On Secure Ratcheting with Immediate Decryption[⋆]

Jeroen Pijnenburg[1] and Bertram Poettering[2] [0000−0001−6525−5141]

[1] Royal Holloway, University of London, Egham Hill, Egham, Surrey, United Kingdom
[2] IBM Research Europe – Zurich, Säumerstr 4, 8803 Rüschlikon, Switzerland
poe@zurich.ibm.com

Abstract. Ratcheting protocols let parties securely exchange messages in environments in which state exposure attacks are anticipated. While, unavoidably, some promises on confidentiality and authenticity cannot be upheld once the adversary obtains a copy of a party's state, ratcheting protocols aim at confining the impact of state exposures as much as possible. In particular, such protocols provide *forward security* (after state exposure, past messages remain secure) and *post-compromise security* (after state exposure, participants auto-heal and regain security).

Ratcheting protocols serve as core components in most modern instant messaging apps, with billions of users per day. Most instances, including Signal, guarantee *immediate decryption* (ID): Receivers recover and deliver the messages wrapped in ciphertexts immediately when they become available, even if ciphertexts arrive out-of-order and preceding ciphertexts are still missing. This ensures the continuation of sessions in unreliable communication networks, ultimately contributing to a satisfactory user experience. While initial academic treatments consider ratcheting protocols without ID, Alwen *et al.* (EC'19) propose the first ID-aware security model, together with a provably secure construction. Unfortunately, as we note, in their protocol a receiver state exposure allows for the decryption of all prior *undelivered* ciphertexts. As a consequence, from an adversary's point of view, intentionally preventing the delivery of a fraction of the ciphertexts of a conversation, and corrupting the receiver (days) later, allows for correctly decrypting all suppressed ciphertexts. The same attack works against Signal.

We argue that the level of (forward-)security realized by the protocol of Alwen *et al.*, and mandated by their security model, is considerably lower than both intuitively expected and technically possible. The main contributions of our work are thus a careful revisit of the security notions for ratcheted communication in the ID setting, together with a provably secure proof-of-concept construction. One novel component of our model is that it reflects the progression of *physical time*. This allows for formally requiring that (undelivered) ciphertexts automatically *expire* after a configurable amount of time.

[⋆] Please find the full version at https://ia.cr/2022/995.

1 Introduction

We consider a communication model between two parties, Alice and Bob, as it occurs in real-world instant messaging (e.g., in smartphone-based apps like Signal). A key principle in this context is that the parties are only very loosely synchronized. For instance, a "ping-pong" alteration of the sender role is not assumed but parties can send concurrently, i.e., whenever they want to. Further, specifically in phone-based instant messaging, a generally unpredictable network delay has to be tolerated: While some messages are received split seconds after they are sent, it may happen that other messages are delivered only with a considerable delay.[3] We refer to this type of communication (with no enforced structure and arbitrary network delays) as *asynchronous*. We say that asynchronous communication has *in-order* delivery if messages always arrive at the receiver in the order they were sent (what Alice sends first is received by Bob before what she sends later); otherwise, if in-order delivery cannot be guaranteed by the network, we say that the communication has *out-of-order* delivery.

The central cryptographic goals in instant messaging are that the confidentiality and integrity of messages are maintained. As communication sessions are routinely long-lived (e.g., go on for months), and as mobile phones are so easily lost, stolen, confiscated, etc., the resilience of solutions against *state exposure attacks* has been accepted as pivotal. In such an attack, the adversary obtains a full copy of the attacked user's program state.[4] We say that a protocol provides *forward security* if after a state exposure the already exchanged messages remain secure (in particular confidential), and we say that it provides *post-compromise security* if after a state exposure the attacked participant heals automatically and regains full security.

Past research efforts succeeded with proposing various security models and constructions for the (in-order) asynchronous communication setting with state exposures [14,26,10,17,18,24,19]. The rule of thumb "the stronger the model the more costly the solution" applies also to the ratcheting domain, and the indicated works can be seen as positioned at different points in the security-vs-cost trade-off space. For instance, the security models of [17,24] are the strongest (for excluding no attacks beyond the trivial ones) but seem to necessitate HIBE-like building blocks [6], while [10,14,18] work with a relaxed healing requirement (either parties do not recover completely or recovery is delayed) that can be satisfied with DH-inspired constructions.

While the works discussed above exclusively consider communication with in-order delivery, popular instant-messaging solutions like Signal are specifically designed to tolerate out-of-order delivery [22, Sec. 2.6] in order to best deal with

---

[3] E.g., delays of hours can occur if a phone is switched off over night or during a long-distance flight.

[4] Program states could leak because of malware executed on the user's phone, by analyzing backup images of a phone's memory that are stored insufficiently encrypted in the cloud, by analyzing memory residues on swap drives, etc. Less technical conditions include that users are legally or illegally coerced to reveal their states.

the needs of users who want to effectively communicate despite temporary network outages, radio dead spots, etc. Given this means that the protocols cannot rely on ciphertexts arriving in the order they were sent, let alone that they arrive at all, the *immediate decryption* (**ID**) property of such protocols demands that independently of the order in which ciphertexts are received, and independently of the ciphertexts that might still be missing, any ciphertext shall be decryptable for immediate display in the moment it arrives.[5] The ID property received first academic attention in an article by Alwen, Coretti, and Dodis (**ACD**) [2]. As the authors point out, while virtually all practical secure messaging solutions do support ID, most rigorous treatments do not. The work of ACD aims at closing this gap. We revisit and refine their results.

The main focus of ACD is on the Double Ratchet (**DR**) primitive which is one of the core components of the Signal protocol [22,13]. DR was specifically developed to allow for simultaneously achieving forward and post-compromise security in ID-supporting instant messaging. ACD contribute a formal security model for this primitive and detail how instant messaging can be constructed from it. This approach, taken by itself, does not guarantee that their solution is secure also in an intuitive sense: As everywhere else in cryptography, if a model turns out to be weak in practical cases, so may be the protocols implementing it. Indeed, we identified an attack that should not be successful against a secure ID-supporting instant messaging protocol, yet if applied against the ACD protocol (or Signal) it leads to the full decryption of arbitrarily selected ciphertexts.

Our attack is surprisingly simple: Assume Alice encrypts, possibly spread over a timespan of months, a sequence of messages $m_1, \ldots, m_L$ and sends the resulting ciphertexts $c_1, \ldots, c_L$ to Bob. An adversary that is interested in learning the target message m_1, arranges that all ciphertexts with exception of c_1 arrive at their destination. By the ID property, Bob decrypts the ciphertexts $c_2, \ldots, c_L$ delivered to him and recovers the messages $m_2, \ldots, m_L$. Further, expecting that the missing c_1 is eventually delivered, he consciously remains in the position to eventually decrypt c_1. But if Bob can decrypt c_1, the adversary, after obtaining Bob's key material via a state exposure, can decrypt c_1 as well, revealing the target message m_1. Note that the attack is not restricted to targeting specifically the first ciphertext; it would similarly work against any other ciphertext, or against a selection of ciphertexts, and the adversary would in all cases fully recover the target messages from just one state exposure. That is, for an adversary who wants to learn specific messages of a conversation secured with Signal or the protocol of ACD, it suffices to suppress the delivery of the corresponding ciphertexts and arrange for a state exposure at some later time. This obviously contradicts the spirit of FS.

Main Conceptual Contributions. Our attack seems to indicate that the immediate decryption (ID) and forward security (FS) goals, by their very nature, are mutually exclusive, meaning that one can have the one or the other, but not both. Our interpretation is less black and white and involves refining both the ID and the FS notions. We argue that, while out-of-order delivery and ID features

[5] In the user interface, placeholders could indicate messages that are still missing.

are indeed necessary to deal with unreliable networks, it also makes sense to put a cap on the acceptable amount of transmission delay. For concreteness, let threshold δ specify a maximum delay that messages traveling on the network may experience (including when transmissions are less reliable). Then ciphertexts that are sent at a time t_1 and arrive at a time t_2 should be deemed useful and decryptable only if $\Delta(t_1, t_2) \leq \delta$, while they should be considered expired and thus disposable if $\Delta(t_1, t_2) > \delta$. Once a threshold δ on the delay is fixed, the ID notion can be weakened to demand the correct decryption of ciphertexts only if the latter are at most δ old, and the FS notion can be weakened to protect past messages under state exposure only if they are older than δ (or already have been decrypted). As we show, once the two notions have been weakened in this sense, they fit together without contradicting each other. That is, this article promotes the idea of integrating a notion of progressing physical time into the ID and FS definitions so that their seemingly inherent rivalry is resolved and one can have both properties at the same time.

Our models and constructions see δ as a configurable parameter. The value to pick depends on the needs of the participants. For instance, if Alice and Bob are political activists operating under an oppressive regime, choosing $\delta < 10$ mins might be useful; more relaxed users might want to choose $\delta = 1$ week. Note that for $\delta = \infty$ our definitions 'degrade' to the no-expiration setting of ACD.

