Puncturable Key Wrapping and Its Applications

Matilda Backendal[0000−0002−8677−8301], Felix Günther[0000−0002−8495−6610], and Kenneth G. Paterson[0000−0002−5145−4489]

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{mbackendal,kenny.paterson}@inf.ethz.ch, mail@felixguenther.info

Abstract. We introduce *puncturable key wrapping* (PKW), a new cryptographic primitive that supports fine-grained forward security properties in symmetric key hierarchies. We develop syntax and security definitions, along with provably secure constructions for PKW from simpler components (AEAD schemes and puncturable PRFs). We show how PKW can be applied in two distinct scenarios. First, we show how to use PKW to achieve forward security for TLS 1.3 0-RTT session resumption, even when the server's long-term key for generating session tickets gets compromised. This extends and corrects a recent work of Aviram, Gellert, and Jager (Journal of Cryptology, 2021). Second, we show how to use PKW to build a protected file storage system with file shredding, wherein a client can outsource encrypted files to a potentially malicious or corrupted cloud server whilst achieving strong forward-security guarantees, relying only on local key updates.

1 Introduction

Key wrapping. Key encryption, or *key wrapping*, is a mechanism often deployed to build symmetric key hierarchies: systems in which the confidentiality and integrity of multiple cryptographic keys are protected by a single (master wrapping) key. The wrapped keys may in turn be used to secure data at a more fine-grained level, e.g., at the level of individual files, messages, or financial transactions. This hierarchical approach eases key management: it allows strong but more expensive protection to be applied to a small number of wrapping keys while limiting the security impact if individual wrapped keys are exposed. Key wrapping is widely used in practice; specific schemes have been standardized by NIST in [24]. Formal foundations for key wrapping were established in [47].

As a pertinent example, when using the pre-shared key (PSK) mode of TLS 1.3 [45] for session resumption, new sessions between client and server are protected by independent, symmetric keys (denoted PSK) established in an earlier session. To reduce storage overhead, servers often use a long-term symmetric encryption key to *wrap* PSKs into so-called *tickets*. These tickets are sent to the client, thereby outsourcing the PSK storage from the server to the client.

Another example of key hierarchies is found in cloud storage systems, where service providers encrypt data before storing it on their servers—so called *encryption at rest*. The encryption is done to meet customer demand and regulatory requirements. To ensure good key-hygiene, best practices stipulate that separate encryption keys be used for separate files (or even parts of large files). To this end, cloud storage providers use a new *data encryption key* (DEK) to encrypt each (part of a) file. The DEK is then wrapped using a *key encryption key* (KEK) and stored together with the encrypted file. Here, using a key hierarchy also allows for a form of *key rotation*, a process in which a key is replaced by a fresh one, and the encrypted data is updated to be secured under the new key. The technique used by all four of Amazon Web Services [4], Google Cloud [30], IBM Cloud [34] and Microsoft Azure [42] is to rotate only the KEK rather than all of the DEKs. This limits the amount of data that needs to be re-encrypted under the new KEK to just the DEKs that were wrapped under the original KEK, rather than the actual files themselves. This approach provides an efficient but security-limited form of key rotation [25].

Forward-secure session resumption and puncturable encryption. Aviram, Gellert, and Jager (AGJ) [1,2] observed that the key hierarchy induced by the ticketing mechanism in TLS 1.3 PSK mode can be used to achieve *forward security* for resumed sessions. By updating the Session Ticket Encryption Key (STEK) after accepting the ticket of a resumed session, and deleting the corresponding PSK, the confidentiality of the session is guaranteed even against an attacker who later compromises the STEK. AGJ formalized this idea with their notion of a forward-secure *session resumption protocol.* The per-session forward security enjoyed by such a resumption protocol is reminiscent of the fine-grained forward security achieved by *puncturable encryption* [31], and indeed, AGJ make use of *puncturable* pseudo-random functions (PPRFs) [13,17,36] for their construction. Their innovation naturally begs the question: Can puncturing be combined with key hierarchies to bring fine-grained forward security also to other applications? This work provides the affirmative response.

Our contributions. We investigate how puncturing can be combined with key wrapping to provide fine-grained forward security in applications using a symmetric key hierarchy. To this end, we introduce a new cryptographic primitive that we call *puncturable key wrapping* (PKW). We provide formal definitions, relations between security notions, and an efficient, generic construction for PKW. We also show how to use PKW in two sample applications: TLS ticketing (inspired by [2], but addressing several shortcomings of that work) and protected file storage. We argue that, while PKW is closely related to existing primitives like PPRFs, it provides a useful abstraction that more intuitively captures what is needed for achieving fine-grained forward security in symmetric key hierarchies. This makes building applications conceptually simpler and less error-prone.[1]

Puncturable key wrapping. A puncturable key-wrapping scheme provides the basic functionality needed for a symmetric key hierarchy: algorithms to wrap

[1] A broad analogy that readers may find useful: PKW is to PPRFs as AEAD is to block ciphers.

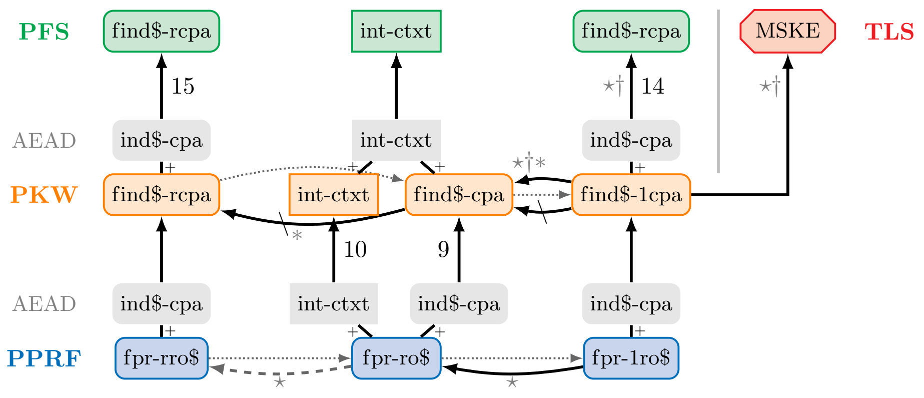

Fig. 1. Security notions and relations for PPRFs, puncturable key-wrapping (PKW), protected file storge (PFS), and TLS ticketing (TLS). Confidentiality/forward security notions are in rounded boxes, integrity notions in rectangular boxes. Solid lines indicate implications, with numbers referencing the respective theorem in this paper (others in [5]) and a plus + when combining several notions. Barred lines denote separations, dotted lines trivial implications, and dashed lines non-tight implications. A star $\star$ or dagger † next to an arrow indicates that the implication holds if puncture invariance (Defs. 2, 5), resp. consistency (Def. 6) is assumed; a $*$ indicates additional assumptions.

and unwrap data encryption keys under a master secret key. Additionally, a puncturing algorithm allows the master secret key to be updated such that specific wrapped data encryption keys are rendered irrecoverable. Our PKW syntax merges classical key wrapping/deterministic authenticated encryption [47] with tag-based puncturable encryption [31]. The resulting primitive allows authenticated headers and uses tags to enable fine-grained puncturing of ciphertexts. The puncturing tags simplify the exposition of PKW and allow for versatile treatments of the targeted applications: e.g., tags may be chosen via a counter when keeping state or ordering is required, or as random strings when meta-data privacy is a concern (cf. [7]). This contrasts with the foundational work on (non-puncturable) key wrapping [47], where randomness needed for secure wrapping is effectively extracted from the wrapped key in the SIV construction.

We introduce four different security notions for PKW schemes (see Figure 1), three relating to confidentiality (find$-cpa: a classical “real-or-random” notion, find$-rcpa: additionally allowing “real” wrappings, and find$-1cpa: a one-time challenge notion) and one to integrity (of ciphertexts, int-ctxt). They are developed with an eye towards applications, catering to the needs of key hierarchies found in cloud storage systems and the TLS ticketing mechanism. Hence, all four are in a multi-key (or multi-user) setting [6]. We also provide a simple and generic construction for a PKW scheme based on a PPRF and a nonce-based AEAD [46] scheme. The core idea is to view the master key as the secret key of a PPRF; wrapping of a selected data encryption key is performed by evaluating the PPRF on the tag to generate a one-time AEAD key, and then using that AEAD key to encrypt the data encryption key. PKW puncturing equates to

PPRF puncturing. Depending on the precise assumptions made on the PPRF, we reach our three different levels of confidentiality for the PKW scheme; the integrity notion requires nothing further of the PPRF. In all cases, standard multi-user notions of AEAD security suffice. Using a misuse-resistant AEAD scheme [47] could further enable batch puncturing of wrappings under the same tag. Full details of our treatment of PKW can be found in Section 3.

PPRFs. While the precise PPRF security notions we require resemble those in prior work [13,17,36,48], they appear to be, strictly speaking, new. This shows how an application-driven analysis can bring to the surface new requirements on existing primitives. In Section 2 (see also Figure 1), we explore the relations between our different PPRF notions and discuss possible instantiations, e.g., using the GGM construction for PRFs (as adapted to PPRFs in e.g. [36]).

To summarize, we obtain a generic instantiation of PKW, achieving a variety of security notions from standard primitives (AEAD schemes and PRGs).

Application: Forward-secure session resumption. Equipped with our new primitive, we revisit the idea of Aviram, Gellert, and Jager (AGJ) [2] for achieving forward security for the zero round-trip time (0-RTT) data that is immediately sent by clients in the TLS 1.3 PSK resumption mode. In Section 4, we show how a find\$-1cpa-secure PKW scheme can readily be deployed for TLS ticketing to yield forward-secure TLS 1.3 0-RTT resumption that is secure in the sense of a multi-stage key exchange (MSKE) protocol [27]; see also Figure 1. Using PKWs in place of PPRFs (as in AGJ) permits us to take a more generic and abstract viewpoint. This not only directly facilitates constructions offering differing functionality and security guarantees, but also enabled us to identify and correct some technical issues arising in the approach of AGJ.

In particular, building TLS ticketing from PKW allows us to seamlessly switch to a more *privacy-friendly* approach, addressing an open problem in [2]: by sampling tags randomly, we are able to make TLS tickets indistinguishable from random, whereas the AGJ proposal uses a counter in the construction, making their tickets potentially linkable to the time of issuance. Thus our approach can alleviate privacy concerns for TLS ticketing, e.g., regarding tracking users on the web by passively observing network traffic.

The integration of a session resumption protocol into the TLS 1.3 resumption handshake is described in [2, Section 4]. Rephrasing the AGJ proposal in the language of puncturable key wrapping led us to discover conceptual and technical issues in the security model, the proposed protocol, and the proof that prevent the proposal of AGJ from being forward secure, as we discuss in Section 4. Specifically, the security model used in [2] does not reflect the ticketing mechanism of a key exchange protocol in how pre-shared secrets are sampled, registered with parties, and potentially corrupted. Furthermore, the proposed protocol encrypts the TLS resumption master secret RMS in the session ticket. Since RMS is used to derive multiple PSK values, this violates forward security (an adversary learning RMS from one ticket can use it to decrypt prior sessions using a PSK derived from the same RMS). However, this can be easily fixed by ticketing the respective PSK instead of RMS. Finally, we identified overlooked steps and missing underlying assumptions in the AGJ security proof, which were surfaced when applying our PKW formalism. We address all these points in our treatment of forward-secure session resumption for TLS 1.3, see Section 4.

Application: Protected file storage. As a second application example, we show in Section 5 how our new PKW primitive can be used in an encrypted file storage system to give forward security to deleted files. This application is motivated by the current trust assumptions in cloud storage systems, where the confidentiality of the stored data rarely extends to the service provider. Indeed, if the master key in the key hierarchy is managed by the cloud, then the service provider can trivially decrypt any file. The aim of our protected file storage (PFS) system is to provide strong security guarantees for the user, even when encrypted files are outsourced to a malicious or corrupted storage system.

Using a PKW scheme, a client can locally encrypt files under separate data encryption keys, wrap the DEKs with its master key (acting as a KEK) and then outsource both the encrypted files and the wrapped keys to the cloud. In addition to relieving the user of the need to store anything beyond the master key for the PKW scheme, our PFS system also allows secure *shredding* of files: by puncturing the master key such that a specific wrapped DEK is rendered irrecoverable, the file encrypted by the DEK is made permanently inaccessible, even if the ciphertext is not actually deleted by the cloud storage provider when the client requests it to be. This means that a motivated attacker with access both to the encrypted files *and* the secret key of the user will not be able to compromise the contents of files that were shredded before the user key was compromised. The system hence provides very strong forward security guarantees for shredded files. Crucially in our approach, there is no need for the user to trust the storage provider to actually delete the shredded files, an assumption which would seldom hold in practice due to the presence of backups for disaster recovery purposes (see, e.g., [30]) or bugs in the deletion process [44].

An additional feature of our PFS system is that, in line with current industry practice, it supports key rotation at the KEK level. Key rotation extends the lifetime of encrypted data, overcomes usage limits of encryption through rekeying, and supports forward security in practice. It is also important given that the PKW schemes we build have a finite puncturing capability; KEK rotation is then used to restore puncturing capability whenever needed. The multi-key aspect of our PKW security notions readily supports this key rotation.

