# Forward Secure Asynchronous Messaging from Puncturable Encryption

Matthew D. Green and Ian Miers
Department of Computer Science
The Johns Hopkins University
Baltimore, USA
[mgreen,imiers]@cs.jhu.edu

***Abstract*—In this paper we investigate new mechanisms for achieving forward secure encryption in *store and forward* messaging systems such as email and SMS. In a forward secure encryption scheme, a user periodically updates her secret key so that past messages remain confidential in the event that her key is compromised. A primary contribution of our work is to introduce a new form of encryption that we name *puncturable encryption*. Using a puncturable encryption scheme, recipients may repeatedly update their decryption keys to revoke decryption capability for selected messages, recipients or time periods. Most importantly, this update process does not require the recipients to communicate with or distribute new key material to senders. We show how to combine puncturable encryption with the forward-secure public key encryption proposal of Canetti *et al.* to achieve practical forward-secure messaging with low overhead. We implement our schemes and provide experimental evidence that the new constructions are practical.**

## I. Introduction

Asynchronous messaging services such as email and SMS have become indispensable for personal and business communications. In recent years, several messaging services have begun to support end-to-end encryption in order to protect content transmitted over insecure communications channels. In this paper we address a specific drawback of many such encryption schemes: they often fail to ensure the confidentiality of *past* messages in the event that a user's key is compromised. The necessary property, known as *forward secrecy*, is vital to protecting user confidentiality. This is particularly important for messaging systems that use decentralized or cloud-based delivery systems that may preserve older messages for indefinite periods of time.

While it is relatively simple to add forward secrecy to interactive protocols such as TLS [23] or OTR [15], it is far more challenging to achieve in asynchronous "store-and-forward" messaging such as encrypted email, SMS, or in messaging systems that offer delivery to offline users (*e.g.,* Apple iMessage and Google Hangouts). Indeed, none of the three most popular encryption protocols for asynchronous messaging, Apple iMessage [21], OpenPGP [17] and S/MIME [36], provide a forward secrecy mechanism.

Addressing this problem is not trivial. Asynchronous messaging systems do not mandate that senders and recipients be online simultaneously, nor do they enforce two-way interaction between parties. Messages may be delayed for substantial periods due to delivery failures and connectivity issues, and some extensions, such as "greylisting" for spam prevention in email [28] and anonymous remailers/mixnets [22], intentionally introduce large delays in delivery.

Existing proposals for adding forward security to encrypted email [39], [8], [40] add increased complexity and new points of failure. They often require highly-available network infrastructure to distribute fresh key material to senders [39], or force changes to client interaction, such as requiring an initial message exchange prior to secure communications [33]. Beyond the added cost, such mechanisms are expensive to scale and may fail against active attackers. As a concrete example of these challenges, the TextSecure protocol used by WhatsApp[1] [4] implements an extremely fine-grained forward secrecy mechanism in which the client uploads hundreds of ephemeral keys to an online server that in turn distributes them to senders [32]. Not only are storage costs substantial, but an attacker can easily exhaust a given recipient's pre-key supply, which causes the mechanism to fail open.

In Eurocrypt 2003, Canetti, Halevi and Katz [18] proposed an alternative approach that does not require changes to the key distribution model. Their proposal defines a *forward secure public key encryption* scheme (FS-PKE) in which users may publish a short, unchanging public key via existing key distribution infrastructure. The novel element of FS-PKE is an efficient *update* procedure by which a user's secret key can, at time period $T$, be altered to revoke decryption capability for any ciphertext encrypted during time period $T' < T$. In principle, this can be used to achieve forward security in existing messaging systems, under the relatively mild requirement that parties share (loosely) synchronized clocks.

Unfortunately FS-PKE has not been widely adopted. In part this is because little work has been conducted to establish the concrete performance characteristics of such a system. More problematically, the Canetti *et al.* key update procedure is a relatively blunt instrument: in removing decryption capability for a given time period, a user necessarily loses access to *all* messages sent during prior time periods. Thus the scheme cannot provide fine-grained deletion of messages, *e.g.,* removing access to individual messages or messages from a single

[1] TextSecure is the encryption protocol used by the WhatsApp communication network, which has over 600 million users worldwide.

DOI 10.1109/SP.2015.26

sender. The practical consequence is that an implementation that aims to preserve user experience must by either risk updating the key before all messages have arrived – or it must leave some messages exposed until the receiver can be certain that it is safe to wind the key forward. When one factors in clock drift and delivery latency, the result may be a period ranging from hours to weeks during which data remains vulnerable.

**Our contributions.** In this work we systematically explore the problem of providing forward secrecy in asynchronous messaging systems. Our overall goal is to develop public key encryption that allows for fine-grained revocation of decryption capability *only for specific messages*, while minimizing cost and storage requirements. As with the Canetti *et al.* approach, our goal is to use short, unchanging public keys. Unlike previous solutions, we require that the secret key update procedure remove access at the level of individual ciphertexts or message senders, while retaining the ability to decrypt all other messages.

To achieve this goal, we employ two new ingredients. First, we introduce a a new form of public-key encryption that supports revocation of individual messages. We refer to this new encryption scheme as *puncturable encryption*. The primitive can be thought of as a form of tag-based encryption [31] which adds an efficient **Puncture** algorithm that, on input the current secret key SK and a tag $t \in \{0,1\}^*$, outputs a new secret key $\mathrm{SK}'$ that will decrypt all ciphertexts *not* encrypted under tag $t$. Secret keys in this scheme can be repeatedly and sequentially punctured at many different points, replicating the experience of normal message deletion.

Second, we show how to merge puncturable encryption into a variant of Canetti *et al.* FS-PKE, modified to allow fine-grained revocation of specific time intervals *without* revoking all previous intervals. By combining these approaches into a unified scheme, we show how to implement *practical* forward-secure public key encryption under reasonable workloads. More specifically, our contributions are as follows:

1) We define puncturable encryption, propose security definitions for the primitive, and offer an efficient construction secure under well-studied assumptions in bilinear groups. Our construction is based on a non-monotonic Attribute Based Encryption (ABE) scheme due to Ostrovsky, Sahai and Waters [35], modified to realize a new key update functionality. After $n$ puncture operations, our construction features $O(1)$ public key, ciphertext size and encryption cost, and $O(n)$ secret key storage and decryption cost.

2) To improve efficiency, we show how to compose puncturable encryption with an optimized FS-PKE construction due to Boneh *et al.* [12] that allows for revocation of individual time periods. This combination realizes the "best of both worlds", allowing a user to instantly delete selected messages with precision, while dramatically reducing decryption cost over puncturable encryption alone. The advantage of this approach is that total decryption and key storage cost now grow linearly only in the maximum number of messages received *within a given time period*, and only logarithmically in the number of time periods.

   Interestingly, our results show that composing these schemes is not simply a matter of running both in parallel, but requires that they be carefully combined such that the resulting keys provide *collusion resistance* against an adversary who seeks to recombine keys from different time periods.

3) Finally, we provide a software implementation of the combined scheme both as a standalone library `libforwardsec` and describe how to integrate it with tools such as Gnu Privacy Guard [5]. We then use the new tools to conduct experiments evaluating the overhead of deploying puncturable and forward security encryption under various simulated usage scenarios.

**Outline of this paper.** The rest of this paper is structured as follows. In the remainder of this section we discuss the intuition behind our constructions. In §II and §III we provide background and formal definitions for the puncturable encryption primitive and in §IV present our main construction. In §V we show how to combine puncturable encryption efficiently with FS-PKE. In §VI, §VII, §VIII and §IX we describe the implementation and evaluation of our proposals. In §X we discuss other applications of these proposals. Finally, in §XI we discuss related work.

### A. Encryption Model

We now briefly explain the framework we use to describe encryption in asynchronous messaging systems. An asynchronous messaging network consists of a set of senders and a set of recipients, all interacting via an insecure channel. For the purposes of this work we will assume that senders have some means to obtain a single authentic public encryption key for each recipient they wish to communicate with. Instantiations of this model include the existing OpenPGP infrastructure, as well as systems like Apple iMessage [17], [21].

Conversations can be broken down into two types of message: *initial* messages between a sender and recipient, and optional *interactive messages*. In an asynchronous system, each conversation consists of at least one initial message, optionally followed by an interactive exchange between the communicating parties. In this work we are primarily concerned with the forward secrecy of initial messages, since the forward secrecy of subsequent interactions may be achieved by using an interactive "ratcheting" protocol such as OTR [15] or TextSecure/Axolotl [32].

The approach we propose in this work is to attach to each initial message a unique identifier, or "tag", generated by the encrypting party. Upon receiving this message, the receiving party may – at its discretion – revoke decryption capability for the received message via a secret key update. Since our puncturable encryption constructions support tags in the space $\{0,1\}^*$, the sender can use any unique string for the message identifier. Example tags might include a GUID (which we use in our experiments), or alternatively a concatenation of sender ID and a monotonically-increasing message counter. On receipt of a message, the recipient can securely revoke decryption capability for *only* that message by "puncturing" the secret key on that tag. Our puncturable constructions also support employing multiple tags per message. In this case, the unique message ID may be supplemented with additional meta-data such as the sender ID. This approach allows receivers to revoke entire classes of message (*e.g.,* all messages from a given sender).

There are two limitations of our approach: $(1)$ the cost of decrypting a message with a given key increases linearly in the number of punctures in that key and $(2)$ an active attacker may block messages from reaching the recipient, thus preventing the recipient from revoking access to these messages. To address these concerns, in Section V we propose to combined puncturable encryption with forward secure encryption. In this approach, the sender simultaneously encrypts each message with *both* a time period and a unique message identifier. The receiver may instantly perform both fine-grained revocation of the messages it receives, but also possesses a coarse-grained update mechanism to revoke messages older than a certain time period in the past. We refer to the time period for which the recipient *can* decrypt as the "decryption window". This dual approach is roughly analogous to the fine- and coarse-grained revocation approach used in TextSecure [32], but does not require distribution of pre-keys to the sender. Crucially for efficiency, decryption time for a message arriving in interval T is linearly only in the punctures done for interval T, not the total number of punctures.

In Section IX we discuss choices for parameters such as message ID format, time period interval length, and the size of decryption windows.

### B. Intuition: Puncturable Encryption

To explain the intuition behind our constructions, let us first address some trivial solutions. We consider schemes in which an encryptor attaches a "tag" such as a message identifier (or time period identifier) to each message sent to a given receiver. The goal of the system is to allow the receiver to selectively revoke the ability to decrypt specific tags. In systems with only a polynomially number of time periods (or "tags") $\mathcal{T}$, it is simple to realize (inefficient) puncturable encryption by generating a unique PKE keypair corresponding to each "tag" a sender might encrypt under. Puncturing the key is simply a matter of deleting the corresponding secret. One can improve upon the $O(|\mathcal{T}|)$ public key size of this construction using identity-based encryption to produce $O(1)$ sized public parameters for use as the public key, deriving IBE decryption keys for all tags $t \in \mathcal{T}$, and destroying the master secret.