Main Technical Contributions. We start with a compact description of our three main technical contributions. We expand on the topics subsequently.

In a nutshell, the contributions of this article are: (1) We introduce the concept of evolving physical time to formal treatments of secure messaging. This allows us to express requirements on the automatic expiration of ciphertexts after a definable amount of time. (2) We propose new security models for secure messaging with immediate decryption (ID). Our approach is to have the security definitions disregard the unavoidable trivial attacks but nothing else; this renders our models particularly strong. By incorporating the progressing of physical time into our notions, our FS and ID definitions are not in conflict with each other. (3) We contribute a proof-of-concept protocol that provably satisfies our security notions. Efficiency-wise our protocol might be less convincing than the ACD protocol and Signal, but it is definitely more secure.

(1) Modeling physical time. Among the many possible approaches to formalizing evolving physical time, the likely most simple option is sufficient for our purposes. In our treatments we assume that participants have access to a local clock device that notifies them periodically through events referred to as *ticks* about the elapse of a configurable amount of time.[6] The clocks of all participants are expected to be configured to the same ticking frequency (e.g., one tick every one minute), but otherwise our synchronization demands are very moderate: The only aspect relevant for us is that when Alice sends a ciphertext at a time t_1 (according to her clock) and Bob receives the ciphertext at a time t_2 (accord-

[6] Modern computing environments provide such a service right away. For instance, in Linux, via the `setitimer` system call or the `alarm` standard library function.

ing to his clock), then we expect that the difference $\Delta(t_1, t_2)$ be meaningful to declare ciphertexts fresh or expired. More precisely, we deem ciphertexts with $\Delta(t_1, t_2) \leq \delta$, for a configurable threshold δ, fresh and thus acceptable, while we consider all other ciphertexts expired and thus discardable. Note here that threshold δ specifies both a maximum on the tolerated network delay and on a possibly emerging clock drift between the sender's and the receiver's clock. The right choice of threshold δ is an implementation detail which controls the robustness-security trade-off.[7] See above for a discussion on how to choose δ.

(2) Security models. We develop security models for secure messaging with out-of-order delivery and immediate decryption (ID). We claim two main improvements over prior definitions: (a) We incorporate physical time into all correctness and security notions. For instance, when formulating the correctness requirements, we do not demand the correct decryption of expired ciphertexts, and our confidentiality definitions deem state exposure based message recovery attacks successful if the targeted ciphertext is expired. (b) We formalize the maximum level of attainable security (under state exposures). Recall that ACD was designed for analyzing Double Ratchet based constructions which were proven to achieve only limited security already in the in-order delivery setting [17,24].[8] In contrast, our models are designed to exclude the unavoidable 'trivial' attacks but nothing else, thus guaranteeing the best-possible security. (In the full version we review examples of such trivial attacks. We also list attacks that are included in our model but excluded by the ACD model.)

(3) Our construction. We propose a proof-of-concept construction that provably satisfies our security definitions. Its cryptographic core is formed by two specialized types of key encapsulation mechanism (KEM): a KeKEM and a KuKEM. In a nutshell, our KeKEM (key-evolving KEM) primitive is a type of KEM where public and secret keys can be linearly updated 'to the next epoch', almost like in forward-secure PKE. In contrast, our KuKEM (key-updatable KEM) primitive allows for updating keys based on provided auxiliary input strings. In both cases, key updates provide forward secrecy, i.e., 'the updates cannot be undone'.[9] Together with additional more standard building blocks like (stateful) signatures,

[7] One might wonder about the resilience of computer clocks against desynchronization attacks where the adversary aims at desynchronizing participants. We note that instant messaging apps are typically run on mobile devices that have access to multiple independent clock sources (e.g., a local clock, NTP, GSM, and GNSS) that can be compared and relied upon when consistent. Only the strongest adversaries can arrange for a common deviation of all these clock sources simultaneously and even in this case our solutions degrade gracefully: If all clocks stop, the security of our solution doesn't degrade below the security defined by ACD.

[8] In a nutshell, DR provides optimal security only if used for ping-pong structured communication [17,24]. In contrast, the constructions of [17,24] provide security for *any* (in-order) communication pattern, though require stronger primitives than DR.

[9] We note that similar KEM variants have been proposed and used in prior work on instant messaging [17,24,6], so in this article we claim novelty for neither the concepts nor the constructions.

we finally obtain a secure instant messaging protocol. In addition to the cryptographic core, a considerable share of our protocol specification is concerned with data management: the KeKEM and KuKEM primitives require that senders and receivers perform their updates in a strictly synchronized fashion; if ciphertexts arrive out of order, careful bookkeeping is required to let the receiver update in the right order and at the right time.

When compared to the constructions of ACD and Signal, our construction is admittedly less efficient, primarily because (a) we employ the KuKEM and KeKEM primitives that seem to require a considerable computational overhead, and (b) the ciphertexts of our protocol are larger. Concerning (a), we note that prior work like [17,24] that achieves strongest possible security for the much less involved in-order instant messaging case uses the same primitives, and that results [6] indicate that their use is actually unavoidable. We conclude from this that the computational overhead that the primitives bring with them seems to represent the due price to pay for the extra security. A similar statement can be made concerning (b): If an instant messaging conversation is such that the sender role strictly alternates between Alice and Bob, then the ciphertext overhead of our protocol, when compared to Signal, is just a couple of bytes per message. If the sender role does not strictly alternate, the ciphertext size grows linearly in the number of messages that the sender still has to confirm to have arrived. Recalling that the non-alternating case is precisely the one where Signal fails to provide best-possible security, the ciphertext overhead seems to be fair given the extra security that is achieved.

Related Work.

We start with providing a more detailed comparison of our results with those of the prior work mentioned above. We first remark that our results generalize the findings of [17,24]: If in our models the physical time is 'frozen', messages are always delivered, and messages are delivered in-order, they express exactly the same security guarantees as [17,24]. It is clear that as soon as time starts ticking our model is stronger: We allow state exposures once ciphertexts 'expire', while this concept does not exist in [17,24]. For out-of-order delivery the picture is more complicated: Note that when messages are delivered in-order, optimal security demands that user states immediately 'cryptographically diverge' when receiving an unauthentic ciphertext, but for out-of-order delivery the situation becomes more nuanced. Consider the scenario where Alice sends a message and is then state-exposed. Using the obtained state information, the adversary could now trivially and perfectly impersonate Alice towards Bob for the second message. That is, if Bob receives the second ciphertext first, there is no (cryptographic) way for him to tell whether it is authentic or not, i.e., to distinguish whether Alice sent or the adversary injected it. If the ciphertext was indeed sent by Alice, correctness would require that Bob remains able to decrypt the first ciphertext. Thus, the latter also has to hold if the ciphertext is unauthentic. Hence, in contrast to the setting with in-order delivery, in the out-of-order setting there are inherent limits to how much the states of Alice and Bob can 'cryptographically diverge' once unauthentic ciphertexts are processed.

Multiple weaker security definitions for secure messaging have been proposed [2,10,14,18]. We provide a brief overview about what makes their security notions suboptimal. In [10,14] the adversary is forbidden to impersonate a user when a secure key is being established. Hence, in this case the authors do not require recovery from a state exposure (which enables an impersonation attack). In [2,18] the construction can take longer than strictly necessary to recover from state exposures. This is encoded in the security games by artificially labeling certain win conditions as trivial. See [9] for an extensive treatment of the limitations of the ACD model. Moreover, in both works the user states are not required to immediately 'cryptographically diverge' for future ciphertexts when accepting an unauthentic ciphertext. We note that an important difference between our KuKEM and Healable key-updating Public key encryption (HkuPke) introduced in [18] is that HkuPke key updates are based on secret update information, while our KuKEM is updated with adversarially controlled associated data.

The security definitions of [2,17,18] assume a slightly different understanding of what it means to expose a participant. Our understanding is that exposures reveal the current protocol state of a participant to the adversary, while their approach is rather that exposures reveal the randomness used for the next sending operation. The two views seem ultimately incomparable, and likely one can find arguments for both sides. One argument that supports our approach is that modern computing environments have RNGs that *constantly* refresh their state based on unpredictable events (e.g., the `RDRAND` instruction of Intel CPUs or the `urandom` device in Linux) so that if one of the situations listed in Footnote 4 leads to a state exposure then it still can be assumed that the randomness used for the next sending operation is indeed safe. A third view considers state exposures to leak a party's state except for signing keys [1], which seems unrealistic (to us).

See [12] for a treatment of secure messaging in the UC setting.

Our work is not the first to consider a notion of physical time in a cryptographic treatment. See [25] for modeling approaches using linear counters, or [11,20,21] for encrypting data 'to the future'.

Recent work in the group messaging setting [4] similarly designs their protocol in a modular way and captures security in game based definitions. A main component, *continuous group key agreement* (CGKA) was first defined in [3] and the analysis of [5] shows, even in the passive case, no known CGKA protocol achieves optimal security without using HIBE.

