Session Resumption Protocols and Efficient Forward Security for TLS 1.3 0-RTT

Nimrod Aviram[1], Kai Gellert[2], and Tibor Jager[2]

[1] Tel Aviv University, nimrodav@mail.tau.ac.il
[2] Paderborn University,
{kai.gellert, tibor.jager}@uni-paderborn.de

Abstract. The TLS 1.3 0-RTT mode enables a client reconnecting to a server to send encrypted application-layer data in "0-RTT" ("zero round-trip time"), without the need for a prior interactive handshake. This fundamentally requires the server to reconstruct the previous session's encryption secrets upon receipt of the client's first message. The standard techniques to achieve this are *Session Caches* or, alternatively, *Session Tickets*. The former provides forward security and resistance against replay attacks, but requires a large amount of server-side storage. The latter requires negligible storage, but provides no forward security and is known to be vulnerable to replay attacks.

In this paper, we first formally define *session resumption protocols* as an abstract perspective on mechanisms like Session Caches and Session Tickets. We give a new generic construction that provably provides forward security and replay resilience, based on puncturable pseudorandom functions (PPRFs). This construction can immediately be used in TLS 1.3 0-RTT and deployed unilaterally by servers, without requiring any changes to clients or the protocol.

We then describe two new constructions of PPRFs, which are particularly suitable for use for forward-secure and replay-resilient session resumption in TLS 1.3. The first construction is based on the strong RSA assumption. Compared to standard Session Caches, for "128-bit security" it reduces the required server storage by a factor of almost 20, when instantiated in a way such that key derivation and puncturing together are cheaper on average than one full exponentiation in an RSA group. Hence, a 1 GB Session Cache can be replaced with only about 51 MBs of storage, which significantly reduces the amount of secure memory required. For larger security parameters or in exchange for more expensive computations, even larger storage reductions are achieved. The second construction combines a standard binary tree PPRF with a new "domain extension" technique. For a reasonable choice of parameters, this reduces the required storage by a factor of up to 5 compared to a standard Session Cache. It employs only symmetric cryptography, is suitable for high-traffic scenarios, and can serve thousands of tickets per second.

Supported by the German Research Foundation (DFG), project JA 2445/2-1, scholarships from The Israeli Ministry of Science and Technology, The Check Point Institute for Information Security, and The Yitzhak and Chaya Weinstein Research Institute for Signal Processing. We thank Nick Sullivan, Sven N. Hebrok and all anonymous reviewers for their valuable comments.

1 Introduction

0-RTT Protocols. A major innovation of TLS 1.3 [39] is its *0-RTT* (zero round-trip time) mode, which enables the resumption of sessions with minimal latency and without the need for an interactive handshake. A 0-RTT protocol allows the establishment of a secure connection in "one-shot", that is, with a single message sent from a client to a server, such that cryptographically protected payload data can be sent immediately ("in 0-RTT") along with the key establishment message, without the need for a latency-incurring prior handshake protocol. This significant speedup of connection establishment yields a smoother Web browsing experience and, more generally, better performance for applications with low-latency requirements. This is particularly noticeable in networks with relatively high latency, such as mobile networks.

The huge practical demand for 0-RTT is exemplified by the fact that many large Internet companies have developed and experimented with such protocols in the recent past, for example Google's QUIC [13] and Facebook's Zero [27] protocols. The content distribution provider Cloudflare has deployed the 0-RTT mode of TLS 1.3 as early as March 2017 at large scale, long before the finalization of the standard [44]. Google and Facebook declared that the cryptography in QUIC and Zero will soon be replaced by TLS 1.3 0-RTT [4,27].

The TLS 1.3 0-RTT Handshake. A full TLS 1.3 handshake (not 0-RTT) is always used in the very first connection between a client and a server. If the server supports 0-RTT, then both the client and server can derive a *Resumption Secret* from their shared key and session parameters. The client will simply store this secret. Naturally, the server then needs to retrieve the Resumption Secret during a subsequent handshake. There are two standard approaches for this, *Session Caches* and *Session Tickets*, which have different advantages and drawbacks. During the first handshake, the server sends to the client either a lookup key pointing to an entry in the Session Cache of the server, or a Session Ticket - depending on the configuration of the server. These approaches essentially work as follows:

Session Caches: The server stores all Resumption Secrets of recent sessions in a local database and issues each client a unique lookup key. When a client reconnects, it includes that lookup key in its 0-RTT messages, enabling the server to retrieve and use the matching Resumption Secret.

Session Tickets: The server uses a long-term symmetric encryption key, called the *Session Ticket Encryption Key* (STEK). Instead of storing the Resumption Secret in a local database, the server encrypts it with the STEK to create a *Session Ticket*. The Session Ticket is stored by the client. When a client reconnects, it includes that Session Ticket in its 0-RTT messages, which enables the server to decrypt it and recover the Resumption Secret. Note that the same STEK is used for many sessions and clients.

On a subsequent 0-RTT handshake, the client will include in its first message either the lookup key or the encrypted Session Ticket, in addition to a Diffie-Hellman key exchange message. The client can also send, in the same message, encrypted application-layer data, termed 0-RTT data. This data will be encrypted with a key derived from

the Resumption Secret and a public client random value, without any input from the server.[3]

In its reply, the server will typically include a Diffie-Hellman key exchange message, and further messages (in either direction) will be encrypted with a key derived also from the DH secret, not only the Resumption Secret. Hence, the only data protected by the Resumption Secret alone is the 0-RTT data. We note that the use of DH is not mandatory, and it is possible to rely only on the Resumption Secret for the security of the entire session; we expect most traffic will use DH as described above.

We stress that the use of Session Caches or Session Tickets is opaque to clients. That is, in either case the server sends a *New Session Ticket* message containing an opaque sequence of bytes, which may be either a lookup key for the Session Cache, or an encrypted Session Ticket, without specifying which is the case.[4] This property ensures that our proposed techniques are compatible with the final TLS 1.3 standard [39] and can be implemented on the server-side without requiring modifications to the protocol or to clients.

Forward Security and Replay Resilience of 0-RTT Protocols. Forward security essentially means that the protocol provides security of sessions, even if an attacker is able to corrupt one party after the session has terminated (e.g., by breaking into a Web server and learning the long-term secret key). Resilience to replay attacks is a fundamental, classical design goal of cryptographic protocols, which prevents an attacker from replaying the same payload data to a server repeatedly.

Both forward security and replay resilience are standard design goals of modern security protocols. However, achieving these properties is well-known to be difficult for 0-RTT protocols. This is because classical ("non-0-RTT") protocols include fresh input from the server (e.g., a Diffie-Hellman message) generated using ephemeral randomness, which provides a leverage to achieve forward security. However, there is no such interactivity in 0-RTT protocols. Furthermore, an attacker is able to replay the 0-RTT key establishment message along with the 0-RTT payload data over and over again to a server, which is not detectable without additional server-side countermeasures.

Forward Security and Replay Resilience of TLS 1.3 0-RTT. With Session Caches the server stores a "unique" Resumption Secret in a local database for each client. In most cases, it is able to delete the Resumption Secret immediately after retrieving it. This provides forward security, as an attacker obtaining the server state cannot decrypt past sessions. It also provides resilience against replay attacks, as the server is not able to decrypt replayed messages.

[3] The above describes typical modes of operation of TLS 1.3. The standard also allows for other modes, e.g. modes that include client authentication. We expect other modes will be used much less often, and therefore they are beyond the scope of this paper.

[4] Confusingly, the message containing this opaque sequence of bytes is always termed a "New Session Ticket Message", for both Session Caches and encrypted self-contained Session Tickets. To our knowledge there is no standard nomenclature, in [39] or elsewhere, for these two different approaches when used in TLS 1.3; see e.g. [39, §8.1]. TLS 1.2 referred to "Session ID Resumption" and "Session Ticket Resumption", but these terms are not used in TLS 1.3.

If Session Tickets are used, then an attacker that obtains access to the server can learn the STEK, and thus decrypt all tickets encrypted with this key to learn the Resumption Keys. Hence, servers using Session Tickets do not provide forward security. They are also generally vulnerable to replay attacks.[5] Since an attacker learning the STEK has catastrophic implications for security, large server operators usually rotate the STEK. Such deployments typically generate a new STEK roughly once per hour, and limit the STEK lifetime to roughly a day [34]. An attacker that learns one STEK can therefore decrypt approximately one hour's worth of traffic. However, most current TLS implementations do not provide out-of-the-box support for STEK rotation, and this (welcome) defensive measure is usually limited to large operators who can afford to modify TLS implementations [32,34]. Long-lived STEKs are unfortunately prevalent, and even among high-profile websites, some reuse the same STEK for many weeks, or even for many months [43].

To summarize, Session Caches are generally forward-secure and replay-resistant, while Session Tickets are not. Naïvely, it would therefore appear that Session Caches are the superior solution. However, Session Caches require the server to store the session state for each (recent) connection. This is often infeasible, in particular for high-traffic server operators. Such server operators often reluctantly use Session Tickets, knowingly forgoing forward secrecy. Additionally, even if forward security is not prioritized by a particular server operator and thus Session Tickets are used, the prevention of replay attacks may still require additional storage at the server, since the only way to prevent replay attacks in this case is to log used tickets.[6] In this context it is sometimes claimed that so-called *idempotent requests*, that is, requests that have the same effect on the server state whether they are served once or several times, are safe to use with TLS 1.3 0-RTT. However, it is well-known [36] and also discussed in the TLS 1.3 specification [39] that even replays of idempotent requests may give rise to attacks that, e.g., reveal the target URL of HTTP requests.

All of these issues are well-known to apply to TLS 1.3 0-RTT and have raised significant concerns about its secure deployability in practice [36]. Eric Rescorla, the main author of the TLS 1.3 RFC draft, acknowledges that this poses a *"difficult application integration issue"* [38]. However, due to the huge practical demand, 0-RTT is also considered *"too big a win not to do"* [38]. Very recently, at EUROCRYPT 2017 [25] and 2018 [15,16], the first 0-RTT protocols that simultaneously achieve forward security and replay resilience were proposed, but these require relatively heavy cryptographic machinery, such as hierarchical or broadcast identity-based encryption, and thus are not yet suitable for large-scale deployment in TLS 1.3.

Our Contributions. We give the first formal definition for secure 0-RTT session resumption protocols, as an abstraction of the constructions currently used in practice in

[5] Unless there is additional server-side logging of tickets that have already been used.

[6] When using resumption, the client must include in its first message the ticket's age, i.e. the time elapsed between receiving the ticket from the server in a previous session. The server expects this time interval to be precise up to a small window of error allowing for propagation delay, typically on the order of 10 seconds. An attacker can perform replay attacks within this time window.

TLS 1.3. We propose new techniques to achieve forward security and replay resilience that are ready-to-use with TLS 1.3 as it is standardized, without any changes to the protocol. Our proposal is based on Session Tickets, and thus requires minimal storage at the server side, but we extend this approach with efficient puncturable pseudorandom functions (PPRFs) that enable us to achieve forward security and replay resilience for Session Tickets. We provide new constructions of PPRFs with short keys and formal security proofs based on standard hardness assumptions. We propose two variants:

1. The first variant is based on the strong RSA assumption. It reduces the server storage by a factor of at least 11 compared to a Session Cache, increases ticket size by a negligible length, and requires the server to perform two exponentiations (one per issuance and one per resumption).
2. The second variant reduces server storage by a factor of up to 5 compared to a Session Cache, while using tickets that are roughly 400 bytes longer than standard tickets. It extends a standard GGM-style [22] binary tree-based PPRF, as described in [10,11,30], with a new *domain extension* idea. It employs only symmetric cryptography, is suitable for very-high-traffic scenarios, and can serve thousands of tickets per second, at the cost of hundreds of megabytes in server storage.

Our Approach. At the base of our approach is the concept of *puncturing* a pseudorandom function (PRF) to obtain a puncturable symmetric-key encryption scheme. Puncturable PRFs are a special case of constrained PRFs [10,11,30], which make it possible to derive constrained keys that allow computation of PRF output only for certain inputs.

In our approach, a server initially maintains a STEK k that allows decryption of any Session Ticket; when receiving ticket t, the server uses k to decrypt t in order to recover the Resumption Secret. Using the puncturing feature of the PPRF, it then derives from k a new key, k', that can decrypt any ticket *except* for t. The server then discards k and stores only k'. It repeats this process for every ticket received. This yields forward secrecy and replay-resistance: an attacker that compromises the server learns a key that is not capable of decrypting past tickets. Similarly, an attacker cannot successfully replay a message, since the server is only able to decrypt each ticket once.

