Optimizing 0-RTT Key Exchange with Full Forward Security

Christian Göth
AIT Austrian Institute of Technology
Vienna, Austria

Sebastian Ramacher
AIT Austrian Institute of Technology
Vienna, Austria

Daniel Slamanig
AIT Austrian Institute of Technology
Vienna, Austria

Christoph Striecks
AIT Austrian Institute of Technology
Vienna, Austria

Erkan Tairi
TU Wien
Vienna, Austria

Alexander Zikulnig
AIT Austrian Institute of Technology
Vienna, Austria

ABSTRACT

Secure communication protocols such as TLS 1.3 or QUIC are doing the heavy lifting in terms of security of today's Internet. These modern protocols provide modes that do not need an interactive handshake, but allow to send cryptographically protected data with the first client message in zero round-trip time (0-RTT). While this helps to reduce communication latency, the security of such protocols in terms of forward security is rather weak.

In recent years, the academic community investigated ways of mitigating this problem and achieving full forward security and replay resilience for such 0-RTT protocols. In particular, this can be achieved via a so-called Puncturable Key Encapsulation Mechanism (PKEM). While the first such schemes were too expensive to be used in practice, Derler et al. (EUROCRYPT 2018) proposed a variant of PKEMs called Bloom Filter Key Encapsulation Mechanism (BFKEM). Unfortunately, these primitives have only be investigated asymptotically and no real benchmarks were conducted. Dallmeier et al. (CANS 2020) were the first to study their practical application within the QUIC protocol. They build upon a specific BFKEM instantiation and conclude that while it comes with significant computational overhead, its practical use is feasible, especially in applications where the increased CPU and memory load can be tolerated.

In this paper, we revisit their choice of the concrete BFKEM instantiation and show that by relying on the concept of Time-based BFKEMs (TB-BFKEMs), also introduced by Derler et al. (EUROCRYPT 2018), one can combine the advantages of having computational efficiency and smaller key sizes. We thereby investigate algorithmic as well as conceptual optimizations with various trade-offs and conclude that our approach seems favorable for many practical settings. Overall, this extends the applicability of 0-RTT protocols with strong security in practice.

CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**.

KEYWORDS

0-RTT key exchange, QUIC, TLS, Bloom Filter Key Encapsulation Mechanism

CCSW '23, November 26, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0259-4/23/11.
https://doi.org/10.1145/3605763.3625246

ACM Reference Format:
Christian Göth, Sebastian Ramacher, Daniel Slamanig, Christoph Striecks, Erkan Tairi, and Alexander Zikulnig. 2023. Optimizing 0-RTT Key Exchange with Full Forward Security. In *Proceedings of the 2023 Cloud Computing Security Workshop (CCSW '23), November 26, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3605763.3625246

1 INTRODUCTION

Forward security is an essential security feature of utmost practical relevance as it mitigates the consequences of secret-key leakage. Most prominently known from key-exchange (KE) protocols, forward security evolves secret-key material over time with the result that, e.g., in the KE setting, older sessions stay secure once a newer key is leaked. Consequently, "store-now-decrypt-later" attacks become more difficult. Looking at practical applications, over 99% of Internet sites[1] offer at least some form of forward security. Moreover, such a feature is included in numerous products from large companies such as Apple[2], Cloudflare[3], Google[4], and Microsoft[5].

The cryptographic literature has a long body of research on the topic of forward security. Besides interactive key-exchange protocols (such as TLS 1.3, QUIC, hybrid KE, or ratcheting) [13, 18, 25, 32, 41], the feature is researched in further areas such as public-key encryption [14, 31], digital signatures [5, 26], search on encrypted data [12], updatable cryptography [43], mobile Cloud backups [19], proxy cryptography [23], new approaches to Tor [36], and distributed key management [24], among others.

Our focus will be on forward security of KE protocols and in particular of zero round-trip time (0-RTT) KE protocols. In such protocols, encrypted data can be sent immediately with the first message from the client to the server. This is highly desirable in practice as it combines strong forward security guarantees with significantly reduced latency of communication. Latter property is of utmost importance in many applications in particular the Internet.[6] However, a naive realization, i.e., taking the receivers public key and already encrypting payload in the first message would not provide forward security. Commonly used key exchange protocols that achieve a strong form of forward security (or, "full forward security") still require one round-trip. Only after the two shares have been exchanged, the shared key can be derived and thus a forward-secure channel be established. Hence, the initiator

[1] Due to Qualys SSL Labs, https://www.ssllabs.com/ssl-pulse/, accessed in Sept. 2023.
[2] https://support.apple.com/en-my/guide/security/sec100a75d12/web
[3] https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details
[4] https://security.googleblog.com/2011/11/protecting-data-for-long-term-with.html
[5] https://azure.microsoft.com/de-de/blog/tlsssl-cipher-suite-enhancements-and-perfect-forward-secrecy
[6] https://blog.cloudflare.com/introducing-0-rtt/

needs to wait for the responder's key share before being able to send the first encrypted message.

Consequently and as we will discuss soon, commonly deployed 0-RTT protocols only provide a reduced form of forward security. This is undesirable and thus forward security has received quite some attention over the last years [16, 21, 22, 33] as it helps to significantly reduce the latency while achieving strong security guarantees and replay protection.

0-RTT protocols in practice. One protocol that has been designed with a focus on 0-RTT is QUIC [40], which is standardized as RFC 9000 by the IETF [35]. Omitting many technical details, the part relevant to 0-RTT and forward security is that it implements a custom cryptographic protocol based on Diffie-Hellman (DH). Essentially, the very first connection of a client to a server requires a 1-RTT KE, during which a medium-lived (typically two days) DH share g^s of the server is authentically delivered to the client. On every subsequent connection, g^s is used as a static DH share of the server with a fresh ephemeral share g^x of the client to initiate a 0-RTT key exchange, i.e., use a symmetric key derived from g^{xs} to encrypt payload already with the first message. The important point is that the server re-uses the share g^s for all sessions within the lifetime of the medium-lived static DH share. Consequently, an attacker which obtains the secret s (which needs to be kept by the server in this time window, e.g., for two days) can break the confidentiality of all sessions that have used g^s. Consequently, only a weak form of forward security can be achieved [37]. Moreover, without additional application-layer countermeasures there is also no inherent replay protection (cf. [18]).

The probably most well-known protocol, the Transport-Layer Security (TLS) protocol, as one of the major innovations in its version 1.3 [39] has also introduced a 0-RTT feature. Again omitting many technical details, the idea is that the client and the server in their very first connection perform a full (non 0-RTT) TLS handshake. From the shared secret obtained during this handshake, they then can derive a so-called resumption secret, which will be used in later sessions. The client simply stores this secret whereas the server then later on resuming a session needs to retrieve it from somewhere. There are two different modes, i.e., session caches (the client stores a lookup key to the secret in a database at the server) and session tickets (the resumption key is encrypted using a server long-term key and is outsourced to the client). Every subsequent connection can then use a 0-RTT handshake, where the client in its first message will either include the lookup key (session caches) or the encrypted session ticket (session tickets) together with a DH share g^x. Additionally, the client can send payload data encrypted with a key derived from the resumption secret and some public client random value. The server can then also send an ephemeral DH share g^s, so that further on g^{xs} can be used to encrypt payload and only the initial data in the first message is solely protected by the resumption secret.

When using session caches, the server can at any point delete the resumption key and one can achieve forward security as well as replay protection. However, the 0-RTT part of all sessions that use the same resumption key can be broken once this secret leaks. Hence, in order to achieve full forward security, one essentially falls back to non-0-RTT modes. Alternatively, if session tickets are used, an attacker compromising the long-term key used to encrypt the resumptions keys can break forward security completely. Moreover, the use of session tickets is also generally vulnerable to replay attacks. We will subsequently not consider protocols with session resumptions and refer the reader to a more in-depth discussion of forward security in such protocols and solutions to these problems to [3, 4].

0-RTT protocols via puncturable encryption. In [33], Günther et al. observed that with puncturable encryption (PE) [29] or, more precisely, puncturable key encapsulation mechanisms (PKEMs), one can realize 0-RTT KE with full forward security and inherent replay protection – something not known to be feasible before. In a PKEM, one can puncture the secret key used to the decrypt a ciphertext in a way, that the punctured secret key can no longer decrypt such a ciphertext. Consequently, one can thus obtain forward security and puncturing the key also provides replay protection. This new approach requires that the client already knows the public key of the server before establishing the connection. In the context of latency-sensitive applications such as the Internet of Things (IoT), we observe that often clients communicate only with a pre-configured server. Therefore, the public key of the server can already be deployed during provisioning of the devices. In case where this is not possible, only during the first connection between a client and the server, a non-0-RTT key exchange has to be performed. The client will receive the public key during this connection and can store it for future use.

Practicality of puncturable key encapsulation. A huge drawback of the aforementioned PKEM by Günther et al. is that the puncturing efficiency is far away from what can be tolerated in a practical setting. Derler et al. [21, 22] proposed a variant of PE called Bloom Filter Key Encapsulation (BFKEM) that achieves very efficient puncturing by tolerating a non-negligible correctness error, something that is perfectly acceptable for the application in forward-secure 0-RTT KE. Their conventional BFKEM construction thereby trades efficiency for quite large secret keys. Besides, they propose a notion of time-based BFKEM (TB-BFKEM) that combines the advantages of having efficient puncturing and smaller keys at the cost of a somewhat more expensive interval puncturing, i.e., switching from one time period to the other. But even this interval puncturing is still orders of magnitude faster then the puncturing of the construction by Günther et al. Unfortunately, all these works only provide asymptotic comparison of efficiency and, thus, it is unclear how such schemes perform in practice.

Integration of PKEM into QUIC. In a recent work, Dallmeier et al. [18] discuss the integration of one particular instantiation of BFKEM from [21] based on the identity-based broadcast encryption (IBBE) scheme by Delerablée [20] into QUIC (see Figure 1 for an illustration of their implementation).

While their concrete choice of the primitives allows to achieve constant-size ciphertexts and keeping the public and secret keys reasonably small, it is computationally expensive and does not allow for many trade-offs. As a consequence, a natural question is whether one can do better in terms of trade-offs that are interesting for practical applications.