Organization. This article considers the security and constructions of what we refer to as bidirectional out-of-order messaging protocols, abbreviated BOOM. In Sect. 3 we define the security model. In Sect. 4 we introduce non-interactive components that we employ in our construction. This includes the mentioned KuKEM and KeKEM primitives. In Sect. 5 we finally present our construction.

2 Notation

We write `T` or 1, and `F` or 0, for the Boolean constants True and False, respectively. For $t_1, t_2 \in \mathbb{N}$ we let $\Delta(t_1, t_2) \coloneqq t_2 - t_1$ if $t_1 \leq t_2$ and $\Delta(t_1, t_2) \coloneqq 0$ if $t_1 > t_2$. For $a, b \in \mathbb{N}$, $a \leq b$, we let $[a \mathinner{..} b] \coloneqq \{a, \ldots, b\}$ and $[b] \coloneqq [0 \mathinner{..} b]$ and $[\![a \mathinner{..} b]\!] \coloneqq [a \mathinner{..} (b-1)]$ and $[\![b]\!] \coloneqq [0 \mathinner{..} (b-1)]$. We further write $[\![\infty]\!]$ for the set of natural numbers $\mathbb{N} = \{0, \ldots\}$. Note that $[\![0]\!]$ represents the empty set.

We specify scheme algorithms and security games in pseudocode. In such code we write $\mathit{var} \leftarrow \mathit{exp}$ for evaluating expression *exp* and assigning the result to variable *var*. If *var* is a set variable and *exp* evaluates to a set, we write $\mathit{var} \overset{\cup}{\leftarrow} \mathit{exp}$ shorthand for $\mathit{var} \leftarrow \mathit{var} \cup \mathit{exp}$ and $\mathit{var} \overset{\cap}{\leftarrow} \mathit{exp}$ shorthand for $\mathit{var} \leftarrow \mathit{var} \cap \mathit{exp}$. A vector variable can be appended to another vector variable with the concatenation operator $\shortparallel$, and we write $\mathit{var} \overset{\shortparallel}{\leftarrow} \mathit{exp}$ shorthand for $\mathit{var} \leftarrow \mathit{var} \shortparallel \mathit{exp}$. We do *not* overload the $\shortparallel$ operator to also indicate string concatenation, i.e., the objects $\mathtt{a} \shortparallel \mathtt{b}$ and `ab` are not the same. We use [ ] notation for associative arrays (i.e., the 'dictionary' data structure): Once the instruction $A[\cdot] \leftarrow \mathit{exp}$ initialized all items of array A to the default value *exp*, individual items can be accessed as per $A[\mathit{idx}]$, e.g., updated and extracted via $A[\mathit{idx}] \leftarrow \mathit{exp}$ and $\mathit{var} \leftarrow A[\mathit{idx}]$, respectively, for any expression *idx*.

Unless explicitly noted, any scheme algorithm may be randomized. We use $\langle\,\rangle$ notation for stateful algorithms: If *alg* is a (stateful) algorithm, we write $y \leftarrow \mathit{alg}\langle \mathit{st}\rangle(x)$ shorthand for $(\mathit{st}, y) \leftarrow \mathit{alg}(\mathit{st}, x)$ to denote an invocation with input x and output y that updates its state *st*. (Depending on the algorithm, x and/or y may be missing.) Importantly, and in contrast to most prior works, we assume that *any* algorithm of a cryptographic scheme may fail or abort, even if this is not explicitly specified in the syntax definition. This approach is inspired by how modern programming languages deal with error conditions via *exceptions*: Any code can at any time 'throw an exception' which leads to an abort of the current code and is passed on to the calling instance. In particular, if in our game definitions a scheme algorithm aborts, the corresponding game oracle immediately aborts as well (and returns to the adversary).

Security games are parameterized by an adversary, and consist of a main game body plus zero or more oracle specifications. The adversary is allowed to call any of the specified oracles. The execution of the game starts with the main game body and terminates when a 'Stop with *exp*' instruction is reached, where the value of expression *exp* is taken as the outcome of the game. If the outcome of a game G is Boolean, we write $\Pr[\mathrm{G}(\mathcal{A})]$ for the probability (over the random coins of G and $\mathcal{A}$) that an execution of G with adversary $\mathcal{A}$ results in the outcome `T` or 1. We define shorthand notation for specific combinations of game-ending instructions: While in computational games we write 'Win' for 'Stop with `T`', in distinguishing games we write 'Win' for 'Stop with b' (where b is the challenge bit). In any case we write 'Lose' for 'Stop with `F`'. Further, for a Boolean condition C, we write 'Require C' for 'If $\neg C$: Lose', 'Penalize C' for 'If C: Lose', 'Reward C' for 'If C: Win', and 'Promise C' for 'If $\neg C$: Win'.

3 Syntax and Security of BOOM

We formalize Bidirectional Out-of-Order Messaging (BOOM) protocols. The scheme API assumes the four algorithms *init*, *send*, *recv*, *tick* and a timestamp decoding function *ts*. The *init*, *send*, *recv* algorithms are akin to prior work and implement instance initialization, message sending, and message receiving, respectively.[10] The *tick* algorithm enables a user's instance to track the progression of physical time: It is assumed to be periodically invoked by the computing platform (e.g., once every second), and has no visible effect beyond updating the instance's internal state. This allows us to model physical time with an integer counter that indicates the number of occurred *tick* invocations of the corresponding participant. Independently of physical time, a notion of logical time is induced by the sequence in which messages are processed by a sender: We track logical time with an integer counter that indicates the number of occurred *send* invocations of the corresponding participant. The logical time associated with a sending operation is also referred to as the operation's sending index. Whenever a ciphertext is produced, we assume a production timestamp is attached to it. Formally, we demand that, given a ciphertext, the timestamp decoding function *ts* recovers the physical time and logical time of the sender at the point when it created the ciphertext by invoking the *send* algorithm. The timestamp notion will prove crucial to formulate conditions related to ciphertext expiration.

We proceed with defining the syntax, the semantics (execution environment and correctness), and the security notions associated with BOOM protocols.

Syntax. A (two-party) BOOM scheme for an associated-data space $\mathcal{AD}$ and a message space $\mathcal{M}$ consists of a state space $\mathcal{ST}$, a ciphertext space $\mathcal{C}$, algorithms *init*, *send*, *recv*, *tick*, and a timestamp decoding function *ts*. Algorithm *init* generates initial states $st_{\mathtt{A}}, st_{\mathtt{B}} \in \mathcal{ST}$ for the participants. Algorithm *send* takes a state $st \in \mathcal{ST}$, an associated-data string $ad \in \mathcal{AD}$, and a message $m \in \mathcal{M}$, and outputs an (updated) state $st' \in \mathcal{ST}$ and a ciphertext $c \in \mathcal{C}$. Algorithm *recv* takes a state $st \in \mathcal{ST}$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and outputs an (updated) state $st' \in \mathcal{ST}$, an acknowledgment set $\mathrm{A} \subseteq \mathbb{N}$, and a message $m \in \mathcal{M}$. (The understanding of output A is that when c was generated by the peer, then for all $i \in \mathrm{A}$ the peer had received the ciphertext with sending index i.) Algorithm *tick* takes a state $st \in \mathcal{ST}$ and outputs an (updated) state $st' \in \mathcal{ST}$. Function *ts* takes a ciphertext $c \in \mathcal{C}$ and recovers a logical timestamp (sending index) $lt \in \mathbb{N}$ and a physical timestamp $pt \in \mathbb{N}$. If $\mathcal{P}(\mathbb{N})$ denotes the powerset of set $\mathbb{N}$, the BOOM API is thus as follows:

$$init \to \mathcal{ST} \times \mathcal{ST} \qquad tick\langle \mathcal{ST} \rangle \qquad ts\colon \mathcal{C} \to \mathbb{N} \times \mathbb{N}$$

$$\mathcal{AD} \times \mathcal{M} \to send\langle \mathcal{ST} \rangle \to \mathcal{C} \qquad \mathcal{AD} \times \mathcal{C} \to recv\langle \mathcal{ST} \rangle \to \mathcal{P}(\mathbb{N}) \times \mathcal{M}$$

[10] More precisely, our *recv* algorithm has a dedicated output for reporting to the invoking user which of the priorly sent own messages have been received by the peer; this output does not exist in prior work.