As core contributions here, we define a syntax for PFS and security notions capturing confidentiality, forward security, and integrity of stored files in a PFS scheme. We show how all of these notions can be achieved by building a PFS scheme from a PKW scheme and an AEAD scheme in a natural and efficient way. We actually provide two different routes to proving our main results on the forward security of PFS, as represented in the first and third column in Figure 1. These routes rely on different security assumptions on the underlying cryptographic components, specifically the PKW scheme used, and result in security theorems with different tightness properties—using a stronger PKW scheme yields a tighter proof of security for the PFS scheme. This in turn relates to the properties required of the underlying PPRF in each of the two routes. While the left, tighter, route requires a PPRF satisfying the strongest security notion (fpr-rro$) as a basic building block—an assumption which, to the best of our knowledge, generally relies on a non-tight (complexity-leveraging) reduction to weaker PPRF notions—it asks less from the building blocks in terms of other properties. Specifically, it avoids the technical requirements of puncture invariance and consistency which we detail in Section 3 and that not all PPRFs may provide, yet which are required for the right, less tight route. The two routes hence show that secure PFS schemes can be constructed from different levels of PKW (and PPRF) schemes; we see this as motivating future work on efficient PKW (or PPRF) constructions that directly fulfill our strong security notions.

We stress that the aim of our PFS system is to showcase how integrating PKWs into existing symmetric key hierarchies can improve security for the *cryptographic core* of secure file storage systems. Building a full-blown system is left to future work.

Further related work. The origins of forward security, in the context of key exchange, date back to Günther [32] and Diffie et al. [23]. A helpful systematization is given by Boyd and Gellert [16].

Green and Miers [31] introduced *puncturable* (public-key) encryption as a means of achieving fine-grained forward security. The ideas of [31] were applied to 0-RTT key exchange and session resumption for TLS 1.3 in [33,22,2] as well as symmetric key exchange [3,15]. The treatment of [15] is for general key exchange, where both parties share a key to a PPRF and puncture it in a semi-synchronized manner. By contrast, our approach to achieving forward security for TLS 1.3 PSK resumption mode using session tickets (in common with [2]) targets the use of puncturable primitives in a "one-sided" setting, where only the server holds the key and performs puncturing operations.

Puncturing techniques have further been used in the context of searchable encryption [52,51]. Fine-grained forward security is also targeted in Derived Unique Keys Per Transaction (DUKPT) [18]: keys are derived in a tree structure and used in a one-time manner, with the aim of improving security against side-channel attacks on weakly protected devices, e.g., payment terminals.

The idea of *secure outsourced storage* is not new. Blaze [10] designed a "Cryptographic File System" already in 1993 to empower users to encrypt their files, preventing remote file servers used for storage from gaining plaintext access to user data. A rich body of work followed suit, improving on and expanding the security guarantees in the direction of, for example, data integrity and file sharing [43], group collaboration [26], access pattern and metadata hiding [20,19] and minimizing trust assumptions [41]. There is also a plethora of services running on top of existing storage systems, for example [39,14]. *Key rotation* for symmetric encryption is widely used by outsourced storage systems in practice, but was only recently formally treated, see [25] and follow-up works [40,37] including work using puncturing [50].

Our approach to secure file storage shares the aim of removing the need to trust the storage provider for confidentiality, but we specifically focus on adding *forward security* for individual files. Boneh and Lipton [12] introduced the idea of using key deletion to revoke access to encrypted files, with an emphasis on file backup systems. Their proposal uses linear data structures to store keys, but lacks the fine-grained forward security and key rotation our PFS scheme offers.

A more recent proposal, BurnBox [54], recognizing the difficulty of truly secure file deletion, introduced self-revocable encryption to limit the power of compelled searches of devices. BurnBox achieves fine-grained forward security for deleted files via a tree-based key hierarchy, storing the root in erasable storage. It further hides file metadata in a protected lookup table, an approach we also suggest for our system. On the surface, these properties make BurnBox very similar to our PFS concept. However, the main goal of BurnBox is not forward security, but the much stronger notion of *compelled access security*, which encompasses temporarily revoking file access when device compromise is expected and further goals such as deletion/revocation obliviousness and timing privacy. This forces BurnBox to use highly application-specific approaches, rely on secure storage, and compromise on efficiency (e.g., of file lookups, in favor of privacy). In contrast, our approach is more generic, requires fewer assumptions, and can directly benefit from optimizations of the underlying PKW or PPRF schemes.

Notation and conventions. For a string $a \in \{0,1\}^*$, $|a|$ denotes its bit length. By $x \leftarrow\$ S$, we denote sampling x uniformly at random (u.a.r.) from a set S of size $|S|$. For sets $\mathcal{S}_1, \mathcal{S}_2$, the shorthand $\mathcal{S}_1 \overset{\cup}{\leftarrow} \mathcal{S}_2$ denotes $\mathcal{S}_1 \leftarrow \mathcal{S}_1 \cup \mathcal{S}_2$. We write $X = (x_1, x_2, \ldots, x_n)$ for an n-tuple and $X \mathrel{+}= x$ or $X \mathrel{-}= x$ for adding, resp. removing, an element x to/from a list or set. By $x \| y$ we denote the concatenation of strings or lists x and y. For an algorithm A, we denote by $y \leftarrow A(x_1, \ldots; r)$ running A on inputs $x_1, \ldots$ and random coins r with output y; by $y \leftarrow\$ A(x_1, \ldots)$ running A on uniformly random coins. The distinguished output $\perp$ indicates rejecting; by convention we require that any algorithm on input $\perp$ also outputs $\perp$.

We use the game-playing framework of [8]. By $\Pr[\mathbf{G}(\mathcal{A}) \Rightarrow x]$ we denote the probability that game $\mathbf{G}$ interacting with adversary $\mathcal{A}$ outputs x; where $\Pr[\mathbf{G}(\mathcal{A})]$ is a shorthand for $\Pr[\mathbf{G}(\mathcal{A}) \Rightarrow \text{true}]$. In games, adversaries implicitly have access to all described oracles unless otherwise indicated, and integer variables, strings, set variables and boolean variables are initialized, respectively, to 0, the empty string ε, the empty set $\emptyset$, and false, unless otherwise specified.

2 Puncturable PRFs, Security Notions, and Relations

Puncturable PRFs (PPRFs) were conceived of independently in [13], [17] and [36]. We recall the definition from Sahai and Waters [48], but restrict our attention to PPRFs with deterministic puncturing algorithms.

Definition 1 (PPRF). *A puncturable pseudorandom function* $\mathsf{PPRF} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Punc})$ *is a triple of algorithms with three associated sets; the secret-key space* $\mathcal{SK}$*, the domain* $\mathcal{X}$ *and the range* $\mathcal{Y}$*.*

- *Via* $sk \leftarrow\!\!\$\ \mathsf{KeyGen}()$*, the probabilistic key generation algorithm* KeyGen*, taking no input, outputs the secret key* $sk \in \mathcal{SK}$*.*
- *Via* $y/\bot \leftarrow \mathsf{Eval}(sk, x)$*, the function evaluation algorithm* Eval*, on input the secret key sk and an element* $x \in \mathcal{X}$ *outputs* $y \in \mathcal{Y}$ *or, to indicate failure,* $\bot$*.*
- *Via* $sk' \leftarrow \mathsf{Punc}(sk, x)$*, the deterministic puncturing algorithm* Punc*, on input the secret key sk and an element* $x \in \mathcal{X}$ *outputs an updated secret key* $sk' \in \mathcal{SK}$*.*

For correctness *we require that for all* $sk \in \mathcal{SK}$ *and all* $x, y \in \mathcal{X}$*:*

(1) $\Pr[\,\mathsf{Eval}(sk_0, x) \neq \bot \mid sk_0 \leftarrow\!\!\$\ \mathsf{KeyGen}()\,] = 1$.
(2) If $sk' \leftarrow \mathsf{Punc}(sk, x)$ *and* $y \neq x$*, then* $\mathsf{Eval}(sk, y) = \mathsf{Eval}(sk', y)$*.*
(3) If $sk' \leftarrow \mathsf{Punc}(sk, x)$*, then* $\mathsf{Eval}(sk', x) = \bot$*.*

Requirement (1) ensures that for any freshly generated secret key sk_0 *and for any* $x \in \mathcal{X}$*,* $\mathsf{Eval}(sk_0, x)$ *will not be* $\bot$*. Requirement (2) says that puncturing any secret key sk on x only affects the evaluation of x. Requirement (3) demands that the evaluation of a punctured point will always be* $\bot$*.*

Requirement (3)—which goes beyond prior PPRF definitions [48,2]—is needed to achieve integrity in applications like TLS ticketing, as we shall see in Section 4. Alternative to phrasing it as a correctness property, one could implicitly demand it in the security games. We find the explicit requirement cleaner and argue that it captures the intuitive understanding that PPRF function evaluation on punctured points should "fail". Alternative concepts such as private puncturable PRFs [11] could achieve similar results, but are harder to construct.

Following [2], we define an additional property of PPRFs called "puncture invariance" which demands that the scheme is insensitive to the order in which punctures are performed. I.e., the puncturing operation is commutative with respect to the resulting secret key. As noted in [2], this property enables reductions that change the order of punctures without an adversary later compromising the secret key noticing; this is necessary for example to have our single-challenge notion (fpr-1ro\$) imply our core PPRF notion (fpr-ro\$), as we shall see.

Definition 2 (PPRF puncture invariance). *A puncturable pseudorandom function* $\mathsf{PPRF} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Punc})$ *is* puncture invariant *if for all keys* $sk \in \mathcal{SK}$ *and all* $x_0, x_1 \in \mathcal{X}$ *it holds that* $\mathsf{Punc}(\mathsf{Punc}(sk, x_0), x_1) = \mathsf{Punc}(\mathsf{Punc}(sk, x_1), x_0)$*.*

PPRF security. We define three security notions for PPRFs, all in the multi-user setting [6], capturing the combined forward security and pseudorandomness goals, or *forward pseudorandomness* (fpr) for short. Let us start with our core *forward pseudorandomness* notion (fpr-ro\$), given in Figure 2. It is an extension of classical PRF security, where the adversary is given oracle access (Ro\$-Eval) either to the real function evaluated on a hidden key, or a lazily-sampled random function. Forward security is captured through access to a puncturing oracle (Punc) as well as corruption oracle (Corr), through which the adversary can obtain secret keys that have been punctured on all challenge points.

```
Game $\mathbf{G}^{\text{fpr-ro\$}}_{\mathsf{PPRF}}(\mathcal{A})$, $\boxed{\mathbf{G}^{\text{fpr-rro\$}}_{\mathsf{PPRF}}(\mathcal{A})}$:
    $b \leftarrow\!\!\$ \{0,1\}$; $u \leftarrow 0$; $\mathsf{T}[\cdot,\cdot] \leftarrow \bot$
    $b^* \leftarrow\!\!\$ \mathcal{A}()$
    Return $b^* = b$

New():
    $u$++; $sk_u \leftarrow\!\!\$ \mathsf{KeyGen}()$
    $\mathcal{C}_u, \mathcal{E}_u, \mathcal{P}_u \leftarrow \emptyset$; $\mathsf{corr}_u \leftarrow$ false

$\boxed{\text{Eval}(i,x)}$:
    If $x \in \mathcal{C}_i$ then return $\bot$
    $y \leftarrow \mathsf{Eval}(sk_i, x)$
    $\mathcal{E}_i \xleftarrow{\cup} \{x\}$
    Return $y$

Punc($i,x$):
    $sk_i \leftarrow \mathsf{Punc}(sk_i, x)$
    $\mathcal{P}_i \xleftarrow{\cup} \{x\}$

Ro$-Eval($i,x$):
    If $x \in \mathcal{E}_i$ or $\mathsf{corr}_i$:
        Return $\bot$
    $y_1 \leftarrow \mathsf{Eval}(sk_i, x)$
    If $y_1 = \bot$: return $\bot$
    If $\mathsf{T}[i,x] = \bot$:
        $\mathsf{T}[i,x] \leftarrow\!\!\$ \mathcal{Y}$
    $y_0 \leftarrow \mathsf{T}[i,x]$
    $\mathcal{C}_i \xleftarrow{\cup} \{x\}$
    Return $y_b$

Corr($i$):
    If $\mathcal{C}_i \not\subseteq \mathcal{P}_i$:
        Return $\bot$
    $\mathsf{corr}_i \leftarrow$ true
    Return $sk_i$
```

```
Game $\mathbf{G}^{\text{fpr-1ro\$}}_{\mathsf{PPRF}}(\mathcal{A})$:
    $b \leftarrow\!\!\$ \{0,1\}$; $u \leftarrow 0$
    $b^* \leftarrow\!\!\$ \mathcal{A}()$
    Return $b^* = b$

New-Ro$-Eval($x$):
    $u$++
    $sk_u \leftarrow\!\!\$ \mathsf{KeyGen}()$
    $y_1 \leftarrow \mathsf{Eval}(sk_u, x)$
    $y_0 \leftarrow\!\!\$ \mathcal{Y}$
    $sk_u \leftarrow \mathsf{Punc}(sk_u, x)$
    Return $(sk_u, y_b)$
```

Fig. 2. Left: Games defining real-or-\$ (fpr-ro\$, without the $\boxed{\text{boxed}}$ Eval oracle) and real-and-real-or-\$ (fpr-rro\$, with $\mathcal{A}$ having access to Eval) forward pseudorandomness. Right: Game defining one-time forward pseudorandomness (fpr-1ro\$) PPRF security. Grey code prevents trivial attacks.

Our second, stronger notion, *forward pseudorandomness with real evaluations* (fpr-rro\$), in addition gives the adversary access to a real evaluation oracle (Eval), capturing that real evaluations do not help distinguishing challenge outputs (even post-corruption).