Unfortunately these simple approaches have secret key storage that is linear in the *total number of allowable tags*, not the current number of punctured tags. Not only is this inefficient, it limits the maximum number of possible tags (or time periods) to be at most polynomial in the security parameter. This clearly rules out exponentially-sized tag spaces, for example, strings such as sender addresses or unique message identifiers. A smaller tag space raises the possibility of tag collisions, where multiple messages from different senders are given the same tag, and thus the second message cannot be decrypted.

To address these issues, we take a different approach. Rather than deleting elements from an existing decryption key, we desire a structure that allows us to *add* new restrictions on what the key can decrypt. The logical building block for our construction is a form of attribute-based encryption scheme that supports *non-monotonic* access formulas. In such schemes, decryption keys may comprise boolean formula containing both positive and negated attributes, *e.g.,* "NOT $t$".

In and of itself non-monotic ABE is not sufficient to construct puncturable encryption, since we must also support the ability to *add* negations to an existing decryption key. A critical observation here is that by formulating a key containing *only* negations, some constructions can be modified to support the creation of new negations within an existing key.

Our concrete proposal begins with an NM-ABE construction due to Ostrovsky, Sahai and Waters [35], which we configure as a form of tag-based encryption supporting a fixed number $d$ of tags per ciphertext. To generate a key pair, a user first produces parameters for an instance of the ABE scheme, publishes the public parameters as her public key $\mathrm{PK}$, derives a decryption key from the master secret key $\mathrm{MSK}$, and destroys the master secret $\mathrm{MSK}$.

At all times subsequent to initial key generation, the recipient's secret key is an ABE decryption key embedding a policy consisting of only negated attributes. To puncture a key at an additional point $t$, the recipient updates her existing secret key to derive a new key that also embeds the negation of $t$. This is possible in the Ostrovsky *et al.* scheme due to the structure of these negated key components. Specifically, within each negated key component, Ostrovsky *et al.* embed one secret share $\lambda$ of the master secret $\alpha$. Due to the nature of this scheme, it is possible to re-share the value $\lambda$ from any given key component, by generating $\lambda'$ and mauling the original key component to embed the share $\lambda - \lambda'$. Simultaneously, one may create a *new* negated key component embedding share $\lambda'$, and bound to the newly punctured tag $t$. This provides our puncture algorithm, which can be operated an arbitrary number of times.

*Combining Puncturable Encryption with FS-PKE.* As mentioned above, to ensure forward secrecy under active attack and allow for more efficient decryption, we need to combine Puncturable Encryption with FS-PKE.

The naïve approach to combining the two schemes is simply to operate FS-PKE in parallel with puncturable encryption, encrypting every plaintext across both systems.[2] However this approach is problematic. There is no obvious mechanism for reducing the complexity of a punctured secret key after winding the FS-PKE key forward – i.e., for removing NOT gates. To solve this problem, during time period $T$ we might instead retain a copy of the initial *unpunctured* secret key for use in time period $T+1$. Unfortunately this poses a new challenge: if an attacker compromises the recipient's computer, she will be able to combine this unpunctured secret key with the FS-PKE secret key for time period $T$, and thus access messages that should be inaccessible.

Our solution to this problem is to cryptographically bind the secret keys for the FS-PKE scheme with those for the punctured encryption scheme. Thus an attacker who obtains the secrets for time period $T$ and $T+1$ cannot recombine any portions of the key to obtain access to messages deleted during the earlier time period. In §V we show how to achieve this combination using an efficient FS-PKE derived from a Hierarchical Identity-Based Encryption scheme of Boneh, Boyen and Goh [12]. Given a maximum number of punctured tags $n$, and a maximum number $\mathcal{L}$ of time periods, the combined scheme gives $O(1)$ sized ciphertexts and public key, and $O(log(\mathcal{L})+n)$ secret key storage and decryption cost. More importantly, we implement this scheme and show that its actual costs are quite practical.

## II. Background

*Notation.* Throughout this paper we will use the following notation. Let $negl(\cdot)$ represent a negligible function. Let $\mathcal{M}$ represent the set of valid plaintexts for a scheme, and let $\mathcal{T}$ represent the set of valid tags.

### A. Bilinear Maps

Let $\mathbb{G}$ and $\mathbb{G}_T$ be two multiplicative cyclic groups of prime order $p$. Let $g$ be a generator of $\mathbb{G}$ and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ be a bilinear map with the properties:

1) Bilinearity: for all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$, we have $e(u^a, v^b) = e(u, v)^{ab}$.
2) Non-degeneracy: $e(g, g) \neq 1$.

We say that $\mathbb{G}$ is a bilinear group if the group operation in $\mathbb{G}$ and the bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ are both efficiently computable. In practice, we may also define bilinear groups in the *asymmetric* setting, where a bilinear map is defined as $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is no efficient isomorphism $\gamma : \mathbb{G}_1 \to \mathbb{G}_2$. We will describe our schemes in the symmetric setting, and in §VI will discuss the process of translating to the asymmetric setting.

The schemes we present in this work are provably secure under the Decisional Bilinear Diffie-Hellman Inversion (DBDHI) (see *e.g.,* [12]) and the Decisional Bilinear Diffie-Hellman assumption (DBDH) [11] in bilinear groups. For reasons of space we will omit a definition of these assumptions here, and refer the reader to the cited works.

## III. Definitions

In this section we provide the syntax and security definitions for puncturable encryption.

### A. Puncturable Encryption

A puncturable encryption scheme is a tuple of probabilistic algorithms $(\mathsf{PPKE.KeyGen}, \mathsf{PPKE.Encrypt}, \mathsf{PPKE.Decrypt}, \mathsf{PPKE.Puncture})$ with the following syntax:

$\mathsf{PPKE.KeyGen}(1^k, d) \to (\mathrm{PK}, \mathrm{SK}_0)$. On input a security parameter $k$, and a maximum number of tags per ciphertext $d$, output a public key PK and an initial secret key $\mathrm{SK}_0$.

$\mathsf{PPKE.Encrypt}(\mathrm{PK}, M, t_1, \ldots, t_d) \to \mathrm{CT}$. On input a public key PK, a plaintext $M$ and a list of tags $t_1, \ldots, t_d$, output the ciphertext CT.

$\mathsf{PPKE.Puncture}(\mathrm{PK}, \mathrm{SK}_{i-1}, t) \to SK_i$. On input a secret key $\mathrm{SK}_{i-1}$ and a tag $t$, output a new secret key $\mathrm{SK}_i$ that can decrypt any ciphertext $\mathrm{SK}'$ can decrypt, except for ciphertexts encrypted with $t$.

$\mathsf{PPKE.Decrypt}(\mathrm{PK}, \mathrm{SK}_i, \mathrm{CT}, t_1, \ldots, t_d) \to \{M\} \cup \{\perp\}$. On input a secret key $\mathrm{SK}_i$ and a ciphertext CT, output a plaintext $M$, or $\perp$ if decryption fails.

We now define correctness and security for puncturable encryption.

### B. Correctness

Correctness is defined by the following experiment. On input $(k, M, n, d, t_1, \ldots, t_n, t_1^*, \ldots, t_d^*)$:

1) Compute $(\mathrm{PK}, \mathrm{SK}_0) \leftarrow \mathsf{PPKE.KeyGen}(1^k, d)$
2) If $n > 0$ then for $i = 1, \ldots, n$ compute $\mathrm{SK}_i = \mathsf{PPKE.Puncture}(\mathrm{SK}_{i-1}, t_i)$.
3) Set $\mathrm{CT} = \mathsf{PPKE.Encrypt}(\mathrm{PK}, M, t_1^*, \ldots, t_d^*)$.

The scheme is correct if for all sufficiently large $k$; $d > 0, n \geq 0$ both polynomial in $k$; $t_1, \ldots, t_n \in \mathcal{T}$, $t_1^*, \ldots, t_d^* \in \mathcal{T} \setminus \{t_1, \ldots, t_n\}$, $M \in \mathcal{M}$ it holds that

$$\mathsf{PPKE.Decrypt}(\mathrm{SK}_n, \mathrm{CT}, t_1^*, \ldots, t_d^*) = M$$

with probability $1 - negl(k)$ taken over the random coins of the experiment.

**Remark.** We allow for a negligible correctness error due to the fact that in our constructions, it is desirable for size of the secret key to be independent of the length of the tag strings. In practice this implies some negligible probability that two different tags will collide.

### C. Security

Security for puncturable encryption is defined by the IND-PUN-ATK game, which we present in Figure 1. This game incorporates both CPA and CCA variants. Intuitively, this is similar to the indistinguishability definition for public key encryption but adds the following new oracles.

On input any tag $t \in \mathcal{T}$, the **Puncture** oracle updates the secret key to revoke tag $t$. The adversary may query this

[2] This encryption would be combined: example, a user might split a message $M$ using a 2-of-2 secret sharing and encrypt each share under one of the two schemes.

**Setup**. On input a security parameter $k$ and a maximum number of tags $d$, the challenger initializes two empty sets $P, C$ and a counter $n = 0$. It runs $(\mathrm{PK}, \mathrm{SK}_0) \leftarrow \mathsf{PPKE.KeyGen}(1^k, d)$ and gives PK to the adversary.

**Phase 1**. Proceeding adaptively, the adversary can repeatedly issue any of the following queries:

- **Puncture**$(t)$: The challenger increments $n$, computes $\mathrm{SK}_n \leftarrow \mathsf{PPKE.Puncture}(\mathrm{SK}_{n-1}, t)$ and adds $t$ to the set $P$.
- **Corrupt**(): The first time the adversary issues this query, the challenger returns the most recent secret key $\mathrm{SK}_n$ to the adversary and sets $C \leftarrow P$. All subsequent queries return $\perp$.
- **Decrypt**$(\mathrm{CT}, t_1, \ldots, t_d)$: If $\mathsf{ATK} = \mathsf{CPA}$ the challenger returns $\perp$. If $\mathsf{ATK} = \mathsf{CCA}$ the challenger computes $M \leftarrow \mathsf{PPKE.Decrypt}(\mathrm{SK}_n, \mathrm{CT}, t_1, \ldots, t_d)$ and returns $M$ to the adversary.

**Challenge**. The adversary submits two messages $m_0, m_1 \in \mathcal{M}$ along with tags $t_1^*, \ldots, t_d^* \in \mathcal{T}$. If the adversary has previously issued a **Corrupt** query and $\{t_1^*, \ldots, t_d^*\} \cap C = \emptyset$, the challenger rejects the challenge. Otherwise the challenger samples a random bit $b$, and returns $\mathrm{CT}^* \leftarrow \mathsf{PPKE.Encrypt}(\mathrm{PK}, M_b, t_1^*, \ldots, t_d^*)$ to the adversary.

**Phase 2**. This phase is identical to Phase 1 with the following restrictions:

- **Corrupt**() returns $\perp$ if $\{t_1^*, \ldots, t_d^*\} \cap P = \emptyset$.
- **Decrypt**$(\mathrm{CT}, t_1, \ldots, t_d)$ returns $\perp$ if $(\mathrm{CT}, t_1, \ldots, t_d) = (\mathrm{CT}^*, t_1^*, \ldots, t_d^*)$.

