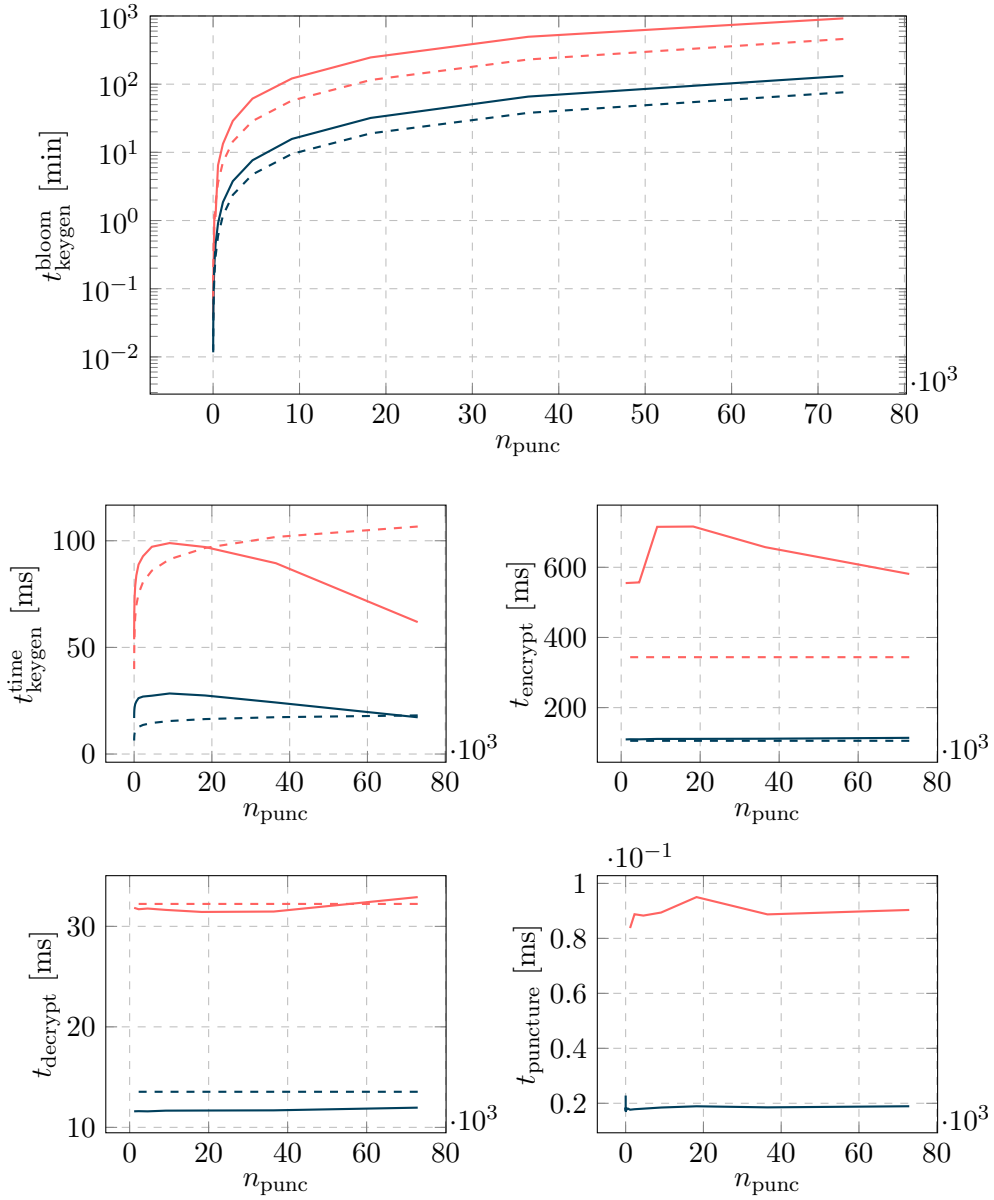


Figure 4.5: Comparison of Java (——) and C (——) run times of TBBFE operations in relation to the number of possible puncturings in a single time interval; BN-382