Open questions. Unfortunately, besides the work by Dallmeier et al. [18], so far there are only asymptotic comparisons between

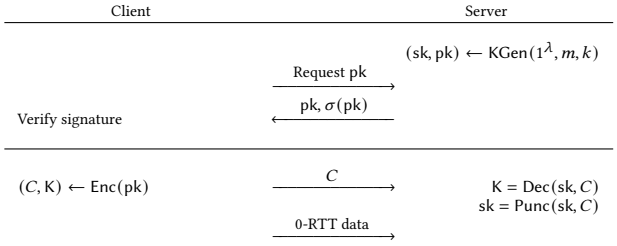


Figure 1: Simplified QUIC handshake protocol as implemented in Dallmeier et al. [18]. KGen **computes the server's secret and public keys** (sk, pk) **based on the Bloom-filter parameter** m **and** k**,** Enc **encapsulates the session key** K **on the client side,** Dec **decapsulates such key on the server's side, while** Punc **punctures the** sk **on the retrieved ciphertext** C**. Moreover,** $\sigma(\mathsf{pk})$ **reflects the server's signature on the public key using a long-term signing key (which has only to be sent if** pk **is not known to the client yet). Using** K**, data can be sent in 0-RTT while achieving forward security and replay protection.**

different constructions and in particular between BFKEMs and TB-BFKEMs. Consequently, it is not entirely clear how good these schemes for parameters of practical interest perform in practice. In doing so, we want to explore different paths for optimizing the use of (TB)-BFKEM by *i*) looking at the underlying concept of a Bloom Filter and *ii*) exploring different design approaches and trade-offs. Moreover, we want to provide an implementation that uses various different optimization and provide extensive benchmarks. This can guide practitioners for making the optimal choices when knowing the constraints of a given application scenario.

Our contribution. Our contribution in this paper is threefold.

First, we revisit the use and parametrization of the Bloom filter in (TB-)BFKEMs. Thereby, we show a number of positive results and in particular:

- Make encapsulation and decapsulation more efficient and ciphertexts smaller. We can reduce the number k of hash functions for the Bloom filter if we allow for a small increase of the secret-key size (scaling in m). For example, reducing k by 45% while increasing m only by 15% (cf. Section 3.2).
- Reduce the secret key size exponentially (in t) while keeping runtimes low and, in particular, independent of t (cf. Section 4.2).
- Further reduce the secret key size by using a ternary instead of binary tree (cf. Section 4.3).
- Further reduce the secret key size by using all nodes instead of just the leaf nodes in the tree (cf. Section 4.4).

Besides the above results, we also investigate some paths that either do not lead to improvements or would require schemes that are not yet available from the literature. We looked at probabilistic puncturing. Unfortunately, such a puncturing approach introduces a false-negative probability and the key size can only be reduced by a factor of $(1-q)$ where q is essentially the false-negative probability (which needs to be kept very low). Moreover, we investigated whether in a TB-BFKEM we can directly manage the secret keys in a binary tree using a hierarchical identity-based encryption (HIBE) scheme [10, 27, 34]. This would only work for HIBE schemes with constant-size secret keys and where the constant is at most 2. Unfortunately, to best of our knowledge, we are not aware of any such HIBE scheme. Due to the lack of space, we report those negative findings in more detail in Appendix A.

Second, we provide an optimized C implementation of a TB-BFKEM from the Boneh-Boyen-Goh (BBG) HIBE [10] where we provide a number of optimizations. In particular, we use a common and highly performant strongly secure one-time signature scheme (i.e., the EdDSA scheme) for the compiler to achieve security against chosen-ciphertext attacks. Moreover, we implement algorithmic measures for performance improvement such as parallelization as well as the use of small pre-computation tables to improve key generation and interval puncturing.

Third, we provide exhaustive benchmarking as well as a comparison with the BFKEM instantiation in context of QUIC by Dallmeier et al. [18]. We show that with TB-BFKEM one can overcome the need to switch public keys every two days. In particular, we can obtain a secret key size that is comparable but smaller to the implementation due to Dallmeier et al. And with the same performance characteristics, our TB-BFKEM approach can easily scale the number of puncturings to support key lifetimes of multiple months or to support significantly more connections per second. We remark that we our comparison is only at the BFKEM level and for a comparison of (TB)-BFKEM based key 0-RTT exchange with the 0-RTT version provided by QUIC, we refer the reader to [18].

2 PRELIMINARIES

In this section, we provide our notation and the building blocks.

Notation. For $n \in \mathbb{N}$, let $[n] := \{1, \ldots, n\}$, and let $\lambda \in \mathbb{N}$ be the security parameter. For a finite set $\mathcal{S}$, we denote by $s \xleftarrow{\$} \mathcal{S}$ the process of sampling s uniformly from $\mathcal{S}$. For an algorithm A, let $y \xleftarrow{\$} A(x)$ be the process of running A on input x with access to uniformly random coins and assigning the result to y. We say an algorithm A is probabilistic polynomial time (PPT) if the running time of A is polynomial in λ. A function f is negligible if its absolute value is smaller than the inverse of any polynomial (i.e., if $\forall c \exists k_0 \forall \lambda \geq k_0 : |f(\lambda)| < 1/\lambda^c$).

2.1 Bloom Filters

A Bloom filter (BF) [8] is a probabilistic data structure for the approximate set membership problem. It allows a succinct representation T of a set $\mathcal{S}$ of elements from a large universe $\mathcal{U}$. For elements $s \in \mathcal{S}$ a query to the BF always answers 1 ("yes"), i.e., its false-negative probability is 0. Ideally, a BF would always return 0 ("no") for elements $s \notin \mathcal{S}$, but the succinctness of the BF comes at the cost that for any query to $s \notin \mathcal{S}$ the answer can be 1, too, but only with small probability (the false-positive probability or fpp).

We will only be interested in the original construction of Bloom filters [8] and omit a general abstract definition. Instead, we describe the construction from [8] directly. For a general definition we refer to [38].

Definition 2.1 (Bloom Filter). A *Bloom filter* BF for set $\mathcal{U}$ consists of algorithms $\mathsf{BF} = (\mathsf{BFGen}, \mathsf{BFUpdate}, \mathsf{BFCheck})$, which are defined as follows.

$\mathsf{BFGen}(m, k)$**:** This algorithm takes as input two integers $m, k \in \mathbb{N}$. It first samples k universal hash functions $H_1, \ldots, H_k$,

where $H_j : \mathcal{U} \to [m]$, defines $H := (H_j)_{j\in[k]}$ and $T := 0^m$, and outputs (H, T).

BFUpdate(H, T, u): Given $H = (H_j)_{j\in[k]}$, $T \in \{0,1\}^m$, and $u \in \mathcal{U}$, this algorithm defines the updated state T' by first assigning $T' := T$. Then, it sets $T'[H_j(u)] := 1$ for all $j \in [k]$, and finally returns T'.

BFCheck(H, T, u): Given $H = (H_j)_{j\in[k]}$, $T \in \{0,1\}^m$, and $u \in \mathcal{U}$, this algorithm returns a bit $b := \bigwedge_{j\in[k]} T[H_j(u)]$.

Bounded false-positive probability. The probability that an element which has not yet been added to the Bloom filter is erroneously "recognized" as being contained in the filter can be made arbitrarily small, by choosing m and k adequately, given (an upper bound on) the size of $\mathcal{S}$.

More precisely, let $\mathcal{S} = (s_1, \ldots, s_n) \in \mathcal{U}^n$ be any vector of n elements of $\mathcal{U}$. Then for any $s^* \in \mathcal{U} \setminus \mathcal{S}$, the false positive probability μ is bounded by

$$\mu := \Pr\left[\mathsf{BFCheck}(H, T_n, s^*) = 1\right] \leq \left(1 - e^{-\frac{(n+1/2)k}{m-1}}\right)^k, \quad (1)$$

where $(H, T_0) \xleftarrow{\$} \mathsf{BFGen}(m, k)$, $T_i = \mathsf{BFUpdate}(H, T_{i-1}, s_i)$ for $i \in [n]$, and the probability is taken over the random coins of BFGen. See Goel and Gupta [28] for a proof of this bound.

Classical choice of parameters. Suppose we are given an upper bound n on the number of elements inserted into the Bloom filter, and an upper bound p on the false-positive probability for this number of elements that we can tolerate. Our goal is to determine the size m of the Bloom filter and the number k of hash functions to achieve a false positive probability of $\mu \leq p$ with respect to n.

Using inequality Equation (1) the choice of

$$m := \left\lceil \frac{-(n+1/2)(\log_2 p + 1)}{\ln 2} \right\rceil + 1 \text{ and } k := \lfloor \frac{(m-1)\ln 2}{n+1/2} \rfloor,$$

yields the desired bound $\mu \leq 2^{-k} \leq p$ on the false-positive probability of the Bloom filter.

2.2 (Hierarchical) Identity-Based Encryption

We present the basic definition of HIB-KEMs.

Definition 2.2. A l-level hierarchical identity-based key encapsulation scheme (HIB-KEM) with identity space $\mathcal{D}^{\leq l}$, ciphertext space $\mathcal{C}$, and key space $\mathcal{K}$ consists of the following four algorithms:

HIBGen(1^λ): Takes as input a security parameter and outputs a key pair $(\mathsf{mpk}, \mathsf{sk}_\varepsilon)$. We say that mpk is the master public key, and sk_ε is the *level-0 secret key*.

HIBDel$(\mathsf{sk}_{\vec{d}'}, d)$: Takes as input secret key $\mathsf{sk}_{\vec{d}'}$ and $d \in \mathcal{D}$, and outputs a secret key $\mathsf{sk}_{\vec{d}'|d}$. (We refer to $|$ as concatenation.)

HIBEnc$(\mathsf{mpk}, \vec{d})$: Takes as input the master public key mpk and an identity $\vec{d} \in \mathcal{D}^{\leq l}$ and outputs a ciphertext $C \in \mathcal{C}$ and a key $\mathsf{K} \in \mathcal{K}$.

HIBDec$(\mathsf{sk}_{\vec{d}}, C)$: Takes as input a secret key $\mathsf{sk}_{\vec{d}}$ and a ciphertext C, and outputs a value $\mathsf{K} \in \mathcal{K} \cup \{\perp\}$, where $\perp$ is a distinguished error symbol.

Correctness for HIB-KEM. We require that for all $\lambda \in \mathbb{N}$, for all $(\mathsf{mpk}, \mathsf{sk}_\varepsilon) \xleftarrow{\$} \mathsf{HIBGen}(1^\lambda)$, for all $d \in \mathcal{D}$, for all $\mathsf{sk}_{\vec{d}'|d} \xleftarrow{\$} \mathsf{HIBDel}(\mathsf{sk}_{\vec{d}'}, d)$, for all $\vec{d} \in \mathcal{D}^{\leq l}$, for all $(C, K) \xleftarrow{\$} \mathsf{HIBEnc}(\mathsf{mpk}, \vec{d})$, we have that $\mathsf{HIBDec}(\mathsf{sk}_{\vec{d}}, C) = K$ holds.

In the following, we want to present different variants of HIBEs which we will later consider for our schemes according to their parameter sizes.

Boneh-Boyen (BB). The first HIBE we want to take a look at is the one presented in [9]. Here a HIBE secret key on level k contains $k+2$ group elements and thus the secret key grows with the depth of an identity. Also Enc gets more expensive when the depth of an identity is increased as it needs to perform basically $k + 2$ exponentiations with a resulting ciphertext of $k + 2$. The algorithm Dec computes $k + 1$ pairings and k exponentiations.