**Guess**. The adversary outputs a guess $b'$. The adversary wins if $b = b'$.

Fig. 1. $\mathsf{IND\text{-}PUN\text{-}ATK}$ security game for puncturable encryption, with $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}\}$.

oracle repeatedly throughout the game, each time producing a new secret key. The **Corrupt** oracle provides the adversary with the most recent state of the secret key held by the challenger. The adversary may challenge on a pair of messages and chosen tags $t_1^*, \ldots, t_d^*$, subject to the restriction that the adversary cannot corrupt the secret key unless she has previously punctured at least one of the tags $t_1^*, \ldots, t_d^*$. This restriction prevents attacks in which the adversary may trivially decrypt the challenge ciphertext.

The CCA variant of the game also adds a decryption oracle. The adversary may call this oracle at any point on input $(\mathrm{CT}, t_1, \ldots, t_d)$ with the sole restriction that $(\mathrm{CT}, t_1, \ldots, t_d) \neq (\mathrm{CT}^*, t_1^*, \ldots, t_d^*)$, i.e., that she does not query on the challenge ciphertext and tags. More formally:

*Definition 3.1 (Security for puncturable encryption):* A puncturable encryption scheme is $\mathsf{IND\text{-}PUN\text{-}ATK}$ secure for $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}\}$ if for all *p.p.t.* adversaries $\mathcal{A}$ and for sufficiently large $k$, it holds that $\mathcal{A}$'s advantage in the $\mathsf{IND\text{-}PUN\text{-}ATK}$ game is bounded by $1/2 + negl(k)$.

## IV. Constructions

We now present constructions that achieve, respectively, CPA-secure and CCA-secure puncturable encryption under reasonable assumptions in bilinear groups.

### A. A CPA-secure construction

Figure 2 presents a CPA-secure construction of Puncturable Encryption based on an Attribute-Based Encryption scheme of Ostrovsky, Sahai and Waters (OSW) [35]. As discussed earlier, the basic construction is an adaptation of the OSW scheme, with the addition of a **Puncture** algorithm that, on input a secret key SK and tag $t$ outputs $\mathrm{SK}'$ with an additional component for the negation of tag $t$. Our key observation is that individual secret key components can be "re-shared" using only public parameters. This process is described in the **Puncture** algorithm.

For space reasons we omit a proof of correctness and move directly to our main security theorem for the security of the CPA construction.

*Theorem 4.1:* The puncturable encryption scheme of Figure 2 is $\mathsf{IND\text{-}PUN\text{-}CPA}$ secure in the random oracle model if the Decisional Bilinear Diffie-Hellman (DBDH) assumption holds in $\mathbb{G}, \mathbb{G}_T$.

The proof of Theorem 4.1 draws extensively from the proof of [35], adding mainly the additional details of simulating the **Puncture** algorithm. We sketch this proof in the full version of this paper.of the full v.

### B. CCA security

The puncturable encryption scheme presented in Figure 2 provides only CPA security. We now describe how to modify this scheme to achieve CCA security. Our approach uses the Fujasaki-Okamoto transform [24] which is commonly used to efficiently transform a CPA-Secure scheme into a CCA-secure one.

Let $\mathcal{M}' \in \{0,1\}^\ell$ be a plaintext space and let $H_1 : \mathbb{G}_T \times \mathcal{M}' \times \mathcal{T}^d \rightarrow \mathbb{Z}_p$ and $H_2 : \mathbb{G}_T \rightarrow \{0,1\}^\ell$ be two independent hash functions. Given the CPA-secure puncturable encryption scheme ($\mathsf{PPKE.KeyGen}$, $\mathsf{PPKE.Encrypt}$, $\mathsf{PPKE.Decrypt}$, $\mathsf{PPKE.Puncture}$) from Figure 2 where $\mathsf{PPKE.Encrypt}$ uses random element $s \in \mathbb{Z}_p$, we define a modified scheme ($\mathsf{PPKE.KeyGen}$, $\mathsf{PPKE.Encrypt}'$, $\mathsf{PPKE.Decrypt}'$, $\mathsf{PPKE.Puncture}$) where $\mathsf{PPKE.Encrypt}'$ and $\mathsf{PPKE.Decrypt}'$ are defined as follows:

> $\mathsf{PPKE.Encrypt}'(\mathrm{PK}, M, t_1, \ldots, t_d)$. First select a random element $\Sigma \in \mathbb{G}_T$, compute $s \leftarrow H_1(\Sigma, M, (t_1, \ldots, t_d))$ and compute $\mathrm{CT}' \leftarrow \mathsf{PPKE.Encrypt}(\mathrm{PK}, \Sigma, t_1, \ldots, t_d)$ using $s$ as the encryption randomness. Now output $\mathrm{CT} = (\mathrm{CT}', H_2(\Sigma) \oplus M)$.
>
> $\mathsf{PPKE.Decrypt}(\mathrm{SK}_i, \mathrm{CT}, t_1, \ldots, t_d)$. First parse CT as $(\mathrm{CT}', S)$ and compute $\Sigma' \leftarrow$

**PPKE.Keygen**$(1^k, d)$. On input a security parameter $k$ and number of tags associated with a ciphertext $d$, choose a group $\mathbb{G}$ of prime order $p$, a generator $g$ and a hash function $H : \{0,1\}^* \to \mathbb{Z}_p$. Chooses random exponents $\alpha, \beta \in \mathbb{Z}_p$ and set $g_1 = g^\alpha$, $g_2 = g^\beta$. Finally sample $r \in \mathbb{Z}_p$ and a degree-$d$ polynomial $q(\cdot)$ subject to the constraint that $q(0) = \beta$. Define $V(x) = g^{q(x)}$. Letting $t_0$ be a distinguished tag not used during normal operation, output:

$$\mathrm{PK} = g, g_1, g_2, g^{q(1)}, \ldots, g^{q(d)} \quad \mathrm{SK}_0 = [(sk_0^{(1)} = g_2^{\alpha + r}, sk_0^{(2)} = V(H(t_0))^r, sk_0^{(3)} = g^r, sk^{(4)} = t_0)]$$

The parameters $g_2, g^{q(1)}, \ldots, g^{q(d)}$ allow any party to compute $V(\cdot)$ by interpolating in the exponent.

**PPKE.Encrypt**$(\mathrm{PK}, M, t_1, \ldots, t_d)$ On input the public parameters PK, a message $\mathcal{M}$ to encrypt and a set of tags $t_1, \ldots, t_d \in \{0,1\}^* \backslash \{t_0\}$. Sample a random $s$ from $\mathbb{Z}_p$ and output

$$\mathrm{CT} = \left( ct^{(1)} = M \cdot e(g_1, g_2)^s, ct^{(2)} = g^s, ct^{(3,1)} = V(H(t_1))^s, \ldots, ct^{(3,d)} = V(H(t_d))^s \right)$$

along with the tags $(t_1, \ldots, t_d)$.

**PPKE.Puncture**$(\mathrm{PK}, \mathrm{SK}_{i-1}, t)$ On input an existing secret key $\mathrm{SK}_{i-1}$ and a tag $t \in \{0,1\}^* \setminus \{t_0\}$. First parse $\mathrm{SK}_{i-1}$ as $[sk_0, sk_1, \ldots, sk_{i-1}]$ and further parses $sk_0$ as $(sk_0^{(1)}, sk_0^{(2)}, sk_0^{(3)}, t_0)$. Next sample $\lambda'$ and $r_0, r_1$ at random from $\mathbb{Z}_p$ and compute:

$$sk_0' = \left( sk_0^{(1)} \cdot g_2^{r_0 - \lambda'}, sk_0^{(2)} \cdot V(H(t_0))^{r_0}, sk_0^{(3)} \cdot g^{r_0}, t_0 \right)$$
$$sk_i = \left( g_2^{\lambda' + r_1}, V(H(t))^{r_1}, g^{r_1}, t \right)$$

It outputs the new key $\mathrm{SK}_i = [sk_0', sk_1, \ldots, sk_{i-1}, sk_i]$.

**PPKE.Decrypt**$(\mathrm{SK}_i, \mathrm{CT}, t_1, \ldots, t_d)$ On input a private key $\mathrm{SK}_i$, a ciphertext CT and a set of tags $t_1, \ldots, t_d$ associated with the ciphertext. Parse the ciphertext CT as $(ct^{(1)}, ct^{(2)}, ct^{(3,1)}, \ldots, ct^{(3,d)})$ and parse $\mathrm{SK}_i$ as $[sk_0, sk_1, \ldots, sk_i]$.
For $j = 0, \ldots, i$ parse $sk_i$ as $(sk_i^{(1)}, sk_i^{(2)}, sk_i^{(3)}, sk_i^{(4)})$. Next compute a set of coefficients $\omega_1, \ldots, \omega_d, \omega_*$ such that $(\omega_* \cdot q(H(sk_i^{(4)}))) + \sum_{k=1}^{d} (\omega_k \cdot q(H(t_k))) = q(0) = \beta$. Finally compute

$$Z_j = \frac{e(sk_j^{(1)}, ct^{(2)})}{e\left(sk_j^{(3)}, \prod_{k=1}^{d} (ct^{(3,k)})^{\omega_j}\right) \cdot e(sk_j^{(2)}, ct^{(2)})^{\omega^*}}$$

and output $M = ct^{(1)} / \prod_{j=0}^{i} Z_j$.

Fig. 2. A CPA-secure puncturable PKE scheme based on the ABE construction of Ostrovsky, Sahai, and Waters [35].

$\mathsf{PPKE.Decrypt}(\mathrm{SK}_i, \mathrm{CT}', t_1, \ldots, t_d)$. Next compute $M' \leftarrow S \oplus H_2(\Sigma)$ and compute $s' \leftarrow H_1(\Sigma, M', (t_1, \ldots, t_d))$. Finally, using $s'$ as the randomness, compute $\mathrm{CT}'' \leftarrow \mathsf{PPKE.Encrypt}(\mathrm{PK}, \Sigma', t_1, \ldots, t_d)$ and if $\mathrm{CT}' \neq \mathrm{CT}''$ return $\perp$. Otherwise return $M'$.

*Theorem 4.2:* The puncturable encryption scheme ($\mathsf{PPKE.KeyGen}$, $\mathsf{PPKE.Encrypt}'$, $\mathsf{PPKE.Decrypt}'$, $\mathsf{PPKE.Puncture}$) is $\mathsf{IND\text{-}PUN\text{-}CCA}$ secure in the random oracle model if the Decisional Bilinear Diffie-Hellman (DBDH) assumption holds in $\mathbb{G}, \mathbb{G}_T$.

The proof of Theorem 4.2 follows the well understood structure described by Fujaki-Okamoto [24]. For space reasons we leave it to the full version of this work.

## V. PUNCTURABLE FORWARD SECURE PKE

While puncturable encryption provides fine-grained control over a recipient's ability to decrypt ciphertexts, the secret key size grows proportionally to the number of revoked ciphertexts tags. This can become unwieldy after a large number of punctures.