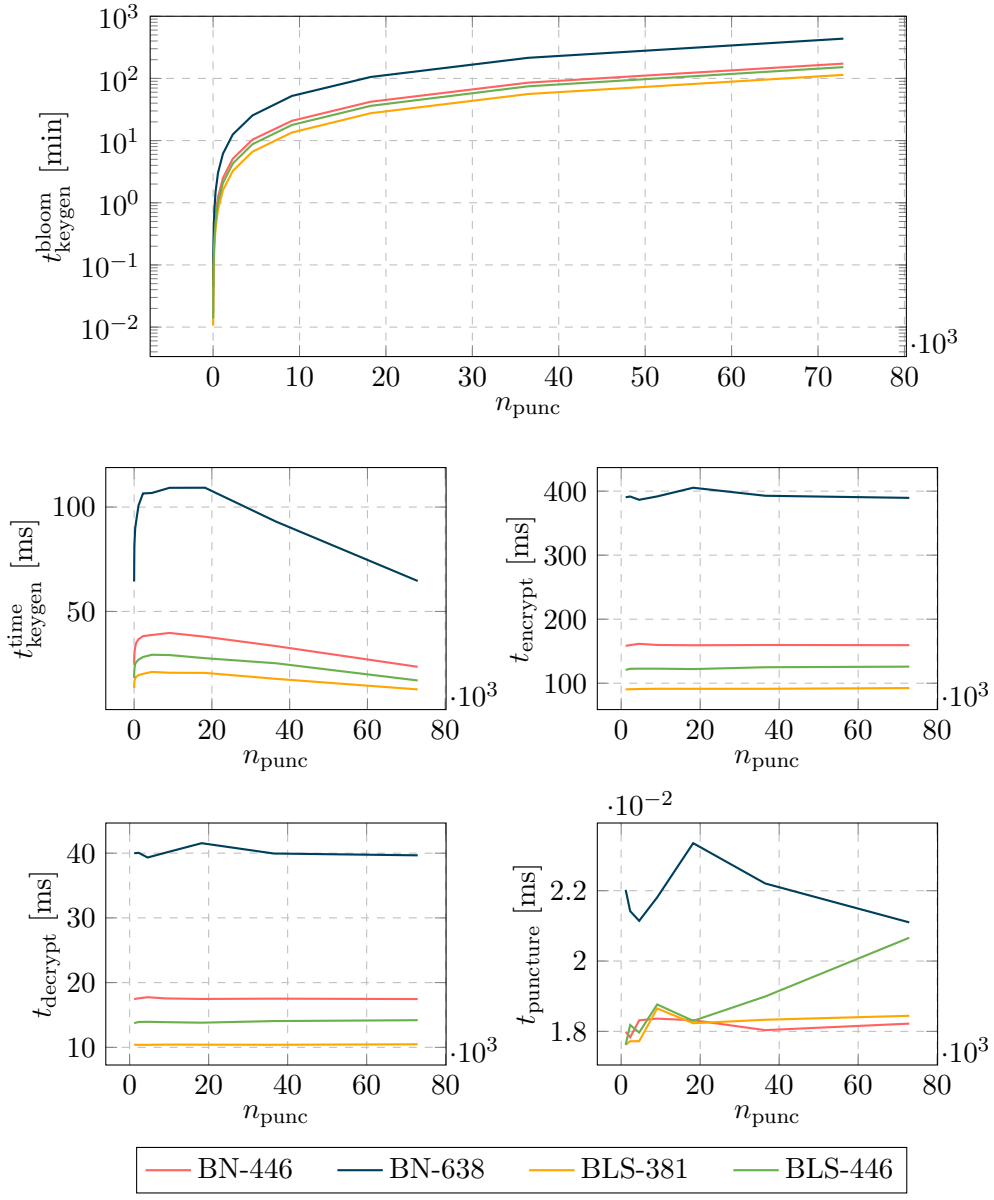


Figure 4.6: Run times of TBBFE operations in relation to the number of possible puncturings per one time interval, C implementation

Table 4.10: Run times of TBBFE functions in C over different elliptic curves; $m_{\text{bloom}} = 2^{18} \Rightarrow n_{\text{bloom}} = 18233$, $n_{\text{time}} = 2^4$

Function	$t_{\text{BN}-382}$	$t_{\text{BN}-446}$	$t_{\text{BN}-638}$	$t_{\text{BLS}-381}$	$t_{\text{BLS}-446}$
$\mathsf{TBBFE.PuncInt}_{\text{bloom}}$ [min]	31.97	42.16	105.49	27.58	36.21
$\mathsf{TBBFE.PuncInt}_{\text{time}}$ [ms]	27.46	37.87	109.25	20.55	27.60
TBBFE.Enc [ms]	110.87	159.27	405.29	91.24	122.07
TBBFE.Dec [ms]	11.67	17.47	41.54	10.41	13.81
TBBFE.PuncCtx [ms]	0.019	0.018	0.023	0.018	0.018

Table 4.11: Estimated run times of TBBFE single bloom tree generation in C; $m_{\text{bloom}} = 2^{20} \Rightarrow m_{\text{bloom}} \approx 15.076 \cdot 10^6$

Elliptic Curve	t_{keygen} [h]
BN-382	31.64
BN-446	41.47
BN-638	104.60
BLS-381	27.35
BLS-446	36.46

Table 4.12: Measured heap allocation of TBBFE secret key in MB

n_{punc}	BN-382/BLS-381	BN/BLS-446	BN-638
72	1.825	2.109	2.958
143	3.639	4.203	5.897
285	7.265	8.392	11.774
570	14.515	16.768	23.525