In our third, weaker notion, *single-challenge forward pseudorandomness* (fpr-1ro\$), the adversary only gets a single challenge evaluation under each key. The challenge is obtained from oracle New-Ro\$-Eval, which on input a domain point x returns either the real function evaluation of x under the (unpunctured) secret key (in the "real" world), or a string drawn u.a.r. from $\mathcal{Y}$ (in the "ideal" world). Additionally the adversary obtains the secret key punctured on x. As usual, the adversary wins if it can distinguish the real world from the ideal one.

Definition 3 (PPRF security (fpr-ro\$, fpr-rro\$, fpr-1ro\$)). *Let* PPRF *be a puncturable pseudorandom function. We define the advantage of an adversary $\mathcal{A}$ against the forward pseudorandomness $X \in \{\text{fpr-ro\$}, \text{fpr-rro\$}, \text{fpr-1ro\$}\}$ of* PPRF *as* $\mathbf{Adv}^{X}_{\mathsf{PPRF}}(\mathcal{A}) = 2\left|\Pr\left[\mathbf{G}^{X}_{\mathsf{PPRF}}(\mathcal{A}) \Rightarrow \text{true}\right] - \frac{1}{2}\right|$, *where game* $\mathbf{G}^{X}_{\mathsf{PPRF}}(\mathcal{A})$ *is given in Figure 2.*

Comparison to prior work. Our PPRF notions resemble those in prior work, but also differ in several ways. For example, fpr-1ro\$ is similar to the non-adaptive notion in [48,2], but restricted to a single challenge. Through a multi-key hybrid argument [6], their notion implies ours. The adaptive "*rand*" notion of [2] most closely corresponds to our fpr-rro\$ notion, but our notion provides the adversary with more flexibility by both allowing multiple real-or-random challenge evaluations under each key (compared to a single evaluation under the single key in [2]) and giving it access both to a separate puncturing oracle (the *rand* experiment only punctures on the single challenge point) and corruption oracle, thereby allowing multiple key compromises of keys punctured on points chosen by the adversary. Our middle notion fpr-ro\$ is, to the best of our knowledge, new.

PPRF relations. Figure 1 (on page 3) shows the relations between our PPRF security notions. The trivial implications (dotted lines) immediately arise from restricting the adversary. As an example, fpr-rro\$ implies fpr-ro\$ because an adversary against the fpr-rro\$ security can simply ignore the EVAL-oracle. Similarly fpr-ro\$ implies fpr-1ro\$.

In the other direction, fpr-1ro\$ implies fpr-ro\$ for any puncture-invariant PPRF PPRF. That is, for any adversary $\mathcal{A}$ against the fpr-ro\$ security of PPRF, there exists an adversary $\mathcal{B}$ running in approximately the same time as $\mathcal{A}$ such that $\mathbf{Adv}^{\text{fpr-ro\$}}_{\mathsf{PPRF}}(\mathcal{A}) \leq q_{ro\$} \cdot \mathbf{Adv}^{\text{fpr-1ro\$}}_{\mathsf{PPRF}}(\mathcal{B})$, via a standard hybrid argument, where puncture invariance ensures that reorderings of punctures do not affect simulation of the later-corrupted secret key.

Via a non-tight reduction, we can also show that fpr-ro\$ implies fpr-rro\$ for a puncture-invariant PPRF. This is again via a hybrid argument, which however now involves guessing the input to the challenge query RO\$-EVAL under each key (so-called complexity leveraging [13,17]), resulting in reduction loss proportional to the size of the PPRF domain.

Instantiations from the literature. One, by now folklore, way of building a PPRF is to use the GGM PRF construction via a tree of pseudorandom-generator (PRG) evaluations [29], extended with a puncturing algorithm, as first noted by [13,17,36]. The core idea to enable puncturing on a domain point x in a GGM PRF is to update the secret key, removing nodes on the path to x in the PRG tree and adding all nodes on the co-path from the root to x. For a more in-depth description and argument of security we refer to [2,36]. Note that the GGM-based construction is correct and puncture invariant, and hence, via our established relations, yields an fpr-ro\$-secure PPRF. Additionally, for this specific construction, adaptive security can be achieved with a loss factor that is only quasi-polynomial in the input length, improving greatly over the exponential loss of complexity leveraging [28]. An alternative construction for a PPRF with security based on the Strong RSA assumption can be found in [2].

3 Puncturable Key Wrapping

We now present our core cryptographic primitive, *puncturable key wrapping* (PKW). With puncturable key wrapping, we merge the notion of key wrapping, originally extensively studied by Rogaway and Shrimpton [47], with tag-based puncturable encryption [31], adapted to the symmetric setting, to capture forward security through puncturing. Puncturable key wrapping, beyond the key K to be wrapped, hence takes a *tag* T used as a pointer for puncturing, as well as optional associated *header* data H which is authenticated along with the wrapped key (akin to associated data in AEAD). In the following, we give syntax, security, and further notions for this new primitive.

Definition 4 (PKW scheme). *A* puncturable key-wrapping scheme $\mathsf{PKW} = (\mathsf{KeyGen}, \mathsf{Wrap}, \mathsf{Unwrap}, \mathsf{Punc})$ *is a 4-tuple of algorithms with four associated sets; the secret-key space $\mathcal{SK}$, the tag space $\mathcal{T}$, the header space $\mathcal{H}$ and the wrap-key space $\mathcal{K}$. Associated to the scheme is a ciphertext-length function* $\mathsf{cl} : \mathbb{N} \to \mathbb{N}$.

- *Via* $sk \leftarrow_{\$} \mathsf{KeyGen}()$*, the probabilistic key generation algorithm* KeyGen*, taking no input, outputs a secret key* $sk \in \mathcal{SK}$.
- *Via* $C/\bot \leftarrow \mathsf{Wrap}(sk, T, H, K)$*, the deterministic wrapping algorithm* Wrap *on input a secret key* $sk \in \mathcal{SK}$*, a tag* $T \in \mathcal{T}$*, a header* $H \in \mathcal{H}$ *and a key* $K \in \mathcal{K}$ *outputs a ciphertext* $C \in \{0,1\}^{\mathsf{cl}(|K|)}$ *or, to indicate failure,* $\bot$.
- *Via* $K/\bot \leftarrow \mathsf{Unwrap}(sk, T, H, C)$*, the deterministic unwrapping algorithm* Unwrap *on input a secret key* $sk \in \mathcal{SK}$*, a tag* $T \in \mathcal{T}$*, a header* $H \in \mathcal{H}$ *and a ciphertext* $C \in \{0,1\}^*$ *returns a key* $K \in \mathcal{K}$ *or, to indicate failure,* $\bot$.
- *Via* $sk' \leftarrow \mathsf{Punc}(sk, T)$*, the deterministic puncturing algorithm* Punc *on input a secret key* $sk \in \mathcal{SK}$ *and a tag* $T \in \mathcal{T}$ *returns a potentially updated secret key* $sk' \in \mathcal{SK}$.

Correctness *of a PKW scheme intuitively demands that a wrapped key can be recovered from its wrapping ciphertext unless the secret key has been punctured on the tag used for the wrapping step, i.e., even if the secret key has been punctured on other tags. Formally, we require that for all* $T \in \mathcal{T}$, $H \in \mathcal{H}$, $K \in \mathcal{K}$*, and all tuples* $\bar{T}_1, \bar{T}_2 \in \mathcal{T}^*$ *where* $T \notin \bar{T}_1$ *and* $T \notin \bar{T}_2$,

$$\Pr\left[\, \mathsf{Unwrap}(sk_{\setminus \bar{T}_1}, T, H, \mathsf{Wrap}(sk_{\setminus \bar{T}_2}, T, H, K)) = K \mid sk \leftarrow_{\$} \mathsf{KeyGen}() \,\right] = 1.$$

Here $sk_{\setminus (T_1, T_2, \ldots, T_n)} = \mathsf{Punc}(\ldots(\mathsf{Punc}(\mathsf{Punc}(sk, T_1), T_2), \ldots), T_n)$ *is shorthand for the secret key obtained via puncturing sk in order on* $T_1, \ldots, T_n \in \mathcal{T}$.

Analogously to Definition 2 for PPRFs, we also define *puncture invariance* for PKW schemes, demanding that the order of punctures does not affect the resulting secret key.

Definition 5 (PKW puncture invariance). *A puncturable key-wrapping scheme* $\mathsf{PKW} = (\mathsf{KeyGen}, \mathsf{Wrap}, \mathsf{Unwrap}, \mathsf{Punc})$ *is* puncture invariant *if for all keys* $sk \in \mathcal{SK}$ *and all tags* $T_0, T_1 \in \mathcal{T}$ *it holds that*

$$\mathsf{Punc}(\mathsf{Punc}(sk, T_0), T_1) = \mathsf{Punc}(\mathsf{Punc}(sk, T_1), T_0).$$

Additionally, we introduce a property of PKW schemes which we call *consistency*, inspired by the definition of consistent puncturable signature schemes in [9]. A consistent PKW scheme is one for which the output of algorithm Wrap only depends on the tag, header and wrap-key input, and not on the (puncturing) state of the secret key—except for when the output is $\bot$ due to puncturing.

```
Game $\mathbf{G}^{\text{find\$-cpa}}_{\mathsf{PKW}}(\mathcal{A})$, $\boxed{\mathbf{G}^{\text{find\$-rcpa}}_{\mathsf{PKW}}(\mathcal{A})}$:
  $b \leftarrow_\$ \{0,1\}$; $u \leftarrow 0$
  $b^* \leftarrow_\$ \mathcal{A}()$
  Return $b^* = b$

New():
  $u{+}{+}$
  $sk_u \leftarrow_\$ \mathsf{KeyGen}()$
  $\mathcal{S}_{PT,u}, \mathcal{S}_{\$T,u}, \mathcal{S}_{T,u} \leftarrow \emptyset$
  $\mathsf{corr}_u \leftarrow$ false

$\boxed{\text{Wrap}(i, T, H, K)}$:
  If $T \in \mathcal{S}_{T,i}$ then return $\bot$
  $C \leftarrow \mathsf{Wrap}(sk_i, T, H, K)$
  $\mathcal{S}_{T,i} \xleftarrow{\cup} \{T\}$
  Return $C$

Ro$-Wrap($i, T, H, K$):
  If $T \in \mathcal{S}_{T,i}$ or $\mathsf{corr}_i$:
    Return $\bot$
  $C_1 \leftarrow \mathsf{Wrap}(sk_i, T, H, K)$
  If $C_1 = \bot$ then return $\bot$
  $C_0 \leftarrow_\$ \{0,1\}^{\mathsf{cl}(|K|)}$
  $\mathcal{S}_{\$T,i} \xleftarrow{\cup} \{T\}$; $\mathcal{S}_{T,i} \xleftarrow{\cup} \{T\}$
  Return $C_b$

Corr($i$):
  If $\mathcal{S}_{\$T,i} \not\subseteq \mathcal{S}_{PT,i}$:
    Return $\bot$
  $\mathsf{corr}_i \leftarrow$ true
  Return $sk_i$

Punc($i, T$):
  $sk_i \leftarrow \mathsf{Punc}(sk_i, T)$
  $\mathcal{S}_{PT,i} \xleftarrow{\cup} \{T\}$
```

```
Game $\mathbf{G}^{\text{find\$-1cpa}}_{\mathsf{PKW}}(\mathcal{A})$:
  $b \leftarrow_\$ \{0,1\}$; $u \leftarrow 0$
  $b^* \leftarrow_\$ \mathcal{A}()$
  Return $b^* = b$

New-Ro$-Wrap($T, H, K$):
  $u{+}{+}$
  $sk_u \leftarrow_\$ \mathsf{KeyGen}()$
  $C_1 \leftarrow \mathsf{Wrap}(sk_u, T, H, K)$
  $C_0 \leftarrow_\$ \{0,1\}^{\mathsf{cl}(|K|)}$
  $sk_u \leftarrow \mathsf{Punc}(sk_u, T)$
  Return $(sk_u, C_b)$
```

Fig. 3. Left and middle: Forward security and privacy find\$-cpa (without access to the $\boxed{\text{boxed}}$ Wrap oracle) / find\$-rcpa (with access to Wrap) of a puncturable key-wrapping scheme PKW. Right: One-time privacy and forward security find\$-1cpa security of a puncturable key-wrapping scheme PKW.
Grey code prevents trivial attacks and ensures that unique tags are used for wrapping.

Definition 6 (PKW consistency). *A puncturable key wrapping scheme* $\mathsf{PKW} = (\mathsf{KeyGen}, \mathsf{Wrap}, \mathsf{Unwrap}, \mathsf{Punc})$ *is* consistent *if for all keys* $K \in \mathcal{K}$, *all headers* $H \in \mathcal{H}$, *all tags* $(T_1, \dots, T_n) \in \mathcal{T}^*$ *and all* $T \in \mathcal{T} \setminus \{T_1, \dots, T_n\}$ *it holds that*

$$\Pr\left[\mathsf{Wrap}(sk, T, H, K) = \mathsf{Wrap}(sk_{\setminus(T_1,\dots,T_n)}, T, H, K) \mid sk \leftarrow_\$ \mathsf{KeyGen}()\right] = 1.$$

Puncture invariance and consistency guarantee a kind of indifference of the PKW scheme with respect to puncturing, allowing sequences of punctures and wrappings to be flexibly reordered without affecting the scheme's future behavior. As we shall see, these properties are important to consider when deploying PKW schemes in, and proving the security of, higher-level applications.