In this section we show how to mitigate this issue by combining puncturable encryption with a forward-secure public key encryption scheme [18] based on an efficient Hierarchical Identity Based Encryption (HIBE) Scheme. In the modified construction, senders encrypt under a time period $T$ and a list of tags $(t_1, \ldots, t_d)$. Receivers may puncture their keys within the current time period (or most recent $n$ time periods), eventually using the forward secure scheme to provide coarse-grained security by winding the key forward. Crucially, in winding the key forward they may eliminate the overhead of storing the punctured key components for past time periods. In this proposal, the total size of the key and cost of decryption are linear in the number of punctures in *only* the current time period(s), and logarithmic in the total number of time periods.

Intuitively, the security definition for this hybrid scheme is similar to the one for puncturable encryption, with the adversary gaining the additional capability to advance to the

**PFSE.Keygen**$(1^k, d, \ell)$. On input a security parameter $k$, the number of tags per ciphertext $d$ and a tree depth $\ell$, first select $\mathbb{G}, \mathbb{G}_T$ of order $p$. Sample random $\alpha, \beta \in \mathbb{Z}_p$ and $g_3, h_1, \ldots, h_\ell \in \mathbb{G}$, and set $g_1 = g^\alpha, g_2 = g^\beta$. Select a hash function $H : \{0,1\} \to \mathbb{Z}_p$ and let $t_0$ be a distinguished tag not used during normal operation. We will implicitly define $\mathrm{PK} = (\mathrm{PK}_{PPKE}, \mathrm{PK}_{PFSE})$ where:

$$\mathrm{PK}_{PPKE} = (g, g_1, g_2, g^{q(1)}, \ldots, g^{q(d)}), \quad \mathrm{PK}_{PFSE} = (g, g_1, g_2, g_3, h_1, \ldots, h_\ell)$$

Now sample $r_1, r_2, r_3 \in \mathbb{Z}_p$ and sample $\alpha_1, \alpha_2 \in \mathbb{Z}_p$ with the restriction that $\alpha_1 + \alpha_2 = \alpha$. Using $\alpha_1$ as the master secret key for the HIBE scheme, compute HIBE keys corresponding to identities “0” and ”1” (*i.e.* the identities to the left and right of the tree root).

$$hsk_{\mathsf{L}} = \textbf{BBG.Keygen}(\mathrm{PK}_{PFSE}, \alpha_1, 0), \;\; hsk_{\mathsf{R}} = \textbf{BBG.Keygen}(\mathrm{PK}_{PFSE}, \alpha_1, 1)$$

Next, compute the initial PPKE share of the key using master key $\alpha_2$ and distinguished tag $t_0$:

$$ppkesk_\emptyset = [(g_2^{\alpha_2 + r_3}, V(H(t_0))^{r_3}, g^{r_3}, t_0)]$$

Set $\mathsf{D}_0 = (0, \{hsk_{\mathsf{L}}, hsk_{\mathsf{R}}\}, ppkesk_\emptyset)$. This initial tuple will be used as a “seed” to obtain the secret key for the first time period, as follows:

$$(tsk_\emptyset^1, \mathsf{D}_1) \leftarrow \textbf{PFSE.NextInterval}(\mathsf{D}_0)$$

Output PK and the initial secret key $\mathrm{SK} = (tsk_\emptyset^1, \mathsf{D}_1)$.

**PFSE.Encrypt**$(\mathsf{PK}, M, T_{cur}, t_1, \ldots, t_d)$. On input $\mathsf{PK}$, a message $M$, a time period $T_{cur}$, a set of tags $t_1, \ldots, t_d \in \{0,1\}^* \setminus t_0$, sample $s \in \mathbb{Z}_p$ and compute a HIBE identity $T_1, \ldots, T_k = \mathsf{IndexToPath}(T_{cur}, \ell)$. Now compute:

$$ct^{(1)} = e(g_1, g_2)^s \cdot M, ct^{(2)} = g^s, ct^{(3,1)} = V(H(t_1))^s, \ldots, ct^{(3,d)} = V(H(t_d))^s, ct^{(4)} = \left(h_1^{T_1} \cdots h_k^{T_k} \cdot g_3\right)^s$$

Output $\mathsf{ct} = (ct^{(1)}, ct^{(2)}, ct^{(3,1)}, \ldots, ct^{(3,d)}, ct^{(4)})$ along with $T_{cur}, (t_1, \ldots, t_d)$.

**PFSE.Puncture**$(\mathrm{PK}, \mathrm{SK}, t)$. On input the current secret key SK, parse $\mathrm{SK} = (tsk_{\mathsf{T}}^i, \mathsf{D}_i)$ where $\mathsf{T}$ represents the set of punctures in the current time period. Further parse $tsk_{\mathsf{T}}^i$ as $(hsk_i, ppkesk_{\mathsf{T}})$ and compute:

$$ppkesk_{\mathsf{T} \cup \{t\}} \leftarrow \textbf{PPKE.Puncture}(\mathrm{PK}_{PPKE}, ppkesk_{\mathsf{T}}, t)$$

Output $tsk_{\mathsf{T} \cup \{t\}}^i = (hsk_i, ppkesk_{\mathsf{T} \cup \{t\}})$.

**PFSE.NextInterval**$(\mathsf{D}_i)$. Parse $\mathsf{D}_i$ as $(i, \mathsf{HSKs}, ppkesk_\emptyset)$ and extract the HIBE key $hsk_{\mathsf{P}}$ corresponding to time period $i$ from $\mathsf{HSKs}$ and derive its left and right keys as follows:

$$hsk_{\mathsf{L}} = \textbf{BBG.Keygen}(hsk_{\mathsf{P}}, 0), \quad hsk_{\mathsf{R}} = \textbf{BBG.Keygen}(hsk_{\mathsf{P}}, 1)$$

Compute a new $\mathsf{HSKs}'$ including the two new keys but without the parent (i.e. $\mathsf{HSKs}' = (\mathsf{HSKs} \setminus hsk_{\mathsf{P}}) \cup \{hsk_L, hsk_R\}$) and set $\mathsf{D}' = (i+1, \mathsf{HSKs}', ppkesk_\emptyset)$.

Second, derive $tsk_\emptyset^i = (hsk_i', ppkesk_\emptyset')$ the key for decrypting messages for time interval $i$ by binding together the HIBE key $hsk_{\mathsf{P}}$ for interval $i$ with randomized version of the P-PKE key $ppkesk_\emptyset$. Parse $hsk_{\mathsf{P}}$ as $(a_0, \ldots)$ and $ppkesk_\emptyset$ as $(sk_0^{(1)}, sk_0^{(2)}, sk_0^{(3)}, t_0)$, sample $\gamma \in \mathbb{G}$, $r \in \mathbb{Z}_p$ at random and compute

$$hsk_{\mathsf{P}}' = (a_0 \cdot g_2^\gamma, \ldots)$$
$$ppkesk_\emptyset' = (sk_0^{(1)} \cdot g_2^{-\gamma + r}, sk_0^{(2)} \cdot V(H(t_0))^r, sk_0^{(3)} \cdot g^r, t_0)$$

Output $(tsk_\emptyset^i, \mathsf{D}')$.

**PFSE.Decrypt**$(tsk_{\mathsf{T}}^i, \mathsf{ct}, t_1, \ldots, t_d)$. Parse $tsk_{\mathsf{T}}^i$ as $(hsk_i, ppkesk_{\mathsf{T}})$ and $\mathsf{ct}$ as $(ct^{(1)}, ct^{(2)}, ct^{(3,1)}, \cdots, ct^{(3,d)}, ct^{(4)})$

$$A \leftarrow \textbf{PPKE.Decrypt}(ppkesk_{\mathsf{T}}, (1, ct^{(2)}, ct^{(3,1)}, \ldots), t_1, \ldots, t_d) \quad , \quad B \leftarrow \textbf{HIBE.Decrypt}(hsk_i, (1, ct^{(2)}, ct^{(4)}))$$

Output $M = \frac{ct^{(1)}}{A \cdot B}$.

Fig. 3. Puncturable Forward Secure Encryption from puncturable encryption and Hierarchical Identity Based Encryption.

secret key to the next time period, and to challenge on an unpunctured key proved it precedes the corrupted interval. We present the full definition in Appendix B.

*FS-PKE and HIBE.* The FS-PKE construction of Canetti *et al.* uses a HIBE scheme as a building block. First, a maximum number of time periods $\mathcal{L}$ is chosen, then a tree depth $\ell = \lceil log(\mathcal{L}+2)\rceil$ is calculated, and each time period $1,\ldots,\mathcal{L}$ is mapped to one node of a binary tree hierarchy of identities using an in-order traversal.[3] Each node of the tree corresponds to a HIBE secret key. When a time interval is complete, HIBE keys for the left and right subtrees are derived using the key for current epoch, which is then deleted.

We can implement the HIBE scheme using any selective-ID secure HIBE. In principle, we could even dual-purpose the Ostrovsky *et al.* NM-ABE scheme from our puncturable encryption construction to build a HIBE, using the key delegation approach proposed by Goyal *et al.* [26]. However, such a construction (and indeed many practical HIBE schemes) would have ciphertext sizes and decryption times that are linear in the identity length, and hence logarithmic in the number of intervals. In contrast, Boneh, Boyen and Goh [12] proposed an efficient construction that features constant-size ciphertexts and decryption times and, interestingly, keys that *decrease* in size linearly as the identity string grows.

*Combining FS-PKE and Puncturable encryption.* Intuitively, our approach in combining the two schemes is to associate with each time interval $T$ a pair of secret keys $(A_T, B_T)$ where $A_T$ represents the key material for the FS-PKE at time $T$ and $B_T$ represents the puncturable encryption key material. The element $B_T$ initially begins with no tags punctured, and will be updated with each subsequent call to **Puncture**. Prior to any punctures occurring, however, we also derive and store a second pair of keys $(A_{T+1}, B_{T+1})$ for the next time period where $A_{T+1}$ represents the FS-PKE key for time period $T+1$, and $B_{T+1}$ is the empty puncturable encryption component. Critically, each pair $(A_i, B_i)$ must be bound together, such that (1) both are required in order to decrypt a ciphertext, and (2) an attacker who obtains $(A_T, B_T), (A_{T+1}, B_{T+1})$ cannot recombine the keys in new combinations.

The process of updating a key from time period $T$ to time period $T+1$ is therefore a matter of discarding the pair $(A_T, B_T)$ and using the pair $(A_{T+1}, B_{T+1})$ to derive another bound pair $(A_{T+2}, B_{T+2})$ for the time period $T+2$.[4] The user applies subsequent puncture operations to $B_{T+1}$. This process may be repeated until the final time period.

As is typical of many IBE/ABE schemes, the HIBE scheme of [12] makes use of a master secret $\alpha$ and a public parameter $g_1 = g^{\alpha}$. The puncturable encryption scheme of §IV also uses a similar construction. Thus our approach is to initially share a single master secret key $\alpha$ as $\alpha_1, \alpha_2$ where $\alpha = \alpha_1 + \alpha_2$. We use $\alpha_1$ as the master secret for the HIBE scheme, and $\alpha_2$ as the secret for the puncturable encryption. At each subsequent time period, we can take the fresh keys for time period $T+1$ and produce a new pair of keys for time period $T+2$ by dynamically updating the sharing of the secret $\alpha$. Crucially, for both [12] and our scheme, $\alpha$ is embedded as a multiplicative factor of the form $g^{\alpha}$. As a result this re-sharing can be computed on any set of derived keys, even after the master secret $\alpha$ has been destroyed.