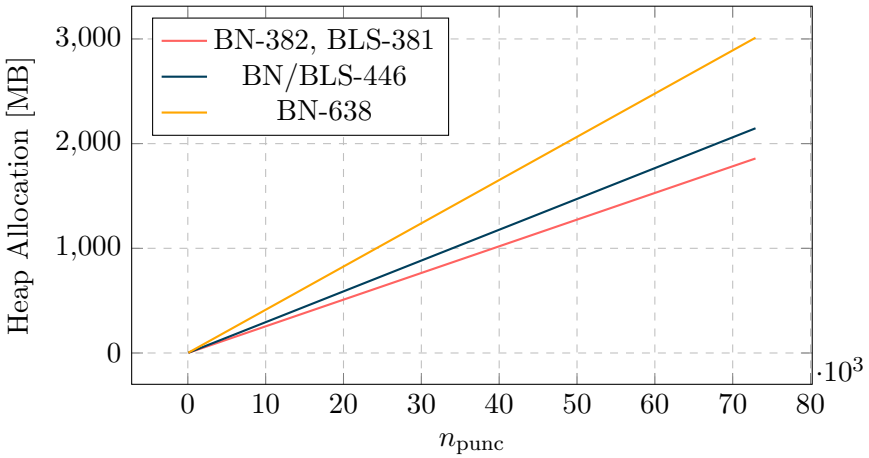


Figure 4.7: Heap allocation of Time-Based BFE secret key. Values for $n_{\text{punc}} > 570$ are estimations.

Similar as we did for Basic BFE, we estimate how much time it would take to generate keys for $n_{\text{punc}} = 2^{20}$ inside a single time interval. The results are presented in Table 4.11. We do not estimate time tree generation times, but those should be negligible in this setup for a reasonable number of intervals considering the size of a bloom tree.

Memory Consumption As described in Section 3.4.2, the final BFE secret key stored in TBBFE is structurally equivalent to the one of the basic scheme with the addition of stored time interval keys. Considering there is one HIBE secret key stored per each Bloom filter position, this key can become rather large. We used the same tools to measure memory consumption of the key as for the Basic

BFE. Results are presented in Table 4.12 and plotted in Figure 4.7. Again, a small key set was profiled while potentially larger keys were estimated from the results. Memory is statically allocated so puncturing keys does not affect the memory allocation during its life span.

For $\mathrm{Pr}_{\mathrm{false}} = 10^{-3}$, Formula 4.13 can be used for the estimation of memory allocation of the secret key.

$$\mathrm{size}_{\mathrm{heap}} \approx \frac{x n_{\mathrm{punc}}}{10^3} \quad [\mathrm{MB}] \tag{4.13}$$

$$x_{\mathrm{BN-382,BLS-381}} = 25.5, x_{\mathrm{BN-446,BLS-446}} = 29.4, x_{\mathrm{BN-638}} = 41.3$$

5 TLS Extension

The presented implementations can now be used as cryptographic primitives for building high-level protocols. As a proof of concept for this thesis, we use Basic BFE to extend the TLS 1.3 protocol with the 0-RTT replay attack secure and forward secret application data transfer. The goal of such an extension would be the faster initialization of a connection between a client and a server. In low-bandwidth areas this could cause noticeable speed improvement without weakening the security.

To avoid breaking existing TLS functionality, and also to comply with the protocol standard, we implemented the additional functionality as a new `bfe_key` extension. The client can send early data as described in the protocol, but this time without owning a previously received PSK. The PSK is dynamically generated on the client-side in the `bfe_key` extension. The key is then encrypted by BFE using the public key of the server and sent out as extension data. The public key is something a client should have obtained earlier, such as a part of server's certificate.

Key Derivation Schedule Figure 5.1 shows the full key derivation schedule of TLS 1.3 with adaptations made with this implementation. Early secrets are always derived prior to the derivation of handshake secrets. In this implementation we replace PSK with a random byte array generated by the client. Additionally, we replace the (EC)DHE shared secret with zero-valued bytes, as forward secrecy is anyhow provided by BFE.

Upon receiving data from the client, the server decrypts the PSK from the extension and uses it to decrypt the early data. The early data is not a part of the extension data, but instead the data is sent after all the extensions as described in TLS 1.3. After decryption the server punctures its secret key

Table 5.1: Captured TLS network packets of full TLS 1.3 handshake with application data

t [ms]	Δt [ms]	Direction	Len. [B]	Message
0	0	C $\longrightarrow$ S	417	Client Hello
1.119	1.101	S $\longrightarrow$ C	200	Server Hello, Change Cipher Spec
5.822	4.485	C $\longrightarrow$ S	72	Change Cipher Spec
9.827	4.005	S $\longrightarrow$ C	1759	Application Data
10.435	0.563	C $\longrightarrow$ S	140	Application Data
46.576	0.020	C $\longrightarrow$ S	130	Application Data
46.708	0.106	S $\longrightarrow$ C	90	Application Data

making it impossible to decrypt the same data again. This ensures replay attack protection and forward secrecy of the early data.

5.1 Implementation

We extended the `GnuTLS 3.6.4` library. It is licensed under LGPLv2.1+ license and is known as one of the more popular C TLS libraries.