Semantics. We give game based definitions of correctness and security. Recall that the form of secure messaging that we consider supports the out-of-order processing of ciphertexts. This property, of course, has to be reflected in all games, rendering them more complex than those of prior works that deal with easier settings. To manage this complexity, we carefully developed our games such that they share, among each other, as many code lines and game variables as possible. In particular, the games can be seen as derived by individualizing a common basic game body in order to express specific aspects of functionality or security. This individualization is done by inserting an appropriate small set of additional code lines.[11] (For instance, the game defining authenticity adds lines of code that identify and flag forgery events.) In the following we explain first the BASIC game and then its refinements FUNC, AUTH, and CONF.[12]

Game BASIC. We first take a quick glance over the BASIC game of Fig. 1, deferring the discussion of details to the upcoming paragraphs. The game body [G00–G18] initializes some variables [G00–G13], invokes the *init* algorithm to initialize states for two users $\mathtt{A}$ and $\mathtt{B}$ [G17], and invokes the adversary [G18]. The adversary has access to four oracles, each of which takes an input $u \in \{\mathtt{A}, \mathtt{B}\}$ to specify the targeted user. The Tick oracle gives access to the *tick* algorithm [T00], the Send oracle gives access to the *send* algorithm [S00,S07], and the Recv oracle, besides internally recovering the logical and physical sending timestamps of an incoming ciphertext [R00], gives access to the *recv* algorithm [R01,R29]. Finally, the Expose oracle reveals the current protocol state of a user to the adversary [E06]. The game variables and remaining code lines are related to monitoring the actions of the adversary, allowing for identifying specific game states and tracking the transitions between them. In particular we identified the user-specific states *in-sync* and *authoritative*, the ciphertext properties *sync-preserving*, *sync-damaging*, *certifying*, and *vouching*, and the transitions *losing sync*, *poisoning*, and *healing*, as relevant in the BOOM setting. We explain these concepts one by one.

We say that protocol actors are synchronized if their views on the communication is consistent. A little more precisely, a participant Alice is in-sync with her peer Bob if all ciphertexts that Alice received are identical with ciphertexts that Bob priorly sent. The complete definition, formalized as part of the BASIC game as discussed below, further requires that the employed associated-data inputs are matching, and that the processing of ciphertexts of an out-of-sync peer also renders the receiver out-of-sync. If Alice is in-sync with Bob, we refer to ciphertexts that Alice can receive without losing sync as sync-preserving; the ciphertexts that would render her out-of-sync are referred to as sync-damaging.[13]

[11] Removing or modifying existing lines will not be necessary. That said, restricting the options to only add new lines might lead to also introducing a small number of redundancies that could allow for simplifications.

[12] The BASIC game itself is not used to model any kind of functionality or security. It merely describes the execution environment.

[13] The in-sync notion first surfaced in [7] in the context of unidirectional channels. It was extended in [23] to handle bidirectional communication and associated-data

```
**Game** BASIC($\mathcal{A}$)
G00 For $u \in \{\mathtt{A},\mathtt{B}\}$:
G01     $lt_u \leftarrow 0$
G02     $pt_u \leftarrow 0$
G03     $is_u \leftarrow \mathtt{T}$
G04     $\mathrm{SC}_u \leftarrow \emptyset$
G05     $\mathrm{CERT}_u \leftarrow \emptyset$
G06     $\mathrm{VF}_u[\cdot] \leftarrow [\![\infty]\!]$
G07     $\mathrm{AU}_u \leftarrow [\![\infty]\!]$
G13     $poisoned_u \leftarrow \mathtt{F}$
G17 $(st_\mathtt{A}, st_\mathtt{B}) \leftarrow init$
G18 Invoke $\mathcal{A}$
```

```
**Oracle** Tick($u$)
T00 $tick\langle st_u \rangle$
T01 $pt_u \leftarrow pt_u + 1$

**Oracle** Send($u, ad, m$)
S00 $c \leftarrow send\langle st_u \rangle(ad, m)$
S02 If $is_u$:
S03     $\mathrm{SC}_u \overset{\cup}{\leftarrow} \{(ad, c)\}$
S06 $lt_u \leftarrow lt_u + 1$
S07 Return $c$

**Oracle** Expose($u$)
E01 If $is_u$:
E02     $\mathrm{VF}_u[\![lt_u]\!] \overset{\cap}{\leftarrow} [\![lt_u]\!]$
E03     $\mathrm{AU}_u \overset{\cap}{\leftarrow} [\![lt_u]\!]$
E06 Return $st_u$
```

```
**Oracle** Recv($u, ad, c$)
R00 $(lt, pt) \leftarrow ts(c)$
R01 $(\mathrm{A}, m) \leftarrow recv\langle st_u \rangle(ad, c)$
R06 If $(ad, c) \in \mathrm{SC}_{\bar{u}}$:
R07     If $is_u$:
R08         $\mathrm{CERT}_u \overset{\cup}{\leftarrow} [lt]$
R09         $\mathrm{AU}_{\bar{u}} \overset{\cup}{\leftarrow} \mathrm{VF}_{\bar{u}}[lt]$
R17 If $(ad, c) \notin \mathrm{SC}_{\bar{u}}$:
R18     If $is_u$:
R19         If $lt \notin \mathrm{AU}_{\bar{u}}$:
R20             $poisoned_u \leftarrow \mathtt{T}$
R23     $is_u \leftarrow \mathtt{F}$
R29 Return $(\mathrm{A}, m)$
```

Fig. 1. Game BASIC. We refer the reader to Footnote 16 for the interpretation of $\mathrm{VF}_u[\![lt_u]\!]$ in line [E02]. We write $\bar{u}$ for the element such that $\{u, \bar{u}\} = \{\mathtt{A}, \mathtt{B}\}$.

As we consider communication algorithms that are stateful, any ciphertext created by a participant may depend on, and may implicitly reflect, the full prior communication history of that participant. That is, if from a sequence of sent ciphertexts only a subset of ciphertexts arrive, then from what *did* arrive the receiver should be able to extract information linked to what was sent before but is still missing. In particular, any ciphertext that is received in-sync should allow for identifying which earlier-sent though later-delivered ciphertexts are authentic. We correspondingly say that in-sync received ciphertexts certify the ciphertexts sent *earlier* by the same sender.

Ciphertexts can also make promises about the future: Every received ciphertext may carry (cryptographic) information that is used to authenticate later ciphertexts (of the same sender, up to their next exposure). Here we say that ciphertexts (cryptographically) vouch for the ones sent *later* by the same participant.

We finally discuss attack classes that are enabled by exposing the states of users: Once a participant's state becomes known by exposure, it is trivial to impersonate the user, simply by invoking the scheme algorithms with the captured state. We refer to states of a participant as authoritative if their actions can *not* be trivially emulated by the adversary in this way. If an impersonation happens right after an exposure, as the adversary can perfectly and permanently emulate all actions of the impersonated party, in addition to all authenticity and confidentiality guarantees being lost, there is also no option to recover into a safe state. We refer to the transition into such a setting, more precisely to the action of exploiting the state exposure of one participant by delivering an impersonating ciphertext to the other participant, as poisoning the latter. A second option of

strings. Our definitions are based on [23], but adapted to tolerate the out-of-order delivery of ciphertexts.

the adversary after exposing a state is to remain passive (in particular, not to poison the partner). In this case the healing property of ratcheting-based secure messaging protocols shall automatically fully restore safe operations.

Coming back to the BASIC game of Fig. 1, we describe how the above concepts are reflected in the game variables and code lines. We start with the game body [G00–G18]. If $u \in \{\mathtt{A}, \mathtt{B}\}$ refers to one of the two participants, integer lt_u ('logical time') reflects the logical time of u; integer pt_u ('physical time') reflects the physical time of u; Boolean flag is_u ('in-sync') indicates whether u is in-sync with their peer $\bar{u}$; set SC_u ('sent ciphertexts') records the associated-data–ciphertext pairs sent by u; set CERT_u ('certified') indicates which of the peer $\bar{u}$'s sending indices have been certified by receiving an in-sync ciphertext from them; for each sending index i, set $\mathrm{VF}_u[i]$ ('vouches for') indicates for which sending indices of u the ciphertext with index i can vouch for; set AU_u indicates for which sending indices participant u is authoritative; flag $poisoned_u$ indicates whether u was poisoned.

We next explain how these variables are updated throughout the game. The cases of lt_u [G01,S06] and pt_u [G02,T01] are clear. Flag is_u is initialized to $\mathtt{T}$ [G03], and cleared [R23] in the moment that u receives a ciphertext that the peer $\bar{u}$ either didn't send, or did send but after becoming out-of-sync [R17] (in conjunction with [S02,S03], see next sentence).[14] Set SC_u is initialized empty [G04] and populated [S03] for each sending operation in which u is in-sync [S02].[15] Set CERT_u is initialized empty [G05] and, when a sync-preserving ciphertext is received [R06,R06insync], populated with all indices prior to, and including, the current one [R08]. All entries of array-of-sets VF_u are initialized to 'all-indices' [G06], expressing that, by default, each sending index cryptographically vouches for its entire future (and past). This changes when u's state is exposed, as impersonating u then becomes trivial; the game reflects this by updating all VF_u entries related to the time preceding the exposure so that the corresponding ciphertexts do not vouch for ciphertexts that are created after the exposure [E02].[16] Set AU_u is initialized to 'all-indices' [G07], and indices are removed from it by exposing u's state, and added back to it by letting u heal; more precisely, while exposing u's state removes all indices starting with the current one (marking the entire future as non-authoritative) [E03], receiving a sync-preserving ciphertext from peer $\bar{u}$ [R06,R06insync] adds the vouched-for entries back [R09] (re-establishing authoritativeness up to the next exposure). Finally, flag $poisoned_u$ is initialized clear [G13], and set [R20] when a sync-damaging ciphertext is received (i.e., one that was not sent by peer $\bar{u}$ [R17] and is the first

[14] The mechanism of considering participants out-of-sync once they process (unmodified) ciphertexts from out-of-sync peers is taken from [23], see Footnote 13.