Boneh-Boyen-Goh (BBG). The BBG HIBE [10] has the opposite behavior compared to the previous one concerning secret key sizes. The private key shrinks with the depth of an identity i.e. a key on level k in a HIBE of depth l contains $2 + l - k$ group elements. We note that decryption takes only 2 pairings and $e(g_1, g_2)$ can be pre-computed for encryption and therefore no pairing needs to be computed for encryption. One of the main advantages is the constant size of ciphertexts for identities on different levels.

Hybrid scheme (BBG-H). The work in [10] also presents a combination of the previous two HIBE schemes. This construction aims to combine those two for a given parameter $\omega \in [0, 1]$. By this one can achieve secret key sizes of $O(l^\omega + l^{1-\omega})$ and ciphertexts of size $O(l^\omega)$ for a HIBE of depth l.

Lattice based HIBE. It is also possible to construct lattice-based HIBEs. A construction with relatively short ciphertext and secret keys is presented in [2]. Their HIBE in the random oracle model has ciphertext sizes of $O(nl^2)$ and secret key sizes of $O(kn^2l^2)$ where n denotes the security parameter.[7] One important point to notice is that especially sizes of secret keys of identities with increasing depth become huge very soon.

3 IMPROVING BLOOM-FILTER KEMS

In this section, we provide improvements for Bloom-Filter KEMs (BFKEMs). We start with recalling the formal model and properties.

3.1 Formal Model and Properties of BFKEM

Definition 3.1 (BFKEM). A Bloom Filter key encapsulation scheme (BFKEM) with key space $\mathcal{K}$ is a tuple (KGen, Enc, Punc, Dec) of PPT algorithms:

KGen$(1^\lambda, m, k)$: Takes as input a security parameter λ, parameters m and k, and outputs a secret and public key (sk, pk) (we assume that $\mathcal{K}$ is implicit in pk, and that pk is implicit in sk).

Enc(pk): Takes as input a public key pk, and outputs a ciphertext C and a symmetric key K.

Punc(sk, C): Takes as input a secret key sk and a ciphertext C, and outputs an updated secret key sk′.

Dec(sk, C): Takes as input a secret key sk and a ciphertext C, and deterministically computes and outputs a symmetric key K or $\perp$ if decapsulation fails.

[7] Note that in the previously mentioned HIBEs based on bilinear maps we are talking about the number of group elements and therefore do not mention the security parameter explicitly in the asymptotic behavior of ciphertext and key sizes.

Correctness. We start by defining correctness of a BFKEM scheme. Basically, here one requires that a ciphertext can always be decapsulated with unpunctured secret keys. However, we allow that if punctured secret keys are used for decapsulation then the probability that the decapsulation fails is bounded by some non-negligible function in the scheme's parameters m, k.

Definition 3.2 (Correctness). We require that the following holds for all $\lambda, m, k \in \mathbb{N}$ and any $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{KGen}(1^\lambda, m, k)$.

For any (arbitrary interleaved) sequence of invocations of

$$\mathsf{sk}_{j+1} \xleftarrow{\$} \mathsf{Punc}(\mathsf{sk}_j, C_j),$$

where $j \in \{1, \ldots, n\}$, $\mathsf{sk}_1 := \mathsf{sk}$, and $(C_j, \mathsf{K}_j) \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk})$, it holds that

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}_{n+1}, C^*) \neq K^*\right] \leq \left(1 - e^{-\frac{(n+1/2)k}{m-1}}\right)^k + \epsilon(\lambda),$$

where $(C^*, \mathsf{K}^*) \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk})$ and $\epsilon(\cdot)$ is a negligible function in λ. The probability is over the random coins of KGen, Punc, and Enc.

The bound $\left(1 - e^{-\frac{(n+1/2)k}{m-1}}\right)^k$ is motivated by the bound achievable by Bloom filters, cf. Equation (1).

Extended correctness. In [21], the definition of extended correctness is provided. In the following, we will only need to define the formal notion of *Impossibility of false-negatives*. We require that the following holds for all $\lambda, m, k, n \in \mathbb{N}$ and any $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{KGen}(1^\lambda, m, k)$.

For any (arbitrary interleaved) sequence of invocations of

$$\mathsf{sk}_{j+1} \xleftarrow{\$} \mathsf{Punc}(\mathsf{sk}_j, C_j)$$

where $j \in \{1, \ldots, n\}$, $\mathsf{sk}_1 := \mathsf{sk}$, and $(C_j, \mathsf{K}_j) \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk})$, it holds that

$$\mathsf{Dec}(\mathsf{sk}_{n+1}, C_j) = \perp \text{ for all } j \leq n. \tag{2}$$

Construction. We will not provide the construction here since it can be found in [21]. But we do want to recall the coarse idea of the construction and asymptotic parameter sizes as well as runtimes.

The secret key contains m identity-based encryption (IBE) secret keys for each identity $i \in [m]$ and therefore grows linearly in the size of parameter m. For encapsulation, the algorithm randomly creates a "unique" (with overwhelming probability) ciphertext component from which k identities are deterministically computed by applying each of the k hash functions of the Bloom filter. Each of these identities is now used for encrypting the same key K with the underlying IBE scheme. Thus, the ciphertext consists of the randomly created ciphertext component and k IBE encryptions sharing the same randomness. The Puncturing of a secret key on a specific ciphertext only deletes the IBE secret keys of the corresponding k identities and is very efficient. Decapsulation Dec needs to perform at most k Bloom-filter checks and (in case at least one of the secret keys has not already been deleted) one IBE decryption.

It is important to emphasize that it is possible to construct a BFKEM from any IBE [16]. In the following, we will only focus on the BFKEM based on hashed Boneh-Franklin IBE [21].

3.2 Non-Classical Choice of Bloom-Filter Parameters

In Section 2.1, we have shown the classical choice of Bloom-filter parameters k and m for given values of μ and n. In the following section, we want to take a look at a different choice for those parameters in order to improve aspects of our BFKEM.

Note that the presented classical choice essentially means (independent of m) for the number of hash functions that $k \approx -\log_2(p)$. For the following considerations, we will only choose values of $p = 2^{-e}$ with $e \in \mathbb{N}$ and therefore we can just assume $k = e = -\log_2 p$.

Choosing larger values for k. We stick to the setting of given values n and p and we get to choose m and k. While not changing the classical value of m, we do want to consider choosing a larger number of hash functions, i.e., $k > -\log_2 p$. We already know that regarding the fpp after exactly n inserts, we will not perform better by this non-optimal choice. But it is still interesting to look at the behavior of the fpp after $\alpha < n$ inserts. As it has already been mentioned in [21, 22], the fpp is overwhelmingly low if only a fraction of the n elements have been added to the Bloom filter. We now want to focus on the behavior at the upper half of insertions, i.e., $n/2 < \alpha < n$. Let us look at the following example to be more precise. Given the parameters $n = 2^{20}$ and $p = 2^{-11}$, the optimal choice of parameters would be $m \approx 2^{24}$ and $k = 11$. In Figure 2a, we visualize the evolution of the fpp after $\alpha \in [n/2, n]$ inserts for $k = 11$ as well as parameters $k > 11$.

We can conclude that we do get an improvement of a lower fpp for a certain range of inserts. But these improvements are only marginal since they are reached at levels where the current fpp is already much lower than our desired upper bound p. In addition, we lose our upper bound when choosing larger values of k. Moreover, for a BFKEM, a larger value of k means worse performance for encapsulation and decapsulation, and typically also larger ciphertexts. Thus, we can conclude that a larger value of k, the gains are insignificant and outweigh by their disadvantages.

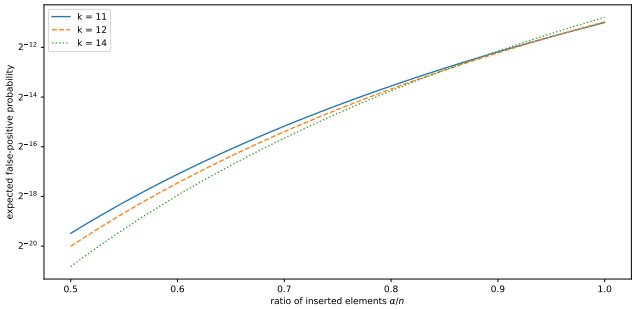
Choosing smaller values for k. Now one can ask whether choosing smaller values of k could be helpful. Taking the same parameters as above, in Figure 2b, we plot the fpp for $k = 11$ as well as parameters $k < 11$.

We can immediately see that for smaller values of k, we do not only lose the upper bound on the fpp after n inserts but it also behaves worse after only $\alpha < n$ inserts. However, we gain a lower runtime for the encapsulation/decapsulation and smaller ciphertexts as mentioned above and in this case it could be worth it to not keep the parameter m fixed but increase it for lower values of k, i.e., trading better performance against higher key size. For a given upper bound p on the fpp, we take another look at inequality Equation (1) and after some manipulations, we can state that this inequality holds for

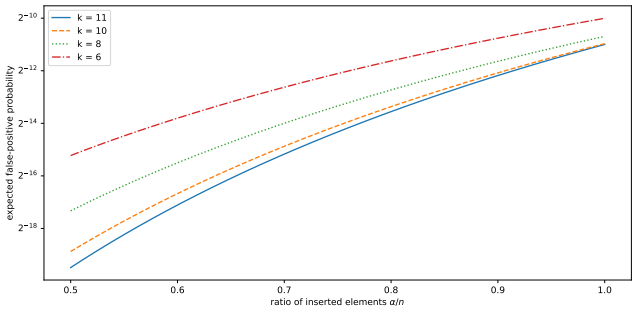
$$m_k \geq 1 - \frac{(n + \frac{1}{2})k}{\ln(1 - \sqrt[k]{p})}.$$

Of course this m_k is bigger than the optimal choice of m if $k < -\log_2(p)$.

Figure 3, we can see how much bigger the bloom filter size m_k is in percentage compared to the optimal choice of m. It is interesting to observe how we only need to increase the size of m by around



(a) $k \geq 11$



(b) $k \leq 11$

Figure 2: fpp after α inserts for $n = 2^{20}, p = 2^{-11}, m \approx 2^{24}$ and different values of k.

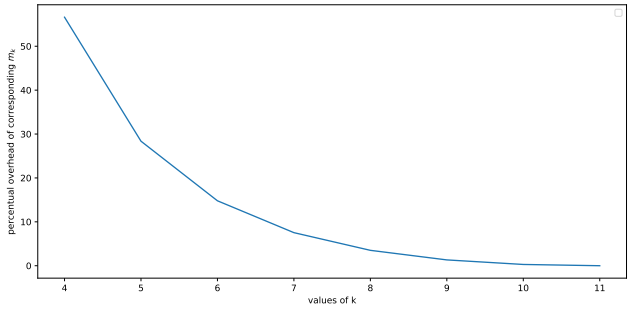


Figure 3: Comparison of needed Bloom filter sizes for $n = 2^{20}, p = 2^{-11}$.

15% when we reduce the number of hash functions to 6 and thereby nearly halve the cost for encapsulation and the size of ciphertexts. Now that we have determined the suitable choice of m_k in order to achieve the upper bound p for the fpp, we should not ignore the fact that the behavior of the fpp is different before n inserts.