The implementation was undertaken in an elementary way. We implemented only the parts directly needed for the proof of concept. This means whenever early data is sent, the new extension is initialized and used, with no fallback system in place.

We extended the `BFE` library with functions for writing and reading the scheme's parameters and keys to and from a file system. This allowed for a client to access the server's public key without having to do any kind of public key obtainment. It additionally simplified implementation as there was no need to extend server initialization functions since key generation was performed offline. Reading the key from a file was done in the `bfe_key` itself.

With all the mentioned shortcuts in place the implementation is not appropriate for usage on production systems. Nonetheless, it proves BFE can be implemented as a part of the TLS 1.3 protocol and provide a testing ground for performance measurements.

Table 5.2: Captured TLS network packets of TLS 1.3 handshake with early data and `bfe_key`

t [ms]	Δt [ms]	Direction	Len. [B]	Message
0	0	C $\longrightarrow$ S	747	Client Hello
97.772	97.751	S $\longrightarrow$ C	127	Server Hello, Change Cipher Spec
97.861	0.074	C $\longrightarrow$ S	72	Change Cipher Spec
97.921	0.052	C $\longrightarrow$ S	106	Application Data
103.097	5.168	S $\longrightarrow$ C	1763	Application Data
103.898	0.781	C $\longrightarrow$ S	92	Application Data
143.340	0.024	C $\longrightarrow$ S	164	Application Data
143.477	0.128	S $\longrightarrow$ C	90	Application Data

5.1.1 Performance Results

TLS performance tests were conducted on a machine with 2 CPU cores (1.8 GHz Intel® Core™ i5 Ivy Bridge) and 6 GB of RAM. Server application was set up on a local socket listening for client data and the client application was executed on the same machine. Network traffic was recorded using `Wireshark`.

Two different test scenarios were executed, one with the full TLS 1.3 handshake followed by application data, and one with the early data and `bfe_key` extension (BLS-381). Captured TLS network packets of the former are shown in Table 5.1 and packets of the latter are shown in Table 5.2. In both tests the same application data was sent.

The handshake with the `bfe_key` extension is noticeably slower in a scenario where the ping time is essentially zero. In other words, the total processing time of packets in the TLS handshake with `bfe_key` is higher than the total processing time of packets in full TLS 1.3 handshake. This was expected as we have introduced additional cryptographic operations in the protocol. The most noticeable performance downgrade is on the `ServerHello` message. This is because prior to sending this message, a server first decrypts the PSK from the `bfe_key` extension. Additional time cost is caused by the trivial implementation of the extension. In a production-ready implementation, the BFE parameters and keys would be loaded in the memory instead of being read from the file system prior to each usage. Reading BFE parameters and secret key from the

file system takes in total 25.77 ms which is the cost of trivial implementation.

Furthermore, the measurements we present are in a perfect zero ping time environment. In the real world where each roundtrip takes time, especially in low-bandwidth conditions, we expect that the early data handshake with `bfe_key` could outperform the full TLS 1.3 handshake with application data. The time difference between handshake start and first application data transfer from client to server, taking into account the system parameters memory read time overhead, provides an effectiveness ping threshold of the extended handshake. The early data handshake with `bfe_key` is more effective than the full TLS 1.3 handshake with application data for

$$t_{\text{ping}} \gtrsim 75\,\text{ms}.$$

5.1.2 Upgrading TLS 1.2 with Early Data

A question that may come up is whether the TLS 1.2 protocol could be extended in a similar fashion to offer an early data mode. In TLS 1.2 a master secret is derived with PRF as

```
master_secret = PRF(pre_master_secret, "master secret",
ClientHello.random + ServerHello.random)
[0..47];
```

A `pre_master_secret` is generated by the client and encrypted with the server's public key before sending to the server. Both client and server can then generate the master secret by using client and server randoms. This provides replay attack protection since even if the the same premaster secret and client random would be sent, the server will generate a new random and a master key will be different each time.

This type of master secret generation is a reason why early data implementation with a `bfe_key` extension or similar can not be implemented in TLS 1.2. It is not possible for a client to obtain a server random without at least one prior roundtrip. Without this random value the master secret cannot be properly derived.