The naïve way to employ this approach in TLS 1.3 0-RTT would be to use public-key puncturable encryption, as in [15,16,25]. However, this approach results in impractically long puncturing times or very long secret keys. Moreover, the most practical constructions require relatively expensive pairing-based cryptography by both the client and the server, thereby obviating a significant benefit of TLS 1.3 0-RTT. Rather than using public-key puncturable encryption, we observe that in TLS 1.3 0-RTT, the server itself generates the tickets it would later need to decrypt. It therefore suffices to use symmetric cryptography, and to maintain a key that allows decryption of only a limited set of ciphertexts, generated by the server itself. To achieve this, we use PPRFs to derive keys for standard TLS 1.3 tickets. Concretely, we describe two new PPRF constructions that are particularly suitable for our application:

– The first builds a new PPRF from the Strong RSA Assumption. The PPRF has a polynomially-bounded input size, but this is sufficient for our application (and probably for certain other PPRF applications as well). Its main distinguishing feature is that its secret key size is independent of the number of puncturings. It con-

sists of an RSA modulus N, a number $g \in \mathbb{Z}_N$, and a bitfield, indicating positions where the PPRF was punctured. Due to the short secret key, our construction may find other applications in applied and theoretical cryptography. Since our primary objective is to provide an as-efficient-as-possible solution for practical protocols such as TLS 1.3 0-RTT, we describe a construction with security proof in the random oracle model [5]. It seems likely that our construction can be lifted to the standard model in a straightforward way, via standard techniques like hardcore predicates [6,8,23], but this would yield less efficient constructions and is therefore outside the scope of this paper.

– The second construction is based on a standard tree-based PPRF [10,11,30], instantiated with a cryptographic hash function, such as SHA-3.
  The size of punctured keys depends linearly on the depth of the tree, which in turn depends on the size of the domain of the PPRF. We describe a new *domain extension* technique that reduces the size of punctured keys by trading secret key size for ticket size, while preserving the puncturing functionality. Domain extension makes it possible to use a PPRF with a smaller domain (and thus smaller punctured keys). To save a factor of up to n in server-side storage, the ticket size rises roughly as $(n-1)!$. Thus, this is only useful for small values of n, but choosing e.g. $n = 5$ can yield significant savings with a modest increase in ticket size on the wire. Concretely, for $n = 5$ and "128-bit security", ticket size is increased by 384 bytes. As discussed in Section 5.5, experiments done by Google estimate that this will impose only a small impact on latency [33].

Large-Scale Server Clusters and Load Balancing. Large TLS server deployments typically consist of many servers that share the same public key. This complicates any logic that relies on the server storing some state, since these servers will typically not share a globally-consistent state. Such discussion is beyond the scope of this paper, and we will assume a single server with consistent storage throughout. When many servers share a Session Cache, the cache is likely to be distributed, and any logic relying on an atomic retrieve-and-delete operation becomes more complex. Therefore, distributed Session Caches are not necessarily replay-resistant nor forward-secure, as this requires synchronous deletion of Resumption Secrets at all servers, and thus synchronized state.[7] However, in such large-scale settings it is highly desirable to minimize the amount of memory that must be consistently synchronized across different servers. Our techniques are therefore useful to that end as well.

Further Applications to Devices with Restricted Resources. Our techniques may also be useful for devices with very restricted resources, such as battery-powered IoT devices with a wireless network connection. For such devices, it is usually extremely expensive to *send* data, because each transmitted bit costs energy, which limits the battery lifetime and thus the range of possible applications. In order to maximize the battery lifetime, it is useful to avoid expensive interactive handshakes and use a 0-RTT protocol whenever data is sent to such devices. Note that here the main gain from using 0-RTT is *not*

[7] When using Session Tickets, the same holds for mechanisms that store used tickets, which are likely to be distributed as well. See [39, §2.3, §8, §E.5], [36,37] for more in-depth discussion.

minimal latency, but rather that no key exchange messages must be sent by the receiver. Ideally, transmitted data should be forward-secure, but such devices have low storage capacity and we cannot use large amounts of storage to achieve forward security.

For such devices, it is reasonable to relax the requirement for very efficient computation, since adding unnecessary transmissions to even a fraction of connections is likely more costly than using moderately more expensive computations. By instantiating our session resumption protocol in a way that puncturing is more expensive (say, five full RSA exponentiations, which may still be reasonable for most IoT devices), we achieve reductions in storage by factors close to 100. Thus, our techniques make it possible to use forward-secure 0-RTT protocols even on such devices. Instead of requiring, say, 1 GB of memory for a session cache, we need only about 10 MBs of memory.

Related Work. Puncturable encryption [24] was used to construct forward-secure instant messaging [24] and 0-RTT protocols [15,16,25], for instance. Green and Miers [24] first proposed puncturable encryption as a practical building block for the case of asynchronous messaging. They used pairing-based puncturable encryption, and as a result observed impractically long processing times for their construction. Günther *et al.* [25] proposed using puncturable encryption for 0-RTT protocols, again proposing concrete constructions based on pairings that are also impractical for high-traffic scenarios. Derler *et al.* [15,16] proposed trading off space in exchange for processing time, with the use of their proposed Bloom Filter Encryption. Their construction essentially precomputes many already-punctured keys, and these keys are used only once, so puncturing becomes simply key deletion. Bloom Filter Encryption may be considered practical for low-traffic scenarios, but supporting a large number of puncturings per key requires precomputation and storage of keys on the order of many gigabytes.

Over the past years there have been several papers formally analyzing the security of TLS 1.2 [7,28,31] and TLS 1.3 [17,21]. Especially noteworthy are the analyses of the 0-RTT mode of TLS 1.3 [21] and QUIC [20] by Fischlin and Günther, who analyze both protocols in a multi-stage key exchange model [20]. Lychev *et al.* [35] further formally analyzed QUIC in a security model that additionally captures the secure composition of authenticated encryption and key exchange. A security definition and construction for QUIC-like 0-RTT protocols were given in [26]. However, all these publications do *not* consider forward secrecy for the very first message in their security models. Hence, we believe that our techniques may also influence the design of protocols providing a 0-RTT key exchange, such as TLS 1.3 and QUIC, in order to achieve forward secrecy for all messages.

Outline. The rest of this paper is organized as follows. In Section 2 we provide formal definitions for secure 0-RTT Session Resumption Protocols. In Section 3 we describe a generic construction, based on abstract PPRFs, and formally prove forward security and replay resilience. Section 4 describes the Strong-RSA-based PPRF and an analysis of the efficiency when used in the protocol construction in Section 3. Section 5 describes the tree-based PPRF and a novel "domain extension" technique for standard binary tree PPRFs, along with an efficiency analysis.

Notation. We denote the security parameter as λ. For any $n \in \mathbb{N}$ let 1^n be the unary representation of n and let $[n] = \{1, \ldots, n\}$ be the set of numbers between 1 and n. Moreover, $|x|$ denotes the length of a bitstring x, while $|\mathcal{S}|$ denotes the size of a set $\mathcal{S}$. We write $x \xleftarrow{\$} \mathcal{S}$ to indicate that we choose element x uniformly at random from set $\mathcal{S}$. For a probabilistic polynomial-time algorithm $\mathcal{A}$ we define $y \xleftarrow{\$} \mathcal{A}(a_1, \ldots, a_n)$ as the execution of $\mathcal{A}$ (with fresh random coins) on input $a_1, \ldots, a_n$ and assigning the output to y.

2 0-RTT Session Resumption Protocols and Their Security

In this section we provide formal definitions for *secure 0-RTT session resumption protocols*. These definitions capture well both our new techniques and the existing solutions already standardized in TLS 1.3. We also expect that the techniques used to formally analyze and verify TLS 1.3 0-RTT [14,18] can be extended to use our abstraction of a session resumption protocol within TLS 1.3.[8] This leads us to believe that our definitions capture a reasonable abstraction of the cryptographic core of the TLS 1.3 0-RTT mode (and likely also of similar protocols that may be devised in the future).

For simplicity, in the following we will refer to pre-shared values as *session keys*, as they are either previously established session keys, or a Resumption Secret derived from a session key, as e.g. in TLS 1.3. The details of how to establish a shared secret and potentially derive a session key from it are left to the individual protocol and are outside the scope of our abstraction. Session keys are elements of a keyspace $\mathcal{S}$.

Definition 1. *A* 0-RTT session resumption protocol *consists of three probabilistic polynomial-time algorithms* $\mathsf{Resumption} = (\mathsf{Setup}, \mathsf{TicketGen}, \mathsf{ServerRes})$ *with the following properties.*

- $\mathsf{Setup}(1^\lambda)$ *takes as input the security parameter* λ *and outputs the server's long-term key* k*.*
- $\mathsf{TicketGen}(k, s)$ *takes as input a long-term key* k *and a session key* s*, and outputs a ticket* t *and a potentially modified long-term key* k'*.*
- $\mathsf{ServerRes}(k, t)$ *takes as input the server's long-term key* k *and the ticket* t*, and outputs a session key* s *and a potentially modified key* k'*, or a failure symbol* $\perp$*.*

Using a Session Resumption Protocol. A 0-RTT session resumption scheme is used by a set of clients C and a set of servers S. If a client and a server share a session key s, the session resumption is executed as follows (cf. Figure 1).

1. The server uses its long-term key k and the session key s to generate a ticket t by running $(t, k') \xleftarrow{\$} \mathsf{TicketGen}(k, s)$. The ticket is sent to the client. Additionally, the server replaces its long-term key k by k' and deletes the session key s and ticket t, i.e. it is not required to keep any session state.

[8] Obtaining a formal security proof for this would be an interesting direction for future research, but is beyond the scope of this work.

2. For session resumption at a later point in time, the client sends the ticket t to the server.
3. Upon receiving the ticket t, the server runs $(s, k') := \mathsf{ServerRes}(k, t)$ to retrieve the session key s. Additionally, k is deleted and replaced by the updated key k'.

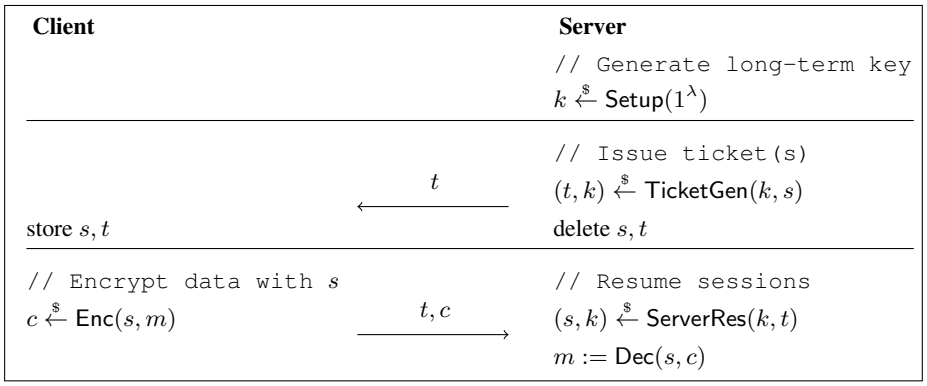

Fig. 1. Execution of a generic 0-RTT session resumption protocol with early data m, where client and server initially are in possession of a shared secret s. Note that procedures TicketGen and ServerRes both potentially modify the server's key k.

Compatibility with TLS 1.3. As explained in Section 1, using either Session Tickets or Session Caches in TLS 1.3 is transparent to clients, i.e. clients are generally unaware of which is used. In either case, the client stores a sequence of bytes which is opaque from the client's point of view. Since all algorithms of a session resumption protocol are executed on the server, while a client just has to store the ticket t (encoded as a sequence of bytes), this generic approach of TLS 1.3 is immediately compatible with our notion of session resumption protocols. Thus, a session resumption protocol can be used immediately in TLS 1.3, without requiring changes to clients or to the protocol. Furthermore, Session Tickets and Session Caches are specific examples of such protocols.

2.1 Security in the Single-Server Setting

We define the security of a 0-RTT session resumption protocol Resumption by a security game $\mathsf{G}^{\text{0-RTT-SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda)$ between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. For simplicity, we will start with a single-server setting and argue below that security in the single-server setting implies security in a multi-server setting. The security game is parametrized by the number of session keys μ (equal to the number of clients in the single-server setting).