Syntactically, a hybrid scheme combines the existing $\mathsf{PPKE.KeyGen}, \mathsf{PPKE.Encrypt}, \mathsf{PPKE.Puncture}, \mathsf{PPKE.Decrypt}$ algorithms with the hierarchical key derivation mechanism of a HIBE scheme. Thus PFSE is a tuple of 6 algorithms$(\mathsf{PFSE.KeyGen},, \mathsf{PFSE.Encrypt}, \mathsf{PFSE.Puncture}, \mathsf{PFSE.Decrypt}, \mathsf{PFSE.NextInterval})$ where

$$\overline{SK}_{n+1} \leftarrow \mathsf{PFSE.NextInterval}(\overline{SK}_n)$$

returns the derived key for the next interval by invoking the HIBE scheme's key derivation function. We present our construction in Figure 3.

*Theorem 5.1:* The scheme of Figure 3 is secure in the sense of Definition in figure C.1 in the random oracle model if the Decisional Bilinear Diffie-Hellman Inversion ($\ell$-DBDHI) assumption holds in $\mathbb{G}, \mathbb{G}_T$.

We sketch a proof of Theorem 5.1 in the full version of this paper.

### A. CCA Security

As in construction of §IV-B, we can apply the Fujisaki-Okamoto transform [24] to the scheme of Figure 2 to obtain a CCA-secure construction. The details of this approach are nearly identical to those given in §IV-B, with the following small modifications. First, the hash function $H_1$ now has the profile $H_1 : \mathbb{G}_T \times \mathcal{M}' \times T_{cur} \times \mathcal{T}^d \to \mathbb{Z}_p$. Second, each of the modified encryption and decryption algorithms each take as input the time period $T_{cur}$ as well as a list of tags, and feed these values into $H_1$. All of the remaining details are as in §IV-B. We leave a formal proof for the full version of this paper.

## VI. Implementation

In this section, we present our implementation of puncturable forward secure encryption, with all technical details needed for implementation.

### A. Symmetric to Asymmetric Conversion

While our scheme is presented in the "symmetric" pairing setting, where the bilinear map $e$ is defined as $e : \mathbb{G}\times\mathbb{G} \to \mathbb{G}_T$, in practice the most practical settings for pairing implementations use asymmetric bilinear groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ where $e$ is defined by $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. In many of these settings, there exists no efficiently computable isomorphism between the groups $\mathbb{G}_1, \mathbb{G}_2$.

The setting we employ for our implementation uses 256-bit Barreto-Naehrig curves [9]. Since the resulting bilinear groups are asymmetric, we must transform our presented

[3] This description places the first time period at the node to the left of the root of the tree.

[4] The user need not immediately discard the previous key and may, if she chooses, retain keys for many time periods. This allows users to "hold the door open" for any late-arriving ciphertexts.

| Key size($\ell$ = max ID length, l = ID length) | | | | Example $l = 30$(no punctures) | | |
|---|---|---|---|---|---|---|
| | puncturable encryption | + HIBE overhead | - overlap | normal | split | CRS |
| PK | $(2+d) \times \lvert\mathbb{G}_1\rvert + (2+d) \times \lvert\mathbb{G}_2\rvert$ | $+(3+\ell) \times \lvert G_1\rvert + (3+\ell) \times \lvert G_2\rvert$ | $-2 \times \lvert\mathbb{G}_1\rvert$ | 4.02 kB | 1.66 kB | 0.42 kB |
| SK | $(3 \times \lvert\mathbb{G}_2\rvert + \lvert\mathbb{Z}_p\rvert) \times punctures$ | $+(2+\ell-k) \times \lvert\mathbb{G}_1\rvert + (\ell-k) \times \lvert\mathbb{G}_2\rvert$ | N/A | 14.02 kB | 16.24 | 14.02 kB |
| CT | $\lvert\mathbb{G}_T\rvert + (d+2) \times \lvert\mathbb{G}_1\rvert)$ | $+\lvert\mathbb{G}_t\rvert + 2 \times \lvert\mathbb{G}_1\rvert$ | $-(\lvert\mathbb{G}_T\rvert + \lvert\mathbb{G}_1\rvert)$ | 0.5 kB | 0.5 kB | 0.5 kB |

TABLE I

KEY SIZES FOR P-PKE, HIBE [13], AND THE COMBINED PFSE SCHEME. BECAUSE OF REDUNDANCIES IN THE TWO SCHEMES (E.G. THE GROUP GENERATOR $g$), THERE IS SOME OVERLAP WE SAVE IN THE COMBINED SCHEME.

schemes into the asymmetric setting. There are many ways to perform this transformation, and efficiency must be taken into account when choosing one. This is because in the BN setting, elements in $\mathbb{G}_1$ are on the order of 256 bits in our case while elements in $\mathbb{G}_2$ are on the order of 1024 bits. As a result, selecting group assignments involves tradeoffs. In effect there are three different goals we can optimize for:

1) Ciphertext size. We attempt to put as many elements in the ciphertext in $\mathbb{G}1$ first.
2) Public key size. We attempt to put the public key elements into $\mathbb{G}1$ first.
3) Secret key size. We attempt to put the secret key elements into $\mathbb{G}1$ first.

For our implementation, we chose to optimize for minimal ciphertext size. This seems appropriate, as secret key storage does not appear to be a significant problem except on highly constrained devices. We performed our group assignment using the Autogroup tool [6], which employs an SMT solver to optimize for appropriate assignments.

### B. Polynomials, evaluation, and recovery coefficients

Our descriptions in Figures 2 and 3 omit several important steps needed to implement the scheme. Specifically, our implementation must (1) select the random polynomial $q(x)$ such that $q(0) = \beta$, (2) compute $V(x)$ without knowledge of the polynomial coefficients, and (3) compute the recovery coefficients $\omega_0 \cdots \omega_d, \omega^*$ for decryption. While these details are not technically novel, we present them in Appendix A for completeness.

### C. Mapping time intervals to a HIBE scheme

In [19], Canetti et. al. detail how to construct a forward secure encryption system from a HIBE scheme by mapping time intervals onto a binary tree using an pre-order traversal and maintaining a stack of keys. Although we use the same pre-order traversal mapping, we use a different algorithm which requires only a dictionary and the ability to convert an index into the tree to a location in the tree and vice-versa. This leads to a simple and natural structure of a secret key as a map from an interval to the key for that interval and we found it easier to work with. The algorithm is defined in Listing 1.

```
def IntervalKeys(pk,sk,i):
  path = indexToPath(i,L)#L=max tree depth
  if len(path) != L:# then not a leaf node
    lkey= hibe.keygen(pk,sk[i],path+[0])
    rkey= hibe.keygen(pk,sk[i],path+[1])
    return ((lkey,pathToIndex(path+[0),L),
            (rkey,pathToIndex(path+[1],L))
```

Listing 1. Key Derivation for Update

### D. Sterile keys

An elegant feature of the HIBE scheme of Boneh et. al is that HIBE keys can be *sterilized* by deleting a few extra elements. These keys can still be used to decrypt messages associated with some identity, but they cannot be used to derive new keys for child intervals.

The ability to sterilize keys gives an important freedom: in PFSE interval sizes can be as short as we want. If we expect latent messages to arrive, we can simply keep a sterilized version of that interval around. This gives us the ability to keep a decryption window around which still maintains forward secrecy. Absent this, intervals could not be kept along and the size of an interval would determine how much latency the scheme can tolerate. This is not desirable as larger intervals will incur more punctures and hence longer decryption times. We leverage this in our implementation.

### E. Software

Our implementation is approximately 4,000 lines of C++ including a C++ wrapper around the RELIC pairing library [7], and extensive unit tests. Serialization is provided by the C++ Cereal [42] serialization library wrapping RELIC's serialization routines for elliptic curve points. To improve performance, we parallelize decryption using OpenMP [34] to parallelize computation of $Z_j$ in the puncturable decryption routines.

## VII. EXTENSIONS AND OPTIMIZATIONS

### A. Outsourced decryption

One potential concern with our scheme is the cost of decrypting, especially on constrained devices such as embedded systems and mobile phones. These concerns may be mitigated if a cloud provider is available to provide computational assistance. For example, there are well known techniques [27], [20] for securely and privately outsourcing pairing computation and ABE decryption to a third party (e.g. a cloud-based server). Since pairings are the dominant cost in the decryption process, and since decryption can be parallelized, this could allow for a substantial reduction in on-device decryption cost. One application of such a scheme would be to implement forward secure encryption within projects such as Google's End-to-End [2], using Google's servers to perform pairing operations.

### *B. Outsourced key storage and updating*

Key sizes in puncturable encryption can grow fairly large (*e.g.,* 900 kB for 1 message a second with up to 1000 seconds of latency). However, not all of this material needs to be kept in secure storage. Instead, the keys themselves can be encrypted and stored in untrusted storage, provided that the decrypter can securely store a short symmetric key, and that this key can be erased and overwritten with a new key during updates. This ensures that old keys cannot be recovered.

### *C. Size of public keys*

When implementing puncturable forward secure encryptionusing symmetric pairings, the HIBE public key contains $\ell$ random group elements. These constitute the bulk of the public key material in puncturable forward secure encryption. To reduce the size of the public key, these elements could be selected using an appropriate hash function, and used as a global constant (formally, a CRS). Alternatively, the same set of $\ell$ elements can be generated globally and shared across all public keys.

Although transforming the scheme into the asymmetric setting makes the scheme for faster and ciphertexts smaller, it makes the situation worse in two ways. First, a corresponding set of $\ell$ elements must exist in both groups. This doubles the size of the public key. Second, because each corresponding element must have the same discrete log (respect to the generator of each group), they cannot be generatedusng a hash function. To generate each pair of elements, it is necessary to pick a random exponent and raise the generator in each group to that power. This requires trusted setup. In the asymmetric setting, this leaves us with a 4.2 kB public key as seen in table I. There are three options for reducing the size of these keys:

1) Move the elements in $\mathbb{G}_2$ into the secret key. We call this the split approach. It increases public key size.
2) Assume trusted set up and generate them as a common parameters.
3) have each user participate in generating the parameters.

The third option holds some potential. A naive approach is to have all users generate $g_1^r, g_2^r$ for each component, prove they are each raised to the same base. and simply take the product of each. It's possible to optimize this so that the common parameters are progressively updated as users register and we need not require all users participate in a single setup protocol at the start.

### *D. Decryption window*

Many forward secrecy applications require the ability to maintain decryption capability for some number of time intervals in the past. In our scheme it is possible to store multiple keys. For example, a recipient may retain keys that allow for decryption of messages received during the past $n$ time periods. Moreover, with the ability to *sterilize* keys, these keys can be altered into a form that does not allow for the creation of keys for subsequent time periods. Thus a recipient can selectively "knock out" the keys for specific time periods, and/or puncture specific message identifiers within those time periods, while retaining the keys for previous time periods. The size of the window is an application choice balancing 1) the need to decrypt latent messages 2) the cost of increased key storage and 3) the risk that an attack might intentionally delay a target message until after they compromise the system and extract keys needed to decrypt it .