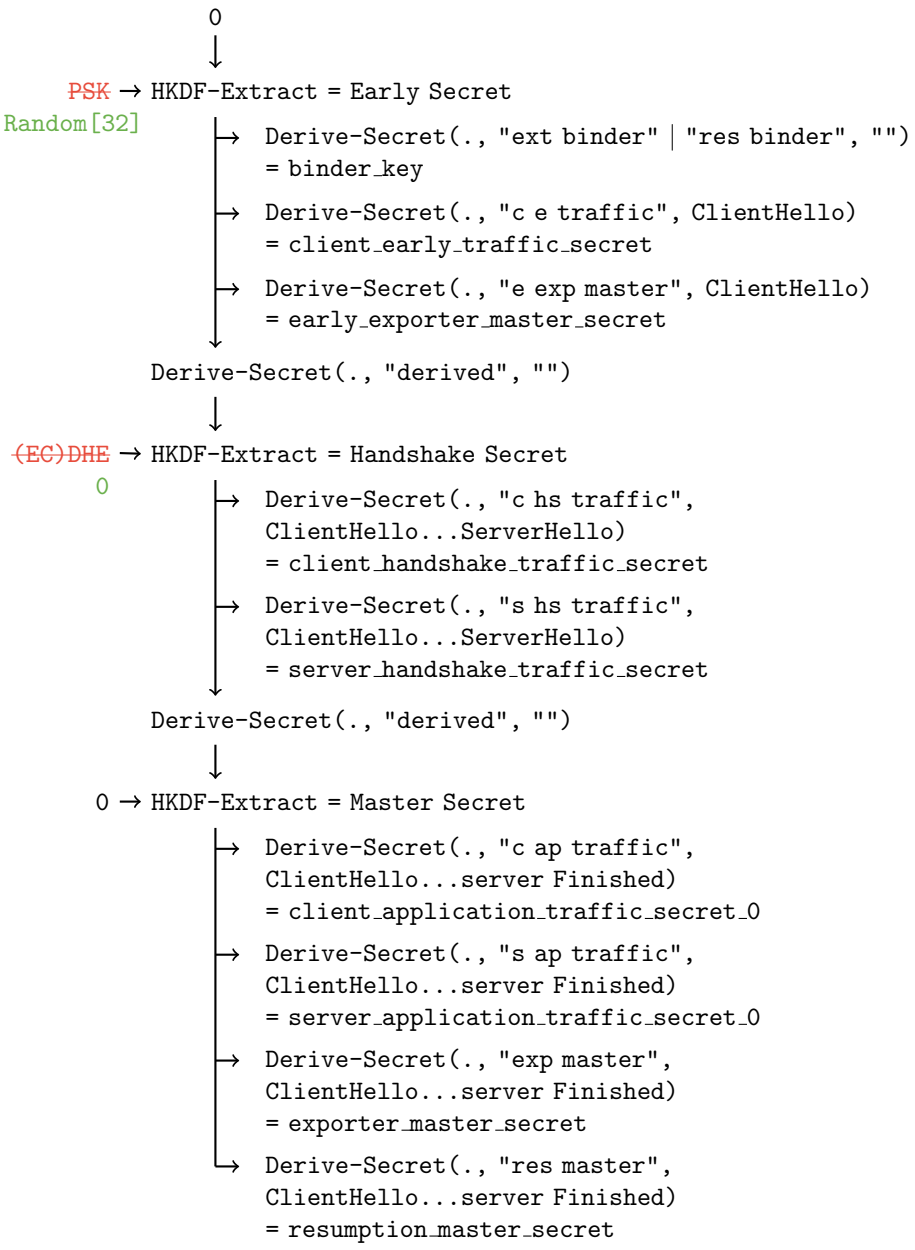


Figure 5.1: Implemented changes in the TLS 1.3 key derivation schedule

To this end, a client could generate a kind of early data secret by using the same PRF without a server random. This secret could then be used for encrypting early data and sending it with the first set of messages together with a variant of the `bfe_key` extension that would carry the BFE encrypted key. While the server would be able to decrypt the early data that way, a master secret derived from both client and server randoms would have to be acquired for further communication. This is something that is easily implemented in TLS 1.3 due to the key derivation process that supports early secrets without the server's contribution. In TLS 1.2 this would require breaking changes to the protocol resulting in its variant that would not be compliant with [RD08] anymore.

6 Conclusion

Implementation and performance of Bloom Filter Encryption together with its applicability to TLS was discussed in this thesis. Chapter 2 explained the preliminaries with an overview of basic secure communication concepts. The focus of the chapter was on presenting the improvements that TLS 1.3 brought compared to its preceding version as these improvements enabled us to integrate our library into the protocol. In Chapter 3, encryption schemes used for construction of BFE were described in detail.

In Chapter 4, implementations of Basic BFE and Time-Based BFE were presented with justifications for specific design decisions. Performance tests were executed for multiple different elliptic curves under different sets of parameters. We presented the results and compared them with expected outcomes. We demonstrated how key generation of Basic BFE benefits from multithreaded execution. To complement the presented implementations, we also mentioned some improvements that could be applied to our implementation in order to improve performance. Multithreaded key generation in TBBFE would be one the improvements that would certainly make a noticeable difference.

In Chapter 5, implementation of the `bfe_key` TLS 1.3 extension was presented. This extension extends the TLS 1.3 handshake by providing a fully forward secret 0-RTT mode. We elaborated performance measurements of the new handshake protocol in detail and compared them with the full TLS handshake. A slow down in TLS handshake package processing is recorded, but we argue such implementation could still be effective in low-bandwidth environments.