1. $\mathcal{C}$ runs $k \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$, samples a random bit $b \xleftarrow{\$} \{0, 1\}$ and generates session keys $s_i \xleftarrow{\$} \mathcal{S}$ for all clients $i \in [\mu]$. Furthermore, it generates tickets t_i and updates key k by running $(t_i, k) \xleftarrow{\$} \mathsf{TicketGen}(k, s_i)$ for all clients $i \in [\mu]$. The sequence of tickets $(t_i)_{i\in[\mu]}$ is sent to $\mathcal{A}$.

2. The adversary gets access to oracles it may query.
   (a) $\mathsf{Dec}(t)$ takes as input a ticket t. It computes $(s_i, k') := \mathsf{ServerRes}(k, t_i)$, returns the session key s_i and replaces $k := k'$. Note that ticket t can either be a ticket of the initial sequence of tickets $(t_i)_{i\in[\mu]}$ or an arbitrary ticket chosen by the adversary.
   (b) $\mathsf{Test}(t)$ takes as input a ticket t. It computes $(s_i, k') := \mathsf{ServerRes}(k, t)$ and outputs $\perp$ if the output of $\mathsf{ServerRes}$ was $\perp$. Otherwise, it updates $k := k'$. If $b = 1$, then it returns the session key s_i. Otherwise, a random $r_i \xleftarrow{\$} \mathcal{S}$ is returned. Note that ticket t can either be a ticket of the initial sequence of tickets $(t_i)_{i\in[\mu]}$ or an arbitrary ticket chosen by the adversary.
   The adversary is allowed to query Test only once.
   (c) Corr returns the current long-term key k of the server. The adversary must not query Test after Corr, as this would lead to a trivial attack.
3. Eventually, adversary $\mathcal{A}$ outputs a guess b^*. Challenger $\mathcal{C}$ outputs 1 if $b = b^*$ and 0 otherwise.

Note that this security model reflects both forward secrecy and replay protection. Forward secrecy is ensured, as an adversary may corrupt the challenger after issuing the Test-query. If the protocol would not ensure forward secrecy, an attacker could corrupt its long-term key and trivially decrypt the challenge ticket. Replay protection is ensured, as an adversary is allowed to issue $\mathsf{Dec}(t_i)$ after already testing $\mathsf{Test}(t_i)$ (as both queries invoke the $\mathsf{ServerRes}$ algorithm). If the protocol would not ensure replay protection, an attacker could use the decryption oracle to distinguish a real or random session key of the Test-query.

Definition 2. *We define the advantage of an adversary $\mathcal{A}$ in the above security game $\mathsf{G}^{\text{0-RTT-SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda)$ as*

$$\mathsf{Adv}^{\text{0-RTT-SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda) = \left|\Pr\left[\mathsf{G}^{\text{0-RTT-SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda) = 1\right] - \frac{1}{2}\right|.$$

We say a 0-RTT session resumption protocol is secure in a single-server environment *if the advantage $\mathsf{Adv}^{\text{0-RTT-SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda)$ is a negligible function in λ for all probabilistic polynomial-time adversaries $\mathcal{A}$.*

3 Constructing Secure Session Resumption Protocols

In this section we will show how session resumption protocols providing full forward security and replay resilience can be constructed. We will start with a generic construction, based on authenticated encryption with associated data and any puncturable pseudorandom function that is invariant to puncturing. Later we describe new constructions of PPRFs, which are particularly suitable for use in session resumption protocols.

3.1 Building Blocks

We briefly recall the basic definition of puncturable pseudorandom functions and authenticated encryption with associated data.

Puncturable PRFs. A puncturable pseudorandom function is a special case of a pseudorandom function (PRF), where it is possible to compute punctured keys which do not allow evaluation on inputs that have been punctured. We recall the definition of puncturable pseudorandom functions and its security from [41].

Definition 3. *A* puncturable pseudorandom function *(PPRF) with keyspace $\mathcal{K}$, domain $\mathcal{X}$ and range $\mathcal{Y}$ consists of three probabilistic polynomial-time algorithms* $\mathsf{PPRF} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Punct})$*, which are described as follows.*

- $\mathsf{Setup}(1^\lambda)$*: This algorithm takes as input the security parameter λ and outputs a description of a key $k \in \mathcal{K}$.*
- $\mathsf{Eval}(k, x)$*: This algorithm takes as input a key $k \in \mathcal{K}$ and a value $x \in \mathcal{X}$, and outputs a value $y \in \mathcal{Y}$.*
- $\mathsf{Punct}(k, x)$*: This algorithm takes as input a key $k \in \mathcal{K}$ and a value $x \in \mathcal{X}$, and returns a punctured key $k' \in \mathcal{K}$.*

Definition 4. *A PPRF is* correct *if for every subset $\{x_1, \ldots, x_n\} = \mathcal{S} \subseteq \mathcal{X}$ and all $x \in \mathcal{X} \setminus \mathcal{S}$, we have that*

$$\Pr\left[\mathsf{Eval}(k_0, x) = \mathsf{Eval}(k_n, x) : \begin{array}{c} k_0 \xleftarrow{\$} \mathsf{Setup}(1^\lambda); \\ k_i = \mathsf{Punct}(k_{i-1}, x_i) \text{ for } i \in [n]; \end{array}\right] = 1.$$

A new property of PPRFs that we will need is that puncturing is "commutative", i.e. the order of puncturing operations does not affect the resulting secret key. That is, for any $x_0, x_1 \in \mathcal{X}, x_0 \neq x_1$, if we first puncture on input x_0 and then on x_1, the resulting key is identical to the key obtained from first puncturing on x_1 and then on x_0. Formally:

Definition 5. *A PPRF is* invariant to puncturing *if for all keys $k \in \mathcal{K}$ and all elements $x_0, x_1 \in \mathcal{X}, x_0 \neq x_1$ it holds that*

$$\mathsf{Punct}(\mathsf{Punct}(k, x_0), x_1) = \mathsf{Punct}(\mathsf{Punct}(k, x_1), x_0).$$

We define two notions of PPRF security. The first notion represents the typical pseudorandomness security experiment with adaptive evaluation queries by an adversary. The second notion is a weaker, non-adaptive security experiment. We show that it suffices to prove security in the non-adaptive experiment if the PPRF is invariant to puncturing and has a polynomial-size domain.

Definition 6. *We define the advantage of an adversary $\mathcal{A}$ in the* rand *(resp.* na-rand*) security experiment $\mathsf{G}^{\mathsf{rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$ (resp. $\mathsf{G}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$) defined in Figure 2 as*

$$\mathsf{Adv}^{\mathsf{rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda) := \left|\Pr\left[\mathsf{G}^{\mathsf{rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda) = 1\right] - \frac{1}{2}\right|,$$

$$\mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda) := \left|\Pr\left[\mathsf{G}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda) = 1\right] - \frac{1}{2}\right|.$$

We say a puncturable pseudorandom function PPRF *is* rand-secure *(resp.* na-rand-secure*), if the advantage $\mathsf{Adv}^{\mathsf{rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$ (resp. $\mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$) is a negligible function in λ for all probabilistic polynomial-time adversaries $\mathcal{A}$.*

$\mathsf{G}^{\mathsf{rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$	$\mathsf{G}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$
$k \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$, $b \xleftarrow{\$} \{0,1\}$, $Q := \emptyset$	$k_0 \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$, $b \xleftarrow{\$} \{0,1\}$
$x^* \xleftarrow{\$} \mathcal{A}^{\mathsf{Eval}(k,\cdot)}(1^\lambda)$	$(x_1, \ldots, x_\ell) \xleftarrow{\$} \mathcal{A}(1^\lambda)$
where $\mathsf{Eval}(k,x)$ behaves like Eval, but sets	$k_i := \mathsf{Punct}(k_{i-1}, x_i)$ for all $i \in [\ell]$
$Q := Q \cup \{x\}$, and runs $k := \mathsf{Punct}(k,x)$	$y_{i,0} \xleftarrow{\$} \mathcal{Y}$, $y_{i,1} := \mathsf{Eval}(k_0, x_i)$ for all $i \in [\ell]$
$y_0 \xleftarrow{\$} \mathcal{Y}$, $y_1 := \mathsf{Eval}(k,x^*)$, $k := \mathsf{Punct}(k,x^*)$	$b^* \xleftarrow{\$} \mathcal{A}(k_\ell, (y_{i,b})_{i\in[\ell]})$
$b^* \xleftarrow{\$} \mathcal{A}(k, y_b)$	return 1 if $b = b^*$
return 1 if $b = b^* \wedge x^* \notin Q$	return 0
return 0	

Fig. 2. Security experiments for PPRFs. The na-rand security experiment for PPRF is left and the rand security experiment is right.

It is relatively easy to prove that na-rand-security and rand-security are equivalent, up to a linear security loss in the size of the domain of the PPRF. In particular, if the PPRF has a polynomially-bounded domain size and is invariant to puncturing, then both are polynomially equivalent.

Theorem 1. *Let* PPRF *be a* na-rand*-secure PPRF with domain* $\mathcal{X}$*. If* PPRF *is invariant to puncturing, then it is also* rand*-secure with advantage*

$$\mathsf{Adv}^{\mathsf{rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda) \leq \frac{\mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)}{|\mathcal{X}|}.$$

Proof. The proof is based on a straightforward reduction. We give a sketch. Let $\mathcal{A}$ be an adversary against the rand security of PPRF. We guess $\mathcal{A}$'s challenge value in advance by sampling $\nu \xleftarrow{\$} \mathcal{X}$ uniformly at random. We initialize the na-rand challenger by sending it ν. In return we receive a challenge y (either computed via Eval or random) and a punctured key k that cannot be evaluated on input ν.

The punctured key k allows us to correctly answer all of $\mathcal{A}$'s Eval queries, except for ν. When the adversary outputs its challenge x^* we will abort if $x^* \neq \nu$. Otherwise, we forward y and a punctured key k' that has been punctured on all values of the Eval queries. Note that the key has a correct distribution, as we require that the PPRF is invariant to puncturing.

Eventually, $\mathcal{A}$ outputs a bit b^* which we forward to the na-rand challenger.

The simulation is perfect unless we abort it, which happens with polynomially-bounded probability $1/|\mathcal{X}|$, due to the fact that $|\mathcal{X}|$ is polynomially bounded. □

Authenticated Encryption with Associated Data. We will furthermore need authenticated encryption with associated data (AEAD) [40], along with the standard notions of confidentiality and integrity.

Definition 7. *An* authenticated encryption scheme with associated data *is a tuple* $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *of three probabilistic polynomial-time algorithms:*

- $\mathsf{KGen}(1^\lambda)$ *takes as input a security parameter* λ *and outputs a secret key* k*.*
- $\mathsf{Enc}(k, m, ad)$ *takes as input a key* k*, a message* m*, associated data* ad *and outputs a ciphertext* c*.*
- $\mathsf{Dec}(k, c, ad)$ *takes as input a key* k*, a ciphertext* c*, associated data* ad *and outputs a message* m *or an failure symbol* $\perp$*.*

An AEAD scheme is called *correct* if for any key $k \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$, any message $m \in \{0,1\}^*$, any associated data $ad \in \{0,1\}^*$ it holds that $\mathsf{Dec}(k, \mathsf{Enc}(k, m, ad), ad) = m$.

Definition 8. *We define the advantage of an adversary* $\mathcal{A}$ *in the* IND-CPA *experiment* $\mathsf{G}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A},\mathsf{AEAD}}(\lambda)$ *defined in Figure 3 as*

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

We say an AEAD scheme AEAD *is* indistinguishable under chosen-plaintext attacks *(*IND-CPA-secure*), if the advantage* $\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A},\mathsf{AEAD}}(\lambda)$ *is a negligible function in* λ *for all probabilistic polynomial-time adversaries* $\mathcal{A}$*.*

Definition 9. *We define the advantage of an adversary* $\mathcal{A}$ *in the* INT-CTXT *experiment* $\mathsf{G}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{AEAD}}(\lambda)$ *defined in Figure 3 as*

$$\mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{AEAD}}(\lambda) := \left| \Pr\left[\mathsf{G}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{AEAD}}(\lambda) = 1\right] \right|.$$

We say an AEAD scheme AEAD *provides* integrity of ciphertexts *(*INT-CTXT-secure*), if the advantage* $\mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{AEAD}}(\lambda)$ *is a negligible function in* λ *for all probabilistic polynomial-time adversaries* $\mathcal{A}$*.*

Additionally, we will need the notion of ε-spreadness for AEAD. ε-spreadness captures the intuition that a ciphertext encrypted under a key k should not be valid under a random key $k' \neq k$.