## VIII. Recommended Usage

Given all of these ingredients, we now have a system that can be used to implement forward secrecy in asynchronous messaging networks. The overall approach we propose is as follows. Each recipient generates a public key for a the hybrid P-PKE scheme, and delivers these keys via the key server. Each party now maintains an open decryption window allowing that party to receive messages with timestamps $> (p - n)$ where $p$ is the current time period and $n$ is the number of time intervals in the decryption window.

In practice, the recipient may implement this by retaining the keys corresponding to the $n$ most recent time periods, using the optimization for retaining keys described in §VII-D. When a new message arrives marked with time period $T$, the recipient identifies the time period key corresponding to $T$. If that key is still available, it uses the key to decrypt the message. Next, the recipient may puncture the corresponding key with the message's unique identifier. At the same time, the recipient's software derives and sterilizes keys as appropriate (this can either be done on the receipt, one per time interval, or in some batched aggregated process, depending on processing and power requirements.) Periodically, the client eliminates old time period keys that are no longer within the decryption window.

## IX. Performance Evaluation

We provide two types of experiments: microbenchmarks demonstrating performance of our hybrid (PFSE) scheme and simulated results illustrating the cost of the schemes in example usage scenarios.

### *A. Microbenchmarks*

We conduct our experiments against three devices:

- A Desktop Intel Core i7-3770K CPU @ 3.5 GHz with 32 GB of RAM running Ubuntu 14.04
- A 2013 Macbook air with an Intel Core i7-4650U CPU @ 1.7 GHz with 8GB of RAM running OSX 10.9.5
- An Android phone with a Qualcomm Snapdragon 801 SoC (a Kriat 400 ARM CPU) @ 2.5 GHz with 3GB of RAM running Cyanogen-mod 11.

All experiments resulted from conducting 50 timing samples on a tree of depth $\ell = 31$ with ciphertexts supporting only $d = 1$ tags. We implemented the CCA-secure variant of the scheme described in §V-A. For these experiments less than 50 kB of memory was used with all keys stored in memory.

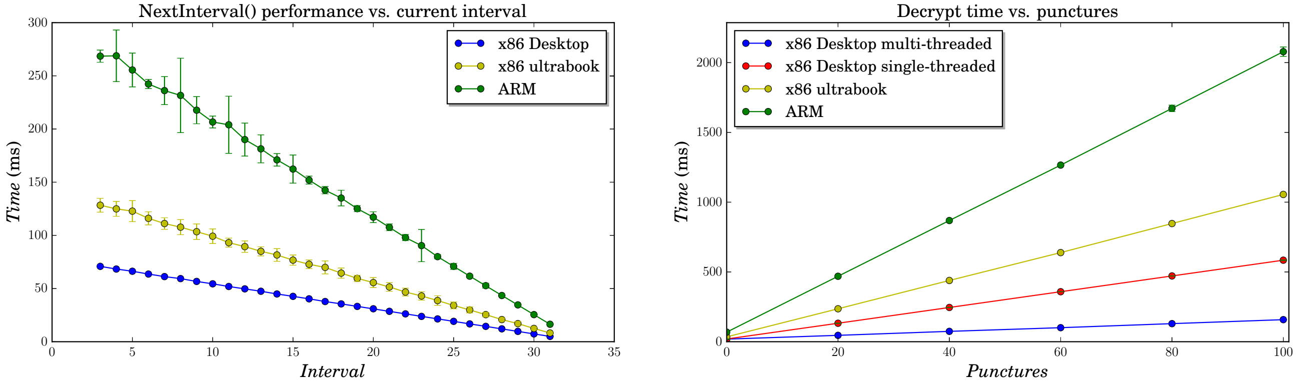

Fig. 4. Performance of PFSE (50 repetitions). Left, the cost of computing the next interval key as intervals advance. As later intervals have shorter HIBE secret keys, this decreases. Right, the cost of decrypting a ciphertext given the number of punctures in a key. Since P-PKE secret keys get larger as they are punctured, this increases.

While the goal of this section is to investigate the performance of our hybrid scheme, our micro benchmarks illustrate the performance of operations related to the two underlying cryptographic components – forward-secure PKE and puncturable encryption. Recall that the puncturable encryption key generation, encryption, and "puncture" operations all perform independently of the current number of punctures previously applied to a secret key. Even when combined with a HIBE scheme to form a PFSE scheme, the cost of these operations remains constant. They do, however, gain a dependency on the total number of allowable time periods. Table II presents microbenchmarks for these operations.

| | (x86)Desktop | (x86)Ultrabook | ARM |
|---|---|---|---|
| Keygen | $196.6 \pm 1.7$ ms | $368.4 \pm 13.9$ ms | $883.2 \pm 56.3$ ms |
| Encrypt | $5.49 \pm 0.1$ ms | $9.9 \pm 1.3$ ms | $22 \pm 0.9$ ms |
| Decrypt | $13.82 \pm 0.01$ ms | $24.8 \pm 2.4$ ms | $55.5 \pm 1.3$ ms |
| Puncture (Initial) | $15.6 \pm 0.1$ ms | $28.8 \pm 2.5$ ms | $68.4 \pm 1.6$ ms |
| Puncture (subsequent) | $9.8 \pm 0.1$ ms | $18.3 \pm 2.2$ ms | $42.2 \pm 0.7$ ms |

TABLE II
MICRO-BENCHMARKS FOR PFSE WITH $2^{32} - 2$ INTERVALS AND $d = 1$ TAGS PER MESSAGE. THIS INDEPENDENT OF THE STATE OF THE KEY.

While PFSE.NextInterval is similarly independent of the number of punctures $n$, it does depend linearly on the length of the current interval key. As keys in the HIBE scheme decrease in size as the identity gets longer (i.e. the key for the first interval is the longest), the process of deriving the next key gets faster. PFSE.Decrypt runs in time $O(n)$ independent of the ID length, the current interval, or even the total number of intervals. However, it does depend on the number of punctures. See Figure 4 for results showing the performance of interval updates and decryption. [5] We note that since decryption depends on *number of punctures* × *number of tags*, Figure 4 also shows the effect of decryption with $d \geq 1$ tags.

### B. PFSE under real world conditions

There are two performance metrics we are concerned with: the size of the secret keys and the amount of time we expect to spend perform cryptographic operations necessary to read messages. Performance of PFSE under real world conditions is determined by 4 parameters:

1) **The distribution of message arrivals.** For our simulations, we assume for simplicity that message arrivals are modeled as a Poisson process with distribution $\lambda$. This assumption clearly does not accurately model bursty communication like chat/SMS where there are many replies – however, for such mechanisms one can use a symmetric key ratchet to achieve forward security. As the message rate increases, we expect performance to decrease.
2) **The duration of each time interval.** The HIBE portion of our scheme maps messages into time intervals for which a specific key is needed to decrypt all messages in that interval. The length of those intervals affects both the size of keys (since for shorter intervals, we need a deeper tree to span the same amount of time), the number of expected punctures (for a fixed message rate, longer intervals mean more messages per interval and hence more punctures) and effort required to derive keys.
3) **Number of intervals** The maximum depth of the HIBE scheme limits the total number of intervals PFSE supports. It also effects the performance of HIBE key derivation.
4) **The "window" of time in which we can decrypt latent messages (all messages prior to this window are permanently inaccessible).** Window size affects the amount of key material that needs to be stored. We expect that the window size in deployed applications will be determined by considerations such as message delivery latency. Latency itself, assuming that it is independent per message does not effect performance.

[5] Due to an implementation quirk, Keygen handles deriving both the children of root and the grand-children. NextInterval therefore starts at interval three, hence the discontinuity in the above graph.

Because the message rate is determined by the type of traffic (e.g. email, sms, etc), and window size only effects key storage, the only two means an application has to tune its performance is through the selection of the interval size and the total number of supported intervals. Combined, these two parameters determine how long a key lasts before it needs to be replaced. In practice, we expect that the amount of time before a key is replaced will be determined by outside requirements. In that case, application developers face a trade off: a key with many short intervals or few long ones.

*1) Simulating real world usage:* To explore PFSE performance under a variety of conditions, we define a simulation parameterized by the three parameters given above. Our simulation uses the data in desktop performance numbers in Figure 4 as a raw input to estimate the computational cost that will be incurred using the cryptography. Results are shown in Figure 5 for the effect of interval size. We set experiments to run for a fixed amount of time (100,000 seconds) and choose parameters so that each public key covers 1 year worth of intervals (i.e. the key with 1 millisecond long intervals has 1000 times as many intervals as the key with 1 second long intervals.).

We subdivide our results into $(1)$ the time spent deriving HIBE keys, $(2)$ the time spent puncturing keys, and $(3)$ the time spent decrypting ciphertexts. A the decrease in efficiency for P-PKE is expected as intervals get larger (since we have more punctures per interval and thus more time spent decrypting). However, surprisingly, we notice a sharp increase in time spent deriving keys as intervals get smaller.

The cause of this extra computation is partly due to an increase in tree depth, as intervals get smaller. Primarily, however, the effect is simply one of spacing: as the interval size decreases, the distance (*i.e.,* number of intervals) between messages goes up. As a result, the number of keys we need to derive increases logarithmically, and because key derivation for each key depends linearly on its depth in the tree, the total amount of work needed increases polylogarithmically as intervals get shorter.

Given these results, how much storage do we expect? Table III shows the maximum private key sizes measured given various message rates. These are taken with a setup similar to the simulated experiments described above but $(1)$ run against the real software with a window size of 1,000 seconds with it aggressively deleting keys immediately once they were outside the window and $(2)$ with each interval getting $E[x]$ messages per interval (i.e. the expected number of messages per interval) rather than a random $X$ messages per interval where $X$ has a Poisson distribution. To ensure accuracy, the experiment simulates 2000 seconds worth of traffic.

*2) Recommended parameter choice and expected performance:* The experiments in Figure 5 suggest that the optimal interval size occurs at the point where the recipient receives 1 message per interval.

The exact point of the trade of may very depending on processor type and the relative efficiency of pairings used in decryption, and point multiplication used primarily in key derivation. Fine-grained tests on the current implementation confirm that 1 message per second is, at least on the test desktop system, a local optimum. Based on our tests, we believe we can deal with message rates of 1 a second and expect decryption times per message of 20 ms and with 99.99% probability, less than 100 ms.

| optimal interval length / message rate | size(kB) |
|---|---|
| 0.001 | 10.56 |
| 0.010 | 55.15 |
| 0.100 | 214.10 |
| 1.000 | 890.44 |

TABLE III
EXPECTED MAX KEYSIZE FOR OPTIMAL INTERVAL SIZE (I.E 1 MESSAGE PER INTERVAL EXPECTED). WINDOW OF 1,000 SECONDS. KEYS SIZED TO SPAN 1 YEAR.