6.1 Future Work

Future work can be done in the areas of code optimization, further testing, and implementation with the TLS. It would be satisfying to see the development of

the `BFE` library continued in the form of an open-source project. There are finer performance improvements that could be made, as well as the multithreaded version of the TBBFE key generation.

Further, it would be interesting to conduct a larger set of tests on different environments, possibly on the devices with less processing power. We would like to see the tests performing key generations of larger keys that are suitable for use on production systems for months, or a year. In the context of TBBFE, further testing of interval puncturing would have to be done to analyze the performance impact of moving the key generation to the right side of the tree.

Interesting conclusions could come to light with a non-trivial implementation of the `bfe_key` extension. Additionally, an implementation of the extension in the library such as the `OpenSSL` could be beneficial for further testing. If the extended TLS would be deployed on a real server, the speed comparisons under different conditions could be made, hopefully proving it indeed has practical benefits in specific scenarios.

Bibliography

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an efficient library for cryptography. https://github.com/relic-toolkit/relic (cited on page 42).

[AGJ19] N. Aviram, K. Gellert, and T. Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 117–150. Springer International Publishing, 2019 (cited on page 3).

[AHO16] M. Abe, F. Hoshino, and M. Ohkubo. Design in Type-I, run in Type-III: fast and scalable bilinear-type conversion using integer programming. In *CRYPTO*, pages 387–415. Springer, 2016. DOI: 10.1007/978-3-662-53015-3_14 (cited on page 54).

[Ama19] Amazon.com, Inc. Alexa, how was Prime Day? Prime Day 2019 surpassed Black Friday and Cyber Monday combined. Press release, July 2019. https://press.aboutamazon.com/news-releases/news-release-details/alexa-how-was-prime-day-prime-day-2019-surpassed-black-friday (cited on page 1).

[BB04] D. Boneh and X. Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In C. Cachin and J. L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 223–238. Springer Berlin Heidelberg, 2004. ISBN: 978-3-540-24676-3 (cited on page 25).

[BBG+06] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ECC) cipher suites for Transport Layer Security (TLS). RFC 4492, May 2006. DOI: 10.17487/RFC4492 (cited on page 9).

[BBG05] D. Boneh, X. Boyen, and E.-J. Goh. Hierarchical identity based encryption with constant size ciphertext. In *Advances in Cryptology—EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 440–456. Berlin: Springer-Verlag, 2005 (cited on pages 24, 26).

[BBS04] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In M. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 41–55. Springer Berlin Heidelberg, 2004. ISBN: 978-3-540-28628-8 (cited on page 17).

[BD18] R. Barbulescu and S. Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, January 2018. ISSN: 1432-1378. DOI: 10.1007/s00145-018-9280-5 (cited on page 18).

[BF03] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. of Computing*, 32(3):586–615, 2003. extended abstract in Crypto'01 (cited on pages 17, 21, 22).

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970 (cited on pages 3, 13).

[BLS01] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '01, pages 514–532. Springer-Verlag, 2001. ISBN: 3-540-42987-5 (cited on page 17).

[BLS03] P. S. L. M. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *Proceedings of the 3rd International Conference on Security in Communication Networks*, SCN'02, pages 257–267. Springer-Verlag, 2003 (cited on page 18).

[BM99] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 431–448. Springer Berlin Heidelberg, 1999 (cited on page 6).

[BMO17] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. ACM Press, 2017. DOI: 10.1145/3133956.3133980 (cited on page 28).

[BN06] P. S. L. M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography*, SAC'05, pages 319–331. Springer-Verlag, 2006. DOI: 10.1007/11693383_22 (cited on pages 18, 40).

[BS04] D. Boneh and H. Shacham. Group signatures with verifier-local revocation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 168–177. ACM, 2004. ISBN: 1-58113-961-6. DOI: 10.1145/1030083.1030106 (cited on page 17).

[BW13] D. Boneh and B. Waters. Constrained pseudorandom functions and their applications. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, pages 280–300. Springer Berlin Heidelberg, 2013 (cited on page 3).

[CHK03] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In E. Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 255–271. Springer Berlin Heidelberg, 2003 (cited on page 6).

[CL04] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In M. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 56–72. Springer Berlin Heidelberg, 2004. ISBN: 978-3-540-28628-8 (cited on page 17).

[CM11] S. Chatterjee and A. Menezes. On cryptographic protocols employing asymmetric pairings — the role of ψ revisited. *Discrete Applied Mathematics*, 159(13):1311–1322, 2011. ISSN: 0166-218X. DOI: 10.1016/j.dam.2011.04.021 (cited on page 40).

[CRR+17] R. Canetti, S. Raghuraman, S. Richelson, and V. Vaikuntanathan. Chosen-ciphertext secure fully homomorphic encryption. In S. Fehr, editor, *Public-Key Cryptography – PKC 2017*, pages 213–240. Springer Berlin Heidelberg, 2017 (cited on page 28).

[Cut10] B. Cutler. Firefox & page load speed – part II. April 5, 2010. URL: https://blog.mozilla.org/metrics/2010/04/05/firefox-page-load-speed-%E2%80%93-part-ii/ (cited on page 2).