Figure 4 illustrates that we still get a higher fpp before n inserts for smaller k even if we increase the size of the bloom filter, m. One has to take that into consideration when using the presented strategy. But we can conclude that this strategy can be a reasonable choice when one wants to achieve a better performance and can accept a slightly bigger key size.

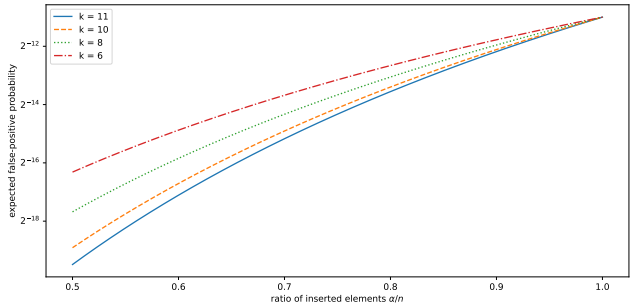


Figure 4: fpp after α inserts for $n = 2^{20}, p = 2^{-11}$ and different values of k with the corresponding m_k.

4 TIME-BASED BLOOM-FILTER KEMS

The ideas presented so far do not provide a huge impact on the desired reduction of key-sizes. Consequently, we take a closer look at the optimization potential for *time-based* BFKEMs (TB-BFKEMs) and note that the previous finding can also be applied in this setting. In this approach, ciphertexts are associated with time slots and one assumes loosely synchronized clocks between sender and receiver of a ciphertext. Besides the trivial approach of designing a TB-BFKEM by assigning an individual key-pair for each time slot, the main advantage of the TB-BFKEM in [21] is to achieve a logarithmic reduction of the secret key by using a *time-tree* based on HIBEs.[8] In particular, one can use 2^t time slots (and their corresponding secret keys) while only needing a time-tree of depth t. In the following, we want to take a closer look at the behavior of key-sizes with a focus on the question if we can indeed accomplish a logarithmic reduction of the key-size when using the time-based approach.

4.1 Model and Construction of TB-BFKEM

We stick to the notations used in [21] and present the definition of TB-BFKEM.

Definition 4.1 (TB-BFKEM). A puncturable forward-secret key encapsulation (TB-BFKEM) scheme is a tuple of the following PPT algorithms:

- $\mathsf{KGen}(1^\lambda, m, k, t')$: Takes as input a security parameter λ, parameters m and k for the Bloom filter, and a parameter t' specifying the number of time slots. It outputs a secret and public key $(\mathsf{sk}, \mathsf{pk})$, where we assume that the key-space $\mathcal{K}$ is implicit in pk and that pk is implicit in sk.
- $\mathsf{Enc}(\mathsf{pk}, \tau)$: Takes as input a public key pk and a time slot τ and outputs a ciphertext C and a symmetric key K.
- $\mathsf{PuncCtx}(\mathsf{sk}, \tau, C)$: Takes as input a secret key sk, a time slot τ, a ciphertext C and outputs an updated secret key sk'.
- $\mathsf{Dec}(\mathsf{sk}, \tau, C)$: Takes as input a secret key sk, a time slot τ, a ciphertext C and deterministically computes and outputs a symmetric key K or $\perp$ if decapsulation fails.

[8] Another aspect worth mentioning is that we also achieve a form of additional delayed forward secrecy between intervals, as the usage of different time slots counters message suppression attacks.

PuncInt(sk, τ): Takes as input a time slot τ and a secret key sk for any time slot $\leq \tau$, and outputs an updated secret key sk′ for the time slot $\tau + 1$.

For formal definitions of security and correctness we refer the reader to [21] as we do not require it for our further treatment here.

Intuition of the construction. For the detailed description of our TB-BFKEM construction, we refer once again to [21]. The idea of the construction is to use a tree of depth $t + 1$ with $t' := 2^t$ level-t nodes each representing one of the t' time slots. Each of those nodes has a fixed number m_t of children representing the Bloom filter of this time slot which makes this tree represent a $(t + 1)$-level HIB-KEM with identity space $\{0, 1\}^t \times [m_t]$. This allows us not only to puncture the secret key on specific ciphertexts by deleting BF-keys but we can also puncture time intervals by traversing the t-th level of the tree from left to right. Since almost all algorithms are basically derived from the known ones of aforementioned BFKEM and HIB BFKEM schemes, we will only focus on the PuncInt algorithm.

PuncInt(sk, τ): Given a secret key $\mathsf{sk} = (T, \mathsf{sk}_{\mathsf{Bloom}}, \mathsf{sk}_{\mathsf{time}})$ for time interval $\tau' \leq \tau$, the time puncturing algorithm proceeds as follows. First, it resets the Bloom filter by setting $T := 0^m$. Then it uses the key delegation algorithm to first compute sk_τ. This key can be computed from the keys contained in $\mathsf{sk}_{\mathsf{time}}$, because sk is a key for time interval $\tau' \leq \tau$. Then it computes

$$\mathsf{sk}_{\tau|d} \xleftarrow{\$} \mathsf{HIBDel}(\mathsf{sk}_\tau, d) \quad \text{for all} \quad d \in [m],$$

and redefines $\mathsf{sk}_{\mathsf{Bloom}} := (\mathsf{sk}_{\tau|d})_{d\in[m]}$. Finally, it updates $\mathsf{sk}_{\mathsf{time}}$ by computing the HIB-KEM secret keys associated to all *right-hand siblings* of nodes that lie on the path from node τ to the root and adds the corresponding keys to $\mathsf{sk}_{\mathsf{time}}$. Then it deletes all keys from $\mathsf{sk}_{\mathsf{time}}$ that lie on the path from τ to the root.

Offline interval puncturing. Note that puncturing between time intervals may become relatively expensive. Depending on the choice of Bloom-filter parameters, in particular on m, this may range between 2^{15} and 2^{25} HIBKEM key delegations. However, the main advantage of TB-BFKEMs over previous constructions of puncturable encryption is that these computations must not be performed "online", i.e., during puncturing, but can actually be computed separately (for instance, parallel on a different computer, or when a server has low workload, etc.).

Remark on CCA Security. We can add another HIBE level to obtain IND-CCA security via the CHK transform [15] in the standard model, and thus to avoid random oracles if required. We note, however, that one cannot straightforwardly apply the CHK transform in a black-box way, but needs to take care that all k HIB-KEM ciphertexts C_j, $j \in [k]$ need to use *the same* verification key of the strong one-time signature used to sign the overall ciphertext. A reasonable choice for the signature scheme is Schnorr [42] or EdDSA [6, 7] or to even use the MAC-based compiler in [11].

4.2 Key Sizes and Runtimes

In the following, we want to focus on the scenario of a given amount of puncturings that we want to perform in a given period of time (e.g. $n = 2^{18}$ puncturings in one day). In the original BFKEM setting, we would need a BF size of around $m = 1.44 \log_2(1/p)n$ yielding a size of the secret key in the order of m. In our example, we could choose $p = 2^{-11}$ and get a value of $m \approx 2^{22}$. For this fixed number of n resp. m, we want to analyze different choices of t in order to split the day into 2^t time slots each one allowing for $n_t = n/2^t$ puncturings reducing the BF size to $m_t = m/2^t$.[9]

Key sizes. We need to emphasize that for the following considerations we instantiate our BFKEM with the hashed Boneh-Franklin IBE and the TB-BFKEM with the BBG HIBE. The reason for the latter is as follows. If we keep the number of time intervals at a reasonable low value, we still have a decent number of BF keys m_t that we need to store at least in the beginning of each time slot. In order to reduce the total size of the secret key, we are therefore interested in using a HIBE in which the HIBE keys are as small as possible for a high level k of the corresponding node. Thus, the chosen BBG HIBE promises the best reduction of the secret-key size among all HIBEs presented in Section 2.2.

The maximum number of keys has to be stored at the beginning of the first time slot – namely one level-k key for each $k \in \{1, \ldots, t\}$ and m_t level-t+1 keys. Using the fact that for BBG of depth l, level-k HIBE keys consist of $2 + l - k$ group elements, this collection of keys gives us a total number of group elements as

$$\sum_{k=1}^{t} 2 + (t + 1) - k + 2m_t = \frac{1}{2}t^2 + \frac{5}{2}t + \frac{m}{2^{t-1}}.$$

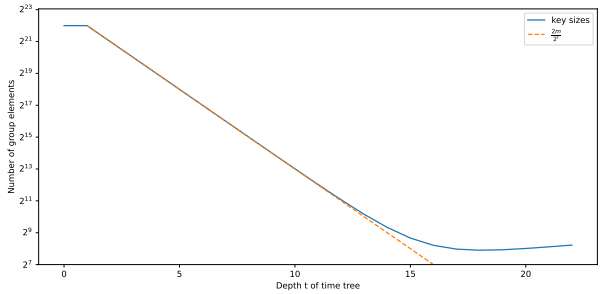


Figure 5: Maximal number of group elements in secret key for $n = 2^{18}$ puncturings and $p = 2^{-11}$ using BBG.

In Figure 5, one can see the evolution of this expression in t for specific values of n and p. For $t = 0$ we are in the classical BFKEM setting, where the maximal key just consists of the $m = 2^{22}$ group elements. After that, the key size decreases (and is even halved in the beginning as one can observe by comparison with the dashed line) until t reaches a value between 18 and 19 and from there on, the key size is slightly increasing. Since the minimizing value t_{min} is hard to determine analytically in the general case, we just mention that computing this minimizer for reasonable values of m yields values ranging from $\log_2(m) - 5$ to $\log_2(m) - 3$. We can see that for such values of t the Bloom filter has a size of only 2^3 to 2^5 which is rather problematic for practical applications. Firstly, the interval puncturing has to be invoked too often and, secondly and more problematic, in the 0-RTT KE application, it requires a very good

[9] Notice that for simplicity we assume a partitioning of the time period such that the puncturings are distributed evenly among the intervals.

estimate on how many puncturings happen per unit time so that the multiple clients know which time interval will be the valid one. So we can take as a result that the expression is strictly decreasing for reasonable values of t – observing that for small values of t it is nearly reducing by half when adding one more layer to the time tree.

Runtimes. We now want to analyze for the algorithms of the TB-BFKEM how their runtimes behave for different values of t.

Encryption performs a total of k HIB-KEM key encapsulations for HIB-KEM identities of level $t+1$ as well as a key and signature generation for the one-time signature scheme. For BBG, each of those needs to compute $t+2$ multiplications and 3 exponentiations, and we can precompute the values of $h_j^{H(0)}$ and $h_j^{H(1)}$ for each $j = 1, \ldots, t$ once and store them for example in the public key. This increases the public key by $2t$ group elements but also helps to make the encryption linearly dependent on t only in the multiplications.

Ciphertext puncturing consists only of updating the BF and deleting elements from the secret key.

Decryption performs at most k membership checks in the Bloom Filter and one HIB-KEM key decapsulation plus the one-time signature verification. For BBG this consists of 2 pairings, 1 computation of an inverse and 3 multiplications so the runtime is basically independent of t.