3.1 PKW Security

Confidentiality. Following Rogaway and Shrimpton [47], we adopt indistinguishability from random bits (ind\$) as the appropriate notion to model confidentiality for (puncturable) key-wrapping schemes. Our three confidentiality notions, formalized in Figure 3, capture forward security in the sense that the confidentiality guarantees hold also after compromise of the secret key, given that it has been appropriately punctured prior to corruption to avoid trivial wins. As before, they are all in the multi-key (or multi-user) setting [6].[2]

Our first security notion, which we call find\$-cpa, can be viewed as a form of ind\$-cpa security adapted to the PKW setting. The adversary is given access to a challenge wrapping oracle RO\$-WRAP, which on input a key index i, a tag T, a header H and a key K chosen by the adversary, returns either an honest wrapping of K under secret key sk_i, or a random bit-string of length $\mathsf{cl}(|K|)$. Forward security is captured via a corruption oracle CORR which allows the adversary to compromise the current version of a secret key sk_i, given that all tags used in challenge queries under sk_i must be punctured on at the time of corruption (via the puncturing oracle PUNC). Focusing on fine-grained forward security, we restrict the adversary to use tags only *once* for wrapping and call this behavior *tag-respecting* (akin to a nonce-respecting adversary in authenticated encryption); this enables puncturing of *individual* ciphertexts.[3]

Guided by the envisioned usage of puncturable key-wrapping schemes, our second, stronger confidentiality notion, find\$-rcpa, additionally gives the adversary access to real wrappings that it does not have to puncture on via an additional oracle WRAP. The rationale behind the notion is that although find\$-cpa provides forward security for all wrapped keys which have been punctured on at the time of compromise, it does not capture the potential leakage from unpunctured ciphertexts which the adversary gains insight into by corrupting. That is, we would like to ensure that there is a form of independence across key wrappings produced with distinct tags. This is motivated by what we believe to be realistic attack scenarios for applications which use a PKW scheme for key management—such as our protected file storage system (to be defined in Section 5). In such a system, normal usage implies the existence of some unpunctured ciphertexts (corresponding to non-shredded files) at any given time, and hence in particular at the time of a key compromise. The idea of find\$-rcpa security is that compromising ciphertexts generated with tags that have not been punctured on, should not give the adversary a higher advantage in distinguishing challenge ciphertexts from random bits.

Lastly, we also introduce a one-time security notion, find\$-1cpa, which only provides the adversary with one challenge output and the punctured secret key, per key. As we will see, together with puncture invariance and consistency, find\$-1cpa turns out to be sufficiently strong to achieve full security in the applications we are interested in.

[2] To focus on forward security, we separate confidentiality (with forward security) and integrity (below) into distinct notions, contrasting with the combined notion in [47]. We give a combined notion in the full version [5], also capturing CCA-style active attacks, and show that it is equivalent to the junction of our separate notions.

[3] We note that a stronger formalization is possible where tag reuse is allowed: by storing and checking the whole tuple (T, H, K) in the sets $\mathcal{S}_{T,i}$ instead of only T, one can demand wraps to look random except when this is impossible due to entirely repeating inputs. This could cater to applications interested in "batch puncturing" [31], i.e., revoking access to multiple wrapped keys via a single puncturing call. Such stronger notions would also require stronger building blocks, as we will see below.

```
Game $\mathbf{G}^{\text{int-ctxt}}_{\mathsf{PKW}}(\mathcal{A})$:
    $\mathsf{win} \leftarrow \text{false}$; $u \leftarrow 0$
    $\mathcal{A}()$
    Return $\mathsf{win}$

$\textsc{New}()$:
    $u$++; $sk_u \leftarrow\$ \mathsf{KeyGen}()$
    $\mathcal{S}_{THC,u}, \mathcal{S}_{T,u}, \mathcal{S}_{PT,u} \leftarrow \emptyset$

$\textsc{Punc}(i, T)$:
    $sk_i \leftarrow \mathsf{Punc}(sk_i, T)$
    $\mathcal{S}_{PT,i} \xleftarrow{\cup} \{T\}$

$\textsc{Wrap}(i, T, H, K)$:
    If $T \in \mathcal{S}_{T,i}$ then return $\perp$
    $C \leftarrow \mathsf{Wrap}(sk_i, T, H, K)$
    If $C = \perp$ then return $\perp$
    $\mathcal{S}_{THC,i} \xleftarrow{\cup} \{(T, H, C)\}$; $\mathcal{S}_{T,i} \xleftarrow{\cup} \{T\}$
    Return $C$

$\textsc{Unwrap}(i, T, H, C)$:
    $K \leftarrow \mathsf{Unwrap}(sk_i, T, H, C)$
    If $K \neq \perp$ and $((T, H, C) \notin \mathcal{S}_{THC,i}$ or $T \in \mathcal{S}_{PT,i})$:
        $\mathsf{win} \leftarrow \text{true}$
    Return $K$
```

Fig. 4. Integrity of ciphertexts of a puncturable key-wrapping scheme PKW. Grey code prevents trivial attacks and ensures that tags are not repeated in wrap queries.

Definition 7 (PKW confidentiality (find\$-cpa**,** find\$-rcpa**,** find\$-1cpa)**).** *Let* PKW *be a puncturable key-wrapping scheme. We define the advantage of an adversary* $\mathcal{A}$ *against the forward indistinguishability* $X \in \{\text{find\$-cpa}, \text{find\$-rcpa}, \text{find\$-1cpa}\}$ *of* PKW *as* $\mathbf{Adv}^{X}_{\mathsf{PKW}}(\mathcal{A}) = 2\left|\Pr\left[\mathbf{G}^{X}_{\mathsf{PKW}}(\mathcal{A}) \Rightarrow \text{true}\right] - \frac{1}{2}\right|$*, where* $\mathbf{G}^{X}_{\mathsf{PKW}}(\mathcal{A})$ *is defined in Figure 3.*

Integrity. In addition to the confidentiality notions we also define (multi-key) *integrity of ciphertexts* (int-ctxt) for PKW schemes as shown in Figure 4. Here, the adversary is given oracle access to wrapping (Wrap), unwrapping (Unwrap), and puncturing (Punc). Its goal is to forge a ciphertext (together with a tag and a header) that was not output by Wrap, or for which the tag was punctured on via Punc, and that unwraps to something other than the error symbol $\perp$. Note that we particularly treat ciphertexts under punctured tags as valid forgery attempts, even if previously output by Wrap. This ensures that after puncturing on a tag, no ciphertext with that tag will be accepted any more, which is sometimes referred to as *replay protection.*

Definition 8 (PKW integrity (int-ctxt)**).** *Let* PKW *be a puncturable key-wrapping scheme. We define the advantage of an adversary* $\mathcal{A}$ *against the* integrity of ciphertexts *of* PKW *as* $\mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{PKW}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\text{int-ctxt}}_{\mathsf{PKW}}(\mathcal{A}) \Rightarrow \text{true}\right]$*, where* $\mathbf{G}^{\text{int-ctxt}}_{\mathsf{PKW}}(\mathcal{A})$ *is defined in Figure 4.*

Notably, in the integrity setting, forging a valid ciphertext becomes trivial if one would allow the adversary to compromise the secret key. Forward security hence seems to only make sense in scenarios where *two copies* of the key are available simultaneously, one "more punctured" than the other. The challenge then would be to forge a ciphertext on a punctured tag T using access to the compromised, more punctured key, such that the ciphertext unwraps under the less punctured key (which has not been punctured on T). This could be interesting, e.g., in a setting where punctured keys are distributed across servers. We leave extending puncturing to the distributed setting as future work.

Relations between PKW notions. We briefly explain how the PKW confidentiality notions are related. See Figure 1 for an overview of all security notions and their relations, and the full version [5] for details and proofs. Beginning from strong to weak: the trivial implications (dotted arrows) arise directly from restricting the adversary. As an example, find\$-rcpa implies find\$-cpa because an adversary against the find\$-rcpa security can simply ignore the Wrap-oracle.

In the opposite direction the relations are more complex. Generally, find\$-1cpa does not imply find\$-cpa. Showing the separation is straightforward: Modify any find\$-1cpa secure scheme so that Wrap outputs a fixed string when receiving an already-punctured tag as input. This makes challenge wraps on punctured tags—which are available in the find\$-cpa game, but not in find\$-1cpa—easily distinguishable. In contrast, for the special case of a PKW scheme that is puncture invariant and consistent, and additionally for which attempting to wrap using a punctured tag always results in $\perp$ (i.e., $\mathsf{Wrap}(sk_{\setminus \bar{T}}, T, \cdot, \cdot) = \perp$ if $T \in \bar{T} \subseteq \mathsf{PKW}.\mathcal{T}$)[4], find\$-1cpa implies find\$-cpa via a hybrid argument.

Lastly, assuming a (forward) secure source of pseudorandomness, such as a fpr-ro\$ secure PPRF, find\$-rcpa is strictly stronger than find\$-cpa. The separation relies on the fact that in the find\$-cpa game, an adversary must puncture on all tags which have been used for wrapping before compromising the secret key; a restriction which is not imposed on tags queried to oracle Wrap in the find\$-rcpa game. This can be used to construct a scheme which leaks a copy of the original, unpunctured secret key when punctured *only once* on a hidden, special tag $\hat{T}$, which can only be learned by wrapping under a different, fixed and publicly known tag T_0. Tag $\hat{T}$ is accessible to an adversary in the find\$-rcpa game via oracle Wrap, but not to a find\$-cpa adversary. The latter can learn $\hat{T}$ only through a Ro\$-Wrap call on T_0, forcing it to also puncture on T_0 and thereby destroying the key copy.

3.2 Instantiating PKW from PPRF and AEAD

Next, we give a generic construction of a PKW scheme, formalized in Figure 5. The construction uses an authenticated encryption scheme with associated data AEAD to encrypt (wrap) keys, using a new AEAD key together with a fixed nonce N_0 for each key-wrap. The keys of AEAD are generated by a pseudorandom function PPRF on input the wrap tag, the key of which is the secret key of the PKW scheme. This allows AEAD keys to be "forgotten" via puncturing the PPRF key, thereby rendering the key-wrap ciphertexts unrecoverable. The construction is inspired by, and re-captures, the generic construction of a "0-RTT session resumption protocol" by Aviram, Gellert, and Jager [2], with the difference that we use a *nonce-based* AEAD scheme, following practically deployed schemes like AES-GCM or ChaCha20-Poly1305, rather than a probabilistic one.

The only technical requirement for our construction is that the range of PPRF matches the key space of AEAD. The key space of the resulting PKW scheme is

[4] The last assumption is necessary for the reduction to simulate a Ro\$-Wrap challenge query on an already punctured tag in the find\$-cpa game.

```
PKW[PPRF, AEAD]:

KeyGen():
1 Return PPRF.KeyGen()

Wrap($sk_p, T, H, K$):
2 $sk_a \leftarrow$ PPRF.Eval($sk_p, T$)
3 $C \leftarrow$ AEAD.Enc($sk_a, N_0, H, K$)
4 Return $C$

Unwrap($sk_p, T, H, C$):
5 $sk_a \leftarrow$ PPRF.Eval($sk_p, T$)
6 $K \leftarrow$ AEAD.Dec($sk_a, N_0, H, C$)
7 Return $K$

Punc($sk_p, T$):
8 $sk'_p \leftarrow$ PPRF.Punc($sk_p, T$)
9 Return $sk'_p$
```

Fig. 5. The PKW[PPRF, AEAD] instantiation of a puncturable key-wrapping scheme based on a puncturable pseudorandom function PPRF and a nonce-based AEAD scheme AEAD (with N_0 a fixed nonce in the nonce space of AEAD).

the key space of PPRF, the tag space the PPRF domain, the header space the associated data space of AEAD, and the wrap-key space the message space of AEAD. The ciphertext-length function cl for PKW is that of AEAD.

Our construction PKW[PPRF, AEAD] achieves puncture invariance and consistency (given PPRF is puncture invariant), all levels of forward indistinguishability (find\$-cpa, find\$-rcpa, and find\$-1cpa) given AEAD ind\$-cpa security and the corresponding strength (fpr-ro\$, fpr-rro\$, resp. fpr-1ro\$) of the underlying PPRF security, as well as integrity of ciphertexts (given PPRF fpr-ro\$ security and AEAD int-ctxt security). For space reasons, we only give security statements for find\$-cpa forward indistinguishability and integrity here, deferring the details of the other results to the full version [5].

Theorem 9 (PKW[PPRF, AEAD] **is** find\$-cpa **secure). *Let* PKW[PPRF, AEAD] *be the PKW scheme in Figure 5. For every adversary $\mathcal{A}$ against the* find\$-cpa-*security of* PKW[PPRF, AEAD] *making at most q_n, $q_{\text{ro\$}}$, q_{corr} and q_p queries to oracles* New, Ro\$-Wrap, Corr *and* Punc, *respectively, there exists adversaries $\mathcal{B}_{\mathsf{pprf}}$ and $\mathcal{B}_{\mathsf{aead}}$ running in approximately the same time as $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\text{find\$-cpa}}_{\mathsf{PKW[PPRF,AEAD]}}(\mathcal{A}) \leq 2 \cdot \mathbf{Adv}^{\text{fpr-ro\$}}_{\mathsf{PPRF}}(\mathcal{B}_{\mathsf{pprf}}) + \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}}).$$

Adversary $\mathcal{B}_{\mathsf{pprf}}$ makes at most q_n, $q_{\text{ro\$}}$, q_{corr}, and q_p queries to oracles New, Ro\$-Eval, Corr, *resp.* Punc. *Adversary $\mathcal{B}_{\mathsf{aead}}$ makes at most $q_{\text{ro\$}}$ queries to oracles* New *and* Ro\$.