[DGJ+18] D. Derler, K. Gellert, T. Jager, D. Slamanig, and C. Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. Cryptology ePrint Archive, Report 2018/199, 2018. https://eprint.iacr.org/2018/199 (cited on pages 2, 3).

[DJS+18] D. Derler, T. Jager, D. Slamanig, and C. Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 425–455. Springer, 2018 (cited on pages 2, 3, 27–29, 32, 33, 38, 41, 49, 53).

[DKL+18] D. Derler, S. Krenn, T. Lorünser, S. Ramacher, D. Slamanig, and C. Striecks. Revisiting proxy re-encryption: forward secrecy, improved security, and applications. In M. Abdalla and R. Dahab, editors, *Public-Key Cryptography – PKC 2018*, pages 219–250. Springer International Publishing, 2018 (cited on page 28).

[ET05] P. Eronen and H. Tschofenig. Pre-shared key ciphersuites for Transport Layer Security (TLS). RFC 4279, December 2005. DOI: 10.17487/RFC4279 (cited on page 9).

[Fac19a] Facebook, Inc. F8 2019 day 1 keynote. Video file, April 2019. https://developers.facebook.com/videos/f8-2019/day-1-keynote/ (cited on page 1).

[Fac19b] Facebook, Inc. Facebook reports first quarter 2019 results. Press release, April 2019. https://investor.fb.com/investor-news/press-release-details/2019/Facebook-Reports-First-Quarter-2019-Results/ (cited on page 1).

[FCA+98] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998 (cited on page 15).

[FO99] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 537–554. Springer Berlin Heidelberg, 1999 (cited on page 32).

[GHJ+17] F. Günther, B. Hale, T. Jager, and S. Lauer. 0-RTT key exchange with full forward secrecy. In *EUROCRYPT (3)*, volume 10212 of *Lecture Notes in Computer Science*, pages 519–548, 2017 (cited on pages 2, 28, 33).

[GM15] M. D. Green and I. Miers. Forward secure asynchronous messaging from puncturable encryption. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 305–320. IEEE Computer Society, 2015. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.26 (cited on pages 27, 28).

[Goo19] Google LLC. HTTPS encryption on the web - Google transparency report. https://transparencyreport.google.com/https/overview, 2019. Accessed: 01/07/2019 (cited on page 1).

[GS02] C. Gentry and A. Silverberg. Hierarchical ID-based cryptography. In Y. Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, pages 548–566. Springer Berlin Heidelberg, 2002 (cited on page 6).

[GS08] J. Groth and A. Sahai. Efficient non-interactive proof systems for bilinear groups. In N. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 415–432. Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-78967-3 (cited on page 17).

[Gün90] C. G. Günther. An identity-based key-exchange protocol. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, pages 29–37. Springer Berlin Heidelberg, 1990 (cited on pages 1, 6).

[HL02] J. Horwitz and B. Lynn. Toward hierarchical identity-based encryption. In L. R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 466–481. Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-46035-0 (cited on page 23).

[Ins18a] Institute for Applied Information Processing and Communications (IAIK). IAIK ECCelerate version 5.0. https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/ECCelerate, 2018 (cited on page 40).

[Ins18b] Institute for Applied Information Processing and Communications (IAIK). IAIK Java Cryptography Extension (IAIK-JCE) version 5.51. `https://jce.iaik.tugraz.at/sic/Products/Core_Crypto_Toolkits/JCA_JCE`, 2018 (cited on page 40).

[Jou00] A. Joux. A one round protocol for tripartite Diffie-Hellman. In *Proceedings of the 4th International Symposium on Algorithmic Number Theory*, ANTS-IV, pages 385–394. Springer-Verlag, 2000. ISBN: 3-540-67695-3 (cited on page 17).

[KM06] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: building a better Bloom filter. In *In Proc. the 14th Annual European Symposium on Algorithms (ESA 2006*, pages 456–467, 2006 (cited on page 39).

[Lys02] A. Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 597–612. Springer Berlin Heidelberg, 2002. ISBN: 978-3-540-45708-4 (cited on page 17).

[Mil86] V. S. Miller. Short programs for functions on curves. In *IBM, Thomas J. Watson Research Center*, 1986 (cited on page 17).

[MRY04] P. MacKenzie, M. K. Reiter, and K. Yang. Alternatives to non-malleability: definitions, constructions, and applications. In M. Naor, editor, *Theory of Cryptography*, pages 171–190. Springer Berlin Heidelberg, 2004 (cited on page 27).

[MVO91] A. Menezes, S. Vanstone, and T. Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 80–89. ACM, 1991. ISBN: 0-89791-397-3. DOI: `10.1145/103418.103434` (cited on page 17).

[MVR99] S. Micali, S. Vadhan, and M. Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 120–130. IEEE Computer Society, 1999. ISBN: 0-7695-0409-4 (cited on page 17).