Definition 10. *An AEAD scheme is* ε-spread *if for all messages* m *and all associated data* ad *it holds that*

$$\Pr_{\substack{k,k' \xleftarrow{\$} \mathsf{KGen}(1^\lambda) \\ k \neq k'}} [\mathsf{AEAD.Dec}(k', \mathsf{AEAD.Enc}(k, m, ad), ad) \neq \perp] \leq \varepsilon.$$

We note that one can easily prove that INT-CTXT-security implies ε-spreadness with negligible ε. However, the "statistical" formulation of Definition 10 will simplify parts of our proof significantly, and therefore we believe it reasonable to make it explicit.

3.2 Generic Construction

Now we are ready to describe our generic construction of a 0-RTT session resumption protocol, based on a PPRF and an AEAD scheme, and to prove its security.

$\mathsf{G}^{\mathsf{IND\text{-}CPA}}_{\mathcal{A},\mathsf{AEAD}}(\lambda)$

$k \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$, $b \xleftarrow{\$} \{0,1\}$

$b^* \xleftarrow{\$} \mathcal{A}^{\mathsf{LoR}(\cdot,\cdot,\cdot)}(1^\lambda)$

where $\mathsf{LoR}(m_0, m_1, ad)$.

returns $\mathsf{Enc}(k, m_b, ad)$.

return 1 if $b = b^*$

return 0

$\mathsf{G}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{A},\mathsf{AEAD}}(\lambda)$

$k \xleftarrow{\$} \mathsf{KGen}(1^\lambda)$, $\mathcal{Q} := \emptyset$, $\mathsf{win} := 0$

$\mathcal{A}^{\mathsf{Enc}(\cdot,\cdot),\mathsf{Dec}(\cdot,\cdot)}(1^\lambda)$

where $\mathsf{Enc}(m, ad)$ returns $\mathsf{Enc}(k, m, ad)$

and sets $\mathcal{Q} := \mathcal{Q} \cup \{(c, ad)\}$,

and where $\mathsf{Dec}(c, ad)$ sets $\mathsf{win} := 1$

if $\mathsf{Dec}(k, c, ad) \neq \perp$ and $(c, ad) \notin \mathcal{Q}$.

return win

Fig. 3. The IND-CPA and INT-CTXT security experiment for AEAD [40].

Construction 1. *Let* $\mathsf{AEAD} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ *be an authenticated encryption scheme with associated data and let* $\mathsf{PPRF} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Punct})$ *be a PPRF with range* $\mathcal{Y}$*. Then we can construct a 0-RTT session resumption protocol* $\mathsf{Resumption} = (\mathsf{Setup}, \mathsf{TicketGen}, \mathsf{ServerRes})$ *in the following way.*

- $\mathsf{Setup}(1^\lambda)$ *runs* $k_{\mathsf{PPRF}} = \mathsf{PPRF.Setup}(1^\lambda)$*, and outputs* $k := (k_{\mathsf{PPRF}}, 0)$*, where "*$0$*" is a counter initialized to zero.*
- $\mathsf{TicketGen}(k, s)$ *takes a key* $k = (k_{\mathsf{PPRF}}, n)$*. It computes* $\kappa = \mathsf{PPRF.Eval}(k_{\mathsf{PPRF}}, n)$*. Then it encrypts the ticket as* $t' \xleftarrow{\$} \mathsf{AEAD.Enc}(\kappa, s, n)$*. Finally, it defines* $t = (t', n)$ *and* $k := (k_{\mathsf{PPRF}}, n + 1)$*, and outputs* (t, k)*.*
- $\mathsf{ServerRes}(k, t)$ *takes* $k = (k_{\mathsf{PPRF}}, n)$ *and* $t = (t', n')$*. It computes a key* $\kappa := \mathsf{PPRF.Eval}(k_{\mathsf{PPRF}}, n')$*. If* $\kappa = \perp$*, then it returns* $\perp$*. Otherwise it computes a session key* $s := \mathsf{AEAD.Dec}(\kappa, t', n')$*. If* $s = \perp$*, it returns* $\perp$*. Else it punctures* $k_{\mathsf{PPRF}} := \mathsf{PPRF.Punct}(k_{\mathsf{PPRF}}, n')$*, and returns* $(s, (k_{\mathsf{PPRF}}, n))$*.*

Note that the associated data n is sent in plaintext, posing a potential privacy leak. This can be circumvented by additionally encrypting n under a dedicated symmetric key. Compromise of this key would only allow an attacker to link sessions by the same returning client, not to decrypt past traffic, therefore this symmetric key needs not be punctured to achieve forward security.[9]

Theorem 2. *If* AEAD *is* ε*-spread and* PPRF *is invariant to puncturing, then from each probabilistic polynomial-time adversary* $\mathcal{A}$ *against the security of* Resumption *in a single-server environment with advantage* $\mathsf{Adv}^{\mathsf{0\text{-}RTT\text{-}SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda)$*, we can construct four adversaries* $\mathcal{B}_{\mathsf{PPRF1}}$*,* $\mathcal{B}_{\mathsf{PPRF2}}$*,* $\mathcal{B}_{\mathsf{AEAD1}}$*, and* $\mathcal{B}_{\mathsf{AEAD2}}$ *such that*

$$\mathsf{Adv}^{\mathsf{0\text{-}RTT\text{-}SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda) \leq \mathsf{Adv}^{\mathsf{rand}}_{\mathcal{B}_{\mathsf{PPRF1}},\mathsf{PPRF}}(\lambda) + \varepsilon + \mu \cdot \Big(\mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{B}_{\mathsf{PPRF2}},\mathsf{PPRF}}(\lambda) + \mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{B}_{\mathsf{AEAD1}},\mathsf{AEAD}}(\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{B}_{\mathsf{AEAD2}},\mathsf{AEAD}}(\lambda)\Big),$$

[9] The natural solution would be to encrypt n using public-key puncturable encryption, but this would be costly, and obviate most of the efficiency benefits described in this work. We are unfortunately unaware of a good solution that achieves session unlinkability in the event of server compromise. We further note that TLS 1.3 0-RTT includes a mechanism named "obfuscated ticket age" that solves a similar session linkability concern; that mechanism as well is not applicable here.

where μ is the number of clients.

Proof. We will conduct this proof in a sequence of games between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. We start with an adversary playing the 0-RTT-SR security game. Over a sequence of hybrid arguments, we will stepwise transform the security game to a game where the Test-query is independent of the challenge bit b. The claim then follows from bounding the probability of distinguishing any two consecutive games. By Adv_i we denote $\mathcal{A}$'s advantage in the i-th game.

Game 0. We define Game 0 to be the original 0-RTT-SR security game. By definition we have

$$\mathsf{Adv}_0 = \mathsf{Adv}^{\text{0-RTT-SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda).$$

Game 1. This game is identical to Game 0, except that we raise an event $\mathsf{abort}_{\mathsf{PPRF}}$, abort the game, and output a random bit $b^* \xleftarrow{\$} \{0,1\}$, if the adversary $\mathcal{A}$ ever queries $\mathsf{Test}(t)$ for a ticket $t = (t', n')$ such that $n' \notin [\mu]$ and $\mathsf{AEAD.Dec}(\kappa, t', n') \neq \bot$, where $\kappa := \mathsf{PPRF.Eval}(k_{\mathsf{PPRF}}, n')$. Since both games proceed identical until abort, we have

$$|\mathsf{Adv}_1 - \mathsf{Adv}_0| \leq \Pr[\mathsf{abort}_{\mathsf{PPRF}}]$$

and we claim that we can construct an adversary $\mathcal{B}_{\mathsf{PPRF1}}$ on the rand-security of the PPRF with advantage at least $\Pr[\mathsf{abort}_{\mathsf{PPRF}}]$.

Construction of $\mathcal{B}_{\mathsf{PPRF1}}$. $\mathcal{B}_{\mathsf{PPRF1}}$ behaves like the challenger in Game 1, expect that it uses the Eval-oracle to generate the keys to encrypt the initial sequence of μ tickets and to answer all Dec-queries by $\mathcal{A}$. Eventually, $\mathcal{A}$ will query $\mathsf{Test}(t)$ for a ticket $t = (t', n')$. $\mathcal{B}_{\mathsf{PPRF1}}$ outputs n' to its PPRF-challenger, which will respond with a punctured key $k := \mathsf{PPRF.Punct}(k, n')$ and a value γ, where either $\gamma := \rho \xleftarrow{\$} \mathcal{Y}$ or $\gamma := \mathsf{PPRF.Eval}(k, n')$.

$\mathcal{B}_{\mathsf{PPRF1}}$ now tries to decrypt the challenge ticket by invoking $\mathsf{AEAD.Dec}(\gamma, t', n')$. If $\gamma = \mathsf{PPRF.Eval}(k, n')$, the decryption will succeed by definition. If $\gamma = \rho$, the decryption will fail with probability $1 - \varepsilon$, since the ε-spreadness of AEAD ensures that $\mathsf{AEAD.Dec}(\rho, t', n') \neq \bot$ for random ρ happens only with probability ε. Hence, $\mathcal{B}_{\mathsf{PPRF1}}$ returns 1 if decryption succeeds and 0 otherwise. Thus, we have

$$\Pr[\mathsf{abort}_{\mathsf{PPRF}}] \leq \mathsf{Adv}^{\mathsf{rand}}_{\mathcal{B}_{\mathsf{PPRF1}},\mathsf{PPRF}}(\lambda) + \varepsilon.$$

Game 2. This game is identical to Game 1, except for the following changes. At the beginning of the experiment the challenger picks an index $\nu \xleftarrow{\$} [\mu]$. It aborts the security experiment and outputs a random bit $b^* \xleftarrow{\$} \{0,1\}$, if the adversary queries $\mathsf{Test}(t)$ with $t = (t', i)$ such that $i \neq \nu$. Since the choice of $\nu \xleftarrow{\$} [\mu]$ is oblivious to $\mathcal{A}$ until an abort occurs, we have

$$\mathsf{Adv}_2 \geq \frac{1}{\mu} \cdot \mathsf{Adv}_1.$$

Game 3. This game is identical to Game 2, except that at the beginning of the game we compute $\kappa_\nu = \mathsf{PPRF.Eval}(k, \nu)$ and then $k := \mathsf{PPRF.Punct}(k, \nu)$. Furthermore, we replace algorithm PPRF.Eval with the following algorithm F_3:

$$F_3(k, i) := \begin{cases} \mathsf{PPRF.Eval}(k, i) & \text{if } i \neq \nu \\ \kappa_\nu & \text{if } i = \nu \end{cases}$$

Everything else works exactly as before. Note that we have simply implemented algorithm PPRF.Eval in a slightly different way. Since PPRF is invariant to puncturing, the fact that κ_ν was computed early, immediately followed by $k := \mathsf{PPRF.Punct}(k, \nu)$, is invisible to $\mathcal{A}$. Hence, Game 3 is perfectly indistinguishable from Game 2, and we have

$$\mathsf{Adv}_3 = \mathsf{Adv}_2.$$

Game 4. This game is identical to Game 3, except that the challenger now additionally picks a random key $\rho \xleftarrow{\$} \mathcal{Y}$ from the range of the PPRF. Furthermore, we replace algorithm F_3 with the following algorithm F_4:

$$F_4(k, i) := \begin{cases} \mathsf{PPRF.Eval}(k, i) & \text{if } i \neq \nu \\ \rho & \text{if } i = \nu \end{cases}$$

Everything else works exactly as before. We will now show that any adversary that is able to distinguish Game 3 from Game 4 can be used to construct an adversary $\mathcal{B}_{\mathsf{PPRF2}}$ against the na-rand-security of the PPRF. Concretely, we have

$$|\mathsf{Adv}_4 - \mathsf{Adv}_3| \leq \mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{B}_{\mathsf{PPRF2}},\mathsf{PPRF}}(\lambda).$$

Construction of $\mathcal{B}_{\mathsf{PPRF2}}$. $\mathcal{B}_{\mathsf{PPRF2}}$ initially picks $\nu \xleftarrow{\$} [\mu]$ and outputs ν to its PPRF-challenger, which will respond with a punctured key $k := \mathsf{PPRF.Punct}(k, \nu)$ and a value γ, where either $\gamma = \mathsf{PPRF.Eval}(k, \nu)$ or $\gamma \xleftarrow{\$} \mathcal{Y}$. Now $\mathcal{B}_{\mathsf{PPRF2}}$ simulates Game 4, except that it uses the following function F in place of F_4.

$$F(k, i) := \begin{cases} \mathsf{PPRF.Eval}(k, i) & \text{if } i \neq \nu \\ \gamma & \text{if } i = \nu \end{cases}$$

Eventually, $\mathcal{A}$ will output a guess b^*. $\mathcal{B}_{\mathsf{PPRF2}}$ forwards this bit to the PPRF-challenger. Note that if $\gamma = \mathsf{Eval}(k, \nu)$, then function F is identical to F_3, while if $\gamma = \rho$ then it is identical to F_4. This proves the claim.