[15] Note that the sending index of any ciphertext is uniquely recoverable (with function ts), implying that each execution of [S03] adds a new element to the set (collisions cannot occur).

[16] Line [E02] should be read as 'For all $0 \leq i < lt_u$: $\mathrm{VF}_u[i] \leftarrow \mathrm{VF}_u[i] \cap [\![lt_u]\!]$' and expresses that all entries of $\mathrm{VF}_u[\cdot]$ that correspond with prior sending indices are trimmed so that they cover no indices that succeed the current one (including).

one making u lose sync [R18]) that was trivially injected after an exposure of peer $\bar{u}$'s state (technically: was crafted for a non-authoritative index [R19]).

This completes the description of the BASIC game. We refine it in the following to obtain three more games, but the basic working mechanisms of the oracles and variables remain the same.

Game FUNC. We specify the expected functionality (a.k.a. correctness) of a BOOM protocol by formulating requirements on how it shall react to receiving valid and invalid ciphertexts. Concretely, in Fig. 2 we specify the corresponding FUNC game as an extension of the BASIC game from Fig. 1. In the figure, the code lines marked with neither $\circ$ nor $\bullet$ are taken verbatim from the BASIC game, and the lines marked with $\circ$ are the ones to be added to obtain the FUNC game. (Ignore the lines marked with $\bullet$ for now.) The FUNC game tests for a total of seven conditions, letting the adversary 'win' if any one of them is not fulfilled. Five of the conditions are checked for all operations (in-sync *and* out-of-sync): The conditions are (1) that the *ts* decoding function correctly indicates the logical and physical creation time of ciphertexts [S01]; (2) that no sending index is received twice (single delivery of ciphertexts) [G10,R02,R25] (set RI_u records 'received indices'); (3) that expired ciphertexts are not delivered (the reported sender's physical time pt is compared with the receiver's physical time pt_u, tolerating a lag of up to δ time units) [R03]; (4) that physical timestamps increase as logical timestamps do [G11,R04,R26] (set RT_u records 'received timestamps');[17] and (5) that the reported acknowledgment set A never shrinks and never lists never-sent indices [G12,R05,R27] (set RA_u records 'received acknowledgments'). Two additional conditions are checked for certified ciphertexts (this includes all in-sync ciphertexts, as they certify themselves [R06,R07,R08]): The conditions are (6) that the *recv* algorithm accurately reports the acknowledgment set A [R13] (recall that set RI_u holds the received indices [G10,R25], allowing to associate this set with each (in-sync) sending operation [G08,S04], so that set $\mathrm{SR}_{\bar{u}}[i]$ [S04,R13] indicates the indices that participant $\bar{u}$ received from u before $\bar{u}$ used sending index i in their sending operation); and (7) that encrypted messages are correctly recovered via decryption [G09,S05,R14] (array SM_u records 'sent messages'). We say that a BOOM protocol is functional if the advantage $\mathbf{Adv}^{\mathrm{func}}(\mathcal{A}) := \Pr[\mathrm{FUNC}(\mathcal{A})]$ is negligibly small for all realistic adversaries $\mathcal{A}$.

Game AUTH. Our authenticity notion focuses on the protection of the integrity of ciphertexts (INT-CTXT). In Fig. 2 we specify the corresponding AUTH game as an extension of the BASIC game from Fig. 1. In the figure, the code lines marked with neither $\circ$ nor $\bullet$ are taken verbatim from the BASIC game, and the lines marked with $\bullet$ are the ones to be added to obtain the AUTH game. (This time, ignore the lines marked with $\circ$.) A BOOM scheme provides AUTH security if any adversarial manipulation (or injection) of ciphertexts is detected and rejected. Taking into account that associated-data strings need to be protected in the same vein, as a first approximation the notion could be formalized by adding

[17] A relation $R \subseteq \mathbb{N} \times \mathbb{N}$ is monotone [R04] if for all $(x, y), (x', y') \in R$ we have $x \leq x' \Rightarrow y \leq y'$.

```
  **Game** FUNC($\mathcal{A}$)   //with ○
  **Game** AUTH($\mathcal{A}$)   //with ●
  G00 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
  G01     $lt_u \leftarrow 0$
  G02     $pt_u \leftarrow 0$
  G03     $is_u \leftarrow \mathtt{T}$
  G04     $\mathrm{SC}_u \leftarrow \emptyset$
  G05     $\mathrm{CERT}_u \leftarrow \emptyset$
  G06     $\mathrm{VF}_u[\cdot] \leftarrow [\![\infty]\!]$
  G07     $\mathrm{AU}_u \leftarrow [\![\infty]\!]$
○ G08     $\mathrm{SR}_u[\cdot] \leftarrow \perp$
○ G09     $\mathrm{SM}_u[\cdot] \leftarrow \perp$
○ G10     $\mathrm{RI}_u \leftarrow \emptyset$
○ G11     $\mathrm{RT}_u \leftarrow \emptyset$
○ G12     $\mathrm{RA}_u \leftarrow \emptyset$
  G17 $(st_\mathtt{A}, st_\mathtt{B}) \leftarrow init$
  G18 Invoke $\mathcal{A}$
  G19 Lose

  **Oracle** Tick($u$)
  T00 $tick\langle st_u \rangle$
  T01 $pt_u \leftarrow pt_u + 1$

  **Oracle** Send($u, ad, m$)
  S00 $c \leftarrow send\langle st_u \rangle(ad, m)$
○ S01 Promise $ts(c) = (lt_u, pt_u)$
  S02 If $is_u$:
  S03     $\mathrm{SC}_u \xleftarrow{\cup} \{(ad, c)\}$
○ S04     $\mathrm{SR}_u[lt_u] \leftarrow \mathrm{RI}_u$
○ S05     $\mathrm{SM}_u[lt_u] \leftarrow m$
  S06 $lt_u \leftarrow lt_u + 1$
  S07 Return $c$

  **Oracle** Recv($u, ad, c$)
  R00 $(lt, pt) \leftarrow ts(c)$
  R01 $(\mathrm{A}, m) \leftarrow recv\langle st_u \rangle(ad, c)$
○ R02 Promise $lt \notin \mathrm{RI}_u$
○ R03 Promise $\Delta(pt, pt_u) \leq \delta$
○ R04 Promise $\mathrm{RT}_u \cup \{(lt, pt)\}$ monotone
○ R05 Promise $\mathrm{RA}_u \subseteq \mathrm{A} \subseteq [\![lt_u]\!]$
  R06 If $(ad, c) \in \mathrm{SC}_{\bar{u}}$:
  R07     If $is_u$:
  R08         $\mathrm{CERT}_u \xleftarrow{\cup} [lt]$
  R09         $\mathrm{AU}_{\bar{u}} \xleftarrow{\cup} \mathrm{VF}_{\bar{u}}[lt]$
○ R12     If $lt \in \mathrm{CERT}_u$:
○ R13         Promise $\mathrm{A} = \mathrm{RA}_u \cup \mathrm{SR}_{\bar{u}}[lt]$
○ R14         Promise $m = \mathrm{SM}_{\bar{u}}[lt]$
  R17 If $(ad, c) \notin \mathrm{SC}_{\bar{u}}$:
● R18     If $is_u$:
● R21         Reward $lt \in \mathrm{AU}_{\bar{u}}$
● R22     Reward $lt \in \mathrm{CERT}_u$
  R23     $is_u \leftarrow \mathtt{F}$
○ R25 $\mathrm{RI}_u \xleftarrow{\cup} \{lt\}$
○ R26 $\mathrm{RT}_u \xleftarrow{\cup} \{(lt, pt)\}$
○ R27 $\mathrm{RA}_u \leftarrow \mathrm{A}$
  R29 Return $(\mathrm{A}, m)$

  **Oracle** Expose($u$)
  E01 If $is_u$:
  E02     $\mathrm{VF}_u[\![lt_u]\!] \xleftarrow{\cap} [\![lt_u]\!]$
  E03     $\mathrm{AU}_u \xleftarrow{\cap} [\![lt_u]\!]$
  E06 Return $st_u$
```

Fig. 2. Games FUNC and AUTH. The FUNC game includes the lines marked with ○ but not the ones marked with ●. The AUTH game includes the lines marked with ● but not the ones marked with ○.