The Puncturing between time intervals is probably the most interesting algorithm to analyze regarding its runtime for variable values of t.[10] In the worst case we need to do 2 key delegations for each of the t levels in the time tree and we always perform another m_t key delegations for the Bloom filter keys on the last level. We want to take a closer look at the cost of a key delegation on a specific level. We need to remark that we always do at least two key delegations from any level $k-1$ key sk_{k-1} – either to both siblings $\mathsf{sk}_{k-1}|0$ and $\mathsf{sk}_{k-1}|1$ or to all of the Bloom filter keys in the case $k = t+1$. We realize in BBG that the computation of the product $h_1^{H(I_1)} \cdots h_{k-1}^{H(I_{k-1})}$ only needs to be performed once for all children of the node with identity $(I_1, \ldots, I_{k-1})$. As already mentioned, we can assume that the values $h_j^{H(0)}$ and $h_j^{H(1)}$ for each $j = 1, \ldots, t$ have been precomputed and we do not need to perform those exponentiations. In total for any $k = 1, \ldots, t+1$ a key delegation from any level $k-1$ key to a level k key takes $5+t-k$ exponentiations.

It is important to note that for different time intervals τ the $\mathsf{PuncInt}(\mathsf{sk}, \tau)$ needs to perform different key delegations on different levels in the time tree. The most effort has to be done when $\tau = t'/2 - 1$ which is exactly the moment when $\mathsf{PuncInt}(\mathsf{sk}, \tau)$ makes the transition from the left subtree to the right subtree. Almost as in the algorithm KGen here we have to do 2 key delegations from each level $k-1$ to k for $k = 2, \ldots, t$ (and additionally m_t key delegations from level t to $t+1$).

Finally, we want to compute the maximal number of exponentiations for $\mathsf{PuncInt}(\mathsf{sk}, \tau)$ (so for the case $\tau = t'/2 - 1$) which sums up to

$$\sum_{k=2}^{t} 5 + t - k + 4m_t = \frac{1}{2}t^2 + \frac{7}{2}t - 4 + 4m_t.$$

Recalling our results from the beginning of this section where we computed the number of group elements in sk, we can conclude that for reasonably small choices of t the term $4m_t = m/2^{t-2}$ is dominating. So again we basically get a reduction by the factor 2 in the number of exponentiations when the value of t is incremented. In Table 1 we summarize the above made computations and observations.

Table 1: Number of operations for algorithms in TB-BFKEM with BBG.

Algorithm	Exponentiations	Pairings	other
Enc	$3k$	0	$O(tk)$
PuncCtx	0	0	$O(k)$
Dec	1	2	$O(k)$
PuncInt	$\frac{m}{2^{t-2}} + \frac{1}{2}t^2 + O(t)$	0	$O(\frac{m}{2^{t-2}} + t^2)$

4.3 Different Arities of the Time Tree

So far, we have only considered time trees of arity 2 since this is the most natural choice also by providing the number of each time slot τ in its binary representation. Still it is interesting if we can gain any improvements by changing the arity of the time tree to 3 or even higher.

Binary vs ternary. If we want to compare a binary tree of depth t to a ternary tree of depth s, we need to make sure that $t \approx \log_2(3)s$. This leads to a similar size of Bloom filters $m/2^t \approx m/3^s$ meaning that the size of Bloom filter keys to be stored should be basically the same for both trees. The interesting part to look at is the overhead of keys that need to be stored in the time tree in the worst case. We recall that for a binary tree of depth t the total number of group elements was given by

$$f_2(t) = \sum_{k=1}^{t} 2 + (t+1) - k = \frac{1}{2}t^2 + \frac{5}{2}t.$$

For a ternary tree the worst case also occurs in the first time slot where 2 level k keys need to be stored on each level $k = 1, \ldots, s$ leading to a total amount of group elements of

$$f_3(s) = \sum_{k=1}^{s} 2 \cdot (2 + (s+1) - k) = s^2 + 5s.$$

Using the assumption $t = \log_2(3)s$ we would like to determine for which values of s the overhead of one method is bigger than the other.

$$f_2(t) \geq f_3(s)$$
$$\Leftrightarrow \qquad s \geq \frac{5 - \frac{5}{2}\log_2(3)}{\frac{1}{2}\log_2(6) - 1} \approx 3.55$$

We can see that for values of $s \geq 4$ respectively $3^s \geq 81$ we get a lower overhead of group elements that we need to store in the time tree.

[10] Anyways one still needs to keep in mind that these computations can be performed separately.

Ternary vs 4-ary. Now we want to investigate whether we can reduce the overall key size even more if we increase the arity. If we compare a 4-ary tree of depth r with a ternary tree of depth s we should ensure that $s \approx \log_3(4)r$. Since in the 4-ary case on each level k for $k = 1, \ldots, r$ we need to store 3 HIBE keys, the overhead is given by

$$f_4(r) = \sum_{k=1}^{r} 3 \cdot (2 + (r+1) - k) = \frac{3}{2}r^2 + \frac{15}{2}.$$

Performing the same computations as above we can conclude that $f_3(s) \geq f_4(r)$ only for values of r greater or equal to 14. Since this corresponds to a total number of time slots of at least $4^{14} = 2^{28}$, we can conclude that in current realistic scenarios a 4-ary tree probably offers no advantages compared to a ternary tree.

Concrete example. We can conclude that for a choice of at least 81 time slots in TB-BFKEM, the usage of ternary trees is basically the best choice regarding maximal key sizes. In the following we want to look at an example with concrete instantiations of parameters and find out how big the difference between the number of group elements is for binary and ternary trees. Let us assume we want to do 2^{23} puncturings in a given time period with a maximal fpp $p = 2^{-11}$. This leads to an optimal choice for m of about 2^{27} and $k = 11$. Choosing a rather high number of time intervals of 2^{19} for the binary tree respectively 3^{12} for the ternary tree yields a Bloom filter size of 256 respectively 253. For the overhead, we can compare $f_2(19) = 228$ and $f_3(12) = 204$ showing that using the ternary tree gives a reduction by 24 group elements to be stored in the case that the key size is maximal.

Average key size. It is easy to see that the average number of overhead group elements that need to be stored, is exactly one half for the binary as well as the ternary tree. For example in the case of the binary tree, this comes from the fact that for a random time slot τ and any $k \in \{1, \ldots, t\}$ the probability is 0.5 that one level-k HIBE secret key needs to be stored. Thus, we can conclude that the ternary tree also is to be preferred concerning the average case.

4.4 Using all Nodes Instead of Only Leafs

So far, we have only dealt with the approach of interpreting each of the 2^t leaf nodes[11] with one time slot. Thus, one could wonder what to save if also the inner nodes, e.g., using an ordering as in [26] which can easily be adapted to the ternary case, as time slots and delegating Bloom filter keys from those keys of identities are shorter than t. For the binary case, we can double the number of time intervals from 2^{t-1} using only leaves to almost 2^t using all nodes. For the ternary case, we obtain an increase from $3^{s-1} = \frac{3^s}{3}$ to almost $\frac{3^s}{2}$ time slots when using all nodes. This corresponds to an increase of a factor of $\frac{3}{2}$.

5 IMPLEMENTATION AND OPTIMIZATIONS

Based on the findings above, we have implemented TB-BFKEM from Derler et al. [21, 22] using the BBG HIBE [10] based on the bilinear pairing library relic[12] in C. Our implementation is available on Github. [13].

Parallelization. Note that during key generation and puncturing, all calls to HIBDel are independent. Hence, they can be performed in parallel without any dependencies on earlier evaluations. Similarly, during encryption the individual ciphertexts are independent of each other. For all three operations, the implementation hence extensively employs OpenMP [17] to saturate all available CPU cores.

sOTS Instantiation. As we require pairings, one possible choice for the strong one-time signature scheme is the one from Groth [30]. But as any other sEUF-CMA secure signature scheme also satisfies the preconditions, we opted for EdDSA [6, 7]. Public keys and signature sizes are thus significantly more compact and highly optimized implementations of EdDSA are available in popular and widely used libraries. We have based our integration of EdDSA on libsodium.[14]

Tree implementation. To validate the theoretical results regarding the arity choice of tree for the management of the time epochs, our tree implementation supports a fixed arity set during compile time. In our implementation, we use every node of the tree. Hence, instead of putting all time epochs into the leaf nodes, we also put them into the intermediate nodes. Note however, this requires that special care needs to be taken when deriving the keys for the Bloom filter to perform domain separation between the keys stored in the Bloom filter and those handling the time epochs.

Pre-computations. Key derivation from BBG as used during key generation and puncturing in the time-based BFKEM requires an extensive amount of exponentiations of a small set of generators. To speed up these operations, we use relic's pre-computation mechanisms for more efficient exponentiations. Note that this efficiency gain comes at the cost of increased memory usage.

6 EVALUATION

For the following evaluation, we fixed some of the parameters of the bloom filter. Specifically, the false positive probability was set to $p = 0.001$ and thereby having a fixed number of $k = 10$ hash functions.

Benchmark results. We performed extensive benchmarks for the practical validation of our results. The benchmarks were performed using relic version 0.6.0 and GCC version 12 on Ubuntu 22.04 running on an Intel Core i7-1265U with 32 GB of RAM. For each algorithm, we conducted the benchmarks by first running a warm up phase and then measuring 50 evaluations. The results are depicted in Figures 6 and 7 and represent the average of the measurements.

From these figures, we can observe that both key generation and secret key sizes greatly benefit from smaller bloom filters which are enabled by using ternary time trees and larger tree depths. Indeed, comparing for example the configuration supporting 8.1 million overall puncturings with ternary trees and a depth of 4, is as fast as a configuration supporting 1.6 million puncturings in a binary tree with a depth of 4. Also for smaller number of puncturings, we

[11] Of course, they are only leaves in the binary tree of size t (the *time-tree*), but each of them has m_t children – representing the Bloom filter.