For one message per interval, it takes on average 50 ms to derive the next key on a desktop. Thus our total expected time to decrypt a message, assuming we naively only derive keys when we get a message[6], is well less than 200 ms. Mobile benchmarks suggest a $4x$ increase in computational cost, increasing the decryption time to under 800 ms. Since the dominant cost of this is key derivation, which we expect to be batched and handled independent of message decryption, the actual expected time to decrypt a message on our ARM processor will be 55 ms, and with 99.99% probability, less than 500 ms.

## X. APPLICATIONS

In this section we discuss two applications and extensions of puncturable encryption within existing systems.

### A. Secure deletion

While the applications we considered in this paper are all limited to messaging protocols, our techniques may be applicable to other types of data storage as well. Where the standard approach to eliminating sensitive information is simply to delete local copies, secure deletion of files in cloud based storage can be potentially more challenging.

For files written and read by a single user, secure deletion can be accomplished trivially by encrypting each file with a unique key and deleting this key when necessary. However, systems with many writers and at least one reader (e.g. file sharing, shared passwords in online password managers[7], and SecureDrop[8] [41]), often require the use of public keys. Unfortunately, in this setting there is no clear way to delete a specific file without deleting the whole private key and losing access to all of the data.

With puncturable encryption, however, we can simple tag the data as appropriate and puncture on the tag to delete the file. There are several interesting options for use as tags:

[6] Unless the real message distribution has very low variance, this isn't recommended: A long gap for messages will effectively deffer key derivation costs until a message arrives. Instead, keys should be derived once per interval or in some batch process.

[7] Such a feature is provided by the cloud based password manager Lastpass.

[8] A system for securely communicating with journalists.

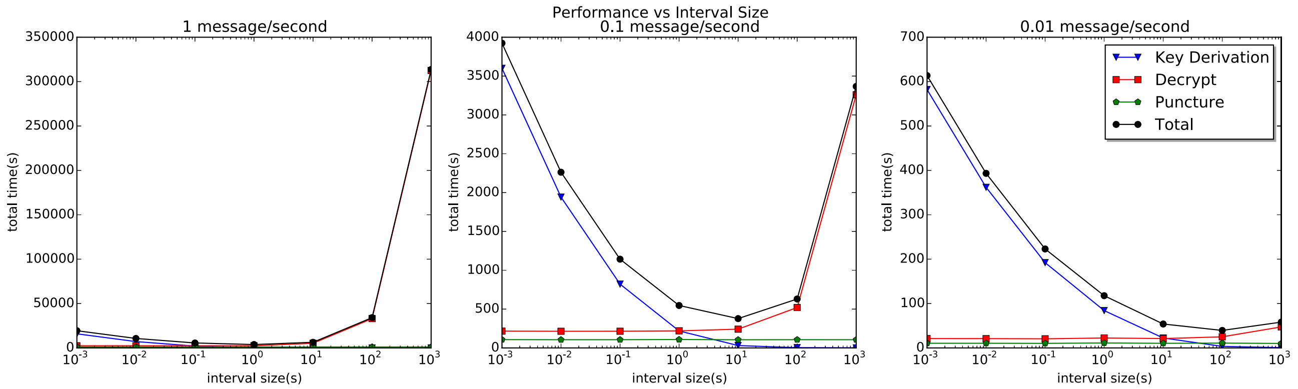

Fig. 5. Total time spent decrypting out of 100,000 seconds on a x86 Desktop. Note, x-axis is log scale and tree depth was adjusted so that the key spanned 1 year.

**Subject tags.** Messages can be tagged with a subject identifier. Once the subject has come to a close (*e.g.,* “selecting a Vice Presidential candidate”), all messages pertaining to the discussion can be deleted.

**Classification tags.** Messages can be tagged with their classification level (e.g. secret, top secret, etc). Negation can either be done globally (e.g. in the case of loosing security clearance), or per time interval to ensure sensitive information is not stored for long periods of time.

**Author.** Messages/files from a given user can be deleted. For example, a lawyer upon terminating their relation ship with a client, or on the death of the client could securely delete all work pertaining to the client to ensure confidentiality.

For data retention, none of these techniques need be applied to the only copy of the data. Instead, copies can be made to more secure archival system and negation used only to ensure that keys used on a daily basis no longer have access.

### *B. Integration with Legacy Applications*

Cryptographic tools for secure messaging, such as GnuPG [5] are integrated into many pieces of software. Even if developers wish to adopt forward security techniques this, many encryption user interfaces are already designed to interoperate with these legacy tools. Ideally, one could bypass client software entirely and merely modify deployed encryption tools such as GnuPG to use `libforwardsec` *without* changing the API of the existing library. We detail how to do this for simple forward security where message tags are random.

Key generation and decryption can be handling opaquely by the library by merely updating existing internal calls. For forward security, tags can be selected randomly by the library and so need not be specified in a call to encrypt. Public keys for the forward secure system may be included as a new form of subkey within existing OpenPGP keys, or they may be treated as an auxiliary component that the tool can retrieve from some external server. or, provided network access is acceptable, from a key server. In the case of the later, if the P-PKE key is signed by the legacy key, then the library can provided opportunistic forward secure encryption even when the sender is not aware the recipient supports it and only has the recipients legacy key.

The legacy model does not, however, provide an obvious solution to actually puncturing keys. The tool could automatically puncture for specific messages each time it is asked to decrypt a message. Alternatively we could modify GnuPG with an additional command of the form `--update-key` that on input either a time period $T$ or tag $t$ will update the current state of the secret key.

## XI. Related Work

There are several types of related work we list in this section.

*Delegation for Attribute Based Encryption.* Delegable or hierarchical Attributed Based Encryption [30], [43], [37], allows a user to modify a key embedding a given access policy into a key embedding a more restrictive one. Our puncture procedure can be considered a variant of delegation. Some work [37] has been done on ciphertext delegation, where an untrusted party can update a ciphertext to be accessible only under a more restricted policy. To the best of our knowledge, however, no scheme supports updating keys to add a negation (*e.g.* not tag 0xDEADBEEF) with only knowledge of that single key and not the master key.

*Revocable IBE.* Another body of work [10], [37], [44] is on revocable Identity Based Encryption. In this setting, a trusted third party issues identities and then through some mechanism (typically either directly updating users keys or posting some public update information) updates only the keys of non-revoked users. This setting does not deal with compromise of the trusted authority and thus can’t be used for forward secure encryption.

*Forward security for messaging.* In addition to the proposals listed earlier in this paper, there are several additional proposals for forward secure messaging that have seen deployment, such TextSecure and Pond [1], [3]. All of these schemes either rely on generating many keys, or, more frequently, using interactive forward secure protocols.

*Puncturable PRFs.* . Puncturable PRFs , dating back to 1984 [25] have received recent attention [14], [16], [29] for use with indistinguishability obfuscation[38]. They allow a PRF key to be punctured so that the PRF can no longer be evaluated on a specific point. Indeed puncturable encryption is intentionally named in a similar vein. However the two are distinct. While puncturable PRFs can be used to create puncturable symmetric key encryption in the obvious way,[9] it's not immediately clear how to use them for public key encryption. Moreover, all existing constructions are either selectively secure [10] or require indistinguishability obfuscation for instantiation.

## XII. Conclusion

In this work we proposed puncturable encryption, a new primitive that allows users to control which ciphertexts their keys may decrypt. By combining this primitive with an efficient FS-PKE scheme, we showed that the two schemes can be used in practice in real messaging systems. This work leaves several open question. One major question is whether puncturable encryption can be realized from alternative building blocks besides Attribute-Based Encryption. In particular, it is interesting to consider whether puncturable encryption can be constructed from elliptic curve groups that do not support pairings. A second set of questions deals with ways to improve the efficiency of this scheme and make it more practical for deployment, including specific techniques for outsourcing the decryption of forward secure ciphertexts.

*Acknowledgements.* This work was supported by: The Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211; the U.S. Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211; The National Science Foundation under award EFRI-1441209; and the Office of Naval Research under contract N00014-11-1-0470.

## References

[1] Forward secrecy for asynchronous messages. https://whispersystems.org/blog/asynchronous-security/. Accessed: 2014-11-13.

[2] Google End-To-End. Available at https://code.google.com/p/end-to-end/.

[3] Pond. https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2. Accessed: 2014-11-13.

[4] Textsecure. https://github.com/WhisperSystems/TextSecure/wiki/ProtocolV2. Accessed: 2014-11-13.

[5] The GNU Privacy Guard. https://www.gnupg.org/.

[6] Joseph A Akinyele, Matthew Green, and Susan Hohenberger. Using SMT solvers to automate design tasks for encryption and signature schemes. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 399–410. ACM, 2013.

[7] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. http://code.google.com/p/relic-toolkit/.

[8] Adam Back and Ben Laurie. Forward Secrecy Extensions for OpenPGP. Available at https://tools.ietf.org/html/draft-brown-pgp-pfs-01, August 2000.

[9] Paulo S.L.M. Barreto and Michael Naehrig. *Pairing-Friendly Elliptic Curves of Prime Order*, volume 3897, pages 319–331. Springer Berlin Heidelberg, 2006.

[10] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. Identity-based encryption with efficient revocation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 417–426. ACM, 2008.

[11] Dan Boneh and Xavier Boyen. Efficient selective-ID secure Identity-Based Encryption without random oracles. In *EUROCRYPT*, pages 223–238, 2004.

[12] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In *Advances in Cryptology–EUROCRYPT 2005*, pages 440–456. Springer, 2005.

[13] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In *EUROCRYPT*, pages 440–456, 2005.

[14] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology-ASIACRYPT 2013*, pages 280–300. Springer, 2013.

[15] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[16] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Public-Key Cryptography–PKC 2014*, pages 501–519. Springer, 2014.

[17] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. Updated by RFC 5581.

[18] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT '03*, pages 255–271, 2003.

[19] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In *CRYPTO*, pages 565–582, 2003.

[20] Benoît Chevallier-Mames, Jean-Sébastien Coron, Noel McCullagh, David Naccache, and Michael Scott. Secure delegation of elliptic-curve pairing. In *CARDIS*, pages 24–35, 2010.

[21] Apple Computer. iOS Security. Available at https://www.apple.com/privacy/docs/iOS_Security_Guide_Oct_2014.pdf, 2014 October.

[22] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, Washington, DC, USA, 2003. IEEE Computer Society.

[23] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[24] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO '99*, volume 1666, pages 537–554, 1999.

[25] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[26] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security*, pages 89–98, 2006.

[27] Matthew Green, Susan Hohenberger, and Brent Waters. Outsourcing the decryption of abe ciphertexts. In *USENIX Security Symposium*, page 3, 2011.

[28] Evan Harris. The Next Step in the Spam Control War: Greylisting. Available at http://projects.puremagic.com/greylisting/whitepaper.html, 2003.

[29] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 669–684. ACM, 2013.

[30] Allison Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *Advances in Cryptology–EUROCRYPT 2010*, pages 62–91. Springer, 2010.

[9] Though perhaps there is no obvious reason to do symmetric puncturable encryption at all given fast symmetric key ratcheting.