Proof. We first leverage the fpr-ro\$ security of PPRF to replace the AEAD keys by random ones, then in a second step apply ind\$-cpa security of AEAD to argue that wrapped PKW[PPRF, AEAD] ciphertexts are indistinguishable from random. The first step consists of a game hop from the original find\$-cpa game, abbreviated $\mathbf{G}_0$, to a game $\mathbf{G}_1$ which replaces the outputs of PPRF by random AEAD keys in the implementation of oracle Ro\$-Wrap. We bound the difference $|\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_1]|$ by the distinguishing advantage of an adversary $\mathcal{B}_{\mathsf{pprf}}$ against the fpr-ro\$ security of PPRF (cf. Definition 3).

Adversary $\mathcal{B}_{\mathsf{pprf}}$ draws a random bit b' and acts as the challenger in game $\mathbf{G}_0$. When $b' = 1$ adversary $\mathcal{B}_{\mathsf{pprf}}$ simulates the "real world" in the PKW game, wrapping the keys output by adversary $\mathcal{A}$. When $b' = 0$, adversary $\mathcal{B}_{\mathsf{pprf}}$ simulates the "random world" and returns random strings in the ciphertext space of the AEAD scheme in response to challenge queries from $\mathcal{A}$. Finally, when adversary $\mathcal{A}$ halts and outputs bit $b^*_{\mathcal{A}}$, adversary $\mathcal{B}_{\mathsf{pprf}}$ returns 1 if $b^*_{\mathcal{A}} = b'$ and 0 otherwise.

Let b denote the random bit drawn by the challenger in the fpr-ro\$ game. When $b = 1$, adversary $\mathcal{B}_{\mathsf{pprf}}$ simulates game $\mathbf{G}_0$ for $\mathcal{A}$. When $b = 0$, the simulation corresponds to game $\mathbf{G}_1$. This gives $\mathbf{Adv}^{\text{fpr-ro\$}}_{\mathsf{PPRF}}(\mathcal{B}_{\mathsf{pprf}}) = |\Pr[\,\mathbf{G}_0\,] - \Pr[\,\mathbf{G}_1\,]|$.

It remains to bound $\Pr[\,\mathbf{G}_1(\mathcal{A})\,]$. A straightforward reduction to the multi-key ind\$-cpa security of AEAD gives $\Pr\left[\,\mathbf{G}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}})\,\right] = \Pr[\,\mathbf{G}_1(\mathcal{A})\,]$ for an adversary $\mathcal{B}_{\mathsf{aead}}$ which simulates game $\mathbf{G}_1$ for adversary $\mathcal{A}$. Adversary $\mathcal{B}_{\mathsf{aead}}$ acts as the challenger in the game, except for when adversary $\mathcal{A}$ makes a query to oracle RO\$-WRAP. To respond to such a query RO\$-WRAP(j, T, H, K), $\mathcal{B}_{\mathsf{aead}}$ first queries oracle NEW to initiate a new AEAD key. Additionally it increments an internal key counter i by one. It then issues a (single) query RO\$$(i, N_0, H, K)$, requesting the challenge to be under the new key. The assumption that adversary $\mathcal{A}$ is tag-respecting ensures that this is a sound simulation. □

Note that for all our forward indistinguishability results, *one-time* multi-user AEAD security suffices, since the uniqueness of tags means that each AEAD encryption is performed under a new key. If we wanted to allow tag-reuse to enable *batch puncturing* (cf. Footnote 3), our $\mathsf{PKW}[\mathsf{PPRF}, \mathsf{AEAD}]$ scheme would need to be instantiated with a *misuse-resistant* AEAD scheme [47] to achieve find\$-cpa security. Interestingly, this straightforward modification is insufficient for find\$-rcpa security: the reuse of tags across real and challenge wrap queries creates a key commitment problem which breaks the reduction. This could potentially be addressed in an idealized model, cf. [35], but we leave this to future work.

Theorem 10 **($\mathsf{PKW}[\mathsf{PPRF}, \mathsf{AEAD}]$ is int-ctxt secure).** *Let* $\mathsf{PKW}[\mathsf{PPRF}, \mathsf{AEAD}]$ *be the PKW scheme in Figure 5. For every adversary $\mathcal{A}$ against the* int-ctxt-*security of* $\mathsf{PKW}[\mathsf{PPRF}, \mathsf{AEAD}]$ *(Def. 8) making at most q_w, q_u, q_p and q_n to oracles* WRAP, UNWRAP, PUNC *and* NEW, *respectively, there exists adversaries $\mathcal{B}_{\mathsf{aead}}$ and $\mathcal{B}_{\mathsf{pprf}}$ running in approximately the same time as $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{PKW}[\mathsf{PPRF},\mathsf{AEAD}]}(\mathcal{A}) \leq \mathbf{Adv}^{\text{fpr-ro\$}}_{\mathsf{PPRF}}(\mathcal{B}_{\mathsf{pprf}}) + \mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}}).$$

Adversary $\mathcal{B}_{\mathsf{pprf}}$ makes at most $q_w + q_u$, q_p, and q_n queries to oracles RO\$-EVAL, PUNC, *resp.* NEW. *Adversary $\mathcal{B}_{\mathsf{aead}}$ makes at most $q_w + q_u$, q_w, and q_u queries to oracles* NEW, ENC, *resp.* DEC.

Proof. We first apply the fpr-ro\$ security of PPRF to replace all AEAD keys by (consistent) random strings, denoting the original game as $\mathbf{G}_0$ and the modified one as $\mathbf{G}_1$. Somewhat similarly to the first step in the proof of Theorem 9, a reduction $\mathcal{B}_{\mathsf{pprf}}$ can bound the introduced difference as $|\Pr[\,\mathbf{G}_0(\mathcal{A})\,] - \Pr[\,\mathbf{G}_1(\mathcal{A})\,]| = \mathbf{Adv}^{\text{fpr-ro\$}}_{\mathsf{PPRF}}(\mathcal{B}_{\mathsf{pprf}})$. Here, $\mathcal{B}_{\mathsf{pprf}}$ uses its challenge oracle RO\$-EVAL to request AEAD keys upon wrapping and unwrapping, and directly relays NEW and PUNC query from $\mathcal{A}$ to its own corresponding oracles. When $\mathcal{A}$ halts, $\mathcal{B}_{\mathsf{pprf}}$ checks and outputs 1 iff $\mathcal{A}$ produced a valid forgery; this yields the first bound.

The second part of the proof now leverages the independent random AEAD keys to reduce a forgery of $\mathcal{A}$ in $\mathbf{G}_1$ to an AEAD multi-key integrity forgery via the following adversary $\mathcal{B}_{\mathsf{aead}}$. Adversary $\mathcal{B}_{\mathsf{aead}}$ simulates $\mathbf{G}_1$ using its oracles Enc and Dec to wrap, resp. unwrap. Each time $\mathcal{A}$ makes a wrap or unwrap query under a new pair (i, T), $\mathcal{B}_{\mathsf{aead}}$ employs a new key index j in the int-ctxt game which it tracks via some table $\mathsf{T}[i, T] = j$. To track puncturing on (i, T), $\mathcal{B}_{\mathsf{aead}}$ sets $\mathsf{T}[i, T] = \bot$ and responds with $\bot$ to any subsequent Wrap/Unwrap calls on (i, T). This way, $\mathcal{B}_{\mathsf{aead}}$ perfectly simulates game $\mathbf{G}_1$ for $\mathcal{A}$. Additionally, $\mathcal{B}_{\mathsf{aead}}$ wins game $\mathbf{G}^{\text{int-ctxt}}_{\mathsf{AEAD}}$ precisely when adversary $\mathcal{A}$ submits a valid forgery in $\mathbf{G}_1$, as the latter means $\mathcal{A}$ unwraps a not previously output ciphertext under a non-punctured key (as otherwise that key was set to $\mathsf{T}[i, T] = \bot$, yielding $\bot$ upon unwrapping), which translates to an AEAD forgery in $\mathcal{B}_{\mathsf{aead}}$'s Dec call. This completes the bound, as now $\mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}}) \geq \Pr[\mathbf{G}_1(\mathcal{A}) \Rightarrow \text{true}]$. $\square$

4 TLS Ticketing

We now turn our attention to applications and begin with the Transport Layer Security (TLS) protocol. We show how the ticketing approach taken in its resumption handshake protocol can be instantiated with a PKW scheme, increasing forward security of resumed sessions. A TLS connection between clients and servers begins with the establishment of a shared symmetric key through a so called *handshake*. For repeated connections, TLS offers a *resumption* handshake mode with better performance, which bootstraps security from a *pre-shared key* (PSK) established in a prior full handshake.

In order to enable a resumption handshake, the so-called "resumption master secret" RMS is derived in a TLS 1.3 handshake and then used to derive (usually multiple) pre-shared keys for later resumptions. For each such pre-shared key, the TLS 1.3 server sends the client a unique nonce N_T, and both derive the pre-shared key as PSK $\leftarrow$ $\mathsf{HKDF.Expand}(\text{RMS}, \texttt{"tls13 resumption"} \| N_T)$ using the HKDF key derivation function [38]. The client will store all PSKs established, but the server may outsource this storage to the client, e.g., by encrypting PSK under a long-term symmetric key, the so-called Session Ticket Encryption Key (STEK), and sending the resulting ciphertext (as the PSK identifier) to the client. This process of outsourcing the server-side resumption state to the client is commonly referred to as *ticketing* [49], and the identifier hence called a *ticket*.

One issue with TLS ticketing is that the tickets are generally not forward secret: if an attacker compromises the STEK, it will be able to recover the PSKs encrypted in prior resumption handshakes, thereby compromising the security of the concerned sessions. While TLS 1.3 allows for ephemeral Diffie–Hellman secrets to be mixed into the key derivation, the so-called "early" or "zero round-trip time" (0-RTT) data that a client can send immediately does not enjoy this protection, and hence would be exposed if the PSK were to be compromised.

Aviram, Gellert, and Jager (AGJ) [2] recently proposed an approach to achieve forward-secure session ticketing, giving forward security even for 0-RTT data, through what they call "session resumption protocols." In this section we

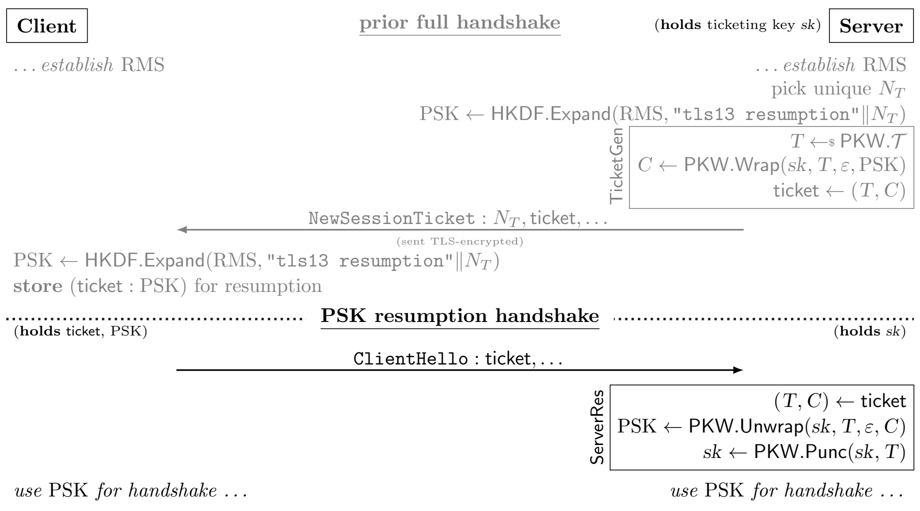
Client
prior full handshake
(holds ticketing key sk)
Server
... establish RMS
... establish RMS
pick unique N_T
PSK ← HKDF.Expand(RMS, "tls13 resumption"$\|N_T$)
TicketGen
$T \leftarrow_{\$} \mathsf{PKW}.\mathcal{T}$
$C \leftarrow \mathsf{PKW.Wrap}(sk, T, \varepsilon, \mathrm{PSK})$
ticket ← (T, C)
NewSessionTicket : N_T, ticket, ...
(sent TLS-encrypted)
PSK ← HKDF.Expand(RMS, "tls13 resumption"$\|N_T$)
store (ticket : PSK) for resumption
PSK resumption handshake
(holds ticket, PSK)
(holds sk)
ClientHello : ticket, ...
ServerRes
(T, C) ← ticket
PSK ← $\mathsf{PKW.Unwrap}(sk, T, \varepsilon, C)$
$sk \leftarrow \mathsf{PKW.Punc}(sk, T)$
use PSK for handshake ...
use PSK for handshake ...