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Federal Information Processing Standards Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, August 2015. DOI: 10.6028/NIST.FIPS.202 (cited on page 40).

[Pay19] PayPal Holdings, Inc. Paypal reports fourth quarter and full year 2018 results. Press release, January 2019. https://investor.paypal-corp.com/news-releases/news-release-details/paypal-reports-fourth-quarter-and-full-year-2018-results (cited on page 1).

[Ram15] F. Ramadhian. Performance analysis of Bloom filter with various hash functions on spell checker, May 2015 (cited on page 39).

[RD08] E. Rescorla and T. Dierks. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, August 2008. DOI: 10.17487/RFC5246 (cited on pages 7, 71).

[Rep08] V. Reporter. The Value of a Millisecond: Finding the Optimal Speed of a Trading Infrastructure, TABB Group, 2008 (cited on page 2).

[Res18] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. RFC 8446, August 2018. DOI: 10.17487/RFC8446 (cited on pages 10, 11, 13).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. ISSN: 0001-0782. DOI: 10.1145/359340.359342 (cited on page 8).

[Sha85] A. Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology*, pages 47–53. Springer Berlin Heidelberg, 1985. ISBN: 978-3-540-39568-3 (cited on page 19).

[Sul17] N. Sullivan. Introducing zero round trip time resumption (0-RTT). March 2017. URL: https://blog.cloudflare.com/introducing-0-rtt/ (cited on page 13).

[SV07] N. Smart and F. Vercauteren. On computable isomorphisms in efficient asymmetric pairing-based systems. *Discrete Applied Mathematics*, 155(4):538–547, 2007. ISSN: 0166-218X. DOI: 10.1016/j.dam.2006.07.004 (cited on page 40).

[Ver01] E. R. Verheul. Self-blindable credential certificates from the Weil pairing. In C. Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 533–551. Springer Berlin Heidelberg, 2001. ISBN: 978-3-540-45682-7 (cited on page 17).

[Wei40] A. Weil. Sur les fonctions algébriques à corps de constantes fini. *Les Comptes Rendus de l'Académie des sciences*, 210:592–594, 1940 (cited on page 17).

Acronyms

(EC)DHE (Elliptic Curve) Diffie-Hellman Ephemeral. 9, 10, 66
0-RTT zero round-trip time. iv, 2, 3, 5, 10, 12, 13, 29, 66, 72

BBGHIBE Boneh-Boyen-Goh HIBE. 24, 26
BCDH Bilinear Computational Diffie-Hellman. 32
BDH Bilinear Diffie-Hellman. v, 18–20, 22
BDHI Bilinear Diffie-Hellman Inversion. 25, 26
BFE Bloom Filter Encryption. vii, viii, 2–4, 17, 29, 31–33, 35, 37–40, 42, 43, 45, 46, 48–53, 55–58, 64–68, 71, 72
BFIBE Boneh-Franklin IBE. 21, 22, 24, 40
BLS Barreto-Lynn-Scott. 18, 49–53, 60, 62–65, 68
BN Barreto-Naehrig. 18, 40, 45, 47–53, 60–65

CDN Content Delivery Network. 2
CPU central processing unit. 40, 43, 52

DH Diffie-Hellman. 8, 9, 17
DHE Diffie-Hellman Ephemeral. 9

ECDHE Elliptic Curve Diffie-Hellman Ephemeral. 9

FO Fujisaki-Okamoto. 32, 33

HIBE Hierarchical Identity Based Encryption. 6, 23, 24, 26, 35, 54, 64
HIBKEM Hierarchical identity based key encapsulation mechanism. 26–28, 37, 38
HTTP Hypertext Transfer Protocol. 13

IBE Identity Based Encryption. 17, 19–21, 23, 29, 40, 52, 54, 83
IND-CCA adaptive chosen ciphertext attack secure. 30, 32–35, 41
IND-CPA chosen plaintext attack secure. 20–24, 30, 32, 34, 35

Acronyms

IND-s-CPA selective chosen plaintext attack secure. 38
IND-sID-CPA selective identity chosen plaintext attack secure IBE. 26

JVM Java Virtual Machine. 45

KEM key encapsulation mechanism. 29

NIST National Institute of Standards and Technology. 40

OW-sID-CPA one-wayness under selective-ID and chosen-plaintext attacks. 27, 28, 38

PFSKEM puncturable forward secret key encapsulation. 33–35, 38
PKEM puncturable key encapsulation mechanism. 29–34
PRF pseudorandom function. 3, 9, 69, 71
PSK pre-shared key. 9, 12, 13, 66, 68

RSA Rivest–Shamir–Adleman. 8

TBBFE Time-Based Bloom Filter Encryption. vii, viii, 33, 35, 36, 38, 39, 54–56, 59–64, 72, 73
TLS Transport Layer Security. iv, vi–viii, 1–4, 6–12, 66–73

VRF verifiable random function. 17

XKCP eXtended Keccak Code Package. 42
XOF extendable output function. 40