the instruction 'Reward $is_u \wedge (ad, c) \notin \mathrm{SC}_{\bar{u}}$' to the Recv oracle.[18] Note however that delivering a forged ciphertext to a participant u is trivial if the state of their peer $\bar{u}$ is exposed, and thus a small refinement is due. Recalling that set $\mathrm{AU}_{\bar{u}}$ lists the sending indices for which participant $\bar{u}$ is authoritative, i.e., their actions not trivially emulatable, we reward the adversary only if the forgery is made for an index contained in this set [R17,R18,R21]. Recall further that in-sync delivered ciphertexts certify prior ciphertexts by the same sender, even if the latter ciphertexts are delivered out-of-sync. In the game we thus reward the adversary also if it forges on a certified index [R17,R22]. We say that a BOOM protocol provides authenticity if the advantage $\mathbf{Adv}^{\mathrm{auth}}(\mathcal{A}) := \Pr[\mathrm{AUTH}(\mathcal{A})]$

[18] The instruction should be read as 'Reward the adversary if it makes an in-sync participant accept an associated-data–ciphertext pair for which at least one of associated-data and ciphertext is not authentic'.

is negligibly small for all realistic adversaries $\mathcal{A}$. We refer the reader to the full version for a formalization of the trivial attack excluded by the AUTH game, and an overview of similar but non-trivial attacks that are allowed.

Games $\mathrm{CONF}^0, \mathrm{CONF}^1$. Our confidentiality notion is formulated in the style of left-or-right indistinguishability under active attacks (IND-CCA). In Fig. 3 we specify corresponding CONF^0 and CONF^1 games. The games are derived from the BASIC game by adding the lines marked with • plus two new oracles: The left-or-right Chal oracle [C00–C08], which behaves similar to the Send oracle but processes one of two possible input messages [C03] depending on bit b that encodes which game CONF^b is played, and the Decide oracle [D00] that lets the adversary control the return value of the game. (A successful adversary manages to correlate this return value with bit b.)

Three new game variables keep track of the actions of the adversary: Variable lx_u ('last exposure', [G14,E04]) indicates the index of the last exposure of user u. Set CH_u ('challenge') represents the set of sending indices for which a challenge query has been posed for u that peer $\bar{u}$ still should be able to validly decrypt. Indices are added to this set in the Chal oracle [G15,C06], and they are removed from it as a reaction to three events. (1) The corresponding ciphertext becomes invalid because the receiver already processed a ciphertext (the same or a different one) with the same index [R28] (see the corresponding guarantee in the FUNC game [G10,R02,R25]). (2) It becomes invalid because it expired based on physical time: To capture the latter condition we denote with

$$\mathrm{ITC}(u) \coloneqq \{lt : \exists ad, c, pt \text{ s.t. } (ad, c) \in \mathrm{SC}_{\bar{u}} \wedge ts(c) = (lt, pt) \wedge \mathit{\Delta}(pt, pt_u) \leq \delta\}$$

('in-time ciphertexts') for participant u the set of sending indices of ciphertexts produced by peer $\bar{u}$ for which the difference between generation time pt and the physical time pt_u of the receiver is less than δ. With the progression of physical time the game removes those indices from set $\mathrm{CH}_{\bar{u}}$ that are not an element of $\mathrm{ITC}(u)$ [T02]. (See the corresponding guarantee in the FUNC game [R03].) (3) Receiving an out-of-sync ciphertext renders u's state incompatible to decrypt *future* challenge queries. Hence all future indices are removed from the challenge set [R24]. Observe this corresponds with [C06]: indices are only added to CH_u for an in-sync peer. Finally, flag xp_u ('exposed') indicates whether the state of u has to be considered known to the adversary after a state exposure. This flag is initially cleared [G16], set when u's state is exposed [E05], and reset if u heals by letting peer $\bar{u}$ receive an in-sync ciphertext created after the last exposure [R10,R11].

We next explain how the new variables help identifying four different trivial attack conditions. The first two conditions consider cases where posing a Chal query needs to be prevented because the receiver state is known due to impersonation or exposure: (1) if participant $\bar{u}$'s state was exposed and $\bar{u}$ is impersonated to u, i.e., u is poisoned, all future encryptions by u for $\bar{u}$ are trivially decryptable, simply because the adversary can emulate *all* actions of $\bar{u}$ [C01]; (2) encryptions by an in-sync sender u for a state-exposed receiver $\bar{u}$ are trivially decryptable (recall that flag $xp_{\bar{u}}$ traces the latter condition) [C02]. The next condition considers cases where posing an Expose query needs to be prevented because an already made Chal query would become trivial to break: (3) if participant $\bar{u}$ generated (challenge) ciphertext c for u, and the latter should still be able to validly decrypt c, then exposing u makes c trivially decryptable [E00]. The last condition is unrelated to exposures: (4) if participant u in-sync decrypts a ciphertext, by correctness the resulting message is identical to the encrypted message, and thus has to be suppressed by the Recv oracle by overwriting it [R15,R16]. (Note how line R16 corresponds with line R14 of FUNC.) This concludes the description of games CONF^b. We say that a BOOM protocol provides confidentiality if the advantage $\mathbf{Adv}^{\text{conf}}(\mathcal{A}) := |\Pr[\text{CONF}^1(\mathcal{A})] - \Pr[\text{CONF}^0(\mathcal{A})]|$ is negligibly small for all realistic adversaries $\mathcal{A}$. We refer the reader to the full version for a formalization of the trivial attacks excluded by the CONF game, and similar but non-trivial attacks that are allowed.

```
**Game** CONF$^b(\mathcal{A})$
  G00 For $u \in \{\mathtt{A},\mathtt{B}\}$:
  G01     $lt_u \leftarrow 0$
  G02     $pt_u \leftarrow 0$
  G03     $is_u \leftarrow \mathtt{T}$
  G04     $\mathrm{SC}_u \leftarrow \emptyset$
  G06     $\mathrm{VF}_u[\cdot] \leftarrow [\![\infty]\!]$
  G07     $\mathrm{AU}_u \leftarrow [\![\infty]\!]$
  G13     $poisoned_u \leftarrow \mathtt{F}$
  G14 •   $lx_u \leftarrow 0$
  G15 •   $\mathrm{CH}_u \leftarrow \emptyset$
  G16 •   $xp_u \leftarrow \mathtt{F}$
  G17 $(st_\mathtt{A}, st_\mathtt{B}) \leftarrow init$
  G18 Invoke $\mathcal{A}$
  G19 Lose

**Oracle** Send$(u, ad, m)$
  S00 $c \leftarrow send\langle st_u\rangle(ad, m)$
  S02 If $is_u$:
  S03     $\mathrm{SC}_u \overset{\cup}{\hookleftarrow} \{(ad, c)\}$
  S06 $lt_u \leftarrow lt_u + 1$
  S07 Return $c$
```

```
**Oracle** Chal$(u, ad, m^0, m^1)$
  C00 Require $|m^0| = |m^1|$
  C01 Penalize $poisoned_u$
  C02 If $is_u$: Penalize $xp_{\bar{u}}$
  C03 $c \leftarrow send\langle st_u\rangle(ad, m^b)$
  C04 If $is_u$:
  C05     $\mathrm{SC}_u \overset{\cup}{\hookleftarrow} \{(ad, c)\}$
  C06     If $is_{\bar{u}}$: $\mathrm{CH}_u \overset{\cup}{\hookleftarrow} \{lt_u\}$
  C07 $lt_u \leftarrow lt_u + 1$
  C08 Return $c$

**Oracle** Expose$(u)$
• E00 Require $\mathrm{CH}_{\bar{u}} = \emptyset$
  E01 If $is_u$:
  E02     $\mathrm{VF}_u[\![lt_u]\!] \overset{\cap}{\hookleftarrow} [\![lt_u]\!]$
  E03     $\mathrm{AU}_u \overset{\cap}{\hookleftarrow} [\![lt_u]\!]$
• E04     $lx_u \leftarrow lt_u$
• E05     $xp_u \leftarrow \mathtt{T}$
  E06 Return $st_u$

**Oracle** Decide$(b')$
  D00 Stop with $b'$
```

```
**Oracle** Recv$(u, ad, c)$
  R00 $(lt, pt) \leftarrow ts(c)$
  R01 $(\mathrm{A}, m) \leftarrow recv\langle st_u\rangle(ad, c)$
  R06 If $(ad, c) \in \mathrm{SC}_{\bar{u}}$:
  R07     If $is_u$:
  R09         $\mathrm{AU}_{\bar{u}} \overset{\cup}{\hookleftarrow} \mathrm{VF}_{\bar{u}}[lt]$
• R10         If $lt \geq lx_{\bar{u}}$:
• R11             $xp_{\bar{u}} \leftarrow \mathtt{F}$
• R15     If $lt \in \mathrm{CH}_{\bar{u}}$:
• R16         $m \leftarrow \diamond$
  R17 If $(ad, c) \notin \mathrm{SC}_{\bar{u}}$:
  R18     If $is_u$:
  R19         If $lt \notin \mathrm{AU}_{\bar{u}}$:
  R20             $poisoned_u \leftarrow \mathtt{T}$
  R23     $is_u \leftarrow \mathtt{F}$
• R24     $\mathrm{CH}_{\bar{u}} \overset{\cap}{\hookleftarrow} [\![lt]\!]$
• R28 $\mathrm{CH}_{\bar{u}} \leftarrow \mathrm{CH}_{\bar{u}} \setminus \{lt\}$
  R29 Return $(\mathrm{A}, m)$

**Oracle** Tick$(u)$
  T00 $tick\langle st_u\rangle$
  T01 $pt_u \leftarrow pt_u + 1$
• T02 $\mathrm{CH}_{\bar{u}} \overset{\cap}{\hookleftarrow} \mathrm{ITC}(u)$
```

Fig. 3. Games CONF^0, CONF^1. See text for the definition of function ITC [T02].