Game 5. This game is identical to Game 4, except that we raise an event $\mathsf{abort}_{\mathsf{AEAD}}$, abort the game, and output a random bit $b^* \xleftarrow{\$} \{0, 1\}$, if the adversary $\mathcal{A}$ ever queries $\mathsf{Test}(t)$ for a ticket $t = (t', \nu) \neq t_\nu$, but $\mathsf{AEAD.Dec}(\rho, t', \nu) \neq \perp$, where $\rho = F_4(k, \nu)$. We have

$$|\mathsf{Adv}_5 - \mathsf{Adv}_4| \leq \Pr[\mathsf{abort}_{\mathsf{AEAD}}]$$

and we claim that we can construct an adversary $\mathcal{B}_{\mathsf{AEAD1}}$ on the INT-CTXT-security of the AEAD with advantage at least $\Pr[\mathsf{abort}_{\mathsf{AEAD}}]$.

Construction of $\mathcal{B}_{\mathsf{AEAD1}}$. $\mathcal{B}_{\mathsf{AEAD1}}$ proceeds exactly like the challenger in Game 5, except that it uses its challenger from the AEAD security experiment to create ticket t_ν. To this end, it outputs the tuple (s_ν, ν) for some $s_\nu \xleftarrow{\$} \mathcal{S}$. The AEAD challenger responds with $t'_\nu := \mathsf{AEAD.Enc}(\rho, s_\nu, \nu)$, computed with an independent AEAD key ρ. Finally, $\mathcal{B}_{\mathsf{AEAD1}}$ defines the ticket as $t_\nu = (t'_\nu, \nu)$. Apart from this, $\mathcal{B}_{\mathsf{AEAD1}}$ proceeds exactly like the challenger in Game 5.

Whenever the adversary $\mathcal{A}$ makes a query $\mathsf{Test}(t)$ with a ticket $t = (t', i)$ with $i \neq \nu$, then we abort, due to the changes introduced in Game 2. If it queries $\mathsf{Test}(t)$ with $t = (t', \nu)$ such that $t \neq t_\nu$, then $\mathcal{B}_{\mathsf{AEAD1}}$ responds with $\perp$ and outputs the tuple (t', ν) to its AEAD challenger. With probability $\Pr[\mathsf{abort}_{\mathsf{AEAD}}]$ this ticket is valid, which yields

$$\mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{B}_{\mathsf{AEAD1}},\mathsf{AEAD}}(\lambda) \geq \Pr[\mathsf{abort}_{\mathsf{AEAD}}].$$

Game 6. This game is identical to Game 5, except that when the adversary queries $\mathsf{Test}(t_\nu)$, then we will always answer with a random value, independent of the bit b. More precisely, recall that we abort if the adversary queries $\mathsf{Test}(t)$, $t = (t', \nu)$ such that $t \neq t_\nu$, due to the changes introduced in Game 5. If the adversary queries $\mathsf{Test}(t_\nu)$, then the challenger in Game 5 uses the bit $b \xleftarrow{\$} \{0, 1\}$ sampled at the beginning of the experiment as follows. If $b = 1$, then it returns the session key s_ν. Otherwise, a random $r_\nu \xleftarrow{\$} \mathcal{S}$ is returned.

In Game 6, the challenger samples another random value $s'_\nu \xleftarrow{\$} \mathcal{S}$ at the beginning of the game. When the adversary queries $\mathsf{Test}(t_\nu)$, then if $b = 1$ the challenger returns s'_ν. Otherwise, it returns a random $r_\nu \xleftarrow{\$} \mathcal{S}$. Note that in either case the response of the $\mathsf{Test}(t_\nu)$-query is a random value, independent of b. Therefore the view of $\mathcal{A}$ in Game 6 is independent of b. Obviously, we have

$$\mathsf{Adv}_6 = 0.$$

We will now show that any adversary who is able to distinguish Game 5 from Game 6 can be used to construct an adversary $\mathcal{B}_{\mathsf{AEAD2}}$ against the IND-CPA-security of AEAD.

Construction of $\mathcal{B}_{\mathsf{AEAD2}}$. Recall that the key used to generate ticket t_ν is $\rho = F_4(k, \nu)$. By definition of F_4, ρ is an independent random string chosen at the beginning of the security experiment. This enables a straightforward reduction to the IND-CPA-security of the AEAD.

$\mathcal{B}_{\mathsf{AEAD2}}$ proceeds exactly like the challenger in Game 6, except for the way the ticket t_ν is created. $\mathcal{B}_{\mathsf{AEAD2}}$ computes $\rho_\nu = F_4(k, \nu)$. Then it outputs (s_ν, s'_ν, ν) to its challenger, which returns

$$t_\nu := \begin{cases} \mathsf{AEAD.Enc}(\rho, s_\nu, \nu) & \text{if } b' = 0 \\ \mathsf{AEAD.Enc}(\rho, s'_\nu, \nu) & \text{if } b' = 1 \end{cases}$$

where ρ is distributed identically to ρ_ν and b' is the hidden bit used by the challenger of the AEAD. Apart from this, $\mathcal{B}_{\mathsf{AEAD2}}$ proceeds exactly like the challenger in Game 6. Eventually, $\mathcal{A}$ will output a guess b^*. $\mathcal{B}_{\mathsf{AEAD2}}$ forwards this bit to its challenger.

Note that if $b' = 0$, then the view of $\mathcal{A}$ is perfectly indistinguishable from Game 5, while if $b' = 1$ then it is identical to Game 6. Thus, we have

$$|\mathsf{Adv}_6 - \mathsf{Adv}_5| \leq \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{B}_{\mathsf{AEAD}2},\mathsf{AEAD}}(\lambda).$$

By summing up probabilities from Game 0 to Game 6, we obtain

$$\begin{aligned}\mathsf{Adv}^{\mathsf{0\text{-}RTT\text{-}SR}}_{\mathcal{A},\mathsf{Resumption}}(\lambda) \leq \mathsf{Adv}^{\mathsf{rand}}_{\mathcal{B}_{\mathsf{PPRF}1},\mathsf{PPRF}}(\lambda) + \varepsilon + \mu \cdot \Big(&\mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{B}_{\mathsf{PPRF}2},\mathsf{PPRF}}(\lambda) \\ &+ \mathsf{Adv}^{\mathsf{INT\text{-}CTXT}}_{\mathcal{B}_{\mathsf{AEAD}1},\mathsf{AEAD}}(\lambda) + \mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathcal{B}_{\mathsf{AEAD}2},\mathsf{AEAD}}(\lambda)\Big).\end{aligned}$$

□

4 A PPRF with Short Secret Keys from Strong RSA

In order to instantiate our generic construction of forward-secure and replay-resilient session resumption protocol with minimal storage requirements, which is the main objective of this paper, it remains to construct suitable PPRFs with minimal storage requirements and good computational efficiency. Note that a computationally expensive PPRF may void all efficiency gains obtained from the 0-RTT protocol.

In this section we describe a PPRF based on the Strong RSA (sRSA) assumption with secret keys that only consist of three elements, even after an arbitrary number of puncturings. More precisely, a secret key consists of an RSA modulus N, an element $g \in \mathbb{Z}_N$ and a bitfield r, indicating positions where the PPRF was punctured. The secret key size is linear in the size of the PPRF's domain, since the bitfield needs to be of the same size as the domain (which is determined at initialization, and does not change over time). Hence, the PPRF's secret key size is independent of the number of puncturings. Moreover, for any reasonable choice of parameters, the bitfield is only several hundred bits long, yielding a short key in practice. Servers can use many instances in parallel with the instances sharing a single modulus, so it is only necessary to generate (and store) the modulus once, at initialization.

Since our primary objective is to provide an efficient practical solution for protocols such as TLS 1.3 0-RTT, the PPRF construction described below is analyzed in the random oracle model [5]. However, we note that we use the random oracle only to turn a "search problem" (sRSA) into a "decisional problem" (as required for a pseudorandom function). Therefore we believe that our construction can be lifted to the standard-model via standard techniques, such as hardcore predicates [6,8,23]. All of these approaches would yield less efficient constructions, and therefore are outside the scope of our work. Alternatively, one could formulate an appropriate "hashed sRSA" assumption, which would essentially boil down to assuming that our scheme is secure. Therefore we consider a random oracle analysis based on the standard sRSA problem as the cleanest and most insightful approach to describe our ideas.

Idea Behind the Construction. The construction is inspired by the *RSA accumulator* of Camenisch and Lysyanskaya [12]. The main idea is the following. Given a modulus $N = pq$, a value $g \in \mathbb{Z}_N$, and a prime number P, it is easy to compute $g \mapsto g^P$

$\mod N$, but hard to compute $g^P \mapsto g \mod N$ without knowing the factorization of N.

In the following let p_i be the i-th odd prime. That is, we have $(p_1, p_2, p_3, p_4, \ldots) = (3, 5, 7, 11, \ldots)$. Let n be the size of the domain of the PPRF. Our PPRF on input ℓ produces an output of the form $H(g^{p_1 \cdot \ldots \cdot p_n / p_\ell})$, where H is a hash function that will be modeled as a random oracle in the security proof. Note that g is raised to a sequence of prime numbers *except for* the ℓ-th prime number. As long as we have access to g, this is easy to compute. However, if we only have access to g^{p_ℓ} instead of g, we are unable to compute the PPRF output without knowledge of the factorization of N. This implies that by raising the generator to certain powers, we prevent the computation of specific outputs. We will use this property to puncture values of the PPRF's domain.

4.1 Formal Description of the Construction

Definition 11. *Let p, q be two random safe primes of bitlength $\lambda/2$ and let $N = pq$. Let $y \xleftarrow{\$} \mathbb{Z}_N^*$. We define the advantage of algorithm $\mathcal{B}$ against the* Strong RSA Assumption *[2] as*

$$\mathsf{Adv}_{\mathcal{B}}^{\mathsf{sRSA}}(\lambda) := \Pr\left[(x, e) \leftarrow \mathcal{A}(N, y) : x^e = y \bmod N\right].$$

The following lemma, which is due to Shamir [42], is useful for the security proof of our construction.

Lemma 1. *There exists an efficient algorithm that, on input $y, z \in \mathbb{Z}_N$ and integers $e, f \in \mathbb{Z}$ such that $\gcd(e, f) = 1$ and $Z^e \equiv Y^f \mod N$, computes $X \in \mathbb{Z}_N$ satisfying $X^e = Y \bmod N$.*

Construction 2. *Let $H : \mathbb{Z}_N \to \{0,1\}^\lambda$ be a hash function and let p_i be the i-th* odd *prime number. Then we construct a PPRF $\mathsf{PPRF} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Punct})$ with polynomial-size $\mathcal{X} = [n]$ in the following way.*

- $\mathsf{Setup}(1^\lambda)$ *computes an RSA modulus $N = pq$, where p, q are safe primes. Next, it samples a value $g \xleftarrow{\$} \mathbb{Z}_N$ and defines $r := 0^n$ and $k = (N, g, r)$. The primes p, q are discarded.*
- $\mathsf{Eval}(k, x)$ *parses $k = (N, g, (r_1, \ldots, r_n))$. If $r_x = 1$, then it outputs $\perp$. Otherwise it computes and returns*

  $$y := H\left(g^{P_x} \bmod N\right).$$

  where p_i is the i-th odd prime and

  $$P_x := \prod_{i \in [n], i \neq x, r_i \neq 1} p_i$$

  is the product of the first n odd primes, except for p_x.
- $\mathsf{Punct}(k, x)$ *parses $k = (N, g, (r_1, \ldots, r_n))$. If $r_x = 1$, then it returns k. If $r_x = 0$, it computes $g' := g^{p_x}$ and $r' = (r_1, \ldots, r_{x-1}, 1, r_{x+1}, \ldots, r_n)$ and returns $k' = (N, g', r')$.*

It is straightforward to verify the correctness of Construction 2 and that it is invariant to puncturing in the sense of Definition 5.

4.2 Security Analysis

We prove the following security theorem in the full version of this paper [1].