Fig. 6. Forward-secure TLS 1.3 0-RTT pre-shared key (PSK) resumption handshake using a puncturable key-wrapping scheme PKW (bottom part), based on a session ticket generated by the server and stored by the client in a prior full handshake (upper part, in gray). The $\boxed{\text{boxed}}$ sections can be read as the PKW-based instantiation of a session resumption protocol [2], with tag sampling and wrapping corresponding to ticket generation (TicketGen) and unwrapping and puncturing corresponding to session resumption (ServerRes); the PKW key *sk* plays the role of the STEK.

revisit their approach and show how their session resumption mechanism can be viewed more simply through the lens of puncturable key wrapping: First of all, their construction is mimicked by our instantiation $\mathsf{PKW}[\mathsf{PPRF}, \mathsf{AEAD}]$ of a PKW from a puncturable PRF and an AEAD scheme, when tags are chosen (and sent as part of the TLS ticket) as *counters*. More importantly, capturing TLS ticketing through the PKW scheme $\mathsf{PKW}[\mathsf{PPRF}, \mathsf{AEAD}]$ allows us to seamlessly switch to a more *privacy-friendly* variant: by choosing the tags as random values, we make the entire TLS ticket random-looking. This avoids the potentially traceable counter element in the AGJ [2] ticketing proposal, thereby addressing privacy concerns for TLS ticketing, e.g., regarding tracking users on the web by passive network observers (see [53] for a broader discussion).

When rephrasing the AGJ integration of a session resumption protocol into the TLS 1.3 resumption handshake [2, Section 4.2, 4.3] as puncturable key wrapping, we found conceptual and technical issues in their proposed protocol, the security model, and the proof. These prevent their proposal from being (forward-)secure as-is. We rectify this situation through the following corrections:

1. Ticketing the right key. In AGJ, the TLS 1.3 resumption master secret RMS is encrypted in the session ticket(s). However, RMS is used to derive *multiple* pre-shared keys PSK for resumption. Ticketing RMS thus violates the goal of forward security: an adversary learning RMS from one ticket can use that value to decrypt prior sessions using a PSK derived from the same RMS.

In our protocol integration (cf. Figure 6), we instead ticket PSK, not RMS, following the TLS 1.3 RFC [45, Section 4.6.1].

2. Accurately modeling tickets and corruption. The security model in AGJ does not reflect the ticketing mechanism of a key exchange protocol in how pre-shared secrets are sampled, registered with parties, and possibly corrupted. This leads to their model, strictly speaking, being unable to capture the ticketing mechanism of TLS resumption.[5] Only allowing server-side corruptions, their model also fails to capture that an adversary might compromise pre-shared secrets stored by clients.
   In our security model, we integrate the protocol's ticketing mechanism and allow the adversary to corrupt both the ticketing mechanism keys of servers, as well as stored secrets of clients.
3. Rectifying proof steps. The security proof for the protocol integration of AGJ [2, Theorem 4] only uses part of the power of their session resumption primitive (i.e., a single challenge where their primitive provides many), and also misses some preliminary steps (esp. the necessity of puncture invariance and consistency, which our PKW formalism brings to light).
   In our proof, we add these missing steps and show that reducing to the weaker one-time PKW security suffices for our integration.
4. Making underlying assumptions precise. The AGJ proof makes two undefined assumptions on the underlying session resumption resp. PPRF scheme. Formally, this leads to an issue with the security proof of their construction, which in turn enables a theoretical violation of the formal integrity claims on their protocol.
   Through our formalism for puncturable key wrapping and PPRFs, we make the necessary assumptions (puncture invariance for PKW, resp. demanding $\perp$ output after puncturing for PPRFs) visible and explicit.

Overall, our exposition stays close to the approach by AGJ, focusing on the necessary corrections. We see this not only as an illustration that puncturable key wrapping is readily applicable to achieve forward-secure 0-RTT session resumption, but also that this conceptual framework helps to avoid errors when integrating puncturing techniques into more complex applications. For space reasons, we defer the technical details of our integration of PKW-based ticketing into TLS as well as the accompanying revised security model, proof, and discussion of assumptions to the full version [5].

5 Protected File Storage

We now turn our focus to our second application, file storage, and show how a PKW scheme can be used to provide (forward) security for remotely stored

[5] E.g., when setting up new pre-shared keys, their model takes the identifier *psid* of the key as an *adversary-provided* input, while *psid* in fact corresponds to the ticket *(honestly) output* by the protocol's ticketing mechanism. This means that their model is actually unable to capture how tickets are generated by (honest) servers.

sensitive data. To this end, we design a *protected file storage* (PFS) system, which provides an interface for local encryption, decryption, and secure file shredding to a privacy-concerned user. The system is inspired by the internals of existing cloud storage services, but the final primitive is oblivious to the actual relationship between data owner and storage provider: in a PFS system, all trust lies with the holder of the secret key. This means that our system can cater both to users who wish to maintain control over the security of their data while offloading storage, *and* to storage providers who perform data encryption as a service.

The PFS interface is aimed at the former case, and hence hides internals of the system such as the key hierarchy to minimize the risk of involuntary misuse by an end user. However, it is still designed to support commonplace attributes of cloud storage systems, such as functionality for key rotation, as well as additionally providing fine-grained forward security for deleted files. This makes our approach conformable for use also by cloud service providers who wish to enhance the security guarantees in their existing systems.

5.1 PFS Syntax

We envision a PFS system to be utilized by a user who holds a set of (plaintext) files that they wish to protect and outsource the storage of. The user generates a local secret key sk via the setup algorithm $\mathsf{Setup}()$. They can then encrypt and decrypt files via algorithms $\mathsf{EncFile}$ and $\mathsf{DecFile}$, where encrypted files are associated with an identifier id, a header h, and a ciphertext C, of which the user stores h and C under the "filename" id at the storage service. (The user may keep a local look-up table mapping human-readable filenames to identifiers id, or decide to offload this table as yet another protected file to the storage service, too. In the latter case, the user only needs to store the identifier of the mapping file.) To shred a file, it suffices to locally run the algorithm $\mathsf{ShredFile}(sk, id)$ on the file identifier to be shredded. This will ensure that the corresponding file is irrecoverable (*forward secure*) from this point on; remote deletion at the service provider is not required to ensure its forward security. Finally, a user may rotate its secret key (e.g., for regulatory purposes or to refresh the key once its usage limit has been reached), which is done through calling a RotKey algorithm, taking the current list of file identifiers and headers as input and updating them with new headers to be replaced at the storage provider.

Definition 11 (PFS scheme). *A* protected file storage *scheme* $\mathsf{PFS} = (\mathsf{Setup}, \mathsf{EncFile}, \mathsf{DecFile}, \mathsf{ShredFile}, \mathsf{RotKey})$ *is a 5-tuple of algorithms with four associated sets; the secret key space* $\mathcal{SK}$*, the file space* $\mathcal{F}$*, the file identifier space* $\mathcal{I}$*, and the header space* $\mathcal{H}$*. Associated to the* PFS *is a ciphertext-length function* $\mathsf{cl}\colon \mathbb{N} \to \mathbb{N}$*.*

- *Via* $sk \leftarrow_{\$} \mathsf{Setup}()$*, the probabilistic setup algorithm* Setup*, taking no input, produces a secret key* $sk \in \mathcal{SK}$*.*
- *Via* $(id, h, C)/\bot \leftarrow_{\$} \mathsf{EncFile}(sk, F)$*, the randomized file encryption algorithm* $\mathsf{EncFile}$ *on input the secret key* $sk \in \mathcal{SK}$ *and a plaintext file* $F \in \mathcal{F}$ *produces a file identifier* $id \in \mathcal{I}$*, a header* $h \in \mathcal{H}$ *and a ciphertext* $C \in \{0,1\}^{\mathsf{cl}(|F|)}$ *or, to indicate failure,* $\bot$*.*

– *Via* $F/\bot \leftarrow \mathsf{DecFile}(sk, id, h, C)$*, the deterministic file decryption algorithm* $\mathsf{DecFile}$ *on input the key* $sk \in \mathcal{SK}$*, a file header* $h \in \mathcal{H}$*, and a ciphertext* $C \in \{0,1\}^*$ *returns a file plaintext* $F \in \mathcal{F}$ *or, to indicate failure,* $\bot$*.*
– *Via* $sk' \leftarrow \mathsf{ShredFile}(sk, id)$*, the deterministic file shredding algorithm* $\mathsf{ShredFile}$ *on input the secret key* $sk \in \mathcal{SK}$ *and a file identifier* $id \in \mathcal{I}$ *returns the updated secret key* $sk' \in \mathcal{SK}$*.*
– *Via* $(sk', (h'_1, \dots, h'_\ell))/(sk', \bot) \leftarrow\!\!{\scriptstyle\$}\ \mathsf{RotKey}(sk, ((id_1, h_1), \dots, (id_\ell, h_\ell)))$*, the randomized key-rotation algorithm* RotKey *on input the secret key* $sk \in \mathcal{SK}$ *and a list of file identifier-header pairs* $(id_1, h_1), \dots, (id_\ell, h_\ell) \in (\mathcal{I} \times \mathcal{H})^*$ *returns the potentially updated secret key* $sk' \in \mathcal{SK}$ *and a sequence of updated headers* $(h'_1, \dots, h'_\ell) \in \mathcal{H}^*$ *or, to indicate failure,* $\bot$*.*

5.2 Confidentiality and Integrity of PFS

A protected file storage scheme should provide confidentiality of the stored files, including their metadata (file identifiers and headers), as well as forward security when files have been shredded. Additionally, key rotation should allow the scheme to recover from corruption, ensuring security of newly encrypted files.

We capture this form of confidentiality through the notion of *forward indistinguishability from random bits* under *real and chosen-plaintext attack* (find\$-rcpa). In the find\$-rcpa security game, given in Figure 7, the adversary is asked to distinguish real from random outputs of a challenge real or \$ encryption oracle RO\$-ENC. We emphasize that indistinguishability here encompasses *both* the file ciphertext *and* metadata (i.e., identifier and header), encoding a strong form of *privacy.* The game further allows the adversary to shred files (via the oracle SHRED) and to rotate keys (via ROTKEY), leading to an update of the headers of all non-shredded files. We encode forward security via a CORR oracle, through which the adversary may ultimately learn the user's current secret key, provided that it shredded all challenge files (to prevent trivial distinguishing attacks) and does not make further challenge queries on that key. Furthermore, we allow new challenge queries *after* a successful key rotation, which captures security being regained after key rotation in which the adversary remained passive, a form of *post-compromise security* [21]. In order to capture potential leakage from unshredded files in the system which a real-world adversary would gain access to when corrupting a user's secret key, the game additionally includes a real encryption oracle ENC, which provides the adversary with honest encryptions of plaintexts of its choice that do not need to be shredded prior to corruption.

Definition 12 (PFS confidentiality (find\$-rcpa)**).** *Let* PFS *be a protected file storage scheme and* $\mathbf{G}^{\text{find\$-rcpa}}_{\mathsf{PFS}}$ *be the game defined in Figure 7. We define the advantage of an adversary* $\mathcal{A}$ *against the* find\$-rcpa *security of* PFS *as* $\mathbf{Adv}^{\text{find\$-rcpa}}_{\mathsf{PFS}}(\mathcal{A}) = 2\left|\Pr\left[\mathbf{G}^{\text{find\$-rcpa}}_{\mathsf{PFS}}(\mathcal{A}) \Rightarrow \text{true}\right] - \frac{1}{2}\right|$.

We also define integrity (of ciphertexts) for a PFS scheme, via the game in Figure 8. The adversary's goal here is to come up with a file tuple (id, h, C) that was not output by the encryption oracle ENC, or has been shredded (using

```
Game $\mathbf{G}^{\text{find\$-rcpa}}_{\mathsf{PFS}}(\mathcal{A})$:
    $b \leftarrow\$ \{0,1\}$; $sk \leftarrow\$ \mathsf{Setup}()$
    $R \leftarrow ()$; $Q \leftarrow ()$
    $\mathcal{S}_{\$id} \leftarrow \emptyset$; $\mathsf{corr} \leftarrow$ false
    $b^* \leftarrow\$ \mathcal{A}()$
    Return $b^* = b$

Ro$-Enc($F$):
    If $\mathsf{corr}=$true then return $\perp$
    $(id_1, h_1, C_1) \leftarrow\$ \mathsf{EncFile}(sk, F)$
    If $(id_1, h_1, C_1) = \perp$:
        Return $\perp$
    $id_0 \leftarrow\$ \mathcal{I}$; $h_0 \leftarrow\$ \mathcal{H}$
    $C_0 \leftarrow\$ \{0,1\}^{\mathsf{cl}(|F|)}$
    $R \mathrel{+}= (id_b, h_b)$
    $\mathcal{S}_{\$id} \xleftarrow{\cup} \{id_b\}$
    Return $(id_b, h_b, C_b)$

Enc($F$):
    $(id, h, C) \leftarrow\$ \mathsf{EncFile}(sk, F)$
    $Q \mathrel{+}= (id, h)$
    Return $(id, h, C)$
```

```
Shred($id$):
    $sk \leftarrow \mathsf{ShredFile}(sk, id)$
    $R \mathrel{-}= (id, *)$; $Q \mathrel{-}= (id, *)$; $\mathcal{S}_{\$id} \leftarrow \mathcal{S}_{\$id} \setminus \{id\}$

RotKey():
    $((id_1, h_1), \dots, (id_{|R|}, h_{|R|})) \leftarrow R$
    $((id_{|R|+1}, h_{|R|+1}), \dots, (id_{|R|+|Q|}, h_{|R|+|Q|})) \leftarrow Q$
    If $b = 0$:
        For $i = 1$ to $|R|$ do $h'_i \leftarrow\$ \mathcal{H}$
        $(sk, (h'_{|R|+1}, \dots, h'_{|R|+|Q|})) \leftarrow\$ \mathsf{RotKey}(sk, Q)$
        If $(h'_{|R|+1}, \dots, h'_{|R|+|Q|}) = \perp$ then return $\perp$
    If $b = 1$:
        $(sk, (h'_1, \dots, h'_{|R|+|Q|})) \leftarrow\$ \mathsf{RotKey}(sk, R \| Q)$
        If $(h'_1, \dots, h'_{|R|+|Q|}) = \perp$ then return $\perp$
    $R \leftarrow ((id_1, h'_1), \dots, (id_{|R|}, h'_{|R|}))$
    $Q \leftarrow ((id_{|R|+1}, h'_{|R|+1}), \dots, (id_{|R|+|Q|}, h'_{|R|+|Q|}))$
    $\mathsf{corr} \leftarrow$ false
    Return $R \| Q$

Corr():
    If $\mathcal{S}_{\$id} \neq \emptyset$ then return $\perp$
    $\mathsf{corr} \leftarrow$ true
    Return $sk$
```

Fig. 7. Confidentiality and forward security (find\$-rcpa) game for a protected file storage scheme PFS. Grey code prevents trivial attacks. Lists R and Q keep track of file identifiers and headers currently in the system for the sake of key rotation.

oracle Shred), yet successfully decrypts (in the decryption oracle Dec). The game further provides access to a key rotation oracle RotKey; in contrast to the find\$-rcpa game, this is strengthened to take adversarially-chosen file identifiers and headers as input. This captures that a malicious storage service might inject forged identifiers and headers into a user's storage or omit files from key rotation.