4 Non-Interactive Primitives

In Sect. 3 we defined the syntax and security of BOOM protocols and we will provide a secure construction in Sect. 5. The current section is dedicated to presenting a set of cryptographic building blocks, in the spirit of public key encryption (PKE) and signature schemes (SS), that will play crucial roles in our construction. Recall that a defining property of a BOOM protocol is that it provides maximum resilience against (continued) state exposure attacks, preventing all but trivial attacks. If a construction would rely on regular PKE or SS schemes as building blocks, the secret keys of the latter would leak on state exposure, which in most cases would inevitably clear the way for an attack on confidentiality or authenticity. We hence employ stateful variants of PKE and SS that process their internal keying material after each use to an updated 'refreshed' version that limits the options of a state-exposing adversary to harm only future operations. Some of the building blocks proposed here additionally fold an *associated data* input into their state, and the assumption is that sender and receiver (i.e., signer and verifier, or encryptor and decryptor) update their states with consistent such inputs.[19]

While the specifics of our building blocks might be different from those of prior work, it can be generally considered well-understood how to construct such primitives. For instance, a forward-secure SS [8], which is a primitive close to one of ours, can be built by coupling each signing operation with the generation of a fresh signature key pair, the public component of which is signed and thus authenticated along with the message; after the signing operation is complete, the original signing key is disposed of and replaced by the freshly generated one. Adding the support of auxiliary associated-data strings into such a scheme is trivial (just authenticate the string along with the message) and is less a cryptographic challenge than an exercise of maintaining the right data structures in the sender/receiver state. Similarly, forward-secure PKE [11], which is a primitive close to one of ours as well, is routinely built from hierarchical identity-based encryption (HIBE) by associating key validity epochs with the nodes of a binary tree. Variants of forward-secure PKE that support key updates that depend on auxiliary associated-data strings have been proposed in prior work as well [17,24], using design approaches that can be seen as minor variations of the original tree-based idea from [11].

For our BOOM construction in Sect. 5 we require three independent forward-secure public key primitives which we refer to as *updatable signature scheme*, *key-updatable KEM*, and *key-evolving KEM*, respectively. We specify their syntax and explain the expected behavior below. We formalize the details and propose concrete constructions in the full version. We note that our security definitions and constructions can be seen as following immediately from the syntax and expected functionality: While the security definitions give the adversary the option to expose the state of any participant any number of times, and formalize the best-possible security that is feasible under such a regime (i.e., maximum resilience against state exposure attacks), the constructions, which all follow the approaches of [8,11,17,24] discussed above, are engineered to re-generate fresh key material whenever an opportunity for this arises.

[19] Unlike regular signature schemes where for each signer there can be many independent verifiers, and unlike regular public key encryption where for each decryptor there can be many encryptors, for the primitives we consider in the current section a strict one-to-one correspondence between sender and receiver is assumed.

4.1 Updatable Signature Schemes (USS)

Like a regular signature scheme, a USS has algorithms $\mathit{gen}, \mathit{sign}, \mathit{vfy}$, where sign creates a signature on a given message and vfy verifies that a given signature is valid for a given message. The particularity of USS is that signing and verification keys can be updated, and that signatures only verify correctly if these updates are performed consistently. More precisely, signing and verification keys are replaced by signing and verification *states*, and update algorithms $\mathit{updss}, \mathit{updvs}$ (for 'update signing state' and 'update verification state', respectively) can update these states to a new version, taking also an associated-data input into account. Multiple such update operations can be performed in succession, on both sides. Signatures of the signer are recognized as valid by the verifier only if the updates of both parties are in-sync, i.e., are performed with the same sequence of update strings. Our security model provides the means to the adversary to expose the state of the parties between any two update operations, and requires unforgeability with maximum resilience to such exposures.

Formally, a key-updatable signature scheme for a message space $\mathcal{M}$ and an associated-data space $\mathcal{AD}$ consists of a signing state space $\mathcal{SS}$, a verification state space $\mathcal{VS}$, a signature space Σ, and algorithms $\mathit{gen}, \mathit{sign}, \mathit{vfy}, \mathit{updss}, \mathit{updvs}$ with APIs

$$\mathit{gen} \to \mathcal{SS} \times \mathcal{VS} \qquad \mathcal{M} \to \mathit{sign}\langle\mathcal{SS}\rangle \to \Sigma \qquad \mathcal{VS} \times \mathcal{M} \times \Sigma \to \mathit{vfy}$$

$$\mathcal{AD} \to \mathit{updss}\langle\mathcal{SS}\rangle \qquad \mathcal{AD} \to \mathit{updvs}\langle\mathcal{VS}\rangle .$$

Note that the vfy algorithm doesn't have an explicit output. The assumption behind this is that the algorithm signals acceptance by terminating normally, while it signals rejection by aborting. (See Sect. 2 on the option of any algorithm to abort.) We expect of a correct USS that for all $(ss, vs) \in [\mathit{gen}]$, if ss and vs are updated by invoking $\mathit{updss}\langle ss\rangle(\cdot)$ and $\mathit{updvs}\langle vs\rangle(\cdot)$ with the same sequence $ad_1, \ldots, ad_l \in \mathcal{AD}$ of associated data, then for all $m \in \mathcal{M}$ and $\sigma \in [\mathit{sign}\langle ss\rangle(m)]$ we have that $\mathit{vfy}(vs, m, \sigma)$ accepts. See the full version for examples of the expected functionality a formalization of correctness and security, and a construction.

4.2 Key-Updatable KEM (KuKEM)

A key-updatable key encapsulation mechanism is a stateful KEM variant with algorithms $\mathit{gen}, \mathit{enc}, \mathit{dec}$ and update properties like for USS: both the encapsulator and the decapsulator can update their public/secret state material with algorithms $\mathit{updps}, \mathit{updss}$ (for 'update public state' and 'update secret state', respectively) that also take an associated-data input into account. The decapsulator, if updated in-sync with the encapsulator, can successfully decapsulate ciphertexts. Our security model formalizes IND-CCA-like security in a model supporting exposing the state of both parties, with the explicit requirement that state exposures neither harm the confidentiality of keys encapsulated for past epochs, nor the confidentiality of keys encapsulated with diverged states.

Formally, a key-updatable key encapsulation mechanism for a key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$, consists of a secret state space $\mathcal{SS}$, a public state space $\mathcal{PS}$, a ciphertext space $\mathcal{C}$, KEM algorithms *gen*, *enc*, *dec* and state update algorithms *updps*, *updss* with APIs

$$gen \to \mathcal{SS} \times \mathcal{PS} \qquad \mathcal{PS} \to enc \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{SS} \times \mathcal{C} \to dec \to \mathcal{K}$$

$$\mathcal{AD} \to updps\langle \mathcal{PS} \rangle \qquad \mathcal{AD} \to updss\langle \mathcal{SS} \rangle .$$

We expect of a correct KuKEM that for all $(ss, ps) \in [gen]$, if ss and ps are updated by invoking $updps\langle ps\rangle(\cdot)$ and $updss\langle ss\rangle(\cdot)$ with the same sequence $ad_1, \ldots, ad_l \in \mathcal{AD}$ of associated data, then for all $(k, c) \in [enc(ps)]$ and $k' \in [dec(ss, c)]$ we have that $k = k'$. See the full version for a formalization of correctness and security, and a construction.

4.3 Key-Evolving KEM (KeKEM)

A key-evolving key encapsulation mechanism consists of algorithms *gen*, *enc*, *dec* like a regular KEM, but, as above, public and secret keys are replaced by public and secret states, respectively, that can be updated. More precisely, the encapsulator's and decapsulator's states can be updated 'to the next epoch' by invoking the *evolveps* (for 'evolve public state') algorithm and the *evolvess* (for 'evolve secret state') algorithm, respectively. Note, however, that if a secret state is updated, the decryptability of ciphertexts generated for older epochs is not automatically lost; rather, ciphertexts associated to multiple epochs remain decryptable until epochs are explicitly declared redundant by invoking the *expire* algorithm.[20] Our security model formalizes IND-CCA-like security in a model supporting exposing the state of both parties, with the explicit requirement that state exposures do not harm the confidentiality of keys encapsulated for expired epochs. Note that our formalization of KeKEMs does not support updating states with respect to an associated-data input.