Theorem 3. *Let* $\mathsf{PPRF} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Punct})$ *be as above with polynomial-size input space* $\mathcal{X} = [n]$*. From each probabilistic polynomial-time adversary* $\mathcal{A}$ *with advantage* $\mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$ *against the* na-rand*-security (cf. Definition 6) we can construct an efficient adversary* $\mathcal{B}$ *with advantage* $\mathsf{Adv}^{\mathsf{sRSA}}_{\mathcal{B}}(\lambda)$ *against the Strong RSA problem, such that*

$$\mathsf{Adv}^{\mathsf{sRSA}}_{\mathcal{B}}(\lambda) \geq \mathsf{Adv}^{\mathsf{na\text{-}rand}}_{\mathcal{A},\mathsf{PPRF}}(\lambda).$$

4.3 Efficiency Analysis

Note that a server is able to create multiple instances of our construction to serve more tickets than one instance is able to. Using multiple instances allows using smaller exponents, but in return, the storage cost grows linearly in the number of instances.

Serving a ticket requires two exponentiations, one for computing the key and one for puncturing. Computing the key requires raising the state g to the power of $\prod_{p \in S} p$ for some subset of primes S. Puncturing requires exponentiating by a single prime. Therefore, all exponentiations feature exponents smaller than $\prod_{i=1}^{n} p_i$. We start by comparing to 2048-bit RSA, which according to the NIST key size recommendations [3] corresponds to "112-bit security", before comparing to larger RSA key sizes.

Worst-case Analysis. We compare to standard exponentiation in the group, i.e. raising to the power of $d \in \mathbb{N}$, where $\log d \approx 2048$. For puncturing to be comparable in the worst-case, we require $\log\left(\prod_{i=1}^{n} p_i\right) \leq 2048$. Choosing p_i to be the i-th odd prime yields $n \leq 232$. An economic server may store only one 2048-bit group element for the current state, and a bitfield indicating which of the 232 primes have been punctured, requiring 2280 bits in total. This allows serving 232 tickets, resulting in a storage cost of 1.22 bytes per ticket. Alternatively, a standard Session Cache would require $112 \cdot 232 = 25984$ bits to serve those 232 tickets, assuming symmetric keys of 112 bits. Therefore, our construction decreases storage size compared to a Session Cache by a factor of $25984/2280 = 11.4$.

Averaged Analysis. Note that in the above worst-case analysis we consider an *upper* bound on the exponentiation cost. That is, we guarantee that a puncturing and key derivation operation is *never* more expensive than a full exponentiation. Indeed, the first key computation raises to the power of $p_1 \cdot \ldots \cdot p_n / p_\ell$, i.e. to the product of $n-1$ primes. However, subsequent key calculations raise to smaller powers, i.e. to the product of $n-2$ primes, then $n-3$, and so on. Therefore serving tickets arriving later is much cheaper than serving the first. In particular in settings where a server uses many PPRF instances in parallel, in order to deal with potentially thousands of simultaneously issued tickets, an alternative and more reasonable efficiency analysis considers the average cost of serving a ticket be comparable to exponentiation in the group. In the worst-case, primes are punctured in order, so p_n is included in the exponent in all key derivations, p_{n-1} in all derivations except the last, etc. Each prime is also used once for

puncturing. Requiring $\sum_{i=1}^{n} i \cdot \log(p_i) \leq n \cdot 2048$ yields a maximum $n = 387$, and a savings factor of $112 \cdot 387/(2048 + 387) = 17.8$. The required storage is therefore 0.79 bytes per ticket.

Considering Other Security Parameters and Efficiency Requirements. Generalizing the above calculations, Table 1 gives concrete parameters for various security levels, following the NIST recommendations for key sizes [3]. Larger key sizes result in larger reductions in storage, especially when requiring average cost similar to exponentiation in the RSA group. We also show the improvement factor in storage when relaxing the above heuristic choice that serving a ticket must not cost more than one full RSA-exponentiation, by considering the case where serving a ticket is cheaper on average than 5 group exponentiations. This demonstrates that the proposed PPRF can yield very significant storage savings in general cryptographic settings, while keeping computation costs on the same order of magnitude as common public key operations. In the context of TLS, however, we expect most server operators would prefer parameters that keep processing time comparable to a single exponentiation. We emphasize that the improvement factor in storage is determined at initialization time, and is deterministic rather than probabilistic. The largest prime used in exponentiations determines how many tickets are served using a single group element. The worst-case and average-case refer to the processing time, not to the savings in storage.

Additional Storage for the Primes. The server will also need to store the first n primes, but this requires negligible additional storage. Storing the primes requires on the order of magnitude of ten kilobytes, where we expect typical caches to use many megabytes. For the minimal storage requirement, we consider 2048-bit RSA while requiring that the worst case puncturing time is cheaper than group exponentiation. In this case $n = 232$ and $p_n = 1471$, therefore all primes fit in 32-bit integers. Storing all the primes would require at most $4 \cdot 232 = 928$ bytes.

The largest value of n for the parameter choices presented in this work is $n = 9704$, for the "average cheaper than 5 exponentiations" case with 15360-bit RSA. $p_{9704} = 101341$. The required additional storage is therefore $4 \cdot 9704 = 38{,}816$ bytes. To reiterate, we expect typical caches to use many megabytes.

Concrete Benchmarks. We now give concrete performance estimates for this construction, using OpenSSL [45]. OpenSSL is a well-known production-grade library that implements the TLS and SSL protocols, as well as low-level cryptographic primitives. For each key size, we measure the computation time of exponentiating by all primes $\prod_{i=1}^{n} p_i$, by calling the OpenSSL "BigNum" exponentiating function. This is analogous to the computation required to serve the first ticket and then puncture the key: Serving requires exponentiating to the power of all primes except one, p_i, and puncturing requires exponentiating to the power of p_i. This is the worst-case, since serving later tickets is cheaper.

We measure the performance of this calculation for two of the above cases, which determine the value of n: 1) Worst-case is cheaper than exponentiation, and 2) The average case is cheaper than exponentiation. We note the latter case is slightly unintuitive:

		Storage Savings Factor		
Symmetric Key Size	Modulus Size	W.C. cheaper than exponentiation	Average cheaper than exponentiation	Average cheaper than 5 exponentiations
112	2048	11.40	17.80	48.92
128	3072	12.28	19.47	54.49
192	7680	16.37	26.52	77.36
256	15360	20.10	33.05	99.12

Table 1. Savings factors for various key sizes. Symmetric and asymmetric key sizes are matched according to the NIST recommendations [3]. Both savings factors denote the reduction in server-side storage required when using Construction 3. Column 3 denotes the reduction in storage achieved under the requirement that serving a single ticket is always cheaper than an exponentiation in the RSA group of respective key size. Column 4 denotes the reduction in storage achieved under the requirement that the average cost for serving a ticket is cheaper than a single exponentiation. Column 5 denotes the reduction in storage achieved under the requirement that the average cost for serving a ticket is cheaper than 5 group exponentiations.

we measure *the worst-case performance, under the requirement that the average case is comparable to one exponentiation in the group*.

Table 2 gives our results. We observe that performance is comparable to, but slower than, RSA decryption. In typical cases, it requires only a few additional milliseconds compared to RSA decryption. We argue the additional latency and computation requirement are small enough to allow the construction to be deployed as-is, in current large scale TLS deployments. It is unsurprising that RSA decryption is faster than our construction, since OpenSSL performs RSA decryption using the Chinese Remainder Theorem.

	Our construction: Decryption + Puncturing		
Modulus Size	W.C. cheaper than exponentiation	Average cheaper than exponentiation	RSA Decryption
2048	2.6	4.7	0.5
3072	8.3	15.2	2.5
4096	19.4	35.8	5.6

Table 2. Worst-case running time for serving a single ticket using our construction, compared to RSA decryption. All times are measured in milliseconds. Measurements were performed on a standard workstation, with a 3.60GHz Intel i7 CPU. All measurements used code from OpenSSL 1.0.2q, released in November 2018. To benchmark our construction we used a short piece of custom code, based on [9], to repeatedly call the OpenSSL exponentiating function. For each parameter choice, we generated 100 random moduli, and performed 100 exponentiations of random group elements to the power of $\prod_{i=1}^{n} p_i$. To benchmark RSA decryption, we used a built-in OpenSSL benchmarking command, "openssl speed" (after applying a small patch that adds support for 3072-bit RSA to the command [29]).

5 Tree-based PPRFs

This section will consider a different approach to instantiating Construction 1 based on PPRFs using trees. At first we will recap the idea behind tree-based PPRFs and explain how we utilize tree-based PPRFs as an instantiation of our session resumption protocol and highlight implications. Finally, we will describe our new "domain extension" technique for PPRFs and analyze its efficiency.

5.1 Tree-based PPRFs

We will briefly recap the main idea behind tree-based PPRFs. It is well known that the GGM tree-based construction of pseudorandom functions (PRFs) from one-way functions [22] can be modified to construct a puncturable PRF, as noted in [10,11,30]. It works as follows.

Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ be a pseudorandom generator (PRG) and let $G_0(k)$, $G_1(k)$ be the first and second half of string $G(k)$, where k is a random seed. The GGM construction defines a binary tree on the PRF's domain, where each leaf represents an evaluation of the PRF. We label each edge with 0 if it connects to a left child, and 1 if it connects to a right child. We label each node with the binary string determined by the path from the root to the node. The PRF value of $x = x_1 \dots x_n \in \{0,1\}^n$ is $(G_{x_n} \circ \dots \circ G_{x_1})(k) \in \{0,1\}^\lambda$, i.e. we compose G according to the path from root to leaf x.

We will briefly describe how this construction can be transformed into a PPRF. In order to puncture the PPRF at input $x = x_1 \dots x_n$ we compute a tuple of n intermediate node evaluations for prefixes $\overline{x_1}, x_1\overline{x_2}, \dots, x_1x_2 \dots \overline{x_n}$ and discard the initial seed k. The intermediate evaluations enable us to still compute evaluations on all inputs but x. Successive puncturing is possible if we apply the above computations to an intermediate evaluation. Note that we have to compute at most $n \cdot m$ intermediate values if we puncture at random, where m is the number of puncturing operations performed.

The PPRF is secure if an adversary is not able to distinguish between a punctured point and a truly random value, even when given the values of all computed "neighbor nodes". This holds as long as the underlying PRG is indistinguishable from random [10,11,30].

5.2 Combining Tree-based PPRFs with Tickets

In our session resumption scenario the tree-based PPRF will act as a puncturable STEK. That is, evaluating the PPRF returns a ticket encryption key. Upon resumption with a ticket we will retrieve the ticket encryption key from the PPRF by evaluating it and puncture the PPRF at that very value to ensure the ticket encryption key cannot be computed twice. Note that each ticket encryption key essentially corresponds to a leaf of the tree. Thus we will subsequently use the terms leaf and ticket (encryption key) interchangeably depending on the context.

For simplicity, we consider tickets which consist of a ticket number i and a ticket lifetime t. Following Construction 1 we will issue the tickets one after another while incrementing the ticket number for each. Note that the ticket number i corresponds to

the i-th leftmost leaf of the tree. The ticket lifetime t determines how long an issued ticket is valid for resumption. That is, if $t' > t$ time has passed, the server will reject the ticket.

We assume that the rate at which tickets are issued is roughly the same as the rate tickets are used for session resumption. This holds as for each session resumption we will issue a new ticket to again resume the session at a later point in time. Similarly, we argue that tickets are roughly used in the same order for resumption as we issued them. Again, if we consider multiple users, repeatedly requesting tickets and resuming sessions, we are able to average the time a user takes until a session is resumed.[10] This yields an implicit window of tickets in usage. The window is bounded left by the ticket lifetime and bounded right by the last ticket the server issued. Within the lifetime of the tree-based PPRF this implicit window will shift from left to right over the tree's leaves. It immediately follows that tickets are also roughly used in that order.

5.3 Efficiency Analysis of the Tree-based PPRF

We will now discuss how the performance of tree-based PPRFs depends on the ticket lifetime. We consider a scenario where the ticket lifetime t equals the number of leaves ℓ. It is also possible to consider a scenario where the ticket lifetime is smaller than the number of leaves. If both number of leaves ℓ and ticket lifetime t are powers of 2, we can divide the leaves in ℓ/t windows, which span a subtree each.[11] The subtrees are all linked with the "upper part" of the tree. A different approach would be to instantiate a new tree when a tree runs out of tickets. We stress that this does not affect our analysis. As soon as one subtree runs out of tickets, the next subtree is used. If the rate at which we issue tickets stays the same, we are able to delete parts of the former tree when issuing tickets of the next one. Hence, for analysis, it is sufficient to consider a single tree.