[12] https://github.com/relic-toolkit/relic

[13] https://github.com/ait-crypto/bfe-bf

[14] https://libsodium.gitbook.io/doc/

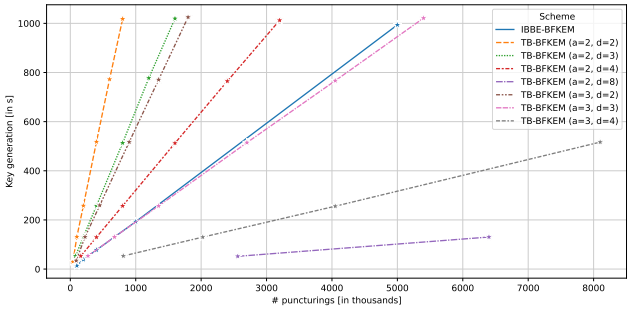


Figure 6: Runtime performance of KGen **for** TB-BFKEM **with tree arity a and depth d, and IBBE-BFKEM from [18] as the bloom filter size increases.**

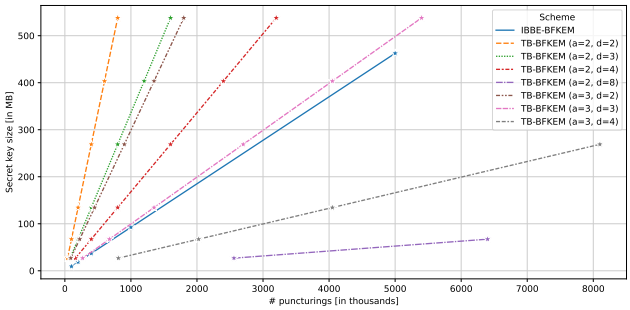


Figure 7: Secret key sizes of TB-BFKEM **with tree arity a and depth d, and IBBE-BFKEM from [18] as the bloom filter size increases.**

can observe similar numbers. Thereby, the measured key sizes of the concrete implementation validate our findings from Section 4.3

The runtime costs for ciphertext puncturing, PuncCtx, are negligible is since the implementation of this function corresponds to an index calculation and clearing of a small area of memory. For PuncInt, note that similar to the performance of KGen, runtime is dominated by deriving keys stored in the bloom filter. The runtime observed for PuncCtx thus coincides with the one from KGen up to a small additive factor.

The runtime performance of Enc of TB-BFKEM depends linearly on k. The size of the bloom filter and the arity of the tree has no impact on the performance. The average was measured at 10 ms for all instances. For Dec, the runtime was observed with numbers between 7 and 8 ms.

Non-classical parameter choices. Finally, we also evaluated the non-classical choices of bloom filter parameters as discussed in Section 3.2. Our implementation confirms the trade-off between secret key and ciphertext sizes. At the cost of increasing the size of the secret key by about 20%, the size of ciphertext can be halved. If Enc is evaluated in a linear fashion, the runtime of Enc is also reduced by the same factor. However, as our implementation fully parallelizes Enc and the benchmarking system supports the concurrent processing of more threads than k, almost no differences in the Enc runtime would be observable. To see the effect in Table 2, we thus limited the parallelism.

Table 2: Enc **runtime and secret key** sk **and ciphertext** C **sizes (as factors of $k = 10$) for non-classical choices for the bloom filter. Bloom filter parameters: $n = 50000$, $p = 0.001$ and hash functions k.**

k	Enc	sk size	C size
5	0.84x	1.20x	0.51x
6	0.90x	1.10x	0.61x
7	0.92x	1.04x	0.71x
8	0.96x	1.02x	0.81x
9	0.98x	1.00x	0.90x
10	1.00x	1.00x	1.00x

Comparison with IBBE-BFKEM [18]. Figures 6 and 7 also include the numbers from the IBBE-BFKEM implementation[15] of [18] evaluated on the same benchmark system. Note that key generation and secret key sizes are on par with a ternary tree configuration with a depth of 3. With higher depths, our approach can support a significantly larger number of puncturings with a secret keys that are more compact and where key generation is more efficient. Enc of the IBBE-BFKEM implementation averages at around 6 ms and Dec averages at around 10 ms. Here, the picture is the opposite of the TB-BFKEM implementation. The sum of both runtimes – which is of importance for authenticated key exchange protocols – is for both implementations at around 16 to 18 ms.

Now let us consider a bloom filter with $n = 60^2 \cdot 24 \cdot 2 = 172800$ expected punctures as recommended in [18]. This parameter choice supports a connection per second for two full day which corresponds to the interval required to achieve forward security in the QUIC implementation that Dallmeier et al. evaluated. Consequently, it also requires the key pair to be regenerated and recertified at least once a day. If we consider more practical lifetimes of keys, e.g., three months as common when using Let's Encrypt [1], we require $60^2 \cdot 24 \cdot 90 = 7776000$ puncturings. With a ternary tree and depth of 4, TB-BFKEM with a bloom filter supporting $n \approx 100000$ puncturings would suffice to satisfy these constraints. With this tree configuration and daily interval puncturings, the secret keys are smaller than those of the IBBE-BFKEM configuration yet we are able to support a significantly larger timespan without the need to switch public keys every day.

Note also, that beside the ability to stretch the key lifetime by increasing the depth of the tree, our approach also allows us to look at other trade-offs. Indeed, by increasing the depth of the tree, we are also able to decrease the size of the bloom filter while supporting the same number of overall puncturings. Thereby, the secret key size decreases and the runtime performance of KGen and PuncInt improves. More concretely, consider $n = 50000$ which reduces secret key sizes and the runtime of KGen and PuncInt by half in comparison to the case above. Assuming hourly interval puncturings, such a choice supports 13 connections per second. Combined

[15]Note that the implementations use different hash functions in the bloom filter which affects the overall performance numbers. The implementation is available at https://gitlab.com/buw-itsc/fs0rtt.

with a ternary tree with a depth of 4, the configuration supports 13 connections per second for two days. Thus, with a smaller key and faster key generation than IBBE-BFKEM, TB-BFKEM supports more puncturings in the same timeframe.

With such a configuration, we can also consider choosing a non-optimal k. Setting $k = 5$ allows us to reduce the ciphertext size by halve and also improves the runtime on the client side. While the secret key size would increase, it is still smaller than a IBBE-BFKEM supporting the same number of puncturings which requires up to 370 MB. With TB-BFKEM and $k = 5$, the secret key is less than half the size with 161 MB.

7 CONCLUSION

In this work, we revisit the use of Bloom-Filter Key Encapsulation Mechanisms (BFKEMs) to implement 0-RTT key exchange with full forward security and replay protection. Our work is motivated by its first practical implementation by Dallmeier et al. [18] within the QUIC protocol and their choice to use a BFKEM based on the identity-based broadcast encryption (IBBE) scheme by Delerablée [20]. In particular, we investigate the use of time-based BFKEM (TB-BFKEM) as an alternative BFKEM approach which can be based on any hierarchical identity-based encryption (HIBE) scheme. In doing so, we explore conceptual optimizations on the underlying Bloom filter, the realization of the time-tree and investigate the trade-offs when instantiating the TB-BFKEM based on the Boneh-Boyen-Goh (BBG) HIBE [10].

Moreover, we explore potential optimizations and trade-offs within a concrete implementation of the scheme. Specifically, we show that the additional degree of freedom of the TB-BFKEM related to the time intervals allows us to support significantly more overall puncturings with the same secret key size or to reduce the secret key size by halve. At the same time, our analysis of non-optimal choices for the number of hash functions also enables us to reduce the costs for clients while still keeping the secret key sizes on the server smaller than in the case of IBBE-BFKEM.

ACKNOWLEDGMENTS

This work was supported by the European commission through ECSEL Joint Undertaking (JU) under grant agreement No 826610 (Comp4Drones), by the Austrian Science Fund (FWF) and netidee SCIENCE under grant agreement P31621-N38 (Profet), and by European Union's Horizon 2020 research and innovation programme under grant agreement No 101019808 (TeamAware).

REFERENCES

[1] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth D. Schoen, and Brad Warren. 2019. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, London, UK, 2473–2487. https://doi.org/10.1145/3319535.3363192

[2] Shweta Agrawal, Dan Boneh, and Xavier Boyen. 2010. Efficient Lattice (H)IBE in the Standard Model. In *EUROCRYPT 2010 (LNCS, Vol. 6110)*, Henri Gilbert (Ed.). Springer, Heidelberg, Germany, French Riviera, 553–572. https://doi.org/10.1007/978-3-642-13190-5_28

[3] Nimrod Aviram, Kai Gellert, and Tibor Jager. 2019. Session Resumption Protocols and Efficient Forward Security for TLS 1.3 0-RTT. In *EUROCRYPT 2019, Part II (LNCS, Vol. 11477)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, Germany, Darmstadt, Germany, 117–150. https://doi.org/10.1007/978-3-030-17656-3_5

[4] Nimrod Aviram, Kai Gellert, and Tibor Jager. 2021. Session Resumption Protocols and Efficient Forward Security for TLS 1.3 0-RTT. *Journal of Cryptology* 34, 3 (July 2021), 20. https://doi.org/10.1007/s00145-021-09385-0

[5] Mihir Bellare and Sara K. Miner. 1999. A Forward-Secure Digital Signature Scheme. In *CRYPTO'99 (LNCS, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 431–448. https://doi.org/10.1007/3-540-48405-1_28

[6] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2011. High-Speed High-Security Signatures. In *CHES 2011 (LNCS, Vol. 6917)*, Bart Preneel and Tsuyoshi Takagi (Eds.). Springer, Heidelberg, Germany, Nara, Japan, 124–142. https://doi.org/10.1007/978-3-642-23951-9_9

[7] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (Sept. 2012), 77–89. https://doi.org/10.1007/s13389-012-0027-1

[8] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[9] Dan Boneh and Xavier Boyen. 2004. Secure Identity Based Encryption Without Random Oracles. In *CRYPTO 2004 (LNCS, Vol. 3152)*, Matthew Franklin (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 443–459. https://doi.org/10.1007/978-3-540-28628-8_27

[10] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. 2005. Hierarchical Identity Based Encryption with Constant Size Ciphertext. In *EUROCRYPT 2005 (LNCS, Vol. 3494)*, Ronald Cramer (Ed.). Springer, Heidelberg, Germany, Aarhus, Denmark, 440–456. https://doi.org/10.1007/11426639_26

[11] Dan Boneh, Ran Canetti, Shai Halevi, and Jonathan Katz. 2007. Chosen-Ciphertext Security from Identity-Based Encryption. *SIAM J. Comput.* 36, 5 (2007), 1301–1328.

[12] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 1465–1482. https://doi.org/10.1145/3133956.3133980

[13] Sonja Bruckner, Sebastian Ramacher, and Christoph Striecks. 2023. Muckle+: End-to-End Hybrid Authenticated Key Exchanges. In *Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16-18, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14154)*, Thomas Johansson and Daniel Smith-Tone (Eds.). Springer, 601–633.

[14] Ran Canetti, Shai Halevi, and Jonathan Katz. 2003. A Forward-Secure Public-Key Encryption Scheme. In *EUROCRYPT 2003 (LNCS, Vol. 2656)*, Eli Biham (Ed.). Springer, Heidelberg, Germany, Warsaw, Poland, 255–271. https://doi.org/10.1007/3-540-39200-9_16

[15] Ran Canetti, Shai Halevi, and Jonathan Katz. 2004. Chosen-Ciphertext Security from Identity-Based Encryption. In *EUROCRYPT 2004 (LNCS, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, Heidelberg, Germany, Interlaken, Switzerland, 207–222. https://doi.org/10.1007/978-3-540-24676-3_13

[16] Valerio Cini, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. 2020. CCA-Secure (Puncturable) KEMs from Encryption with Non-Negligible Decryption Errors. In *ASIACRYPT 2020, Part I (LNCS, Vol. 12491)*, Shiho Moriai and Huaxiong Wang (Eds.). Springer, Heidelberg, Germany, Daejeon, South Korea, 159–190. https://doi.org/10.1007/978-3-030-64837-4_6

[17] David Clark. 1998. OpenMP: a parallel standard for the masses. *IEEE Concurr.* 6, 1 (1998), 10–12. https://doi.org/10.1109/4434.656771

[18] Fynn Dallmeier, Jan P. Drees, Kai Gellert, Tobias Handirk, Tibor Jager, Jonas Klauke, Simon Nachtigall, Timo Renzelmann, and Rudi Wolf. 2020. Forward-Secure 0-RTT Goes Live: Implementation and Performance Analysis in QUIC. In *CANS 20 (LNCS, Vol. 12579)*, Stephan Krenn, Haya Shulman, and Serge Vaudenay (Eds.). Springer, Heidelberg, Germany, Vienna, Austria, 211–231. https://doi.org/10.1007/978-3-030-65411-5_11

[19] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. 2020. SafetyPin: Encrypted Backups with Human-Memorable Secrets. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 1121–1138.