[10] Like the Boneh et. al HIBE scheme, this can be converted to full security via complexity leveraging at the cost of an exponential loss in security. However, for puncturable encryption from prfs, this is exponential in the tag space for messages (i.e. we loose at least 80 bits of security). For PFSE, on the other hand, it is at worst exponential in the tree depth for the HIBE component (i.e. we loose 32 bits if we support $2^{32}$ time intervals).

[31] Philip MacKenzie, Michael K. Reiter, and Ke Yang. Alternatives to non-malleability: Definitions, constructions, and applications. In Moni Naor, editor, *TCC '04*, volume 2951, pages 171–190. Springer, 2004.

[32] Moxie Marlinspike. Forward Secrecy for Asynchronous Messages. Available at https://whispersystems.org/blog/asynchronous-security/, August 2013.

[33] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560 (Proposed Standard), June 1999. Obsoleted by RFC 6960, updated by RFC 6277.

[34] OpenMP Architecture Review Board. OpenMP application program interface.

[35] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based encryption with non-monotonic access structures. In *ACM CCS '07*, pages 195–203, 2007.

[36] B. Ramsdell and S. Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard), January 2010.

[37] Amit Sahai, Hakan Seyalioglu, and Brent Waters. Dynamic credentials and ciphertext delegation for attribute-based encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 199–217. Springer, 2012.

[38] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: Deniable encryption, and more. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 475–484. ACM, 2014.

[39] B. Schneier and C. Hall. An improved e-mail security protocol. In *13th Annual Computer Security Applications Conference*, pages 232–238. ACM Press, 1997.

[40] Hung-Min Sun, Bin-Tsan Hsieh, and Hsin-Jia Hwang. Secure e-mail protocols providing perfect forward secrecy. *Communications Letters, IEEE*, 9(1):58–60, Jan 2005.

[41] Aaron Swartz and Kevin Poulsen.

[42] Randolph Voorhies and Shane Grant. cereal - A C++11 library for serialization. http://uscilab.github.io/cereal/index.html.

[43] Guojun Wang, Qin Liu, and Jie Wu. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 735–737. ACM, 2010.

[44] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Attribute based data sharing with attribute revocation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 261–270. ACM, 2010.

## Appendix A
## Polynomial selection and interpolation

Our descriptions in Figures 2 and 3 omit several steps needed to implement the scheme. Specifically, our implementation must $(1)$ select the random polynomial $q(x)$ such that $q(0) = \beta$, $(2)$ compute $V(x)$ without knowledge of the polynomial coefficients, and $(3)$ compute the recovery coefficients $\omega_0 \cdots \omega_d, \omega^*$ for decryption. We present these details below for completeness.

Recall that a polynomial of degree $d$ is uniquely defined by a set of points $(x_0, y_0), (x_1, y_1) \cdots (x_{d+1}, y_{d+1})$. The Lagrange form of the polynomial allows the computation of a point $x$ on the polynomial using only $d + 1$ points as follows:

$$q(x) = L(x, xc, yc) = \sum_{j=0}^{d} (yc[i] \cdot l(x, j, xc))$$

where $xc = [x_0, \cdots, x_{d+1}]$ and $yc = [y_0, \cdots, y_{d+1}]$ and the Lagrange basis polynomial $l(\cdots) is$

$$l(x, j, xc) = \prod_{\substack{0 \leq < m < d \\ m \neq j}} \frac{x - xc[m]}{xc[j] - xc[m]}$$

In our case the arithmetic above is in $\mathbb{Z}_p$.

Using the Lagrange form of a polynomial, sampling a random degree $d$ polynomial $q(x) = \beta$ consists of sampling $d$ random values $r_1, \ldots, r_d$ from $\mathbb{Z}_p$, setting points $(1, r_1), (2, r_2) \cdots, (d, r_d)$ and setting the final point as $(0, \beta)$ to ensure $q(0) = \beta$.

Lagrange interpolation does not directly yield a definition of $V(x)$ as we have only the public values $g^{q(0)}, \cdots, g^{q(d)}$ to work with. However, we can easily compute $V(x)$ as:

$$V(x) = g^{q(x)} = g^{\sum_{j=0}^{d} y_j l(x,j,xc)} = \prod_{i=0}^{d} (g^{q(i)})^{l(x,j,xc)}$$

where $l(x, j, xc)$ is defined as above. This makes use only of the public values $g^{q(0)}, \cdots, g^{q(d)}$.

Although we defined Langrange interpolation for a sequential set of x coordinates,$x_0 = 1, x_2 = 1, \cdots$, it works for arbitrary points. The condition for the recovery coefficient—find $\omega_0, \cdots \omega_d, \omega_*$ such that

$$(\omega_* \cdot q(H(sk_i^{(4)}))) + \sum_{k=1}^{d} (\omega_k \cdot q(H(t_k))) = q(0) = \beta$$

—is effectively asking for the coefficients necessary to compute $q(x)$ at $0$ given points on the polynomial $(t_0, q(t_0)), \cdots (t_d, q(t_d))$. Since we only need the recovery coefficients, we merely need th Legrange basies and thus do not need the y cordinates at all. As a result, we can compute $\omega_i = l(t_i, i, [t_0, \cdots, t_d, sk^{(4)}])$ and $\omega_* = l(sk^{(4)}, d+1, [t_0, \cdots, t_d, sk^{(4)}])$.

# Appendix B
## HIBE Construction of Boneh et. al

**BBG.Setup**$(\mathbb{G}, \ell, \alpha)$. To produce parameters for a HIBE scheme at most depth $l$ given secret $\alpha$, select a random generator $\hat{g} \in \mathbb{G}$ and set $\hat{g_1} = \hat{g}^{\hat{\alpha}}$. Next pick random elements $\hat{g}_2, \hat{g}_3, \hat{h}_1 \cdots \hat{h}_\ell \in \mathbb{G}$ and output :

$$MPK = (\hat{g}, \hat{g}_1, \hat{g}_2, \hat{g}_3, \hat{h}_1, \cdots, \hat{h_\ell}); \quad MSK = \hat{g_2}^{\hat{\alpha}}$$

**BBG.Keygen**$(\hat{sk}_{ID|k-1}, \mathsf{suffix})$. To generate a a private key $\hat{sk}_{ID|k}$ for an identity $ID||sufix = (I_1, \cdots, I_k)$ of length $k \leq \ell$, using the master secret key, sample a random $r \leftarrow \mathbb{Z}_p$ and output:

$$\hat{sk}_{ID|k} = \left(\hat{g_2}^{\hat{\alpha}} \cdot (\hat{h}_1^{I_1} \cdots \hat{h}_k^{I_k} \cdot \hat{g}_3)^r, \hat{g}^r, \hat{h}_k^r, \cdots, \hat{h}_l^r\right) = (a_0, a_1, b_k, \cdots, b_l)$$

To derive a key incrementally given the parent key $\hat{sk}_{ID|k-1}$, sample a random $t \in \mathbb{Z}_p$ and output:

$$\hat{sk}_{ID|k} = \left(a_0 \cdot b_k^{I_k} \cdot (\hat{h}_1^{I_1} \cdots \hat{h}_k^{I_k} \cdot \hat{g}_3)^t, a_1 \cdot, \hat{g}^t, b_{k+1}\hat{h}_{k+1}^t, \cdots, b_\ell \cdot \hat{h_l}^t\right)$$

**BBG.Encrypt**$(\hat{\mathrm{PK}}, ID, M)$. To encrypt a message $M \in \mathbb{G}_T$ under ID, pick a random $s \in Z_p$ and output:

$$\hat{\mathrm{CT}} = \left(e(\hat{g}_1, \hat{g}_2)^s \cdot M, \hat{g}^s, (\hat{h}_1^{I_1} \cdots \hat{h}_k^{I_k} \cdot \hat{g}^3)^s\right)$$

**BBG.Decrypt**$(\hat{sk}_{ID}, \hat{\mathrm{CT}})$ To decrypt a ciphertext $\hat{\mathrm{CT}} = (A, B, C)$ output:

$$M = e(a_1, C)/e(B, a_0)$$

Fig. B.1. The HIBE construction of Boneh, Boyen and Goh [12].

# Appendix C
## IND-PFSE-ATK game for PFSE

**Setup**. On input a security parameter $k$, a maximum number of tags $d$, and a number of intervals $\ell$ the challenger initializes two empty sets $P, C$ and counter $n = 0, ex = 0$. It runs $(\overline{\mathrm{PK}}, \overline{SK}) \leftarrow \mathsf{PFSE.KeyGen}(1^k, d, \ell)$ and gives PK to the adversary.

**Phase 1**. Proceeding adaptively, the adversary can repeatedly issue any of the following queries:

- **Puncture**$(t)$: The challenger computes $SK_{n+1} \leftarrow \mathsf{PFSE.Puncture}(SK_n, t)$ and adds $t$ to the set $P$.
- **Corrupt**(): The challenger returns the most recent secret key $SK_n$ to the adversary, sets $C \leftarrow P$ and sets $ex = b$.
- **Decrypt**$(\overline{\mathrm{CT}}, t_1, \ldots, t_d)$: If $\mathsf{ATK} = \mathsf{CPA}$ the challenger returns $\perp$. If $\mathsf{ATK} = \mathsf{CCA}$ the challenger computes $M \leftarrow \mathsf{PFSE.Decrypt}(\overline{SK}_n, \overline{\mathrm{CT}}, t_1, \ldots, t_d)$ and returns $M$ to the adversary.
- **NextInterval**(): The challenger sets $P = \emptyset$, increments $n$ and computes $\overline{SK}_{n+1} \leftarrow \mathsf{PFSE.NextInterval}(\overline{SK}_n, n+1)$

**Challenge**. The adversary submits two messages $m_0, m_1 \in \mathcal{M}$ along with tags $t_1^*, \ldots, t_d^* \in \mathcal{T}$ and an interval $0 \leq i \leq \ell$. If the adversary has previously issued a $\mathbf{Corrupt}$ query and $\{t_1^*, \ldots, t_d^*\} \cap C = \emptyset$ or $ex \leq i$, the challenger rejects the challenge. Otherwise the challenger samples a random bit $b$, and returns $\mathrm{CT}^* \leftarrow \mathsf{PFSE.Encrypt}(\overline{\mathrm{PK}}, M_b, i, t_1^*, \ldots, t_d^*)$ to the adversary.

**Phase 2**. This phase is identical to Phase 1 with the following restrictions:

- **Corrupt**() returns $\perp$ if $\{t_1^*, \ldots, t_d^*\} \cap P = \emptyset \vee n \leq i$.
- **Decrypt**$(\mathrm{CT}, t_1, \ldots, t_d)$ returns $\perp$ if $(\mathrm{CT}, t_1, \ldots, t_d) = (\mathrm{CT}^*, t_1^*, \ldots, t_d^*)$.

**Guess**. The adversary outputs a guess $b'$. The adversary wins if $b = b'$.

Fig. C.1. $\mathsf{IND\text{-}PFSE\text{-}ATK}$ security game for PFSE, with $\mathsf{ATK} \in \{\mathsf{CPA}, \mathsf{CCA}\}$.