Definition 13 (PFS integrity (int-ctxt)). *Let* PFS *be a protected file storage scheme and* $\mathbf{G}^{\text{int-ctxt}}_{\mathsf{PFS}}$ *be the game defined in Figure 8. We define the advantage of an adversary* $\mathcal{A}$ *against the* int-ctxt *security of* PFS *as* $\mathbf{Adv}^{\text{int-ctxt}}_{\mathsf{PFS}}(\mathcal{A}) = \Pr\left[\mathbf{G}^{\text{int-ctxt}}_{\mathsf{PFS}}(\mathcal{A}) \Rightarrow \text{true}\right]$.

5.3 Instantiating PFS from PKW and AEAD

We now construct a generic PFS scheme PFS[PKW, AEAD] from a puncturable key-wrapping scheme PKW, which will handle the key management, and an authenticated encryption scheme with associated data AEAD, handling the actual file encryption. The construction, formalized in Figure 9, works as follows.

```
Game $\mathbf{G}^{\text{int-ctxt}}_{\mathsf{PFS}}(\mathcal{A})$:
  $sk \leftarrow\!\!\$\ \mathsf{Setup}()$
  $\mathcal{S} \leftarrow \emptyset$; $\mathsf{win} \leftarrow \mathsf{false}$
  $\mathcal{A}()$
  Return $\mathsf{win}$

$\textsc{Enc}(F)$:
  $(id, h, C) \leftarrow\!\!\$\ \mathsf{EncFile}(sk, F)$
  $\mathcal{S} \xleftarrow{\cup} \{(id, h, C)\}$
  Return $(id, h, C)$

$\textsc{Dec}(id, h, C)$:
  $F \leftarrow \mathsf{DecFile}(sk, id, h, C)$
  If $(id, h, C) \notin \mathcal{S}$ and $F \neq \perp$:
    $\mathsf{win} \leftarrow \mathsf{true}$
  Return $F$

$\textsc{Shred}(id)$:
  $sk \leftarrow \mathsf{ShredFile}(sk, id)$
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(id, *, *)\}$

$\textsc{RotKey}(((id_1, h_1), \ldots, (id_\ell, h_\ell)))$:
  $(sk, (h'_1, \ldots, h'_\ell)) \leftarrow\!\!\$\ \mathsf{RotKey}(sk, ((id_1, h_1), \ldots, (id_\ell, h_\ell)))$
  If $(h'_1, \ldots, h'_\ell) = \perp$ then return $\perp$
  $\mathcal{S}_{\mathsf{new}} \leftarrow \emptyset$
  For $(id, h, C) \in \mathcal{S}$ do:
    If $\exists i \in \{1, \ldots, \ell\}$ s.t. $(id, h) = (id_i, h_i)$:
      $\mathcal{S}_{\mathsf{new}} \xleftarrow{\cup} \{(id_i, h'_i, C)\}$
  $\mathcal{S} \leftarrow \mathcal{S}_{\mathsf{new}}$
  Return $((id_1, h'_1), \ldots, (id_\ell, h'_\ell))$
```

Fig. 8. Integrity of ciphertexts game for a protected file storage scheme PFS. Grey code prevents trivial attacks.

Setup generates a PKW key *sk*, which—for reference to cloud storage and its key-wrapping functionality—we refer to as the key encryption key (KEK).

EncFile first samples an AEAD "data encryption key" (DEK) and a file identifier *id* at random, and wraps DEK under the KEK into a file header *h*, using *id* as tag.[6] It then AEAD-encrypts the file plaintext under DEK and a random[7] nonce N into a ciphertext C; $N \| C$ constitutes the PFS file ciphertext.

DecFile inverts file encryption by first unwrapping the DEK from the header and then using it to decrypt the file ciphertext.

ShredFile punctures the KEK *sk* on a file identifier *id*, using the PKW puncturing algorithm. This effectively prevents future unwrapping of the DEK wrapped with tag *id*, and hence file decryptions of files with this identifier.

RotKey first unwraps the DEKs in all headers it is handed, then samples a fresh KEK to re-wrap them. The PKW tags are re-used in this process, ensuring that encrypted files keep their identifiers across key rotations.

For PFS[PKW, AEAD], we establish confidentiality in Theorems 14 and 15 (below) and integrity (in the full version [5]). Notably, our two confidentiality results follow different paths: Theorem 14 employs weak one-time (find\$-1cpa) PKW security in a hybrid together with puncture invariance and consistency. Theorem 15 in contrast shows our construction achieves the same goal in a tight manner if the underlying PKW scheme meets the stronger find\$-rcpa notion.

[6] Our construction leaves the PKW header empty. In practice, this field may be used to authenticate control data of the DEK, such as expiration date or permitted usage.

[7] Our construction only uses a single AEAD nonce N per any one data encryption key DEK, which would allow using a fixed nonce. We still sample a random nonce to enable file updates/re-encryption as a potential extension to our construction.

```
Setup():
  $sk \leftarrow\$ \mathsf{PKW.KeyGen}()$
  Return $sk$

EncFile($sk, F$):
  $K \leftarrow\$ \{0,1\}^k$; $id \leftarrow\$ \{0,1\}^t$
  $h \leftarrow \mathsf{PKW.Wrap}(sk, id, \varepsilon, K)$
  If $h = \perp$ then return $\perp$
  $N \leftarrow\$ \{0,1\}^n$
  $C \leftarrow \mathsf{AEAD.Enc}(K, N, \varepsilon, F)$
  Return $(id, h, N \| C)$

DecFile($sk, id, h, N \| C$):
  $K \leftarrow \mathsf{PKW.Unwrap}(sk, id, \varepsilon, h)$
  $F \leftarrow \mathsf{AEAD.Dec}(K, N, \varepsilon, C)$
  Return $F$

ShredFile($sk, id$):
  $sk' \leftarrow \mathsf{PKW.Punc}(sk, id)$
  Return $sk'$

RotKey($sk_{\mathsf{old}}, (id_1, h_1), \ldots, (id_\ell, h_\ell)$):
  $sk_{\mathsf{new}} \leftarrow\$ \mathsf{PKW.KeyGen}()$
  For $i = 1$ to $\ell$ do:
    $K_i \leftarrow \mathsf{PKW.Unwrap}(sk_{\mathsf{old}}, id_i, \varepsilon, h_i)$
    $h'_i \leftarrow \mathsf{PKW.Wrap}(sk_{\mathsf{new}}, id_i, \varepsilon, K_i)$
    If $h'_i = \perp$ then return $(sk_{\mathsf{old}}, \perp)$
  Return $(sk_{\mathsf{new}}, (id_1, h'_1), \ldots, (id_\ell, h'_\ell))$
```

Fig. 9. Construction of a protected file storage scheme $\mathsf{PFS[PKW, AEAD]}$ from a puncturable key-wrapping scheme PKW and an AEAD scheme AEAD. The PKW scheme has wrap-key space $\{0,1\}^k$ and tag space $\{0,1\}^t$. The AEAD scheme has key space $\{0,1\}^k$ and nonce space $\{0,1\}^n$. Hence, for the resulting PFS scheme, $\mathcal{I} = \{0,1\}^t$, $\mathcal{H} = \{0,1\}^{\mathsf{PKW.cl}(k)}$, and $\mathsf{PFS.cl}(|F|) = n + \mathsf{AEAD.cl}(|F|)$.

While the latter notion is currently only known to be achievable from strong (fpr-rro\$) PPRF security, the route of Theorem 15 may still be interesting as it does not require puncture invariance and consistency, properties which we expect schemes with non-perfect correctness (e.g., employing Bloom filters), would not achieve. We only give one proof sketch here and provide the full proofs in [5].

Theorem 14 **(**$\mathsf{PFS[PKW, AEAD]}$ **is** find\$-rcpa **secure, via PKW** find\$-1cpa**).** *Let* $\mathsf{PFS[PKW, AEAD]}$ *be the PFS construction in Figure 9 with file identifier space* $\mathcal{I} = \{0,1\}^t$*. If* PKW *is puncture invariant and consistent (Definitions 5 and 6), then for every adversary* $\mathcal{A}$ *against the* find\$-rcpa *security (Definition 12) of* $\mathsf{PFS[PKW, AEAD]}$ *making at most* $q_{\mathrm{ro\$}}$*,* q_e*, resp.* $m-1$ *queries in total to its oracles* Ro\$-Enc*,* Enc*, and* RotKey*, and at most* q_s *queries to oracle* Shred *between each query to the key rotation oracle* RotKey*, there exists adversaries* $\mathcal{B}_{\mathsf{pkw}}$ *and* $\mathcal{B}_{\mathsf{aead}}$ *running in approximately the same time as* $\mathcal{A}$ *such that* $\mathbf{Adv}^{\text{find\$-rcpa}}_{\mathsf{PFS[PKW,AEAD]}}(\mathcal{A}) \leq 2q_{\mathrm{ro\$}}\Big(\frac{(2q_s+q_e+q_{ro\$}-1)}{2^t} + m \cdot \mathbf{Adv}^{\text{find\$-1cpa}}_{\mathsf{PKW}}(\mathcal{B}_{\mathsf{pkw}}) + m \cdot \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}})\Big)$*. Adversary* $\mathcal{B}_{\mathsf{pkw}}$ *makes at most* m *queries to oracle* New-Ro\$-Wrap*. Adversary* $\mathcal{B}_{\mathsf{aead}}$ *makes one query each to its oracles* New *and* Ro\$*.*

Proof idea. The proof proceeds by a series of six game hops, starting with game $\mathbf{G}_0 = \mathbf{G}^{\text{find\$-rcpa}}_{\mathsf{PFS[PKW,AEAD]}}$. Let $\mathbf{Adv}_i(\mathcal{A}) := 2\left|\Pr[\,\mathbf{G}_i(\mathcal{A})\,] - \frac{1}{2}\right|$ for $i \in \{0, \ldots, 6\}$. By *key phase* we denote the period between two consecutive key rotation queries.

$\mathbf{G}_0 \to \mathbf{G}_1$: We begin by excluding, via a bad event [8], that the (real- or ideal-world) challenge file identifier coincides with one already shredded in the current key phase, since the output of wrapping with such an identifier as tag is undefined and hence possibly distinguishable from the ideal-world behavior. The probability of this happening is upper-bounded by $2q_{ro\$} \cdot \frac{q_s}{2^t}$.

$\mathbf{G}_1 \to \mathbf{G}_2$: We reduce the $q_{ro\$}$ Ro\$-Enc challenge queries to a single one via a hybrid argument, yielding an adversary $\mathcal{A}'$ making a single query to Ro\$-Enc and at most $q_e + q_{ro\$} - 1$ queries to Enc, such that $\mathbf{Adv}_1(\mathcal{A}) = q_{ro\$} \cdot \mathbf{Adv}_2(\mathcal{A}')$.

$\mathbf{G}_2 \to \mathbf{G}_3$: Next, we exclude that PKW tags used for the (at most $q_e + q_{ro\$} - 1$) real encryption queries prior to the challenge query collide with the (single) challenge tag, a bad event occurring with probability at most $\frac{q_e + q_{ro\$} - 1}{2^t}$.

$\mathbf{G}_3 \to \mathbf{G}_4$: The challenger now guesses in which of the at most m key phases the challenge encryption occurs; silencing the output otherwise loses a factor of m.

$\mathbf{G}_4 \to \mathbf{G}_5$: We can now apply the find\$-1cpa security of PKW through a reduction $\mathcal{B}_{\mathsf{pkw}}$ to replace the header in the challenge encryption by a random string. This step requires PKW's puncture invariance and consistency to reorder the challenge PKW wrap in the reduction; the latter makes at most m queries to oracle New-Ro\$-Wrap and yields $|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_5]| \leq \mathbf{Adv}^{\text{find\$-1cpa}}_{\mathsf{PKW}}(\mathcal{B}_{\mathsf{pkw}})$.