[20] Cécile Delerablée. 2007. Identity-Based Broadcast Encryption with Constant Size Ciphertexts and Private Keys. In *ASIACRYPT 2007 (LNCS, Vol. 4833)*, Kaoru Kurosawa (Ed.). Springer, Heidelberg, Germany, Kuching, Malaysia, 200–215. https://doi.org/10.1007/978-3-540-76900-2_12

[21] David Derler, Kai Gellert, Tibor Jager, Daniel Slamanig, and Christoph Striecks. 2021. Bloom Filter Encryption and Applications to Efficient Forward-Secret 0-RTT Key Exchange. *J. Cryptol.* 34, 2 (2021), 13. https://doi.org/10.1007/s00145-021-09374-3

[22] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. 2018. Bloom Filter Encryption and Applications to Efficient Forward-Secret 0-RTT Key Exchange. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Heidelberg, Germany, Tel Aviv, Israel, 425–455. https://doi.org/10.1007/978-3-319-78372-7_14

[23] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. 2018. Revisiting Proxy Re-encryption: Forward Secrecy, Improved Security, and Applications. In *PKC 2018, Part I (LNCS, Vol. 10769)*,

Michel Abdalla and Ricardo Dahab (Eds.). Springer, Heidelberg, Germany, Rio de Janeiro, Brazil, 219–250. https://doi.org/10.1007/978-3-319-76578-5_8
[24] David Derler, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. 2021. Fine-Grained Forward Secrecy: Allow-List/Deny-List Encryption and Applications. In *FC 2021, Part II (LNCS, Vol. 12675)*, Nikita Borisov and Claudia Díaz (Eds.). Springer, Heidelberg, Germany, Virtual Event, 499–519. https://doi.org/10.1007/978-3-662-64331-0_26
[25] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. 1992. Authentication and Authenticated Key Exchanges. *Des. Codes Cryptogr.* 2, 2 (1992), 107–125.
[26] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. 2020. Pixel: Multi-signatures for Consensus. In *USENIX Security 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2093–2110.
[27] Craig Gentry and Alice Silverberg. 2002. Hierarchical ID-Based Cryptography. In *ASIACRYPT 2002 (LNCS, Vol. 2501)*, Yuliang Zheng (Ed.). Springer, Heidelberg, Germany, Queenstown, New Zealand, 548–566. https://doi.org/10.1007/3-540-36178-2_34
[28] Ashish Goel and Pankaj Gupta. 2010. Small subset queries and bloom filters using ternary associative memories, with applications. In *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010*, Vishal Misra, Paul Barford, and Mark S. Squillante (Eds.). ACM, 143–154. https://doi.org/10.1145/1811039.1811056
[29] Matthew D. Green and Ian Miers. 2015. Forward Secure Asynchronous Messaging from Puncturable Encryption. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 305–320. https://doi.org/10.1109/SP.2015.26
[30] Jens Groth. 2006. Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures. In *ASIACRYPT 2006 (LNCS, Vol. 4284)*, Xuejia Lai and Kefei Chen (Eds.). Springer, Heidelberg, Germany, Shanghai, China, 444–459. https://doi.org/10.1007/11935230_29
[31] Jens Groth. 2021. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339. https://eprint.iacr.org/2021/339.
[32] Christoph G. Günther. 1990. An Identity-Based Key-Exchange Protocol. In *EUROCRYPT'89 (LNCS, Vol. 434)*, Jean-Jacques Quisquater and Joos Vandewalle (Eds.). Springer, Heidelberg, Germany, Houthalen, Belgium, 29–37. https://doi.org/10.1007/3-540-46885-4_5
[33] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 2017. 0-RTT Key Exchange with Full Forward Secrecy. In *EUROCRYPT 2017, Part III (LNCS, Vol. 10212)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.). Springer, Heidelberg, Germany, Paris, France, 519–548. https://doi.org/10.1007/978-3-319-56617-7_18
[34] Jeremy Horwitz and Ben Lynn. 2002. Toward Hierarchical Identity-Based Encryption. In *EUROCRYPT 2002 (LNCS, Vol. 2332)*, Lars R. Knudsen (Ed.). Springer, Heidelberg, Germany, Amsterdam, The Netherlands, 466–481. https://doi.org/10.1007/3-540-46035-7_31
[35] Jana Iyengar and Martin Thomson. 2021. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor.
[36] Sebastian Lauer, Kai Gellert, Robert Merget, Tobias Handirk, and Jörg Schwenk. 2020. T0RTT: Non-Interactive Immediate Forward-Secret Single-Pass Circuit Construction. *PoPETs* 2020, 2 (April 2020), 336–357. https://doi.org/10.2478/popets-2020-0030
[37] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. 2015. How Secure and Quick is QUIC? Provable Security and Performance Analyses. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 214–231. https://doi.org/10.1109/SP.2015.21
[38] Moni Naor and Eylon Yogev. 2015. Bloom Filters in Adversarial Environments. In *CRYPTO 2015, Part II (LNCS, Vol. 9216)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 565–584. https://doi.org/10.1007/978-3-662-48000-7_28
[39] Eric Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor.
[40] Jim Roskind. 2012. QUIC: Design Document and Specification Rationale. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/.
[41] Paul Rösler, Daniel Slamanig, and Christoph Striecks. 2023. Unique-Path Identity Based Encryption with Applications to Strongly Secure Messaging. In *EUROCRYPT 2023, Part V (LNCS, Vol. 14008)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Heidelberg, Germany, Lyon, France, 3–34. https://doi.org/10.1007/978-3-031-30589-4_1
[42] Claus-Peter Schnorr. 1991. Efficient Signature Generation by Smart Cards. *Journal of Cryptology* 4, 3 (Jan. 1991), 161–174. https://doi.org/10.1007/BF00196725
[43] Daniel Slamanig and Christoph Striecks. 2021. Puncture 'Em All: Updatable Encryption with No-Directional Key Updates and Expiring Ciphertexts. Cryptology ePrint Archive, Report 2021/268. https://eprint.iacr.org/2021/268.

A NEGATIVE RESULTS

A.1 Probabilistic Puncturing

We recall (cf. Section 3.1) that a BFKEM allows a non-negligible correctness error being essentially identical to the fpp of the Bloom filter.

In the following we want to consider a new idea of puncturing ciphertexts that would be suitable for another kind of applications where we can even tolerate the possibility of false-negatives. In this scenario with a certain probability a ciphertext can be decrypted although the secret key has already been punctured on this ciphertext and therefore the decryption should not be successful We need to realize that we weaken the forward security of the BFKEM by allowing for false-negatives. In both strategies discussed below we will delete entries of the secret key only with a certain probability in order to be able to perform more puncturings.

First strategy. We use essentially the same model as in the original BFKEM, but adapt the Puncturing algorithm using a new parameter q which denotes the probability of not Puncturing on a ciphertext:

$\underline{\mathsf{Punc}'(\mathsf{sk}, C, q)}$: With probability $1-q$ perform $\mathsf{Punc}(\mathsf{sk}, C)$.

We do not achieve the Impossibility of false-negatives according to Equation (2) anymore by choosing this algorithm. For $q > 0$ there is a false-negative probability $\nu > 0$ for the event of $\mathsf{Dec}(sk_{n+1}, C) \neq \perp$ for a randomly chosen $j \leq n$. We want to find out the value of ν for given values of q, k and n.

As in the definition of Impossibility of false-negatives Equation (2) we consider the tuple $S = (C_1, \ldots, C_n)$ of all ciphertexts generated by $\mathsf{Enc}(pk)$. By S^- we denote those elements of S where in $\mathsf{Punc}'(\mathsf{sk}, C_j, q)$ the algorithm $\mathsf{Punc}(\mathsf{sk}, C_j)$ has not been performed.

Using this notation we get

$$\begin{aligned}\nu &= \Pr(\mathsf{Dec}(sk_{n+1}, C_j) \neq \perp)\\ &= \Pr(\mathsf{BFCheck}(H, T, C_j[0]) = 0 \wedge C_j \in S^-)\\ &= \Pr(C_j \in S^-) \cdot \Pr(\mathsf{BFCheck}(H, T, C_j[0] = 0) | C_j \in S^-)\\ &\geq q \cdot (1 - 2^{-k}).\end{aligned}$$

Assuming a realistically high choice of k in order to achieve a maximal fpp of at most 10^{-2} we have created a pretty tight lower bound for ν and therefore can approximate $\nu \approx q$. Before evaluating the meaning of these results let us look at the second strategy of probabilistic puncturing.

Second strategy. In this approach we can describe the puncturing in an informal way as

$\underline{\mathsf{Punc}'(\mathsf{sk}, C, q)}$: Given a ciphertext $C := \left(g_1^r, (E(y_j) \oplus \mathsf{K})_{j\in[k]}\right)$ and secret key $\mathsf{sk} = (T, (\mathsf{sk}[i])_{i\in[m]})$, the algorithm first updates the Bloom filter according to $T' = \mathsf{BFUpdate}'(H, T, g_1^r, q)$. Then the keys $\mathsf{sk}[i]$ of indices i that have been set to 1, are now deleted from the secret key sk.

This is almost identical to the original puncturing apart from using a different algorithm for updating the BF defined by

$\underline{BFUpdate'(H, T, u, q)}$: Given $H = (H_j)_{j\in[k]}, T \in \{0,1\}^m$ and $u \in \mathcal{U}$, this algorithm defines the updated state T' by first assigning $T' := T$. Then, it sets $T'[H_j(u)] := 1$ with probability $1-q$ for each $j \in [k]$ and finally returns T'.