If we were to puncture leaves strictly from left to right, we would need to store at most $\log(\ell)$ leaves (one leaf per layer). Note that if we puncture leaves at random, we would need to store at most $p \cdot \log(\ell)$ nodes, where p is the number of punctures performed. We can also bound the number of nodes we need to store by $p \cdot \log(\ell) \leq \ell/2$. This is due to the tree being binary. Essentially each node (except for the lowest layer) represents at least two leaves. To be more precise, in a tree with L layers, storing a node on layer i allows evaluating its 2^{L-i} children. Thus it is preferable to store those nodes instead of storing leaves in order to save memory. In the worst-case only every second leaf is punctured. This results in precomputation of all other leaves without being able to save memory by only storing an intermediary node. Note that this would actually resemble a Session Cache, where all issued tickets are stored. However, note

[10] Cloudflare have suggested that these assumptions seem reasonable. Unfortunately, they cannot provide data on returning clients' behavior yet.

[11] When implementing tree-based PPRFs in session resumption scenarios, such windows should not be implemented as they only add management overhead to the algorithm instead of providing notable advantages. It is sufficient to use a tree-based PPRF as is and puncture leaves for which the ticket's lifetime has expired. This way we achieve an implicit implementation of a sliding window scenario that ensures all established bounds still hold.

that a session cache needs to store each ticket when it has been issued, whereas our construction only needs to increase its storage if a ticket is used for resumption. Thus, our tree-based construction performs (memory-wise) at least as well as a Session Cache. In practice, where user behavior is much more random, our approach is *always* better than Session Caches.

The tree-based PPRF performs more computations compared to a Session Cache. When issuing tickets we need to compute all nodes from the closest computed node to a leaf. For puncturing we need to compute the same, plus computation of some additional sibling nodes. However, when instantiating the construction with a cryptographic hash function, such as SHA-3, evaluation and puncturing of the PPRF consists only of several hash function evaluations. This makes our construction especially suitable for high-traffic scenarios.

Table 3 gives worst-case secret key sizes based on the above analysis. However, we expect the secret key size to be much smaller in practice. Unfortunately, we are not able to estimate the average key size as this would depend on the exact distribution of returning clients' arrival times.

Tickets per Second r	Ticket Lifetime t	Worst-case Secret Key Size $\|k\|$
16	1 hour	461 kB
16	1 day	11.06 MB
128	1 hour	3.69 MB
128	1 day	88.47 MB
1024	1 hour	29.49 MB
1024	1 day	707.79 MB

Table 3. Worst-case size of secret key depending on the rate of tickets per second and the ticket lifetime assuming 128 bit ticket size. The worst-case secret key size is computed as $|k| = 128rt/2$.

5.4 Generic Domain Extension for PPRFs

Most forward-secure and replay-resilient 0-RTT schemes come with large secret keys (possibly several hundred megabytes) when instantiated in a real-world environment [15,16,25]. This is especially problematic if the secret key needs to be synchronized across multiple server instances. Therefore it is often desirable to minimize the secret key size.

In this section we will describe a generic domain extension. In the context of our work, the domain extension reduces the size of punctured keys by trading secret key size for ticket size, while preserving the puncturing functionality.

Idea Behind the Construction. Our session resumption protocol uses the output of the PPRF as a ticket encryption key. Normally, a PPRF only allows one output per input as it is designed to be a function. Our protocol, however, does not rely on this property. Instead of only using one ticket encryption key we could generate multiple ticket encryption keys. Ticket issuing would work as follows. First, we generate an interme-

diary symmetric key to encrypt the Resumption Secret[12]. The intermediary symmetric key is then encrypted under each of the ticket encryption keys. The ticket will consist of one encryption of the Resumption Secret and several (redundant) encryptions of the intermediary symmetric key.

As long as the PPRF is able to recompute *at least one* of those ticket encryption keys, the server will still be able to resume the session. This allows us to construct a wrapper around the PPRF that extends the PPRF's domain by relaxing the requirement that every input has only a single output.

Before formally describing our construction, we will provide an example to illustrate the idea. Let $\mathcal{X}$ be the PPRF's domain. We will extend the domain to $\mathcal{X} \times [n]$ with a domain extension factor of n. That is, we will allow $(x, i), i \in [n]$ for any $x \in \mathcal{X}$ as input. Let $G : \{0,1\}^\lambda \to \{0,1\}^{n\lambda}$ be a pseudorandom generator and let $G_j(x)$ be the j-th bitstring of size λ of G on input x. We define the evaluation of (x, i) as all possible compositions of G_j which end with G_i. That is, for any input (x, i) there will be $(n-1)!$ different outputs, as there are $(n-1)!$ ways to compose G_j with $j \neq i$. The possible compositions of PRGs can be illustrated as a tree as shown in Figure 4.

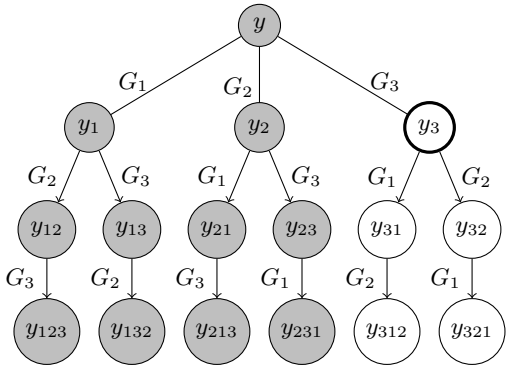

Fig. 4. Possible composition of PRGs for $n = 3$ illustrated as a tree. Each path from parent to child illustrates an evaluation of the PRG shown next to the path. Upon puncturing $(x, 3)$, the value y_3 is computed and stored and y is discarded. Thus, only the white nodes are computable, whereas the gray nodes cannot be computed without inverting G_3.

After puncturing the PPRF's key for a value (x, i), it must not be possible to evaluate the value anymore. This requires a mechanism to ensure that composing the PRGs which end with G_i is no longer possible. We achieve this by forcing an evaluation of $y_i := G_i(y)$, where y is the evaluation of the underlying PPRF on input x. In order to render recomputation of y impossible, we additionally need to puncture the PPRF's key on value x and delete the computed y. Formally, the construction is defined as follows.

[12] Typically, a ticket contains not only the Resumption Secret but also the chosen cipher suite and other additional session parameters, and is thus larger than just the Resumption Secret. Therefore it is reasonable to encrypt this data only once, while encrypting the shorter intermediary symmetric key multiple times. This makes the ticket as short as possible.

Construction 3. *Let* $G : \{0,1\}^\lambda \to \{0,1\}^{n\lambda}$ *be a PRG and let* $G_i(k)$ *be the* i*-th bitstring of size* λ *of* G*. Let* $\mathsf{PPRF}' = (\mathsf{Setup}', \mathsf{Eval}', \mathsf{Punct}')$ *be a PPRF with domain* $\mathcal{X}$*. We construct a domain extended PPRF* $\mathsf{DE} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Punct})$ *with domain* $\mathcal{X} \times [n]$ *for* $n \in \mathbb{N}$ *as follows.*

- $\mathsf{Setup}(1^\lambda)$ *computes* $k_{\mathsf{PPRF}} := \mathsf{Setup}'(1^\lambda)$*. Next, it defines an empty list* $\mathcal{L} = \emptyset$*. Output is* $k = (k_{\mathsf{PPRF}}, \mathcal{L})$
- $\mathsf{Eval}(k, x)$ *parses* $x = (x_{\mathsf{PPRF}}, x_{\mathsf{ext}}) \in \mathcal{X} \times [n]$ *and* $k = (k_{\mathsf{PPRF}}, \mathcal{L})$*. It computes* $y := \mathsf{Eval}(k_{\mathsf{PPRF}}, x_{\mathsf{PPRF}})$.
  If $y = \bot$*, it checks whether* $\exists x_{\mathsf{PPRF}}$ *with* $(x_{\mathsf{PPRF}}, y', (r_1, \ldots, r_n)) \in \mathcal{L}$*. If it exists, assign* $y := y'$*. Otherwise it outputs* $\bot$*.*
  Furthermore, it defines a set $\mathcal{R} = \{i \in [n] \mid r_i = 1\}$*. If* r_i *are undefined, set* $\mathcal{R}$ *is empty. Next, it computes*

$$\mathcal{Y} = \{(G_{i_{n-|\mathcal{R}|-1}} \circ \ldots \circ G_{i_1})(y)\},$$

  where $(i_1, \ldots, i_{n-|\mathcal{R}|-1})$ *are all* $(n - |\mathcal{R}| - 1)!$ *possible permutations of elements in* $[n] \setminus (\mathcal{R} \cup \{x_{\mathsf{ext}}\})$*. Output is* $\mathcal{Y}$*.*
- $\mathsf{Punct}(k, x)$ *parses* $k = (k_{\mathsf{PPRF}}, \mathcal{L})$ *and* $x = (x_{\mathsf{PPRF}}, x_{\mathsf{ext}}) \in \mathcal{X} \times [n]$*. It computes* $y := \mathsf{Eval}(k_{\mathsf{PPRF}}, x_{\mathsf{PPRF}})$*. If* $y \neq \bot$*, it appends* $\mathcal{L}' = \mathcal{L} \cup \{x_{\mathsf{PPRF}}, y, (r_1, \ldots, r_n)\}$*, where* $r_i = 0$*, but* $r_{x_{\mathsf{ext}}} = 1$*. Additionally, it punctures* $k'_{\mathsf{PPRF}} := \mathsf{Punct}'(k_{\mathsf{PPRF}}, x_{\mathsf{PPRF}})$*.*
  If $y = \bot$ *and* $\nexists x_{\mathsf{ext}}$ *with* $(x_{\mathsf{ext}}, y', r) \in \mathcal{L}$*, it outputs* k*.*
  Otherwise it retrieves $\ell = (x_{\mathsf{ext}}, y', (r_1, \ldots, r_n)) \in \mathcal{L}$*. If* $r_i = 1$ *for all* $i \in [n] \setminus \{x_{\mathsf{ext}}\}$*, remove* ℓ *from* $\mathcal{L}$*. Else it updates* $\ell \in \mathcal{L}$ *by computing* $\mathcal{L}' = (\mathcal{L} \setminus \{\ell\}) \cup \{\ell'\}$*.*
  Output is $k = (k'_{\mathsf{PPRF}}, \mathcal{L}')$*.*

5.5 Efficiency Analysis of the Generic Domain Extension

Increased Ticket Size. Note that a ticket is longer than a standard ticket by $(n-1)!$ encrypted blocks. Assuming 128-bit AES, and choosing $n = 5$, this translates to $4! \cdot 16 = 384$ additional bytes. This is likely to be insignificant on the modern Internet. For example, Google has pushed for increasing the maximum initial flight from 4 TCP packets to 10 [19], as most server responses span several packets already (a typical full packet is about 1500 bytes). A basic experiment performed by Google and Cloudflare in 2018 measured a similar scenario: It added 400 bytes for both the client's and server's first flights [33]. They observed relatively small additional latencies: 2–4 milliseconds in the median, and less than 20 milliseconds for the 95th percentile.[13] However, choosing $n = 6$ or larger is likely to be not cost-effective. This would translate to $5! \cdot 16 = 1920$ additional bytes, larger than a standard TCP packet.

Storage Requirements. Comparing the storage requirements of the tree-based construction to standard Session Caches depends on the specific distribution of returning clients. In the best case, tickets arrive in large contiguous blocks. In this case, a tree-based construction uses negligible storage (logarithmic in the number of tickets), making the savings factor in storage huge. However, this is unrealistic in practice. In the worst-case,

[13] The relevant experiment is denoted as "Phase Two"; "Phase One" only added bytes to the client's first flight.

tickets arrive in blocks of $n-1$ tickets of the form (x_{PPRF}, i) for $i \in [n-1]$, adversarially rendering the domain extension technique useless as each subtree is reduced to a single node. As before, this is unrealistic in practice.

We have therefore resorted to simulations in order to assess the improvement in storage requirements. Our simulation constructs two trees: a standard binary tree with ℓ layers, and a domain-extended tree with $n = 4$. For the domain-extended tree, the first $\ell - 2$ layers are constructed as a standard binary tree, and the last $\log(4) = 2$ layers are represented by the domain extension.