$\mathbf{G}_5 \to \mathbf{G}_6$: Finally, we replace the challenge file ciphertext with a random string via a reduction $\mathcal{B}_{\mathsf{aead}}$ to the AEAD scheme's ind\$-cpa security, which yields $|\Pr[\mathbf{G}_5] - \Pr[\mathbf{G}_6]| \leq \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}})$. After this step, $\mathbf{Adv}_6(\mathcal{A}) = 0$. □

Theorem 15 ($\mathsf{PFS[PKW, AEAD]}$ is find\$-rcpa secure, via PKW find\$-rcpa). *Let $\mathsf{PFS[PKW, AEAD]}$ be the PFS construction in Figure 9 with file identifier space $\mathcal{I} = \{0,1\}^t$. For every adversary $\mathcal{A}$ against the find\$-rcpa security (Definition 12) of $\mathsf{PFS[PKW, AEAD]}$ making at most $q_{ro\$}$, q_e, q_{corr}, resp. q_{rk} queries in total to its oracles Ro\$-Enc, Enc, Corr and RotKey, and at most q_s queries to oracle Shred between each query to oracle RotKey, there exists adversaries $\mathcal{B}_{\mathsf{pkw}}$ and $\mathcal{B}_{\mathsf{aead}}$ running in approximately the same time as $\mathcal{A}$ such that $\mathbf{Adv}^{\text{find\$-rcpa}}_{\mathsf{PFS[PKW,AEAD]}}(\mathcal{A}) \leq 2 \cdot \left(\frac{2q_{ro\$}q_s}{2^t} + \frac{(q_e+q_{ro\$})^2}{2^{t+1}} + \mathbf{Adv}^{\text{find\$-rcpa}}_{\mathsf{PKW}}(\mathcal{B}_{\mathsf{pkw}}) + \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{aead}})\right)$. Adversary $\mathcal{B}_{\mathsf{pkw}}$ makes at most $q_{rk}+1$, $q_{ro\$}(q_{rk}+1)$, $q_e(q_{rk}+1)$, q_{corr} and $q_{rk} \cdot q_s$ queries to oracles New, Ro\$-Wrap, Wrap, Corr and Punc, respectively. Adversary $\mathcal{B}_{\mathsf{aead}}$ makes at most $q_{ro\$}$ queries each to its oracles New and Ro\$.*

6 Discussion and Future Work

Our approach to PKW integrates a flexible tag-based approach [31] with classical key wrapping [47]. We build PKW generically from PPRF and AEAD, focusing on applications which require fine-grained forward security. For applications where batch puncturing might be useful, deploying nonce-misuse resistant AEAD would allow tags to be reused, achieving a stronger version of our main find\$-cpa security notion. Interestingly, proving the (even stronger) find\$-rcpa security of such an instantiation runs into a key commitment problem; whether resolving this needs idealized models (cf. [35]) or can be done in the standard model is an interesting open problem.

Our PKWs and the PPRFs they are built from are not private [11]; we could potentially obtain improved privacy after client compromise for our PFS system if they were, cf. [54]. Finding practically efficient private PPRFs and building private PKW schemes from them is an open problem whose solution would have immediate applications.

Our work on TLS session resumption assumes the server's key is held and operated on by a single server. Yet distributed server environments are common in TLS deployments, to reduce latency and improve scalability. It would be useful to extend our work to this setting. The challenge is to maintain appropriate synchronization amongst the punctured keys held by the servers.

The applications we treat in this work are a sample from the set of possible use-cases for PKW. They already demonstrate that it is a useful abstraction. Examining further potential applications where puncturable key wrapping can be integrated, such as in symmetric key exchange [15] and DUKPT [18], would be interesting future work.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. Felix Günther has been supported in part by Research Fellowship grant GU 1859/1-1 of the German Research Foundation (DFG).

References

1. Aviram, N., Gellert, K., Jager, T.: Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 117–150. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17656-3_5
2. Aviram, N., Gellert, K., Jager, T.: Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. Journal of Cryptology **34**(3), 20 (Jul 2021). https://doi.org/10.1007/s00145-021-09385-0
3. Avoine, G., Canard, S., Ferreira, L.: Symmetric-key authenticated key exchange (SAKE) with perfect forward secrecy. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 199–224. Springer, Heidelberg (Feb 2020). https://doi.org/10.1007/978-3-030-40186-3_10
4. AWS: Protecting data using client-side encryption. http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingClientSideEncryption.html
5. Backendal, M., Günther, F., Paterson, K.G.: Puncturable key wrapping and its applications. Cryptology ePrint Archive, Paper 2022/1209 (2022), https://eprint.iacr.org/2022/1209
6. Bellare, M., Boldyreva, A., Micali, S.: Public-key encryption in a multi-user setting: Security proofs and improvements. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 259–274. Springer, Heidelberg (May 2000). https://doi.org/10.1007/3-540-45539-6_18
7. Bellare, M., Ng, R., Tackmann, B.: Nonces are noticed: AEAD revisited. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part I. LNCS, vol. 11692, pp. 235–265. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26948-7_9

8. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (May / Jun 2006). https://doi.org/10.1007/11761679_25
9. Bellare, M., Stepanovs, I., Waters, B.: New negative results on differing-inputs obfuscation. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 792–821. Springer, Heidelberg (May 2016). https://doi.org/10.1007/978-3-662-49896-5_28
10. Blaze, M.: A cryptographic file system for UNIX. In: Denning, D.E., Pyle, R., Ganesan, R., Sandhu, R.S., Ashby, V. (eds.) ACM CCS 93. pp. 9–16. ACM Press (Nov 1993). https://doi.org/10.1145/168588.168590
11. Boneh, D., Lewi, K., Wu, D.J.: Constraining pseudorandom functions privately. In: Fehr, S. (ed.) PKC 2017, Part II. LNCS, vol. 10175, pp. 494–524. Springer, Heidelberg (Mar 2017). https://doi.org/10.1007/978-3-662-54388-7_17
12. Boneh, D., Lipton, R.J.: A revocable backup system. In: USENIX Security 96. USENIX Association (Jul 1996)
13. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 280–300. Springer, Heidelberg (Dec 2013). https://doi.org/10.1007/978-3-642-42045-0_15
14. Boxcryptor: Boxcryptor security for your cloud. https://www.boxcryptor.com/
15. Boyd, C., Davies, G.T., de Kock, B., Gellert, K., Jager, T., Millerjord, L.: Symmetric key exchange with full forward security and robust synchronization. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part IV. LNCS, vol. 13093, pp. 681–710. Springer, Heidelberg (Dec 2021). https://doi.org/10.1007/978-3-030-92068-5_23
16. Boyd, C., Gellert, K.: A modern view on forward security. Comput. J. **64**(4), 639–652 (2021). https://doi.org/10.1093/comjnl/bxaa104
17. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519. Springer, Heidelberg (Mar 2014). https://doi.org/10.1007/978-3-642-54631-0_29
18. Brier, E., Peyrin, T.: A forward-secure symmetric-key derivation protocol - how to improve classical DUKPT. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 250–267. Springer, Heidelberg (Dec 2010). https://doi.org/10.1007/978-3-642-17373-8_15
19. Chen, W., Hoang, T., Guajardo, J., Yavuz, A.A.: Titanium: A metadata-hiding file-sharing system with malicious security. In: NDSS 2022. The Internet Society (2022). https://doi.org/10.14722/ndss.2022.24161
20. Chen, W., Popa, R.A.: Metal: A metadata-hiding file-sharing system. In: NDSS 2020. The Internet Society (Feb 2020)
21. Cohn-Gordon, K., Cremers, C.J.F., Garratt, L.: On post-compromise security. In: Hicks, M., Köpf, B. (eds.) CSF 2016 Computer Security Foundations Symposium. pp. 164–178. IEEE Computer Society Press (2016). https://doi.org/10.1109/CSF.2016.19
22. Derler, D., Jager, T., Slamanig, D., Striecks, C.: Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 425–455. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_14
23. Diffie, W., van Oorschot, P.C., Wiener, M.J.: Authentication and authenticated key exchanges. Designs, Codes and Cryptography **2**(2), 107–125 (Jun 1992)

24. Dworkin, M.: Recommendation for block cipher modes of operation: Methods for key wrapping. NIST Special Publication SP 800-38F (2012), https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf
25. Everspaugh, A., Paterson, K.G., Ristenpart, T., Scott, S.: Key rotation for authenticated encryption. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part III. LNCS, vol. 10403, pp. 98–129. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63697-9_4
26. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: Sporc: Group collaboration using untrusted cloud resources. In: OSDI 20210 (2010)
27. Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google's QUIC protocol. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014. pp. 1193–1204. ACM Press (Nov 2014). https://doi.org/10.1145/2660267.2660308
28. Fuchsbauer, G., Konstantinov, M., Pietrzak, K., Rao, V.: Adaptive security of constrained PRFs. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 82–101. Springer, Heidelberg (Dec 2014). https://doi.org/10.1007/978-3-662-45608-8_5
29. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: 25th FOCS. pp. 464–479. IEEE Computer Society Press (Oct 1984). https://doi.org/10.1109/SFCS.1984.715949
30. Google: Encryption at rest in Google Cloud. https://cloud.google.com/security/encryption/default-encryption
31. Green, M.D., Miers, I.: Forward secure asynchronous messaging from puncturable encryption. In: 2015 IEEE Symposium on Security and Privacy. pp. 305–320. IEEE Computer Society Press (May 2015). https://doi.org/10.1109/SP.2015.26
32. Günther, C.G.: An identity-based key-exchange protocol. In: Quisquater, J.J., Vandewalle, J. (eds.) EUROCRYPT'89. LNCS, vol. 434, pp. 29–37. Springer, Heidelberg (Apr 1990). https://doi.org/10.1007/3-540-46885-4_5
33. Günther, F., Hale, B., Jager, T., Lauer, S.: 0-RTT key exchange with full forward secrecy. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 519–548. Springer, Heidelberg (Apr / May 2017). https://doi.org/10.1007/978-3-319-56617-7_18
34. IBM: Protecting data with envelope encryption. https://cloud.ibm.com/docs/key-protect?topic=key-protect-envelope-encryption
35. Jaeger, J., Tyagi, N.: Handling adaptive compromise for practical encryption schemes. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 3–32. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56784-2_1
36. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013). https://doi.org/10.1145/2508859.2516668
37. Klooß, M., Lehmann, A., Rupp, A.: (R)CCA secure updatable encryption with integrity protection. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 68–99. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2_3
38. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34
39. Lau, B., Chung, S.P., Song, C., Jang, Y., Lee, W., Boldyreva, A.: Mimesis aegis: A mimicry privacy shield-A system's approach to data privacy on public cloud. In:

Fu, K., Jung, J. (eds.) USENIX Security 2014. pp. 33–48. USENIX Association (Aug 2014)

40. Lehmann, A., Tackmann, B.: Updatable encryption with post-compromise security. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 685–716. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_22
41. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud storage with minimal trust. ACM Trans. Comput. Syst. **29**(4) (dec 2011). https://doi.org/10.1145/2063509.2063512
42. Microsoft: Azure Data Encryption at rest. https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-atrest
43. Miller, E., Long, D., Freeman, W., Reed, B.: Strong security for distributed file systems. In: Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference. pp. 34–40 (2001). https://doi.org/10.1109/IPCCC.2001.918633
44. Nichols, S.: Dropbox: Oops, yeah, we didn't actually delete all your files this bug kept them in the cloud. https://www.theregister.com/2017/01/24/dropbox_brings_old_files_back_from_dead/ (2017)
45. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard) (Aug 2018), https://www.rfc-editor.org/rfc/rfc8446.txt
46. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) ACM CCS 2002. pp. 98–107. ACM Press (Nov 2002). https://doi.org/10.1145/586110.586125
47. Rogaway, P., Shrimpton, T.: A provable-security treatment of the key-wrap problem. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 373–390. Springer, Heidelberg (May / Jun 2006). https://doi.org/10.1007/11761679_23
48. Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: Shmoys, D.B. (ed.) 46th ACM STOC. pp. 475–484. ACM Press (May / Jun 2014). https://doi.org/10.1145/2591796.2591825
49. Salowey, J., Zhou, H., Eronen, P., Tschofenig, H.: Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard) (Jan 2008), https://www.rfc-editor.org/rfc/rfc5077.txt, obsoleted by RFC 8446, updated by RFC 8447
50. Slamanig, D., Striecks, C.: Puncture 'em all: Updatable encryption with no-directional key updates and expiring ciphertexts. Cryptology ePrint Archive, Report 2021/268 (2021), https://eprint.iacr.org/2021/268
51. Sun, S.F., Steinfeld, R., Lai, S., Yuan, X., Sakzad, A., Liu, J.K., Nepal, S., Gu, D.: Practical non-interactive searchable encryption with forward and backward privacy. In: NDSS 2021. The Internet Society (Feb 2021)
52. Sun, S., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 763–780. ACM Press (Oct 2018). https://doi.org/10.1145/3243734.3243782
53. Sy, E., Burkert, C., Federrath, H., Fischer, M.: Tracking users across the web via TLS session resumption. In: ACSAC 2018. pp. 289–299. ACM (2018). https://doi.org/10.1145/3274694.3274708
54. Tyagi, N., Mughees, M.H., Ristenpart, T., Miers, I.: BurnBox: Self-revocable encryption in a world of compelled access. In: Enck, W., Felt, A.P. (eds.) USENIX Security 2018. pp. 445–461. USENIX Association (Aug 2018)