Again it is interesting to determine the false-negative probability ν.

$$\begin{aligned}\nu &= \Pr(\mathsf{Dec}(sk_{n+1}, C_j) \neq \perp)\\ &= \Pr(\mathsf{BFCheck}(H, T, C_j[0]) = 0)\\ &= 1 - \Pr(\mathsf{BFCheck}(H, T, C_j[0]) = 1)\\ &= 1 - \Pr(\forall i \in [k] : H_i(C_j[0]) = 1).\end{aligned}$$

Defining $U(C_j) := \{l \in [k] : H_l(C_j[0])$ updated during $\mathsf{BFUpdate}'\}$ we can rewrite this as

$$\begin{aligned}&= 1 - \left(\sum_{I \subseteq [k]} \Pr(U(C_j) = I \wedge \forall l \in I : H_l(C_j[0]) = 1)\right)\\ &= 1 - \left(\sum_{I \subseteq [k]} \Pr(U(C_j) = I) \cdot \Pr(\forall l \in I : H_l(C_j[0]) = 1 | U(C_j) = I)\right)\\ &= 1 - \left(\sum_{I \subseteq [k]} (1-q)^{|I|} \cdot q^{k-|I|} \cdot \left(1 - e^{-\frac{(1-q)kn}{m}}\right)^{k-|I|}\right)\\ &= 1 - \left(\sum_{i=0}^{k} \binom{k}{i} (1-q)^{k-i} q^i \cdot \left(1 - e^{-\frac{(1-q)kn}{m}}\right)^{i}\right).\end{aligned}$$

Computational results for reasonable values of k show that this value of ν is at least three times higher than the one in the first strategy. It is easy to see that in both strategies (given a fixed value of q) n puncturings correspond to $(1-q)n$ puncturings in the original strategy concerning inserts that are made into the Bloom filter. Therefore the size of the secret key which grows in the order of m can also be reduced by a factor of $(1-q)$. As both strategies lead to the same benefit the first strategy should be preferred because of its lower fnp.

Unfortunately even for the first strategy the value of q can only be chosen really low since it almost directly determines the false-negative probability. If we want to allow for a fnp of at most 10^{-2}, meaning that 1 out of 100 ciphertexts can be decrypted although it should not, the size of the secret key reduces by only 1%. This example shows that for any reasonably small choice of the fnp, the benefit of a reduced key-size is only at a small level in both of the presented strategies.

A.2 Key Management Using Binary Trees

In the following we want to look at another way of storing the group elements in the secret key. As mentioned in Section 3.1 we are focusing on the BFKEM based on hashed Boneh-Franklin IBE. For hashed Boneh-Franklin IBE the secret keys consist of exactly one group element for each identity. Thus we originally store m group elements one by one and the size of the secret key grows linearly in the size of m i.e. linearly in the size of the number of puncturings n. In order to significantly reduce the key-size we should be looking for a data structure that allows us to store the elements of sk in a sublinear way.

Intuition of the Construction. The idea is based on using a l-level hierarchical identity-based key encapsulation scheme (HIB-KEM) with identity space $\{0,1\}^l$ in order to build a l-level HIB BFKEM.

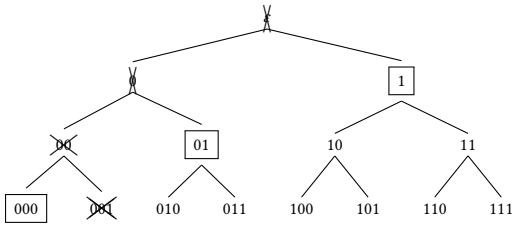


Figure 8: The first hash function maps to the identity 001**.**

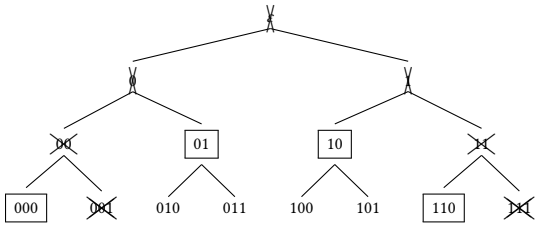


Figure 9: The second hash function maps to the identity 111**.**

KGen instantiates a Bloom filter of size $m = 2^{l}$[16] with k hash functions. It then runs $(\mathsf{mpk}, \mathsf{sk}_\epsilon) \xleftarrow{\$} \mathsf{HIBGen}(1^\lambda)$ to generate a key pair.

Enc does initially the same as the basic BFKEM Enc i. e. it randomly creates a ciphertext component and determines k identities from this. Then it performs k HIBE encryptions of the same key K using each of the identities.

Punc of the secret key on a ciphertext deterministically computes the k identities generated by the ciphertext component. Then for each of the k identities it computes the HIB-KEM secret keys associated to all *siblings* of nodes that lie on the path from the corresponding leaf node to the root until it reaches a node whose key is part of the sk and adds these to the sk. Then it deletes the key of the node, where it stopped such that it is not possible to delegate the key of the initial identity anymore.

Figures 8 and 9 illustrate this algorithm applied to the initial secret key sk_ϵ by a small example with 8 leaf nodes and $k = 2$ hash functions.

Dec checks similarly to the original BFKEM Dec for at most k identities if the corresponding secret key can be delegated and (in this case) delegates this specific key and performs one HIBE Decryption.

Remark. Notice that the level-l keys needed for Dec have to be delegated first along the way of decrypting several ciphertexts and that during the process of puncturing there are more new keys (of level $< l$) to be stored than keys to be deleted. This way intuitively the size of the secret key gets bigger as we puncture more ciphertexts in contrast to our original BFKEM.[17] Since we want to achieve a method of managing the secret key in sublinear

[16] For the purpose of simplicity we assume for the rest of this Section that the size of m is a power of 2 given as $m = 2^l$. Also we will start indexing at 0 in order to represent the m Bloom filter indices as l-bit binary strings. This way we will interpret each identity as a leaf node of a binary tree with depth l.

[17] In the standard BFKEM the puncturing only consists of the deletion operation. Therefore the size of the secret key decreases monotonically with each puncturing.

size, we want to take a closer look at the growth of the secret key in HIB-BFKEM depending on the number of puncturings. Before taking into consideration the fact that HIBE-keys are of different size on the different levels we are going to take a first look at only the number of total HIBE-keys that need to be stored. We will also differ between the number of puncturings and the number of *inserts* into the tree – where one puncturing consists of k inserts.

Best Case. It is easy to see that in the best case the leaf nodes of the binary tree are traversed from e.g. left to right. Starting with an insert at (the binary representation of) 0 and continuing with $1, 2$ etc. one can show inductively that a maximum number of $l-1$ HIBE-keys is never exceeded namely by having always at most one key on each level $1 \leq k \leq l$. Since we assume the puncturings/ inserts to be uniformly random, the Best Case in which we can choose the order of the inserted leaf nodes is not too relevant and thus we do not go further into details here.

Worst Case. Although the same holds true for the worst case, it is still worth determining the maximum size that the secret key can reach at all. An immediate upper bound for the maximum number of HIBE-keys is given by $2m - 1 = 2^{l+1} - 1$ which is exactly the number of nodes in a perfect binary tree of depth l. If we aim for computing a stricter upper bound, we should look for an order of inserts that maximizes the secret key. In order to do so we choose the order in a sort of greedy way – such that the number of HIBE-keys added at each step is maximal. Starting at any random leaf node (e.g. 0) we consider its corresponding subtree of depth $l-1$ and call it $T_1^{(1)}$. For the next insert we choose any leaf node (e.g. $m-1$) of the other remaining subtree $T_1^{(2)}$ of depth $l-1$. Now we consider the subtrees $T_2^{(1)}$ and $T_2^{(2)}$ of depth $l-2$ each containing one of the two chosen leaf nodes. There are 2 more subtrees of depth $l-2$. From both we can choose one more leaf node for our next two inserts. Continuing like this we get 4 more inserts from the subtrees of depth $l-3$ that have not been reached so far.

After 2^d inserts for $d \in \{0, \ldots, l-2\}$ we have added one leaf node in every subtree of depth $l-d$ and after that process the secret key consist of

$$\begin{aligned} &l - 1 + \sum_{i=0}^{d-1} 2^i (l - 2 - i) \\ =&2^d (l - d) - 1 \end{aligned}$$

HIBE keys. For $d = l - 2$ this gives us the number of HIBE keys after $m/4$ inserts, which is exactly $m/2 - 1$.

Average Case. We now know that on average the maximal size of the secret key lies in between $\log_2(m) - 1$ and $m/2 - 1$. As a first step we want to find out the expected value for the random variable N_i denoting the growth[18] of the secret key during the i-th insert. At first we define for $k = 1, \ldots, l$ the events

$$A_k^{(i)} := \forall j = 1, \ldots, i-1 : s_i[:k] \neq s_j[:k]$$

where $s_j[:k]$ denotes the first k bits of the j-th insert.

Using

$$\Pr(A_k^{(i)} \cap \neg A_{k-1}^{(i)}) \overset{A_{k-1}^{(i)} \subseteq A_k^{(i)}}{=} \Pr(A_k^{(i)}) - \Pr(A_{k-1}^{(i)})$$

we can compute

$$\begin{aligned} \mathbb{E}(N_i) &= \sum_{k=1}^{l} (l - 1 - k) \Pr(A_k^{(i)} \cap \neg A_{k-1}^{(i)}) \\ &= \sum_{k=1}^{l} (l - 1 - k) \left(\Pr(A_k^{(i)}) - \Pr(A_{k-1}^{(i)}) \right) \\ &= \sum_{k=1}^{l} (l - 1 - k) \left((\frac{2^k - 1}{2^k})^{i-1} - (\frac{2^{k-1} - 1}{2^{k-1}})^{i-1} \right). \end{aligned}$$

With these values for each insert i we can simply compute the expected value of the number of HIBE-keys in the secret key after N inserts by

$$\mathbb{E}(S_N) = 1 + \sum_{i=1}^{N} \mathbb{E}(N_i).$$

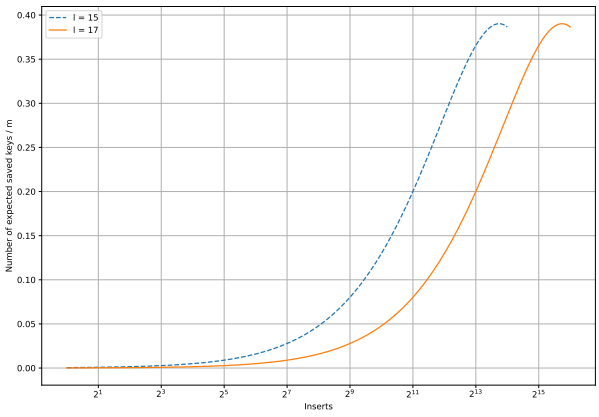


Figure 10: Expected number of HIBE-keys after variable number of inserts for two different depths l of binary trees

In Figure 10 the development of the expected size of the secret key is shown. It is interesting to see that in both cases for different depths of the binary tree the secret key contains a maximal number of HIBE-keys of approximately $0.39m$.

Concerning the reduction of the secret key size this result seems to be promising at first sight. But one would need for example a HIBE with keys of constant size – if there existed a HIBE where a key on each level consists of at most 2 group elements, we would get a reduction in the key-size of at least 22%. When taking into account the size of keys in currently used HIBEs (c.f. Section 2.2) it shows that this construction is unrealistic until now and we do not succeed yet by using this strategy.

[18] Note that this growth can also be negative.