We simulated the storage requirements for trees of 10,000 tickets.[14] We focused on the relationship between ticket puncturing rate and savings in storage. The ticket puncturing rate denotes the percentage of tickets that are punctured, out of the 10,000 outstanding tickets. This can also be thought of as the percentage of returning clients. After fixing the puncturing rate to r, we simulate the arrival of $r\%$ of clients according to two distributions: Gaussian and uniform. With the uniform distribution, the next ticket to be punctured is sampled uniformly out of the outstanding tickets. With the Gaussian distribution, the next ticket to be punctured is sampled using a discrete Gaussian distribution with mean $\mu = 5000$ and standard deviation σ (for varying values of σ). We then simulate the state of both trees after puncturing the sampled ticket. We repeatedly sample tickets and puncture them, until we reach the desired puncturing rate. We then report the ratio between the storage for the standard binary tree and the storage for the domain-extended tree, in their final states.

Intuitively, the Gaussian distribution aims to simulate the assumption where tickets arrive in some periodic manner. For example, assume the tickets most likely to arrive are the tickets issued roughly one hour ago. Then the distribution of arriving tickets will exhibit a noticeable mode ("peak"), where tickets close to the mode are much more likely to arrive than tickets far from it. The Gaussian distribution is a natural fit for this description. On the other hand, the uniform distribution makes no assumptions on which ticket is likely to arrive next. In personal communication, Cloudflare have advised us that it is reasonable to assume tickets are redeemed roughly in order of issuance (they do not have readily-available data on returning clients' behavior). This motivated our use of Gaussian distributions. We hope to see additional research in this area. In particular, it would be helpful if large server operators could release anonymized datasets that allow simulating the behavior of returning clients in practice.

Using our domain extension technique with $n = 4$ results in a typical factor of 1.4 (or more) reduction in storage compared to a tree-based PPRF. Figure 5 plots the results when using the uniform distribution and a Gaussian distribution with $\sigma = 2000$. We encountered similar results when using other values for σ. We estimate ticket redeeming rates in large-scale deployments are roughly 50%. We therefore focus on cases where the puncturing rate is at least 40% and at most 60%. We note that in the worst-case, the domain extension performs as well as the binary tree.

[14] We note that results for trees of 10,000 tickets should closely follow results for larger tree sizes. Trees are quickly split into smaller sub-trees when puncturing, regardless of the initial tree size. In the first puncturing operation we delete the root and store smaller sub-trees with at most half the nodes in each, and so forth.

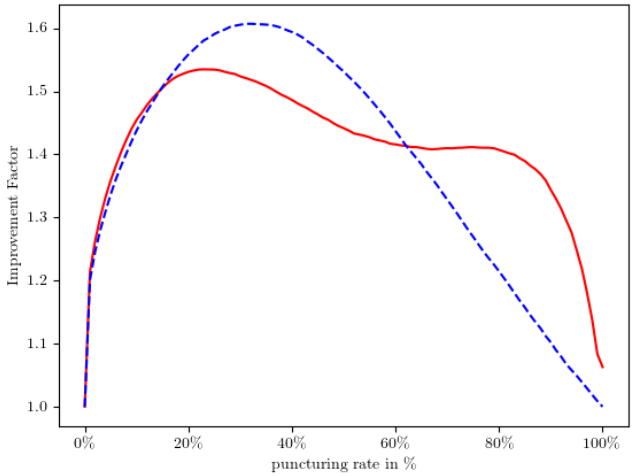

Fig. 5. Average storage improvement factor of the domain-extended binary tree (with $n = 4$) compared to a standard binary tree, depending on the ticket puncturing rate. All simulations used trees of 10,000 tickets. The dashed blue line (resp. continuous red line) shows the storage improvement when modeling client's arrivals with a uniform distribution (resp. discrete Gaussian distribution with mean $\mu = 5000$ and standard deviation $\sigma = 2000$).

6 Comparison of Solutions and Conclusion

Comparison of Solutions. To summarize this work, Table 4 compares our two constructions with the standard solutions of Session Tickets and Session Caches.

Solution	Forward Security	Replay Protection	Storage per Ticket	Dominant Cost	See Section
Session Tickets	After $\approx$ 1 day	No	Negligible	Symmetric encryption	1
Session Caches	Yes	Yes	$\approx$ 20–30 bytes	Database access	1
sRSA-based PPRF	Yes	Yes	$\approx$ 0.8–1.2 bytes	Group exponentiation	4.3
Tree-based PPRF	Yes	Yes	$\leq$ 20–30 bytes	Database access	5.3

Table 4. Comparison of security guarantees and dominant cost for Session Tickets, Session Caches, and our two constructions. For Session Tickets, we assume a deployment that rotates STEKs, as in [34]. For Session Caches, we assume each key is 128 bits (16 bytes) long. The unique ticket identifier, and other storage overhead, will typically require a few more bytes. We therefore estimate total storage per key as 20–30 bytes. For the Tree-based PPRF, actual storage per ticket highly depends on returning clients behavior. However, this solution always requires at most as much storage as a Session Cache.

Conclusion. In most facets, TLS 1.3 offers significant improvements in security compared to earlier TLS versions. However, when 0-RTT mode is used, it surprisingly weakens standard security guarantees, namely forward security and replay resilience. This was noted as the protocol was standardized, but the latency reduction from 0-RTT was considered "too big a win not to do" [38].

This paper presented formal definitions for secure 0-RTT Session Resumption Protocols, and two new constructions that allow achieving the aforementioned security guarantees at a practical cost. We expect continued research in the coming years in this area, of achieving secure 0-RTT traffic as cheaply as possible. Currently, many large server operators serve 0-RTT traffic using STEK-encrypted Session Tickets. As more Internet traffic becomes 0-RTT traffic, this solution rolls back the security guarantees offered to everyday secure sessions.

References

1. Aviram, N., Gellert, K., Jager, T.: Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. Cryptology ePrint Archive (2019), https://eprint.iacr.org
2. Bari, N., Pfitzmann, B.: Collision-free accumulators and fail-stop signature schemes without trees. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 480–494. Springer, Heidelberg, Germany, Konstanz, Germany (May 11–15, 1997)
3. Barker, E.: Recommendation for key management part 1: General (revision 4). NIST special publication (2016)
4. Behr, M., Swett, I.: Introducing QUIC support for HTTPS load balancing (2018), https://cloudplatform.googleblog.com/2018/06/Introducing-QUIC-support-for-HTTPS-load-balancing.html
5. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Ashby, V. (ed.) ACM CCS 93. pp. 62–73. ACM Press, Fairfax, Virginia, USA (Nov 3–5, 1993)
6. Bellare, M., Stepanovs, I., Tessaro, S.: Poly-many hardcore bits for any one-way function and a framework for differing-inputs obfuscation. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 102–121. Springer, Heidelberg, Germany, Kaoshiung, Taiwan, R.O.C. (Dec 7–11, 2014)
7. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zanella Béguelin, S.: Proving the TLS handshake secure (as it is). In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 235–255. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2014)
8. Blum, L., Blum, M., Shub, M.: A simple unpredictable pseudo-random number generator. SIAM J. Comput. 15(2), 364–383 (1986), https://doi.org/10.1137/0215025
9. Böck, H.: Fuzz-compare the OpenSSL function BN_mod_exp() and the libgcrypt function gcry_mpi_powm(), https://github.com/hannob/bignum-fuzz/blob/master/openssl-vs-gcrypt-modexp.c
10. 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, Germany, Bengalore, India (Dec 1–5, 2013)
11. 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, Germany, Buenos Aires, Argentina (Mar 26–28, 2014)

12. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2002)
13. Chang, W.T., Langley, A.: QUIC crypto (2014), https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g
14. Cremers, C., Horvat, M., Scott, S., van der Merwe, T.: Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In: 2016 IEEE Symposium on Security and Privacy. pp. 470–485. IEEE Computer Society Press, San Jose, CA, USA (May 22–26, 2016)
15. Derler, D., Gellert, K., Jager, T., Slamanig, D., Striecks, C.: Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. Cryptology ePrint Archive, Report 2018/199 (2018), https://eprint.iacr.org/2018/199
16. 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, Germany, Tel Aviv, Israel (Apr 29 – May 3, 2018)
17. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In: Ray, I., Li, N., Kruegel:, C. (eds.) ACM CCS 15. pp. 1197–1210. ACM Press, Denver, CO, USA (Oct 12–16, 2015)
18. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081 (2016), http://eprint.iacr.org/2016/081
19. Dukkipati, N., Refice, T., Cheng, Y., Chu, J., Herbert, T., Agarwal, A., Jain, A., Sutin, N.: An argument for increasing TCP's initial congestion window. Computer Communication Review 40(3), 26–33 (2010)
20. 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 14. pp. 1193–1204. ACM Press, Scottsdale, AZ, USA (Nov 3–7, 2014)
21. Fischlin, M., Günther, F.: Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. pp. 60–75. IEEE (2017), https://doi.org/10.1109/EuroSP.2017.18
22. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM 33(4), 792–807 (Aug 1986), http://doi.acm.org/10.1145/6490.6503
23. Goldreich, O., Levin, L.A.: A hard-core predicate for all one-way functions. In: 21st ACM STOC. pp. 25–32. ACM Press, Seattle, WA, USA (May 15–17, 1989)
24. 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, San Jose, CA, USA (May 17–21, 2015)
25. Günther, F., Hale, B., Jager, T., Lauer, S.: 0-RTT key exchange with full forward secrecy. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 519–548. Springer, Heidelberg, Germany, Paris, France (Apr 30 – May 4, 2017)
26. Hale, B., Jager, T., Lauer, S., Schwenk, J.: Simple security definitions for and constructions of 0-RTT key exchange. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 17. LNCS, vol. 10355, pp. 20–38. Springer, Heidelberg, Germany, Kanazawa, Japan (Jul 10–12, 2017)
27. Iyengar, S., Nekritz, K.: Building zero protocol for fast, secure mobile connections (2017), https://code.fb.com/android/building-zero-protocol-for-fast-secure-mobile-connections/
28. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 273–293. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012)

29. Kario, H.: Add 3072, 7680 and 15360 bit RSA tests to openssl speed, https://groups.google.com/forum/#!topic/mailing.openssl.dev/bv8t7QcXrqg
30. 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 13. pp. 669–684. ACM Press, Berlin, Germany (Nov 4–8, 2013)
31. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 429–448. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2013)
32. Langley, A.: How to botch TLS forward secrecy (2013), https://www.imperialviolet.org/2013/06/27/botchingpfs.html
33. Langley, A.: Post-quantum confidentiality for TLS (2018), https://www.imperialviolet.org/2018/04/11/pqconftls.html
34. Lin, Z.: TLS Session Resumption: Full-speed and Secure (2015), https://blog.cloudflare.com/tls-session-resumption-full-speed-and-secure/
35. Lychev, R., Jero, S., Boldyreva, A., Nita-Rotaru, C.: How secure and quick is QUIC? Provable security and performance analyses. In: 2015 IEEE Symposium on Security and Privacy. pp. 214–231. IEEE Computer Society Press, San Jose, CA, USA (May 17–21, 2015)
36. MacCarthaigh, C.: Security Review of TLS 1.3 0-RTT. https://github.com/tlswg/tls13-spec/issues/1001, accessed: 2018-07-29
37. Rescorla, E.: TLS 0-RTT and Anti-Replay (2015), https://www.ietf.org/mail-archive/web/tls/current/msg15594.html
38. Rescorla, E.: TLS 1.3 (2015), http://web.stanford.edu/class/ee380/Abstracts/151118-slides.pdf
39. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (2018), https://rfc-editor.org/rfc/rfc8446.txt
40. Rogaway, P.: Authenticated-encryption with associated-data. In: Atluri, V. (ed.) ACM CCS 02. pp. 98–107. ACM Press, Washington D.C., USA (Nov 18–22, 2002)
41. 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, New York, NY, USA (May 31 – Jun 3, 2014)
42. Shamir, A.: On the generation of cryptographically strong pseudorandom sequences. ACM Trans. Comput. Syst. 1(1), 38–44 (Feb 1983), http://doi.acm.org/10.1145/357353.357357
43. Springall, D., Durumeric, Z., Halderman, J.A.: Measuring the security harm of TLS crypto shortcuts. In: Proceedings of the 2016 Internet Measurement Conference. pp. 33–47. ACM (2016)
44. Sullivan, N.: Introducing Zero Round Trip Time Resumption (2017), https://blog.cloudflare.com/introducing-0-rtt/
45. The OpenSSL Project: OpenSSL: The open source toolkit for SSL/TLS, www.openssl.org