Formally, a key-evolving key encapsulation mechanism for a key space $\mathcal{K}$ consists of a secret state space $\mathcal{SS}$, a public state space $\mathcal{PS}$, a ciphertext space $\mathcal{C}$, KEM algorithms *gen*, *enc*, *dec* and state update algorithms *evolveps*, *evolvess*, *expire* with APIs

$$\mathbb{N} \to gen \to \mathcal{SS} \times \mathcal{PS} \qquad \mathcal{PS} \to enc \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{SS} \times \mathbb{N} \times \mathcal{C} \to dec \to \mathcal{K}$$

$$evolveps\langle \mathcal{PS} \rangle \qquad evolvess\langle \mathcal{SS} \rangle \qquad expire\langle \mathcal{SS} \rangle .$$

In the KeKEM setting it makes sense to number the epochs. Note that the *dec* algorithm expects, besides the secret state and the ciphertext, an explicit indication of the epoch number for which the ciphertext was created. For simplicity, one would like to provide an absolute time to the *dec* algorithm, e.g. Unix time, rather than the time offset relative to the generation time. For this

[20] The *expire* algorithm expires always to oldest currently supported epoch. That is, active epochs of KeKEMs always span a continuous interval.

reason, the *gen* algorithm takes in an epoch number which can be used to specify the generation time and thus the first epoch need not necessarily start at zero. Then the state can internally compute the relative offset on decapsulation. As the full definition is quite involved and thus deferred to the full version, we illustrate the functionality of a correct KeKEM using an example: If we invoke $(ss, ps) \leftarrow gen(5)$ to generate a state pair (and associating the number 5 with the first state), then invoking 2-times $evolveps\langle ps\rangle$ followed by $(k, c) \leftarrow enc(ps)$, and then 4-times $evolvess\langle ss\rangle$, then invoking $dec(ss, 7, c)$ will return k until $expire\langle ss\rangle$ has been invoked for the third time (expiring epochs 5, 6, and finally 7). See the full version for a formalization of correctness and security, and a construction.

5 Interactive Primitives and BOOM

This section exposes our Bidirectional Out-of-Order Messaging (BOOM) protocol, in three steps. In Sect. 5.1 we first present a BOOM-signature scheme, which uses the USS introduced in Sect. 4.1 as building block. This scheme will be used by our final BOOM construction in a black box manner by calling its *sign* and *vfy* procedures on each message to add an authenticity layer. Next, we present a BOOM-KEM scheme in Sect. 5.2. Our final BOOM construction will query the BOOM-KEM in a black box manner by calling its *enc* and *dec* procedures to obtain encryption keys for each message. The BOOM-KEM uses the KuKEM and KeKEM building blocks introduced in Sect. 4.2 and Sect. 4.3, to ensure the BOOM scheme can achieve confidentiality with its keys. The BOOM construction will additionally invoke its *upd* procedure to reflect the passing of time and the *expire* procedure to indicate we no longer wish to be able to obtain 'old' decryption keys.

Despite the strong building blocks defined in Sect. 4, our BOOM protocols are complex and involved. These difficulties stem from the data structures required to manage out-of-order delivery of ciphertexts. These data structures obscure the cryptographically novel core of our construction and render it difficult to interpret. Therefore, we have separated the authenticity tool and the confidentiality tool and present them in their own right. Note that this modularization implies certain data structures will be duplicated across each tool, but an implementation could consolidate them.

5.1 BOOM-Signature Scheme

In Sect. 5.3 we will use a specialized signature scheme to achieve authenticity for our BOOM construction. In this section we describe the inner workings of this cryptographic tool.

Syntax. A BOOM-signature scheme for a message space $\mathcal{M}$ consists of a state space $\mathcal{ST}$, a signature space Σ, algorithms *init*, *sign*, *vfy*, and a (logical) timestamp decoder *ts* as follows:

$$init \to \mathcal{ST} \times \mathcal{ST} \quad \mathcal{M} \to sign\langle \mathcal{ST}\rangle \to \Sigma \quad \mathcal{M} \times \Sigma \to vfy\langle \mathcal{ST}\rangle \quad ts\colon \Sigma \to \mathbb{N}.$$

Construction. We provide a construction for a BOOM-signature scheme in Fig. 4. The construction consists of four procedures: *init*, *sign*, *vfy* and *ts*. The *init* procedure initializes the states for two users A and B. The *sign* procedure is stateful and will output a signature σ for any message m, updating its state in the process. The *vfy* procedure is also stateful and will verify any pair $(m, \sigma) \in \mathcal{M} \times \Sigma$. If σ is a correct signature on m, the state will update and *vfy* will return control to the caller. If the signature does not correctly verify, the *vfy* procedure will abort. The *ts* function returns the logical time (measured in signer invocations).

On a very high level, *sign* generates a fresh USS key pair every iteration to recover from (potential) state exposures and signs the hash of its sent transcript, while *vfy* updates its state with the messages that have been received, so states will diverge if the adversary injects a message, while managing out of order delivery. We will now describe the variables and code lines in more detail.

```
Proc init
i00 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
i01     $lt_u \leftarrow 0$; $lt^*_u \leftarrow 0$
i02     $\mathrm{S}_u[\cdot] \leftarrow \bot$; $\mathrm{V}_u[\cdot] \leftarrow \bot$
i03     $(ss_u, vs^*_u) \leftarrow \mathrm{USS}.gen$
i04     $\mathrm{P}_u \leftarrow \emptyset$
i05     $\mathrm{AS}_u[\cdot] \leftarrow \bot$
i06     $\mathrm{AS}_u[lt_u] \leftarrow H()$
i07     $av_u \leftarrow H()$
i08     $st_u := (\ldots)$
i09 Return $(st_\mathtt{A}, st_\mathtt{B})$

Proc $sign\langle st_u \rangle(m)$
s00 $(ss, vs) \leftarrow \mathrm{USS}.gen$
s01 $h \leftarrow H(m \parallel lt_u \parallel \mathrm{P}_u \parallel vs \parallel \mathrm{S}_u[\![lt_u]\!])$
s02 $\sigma \leftarrow \mathrm{USS}.sign\langle ss_u \rangle(h)$
s03 $\mathrm{AS}_u[lt_u + 1] \leftarrow H(\mathrm{AS}_u[lt_u] \parallel h \parallel \sigma)$
s04 $\mathrm{S}_u[lt_u] \leftarrow (h, \mathrm{P}_u, vs, \sigma)$
s05 $\sigma \overset{\parallel}{\leftarrow} lt_u \parallel \mathrm{P}_u \parallel vs \parallel \mathrm{S}_u[\![lt_u]\!]$
s06 $lt_u \leftarrow lt_u + 1$
s07 $(ss_u, \mathrm{P}_u) \leftarrow (ss, \emptyset)$
s08 Return $\sigma$

Proc $ts(\sigma)$
t00 Parse $\sigma \parallel lt \parallel \mathrm{P} \parallel vs \parallel \mathrm{S}[\![lt]\!] \leftarrow \sigma$
t01 Return $lt$

Proc $vfy\langle st_u \rangle(m, \sigma)$
v00 Parse $\sigma \parallel lt \parallel \mathrm{P} \parallel vs \parallel \mathrm{S}[\![lt]\!] \leftarrow \sigma$
v01 $h \leftarrow H(m \parallel lt \parallel \mathrm{P} \parallel vs \parallel \mathrm{S}[\![lt]\!])$
v02 While $lt^*_u \le lt$:
v03     If $lt^*_u = lt$:
v04         $(\mathrm{P}', vs') \leftarrow (\mathrm{P}, vs)$
v05         $(h', \sigma') \leftarrow (h, \sigma)$
v06     Else:
v07         $(h', \mathrm{P}', vs', \sigma') \leftarrow \mathrm{S}[lt^*_u]$
v08     $vs^* \leftarrow vs^*_{\bar{u}}$
v09     For $i \in \mathrm{P}'$:
v10         $\mathrm{USS}.updvs\langle vs^* \rangle(\mathrm{AS}_u[i])$
v11     Require $\mathrm{USS}.vfy(vs^*, h', \sigma')$
v12     $vs^*_{\bar{u}} \leftarrow vs'$
v13     $av_u \leftarrow H(av_u \parallel h' \parallel \sigma')$
v14     $\mathrm{V}_u[lt^*_u] \leftarrow (h', \sigma')$
v15     $\mathrm{USS}.updss\langle ss_u \rangle(av_u)$
v16     $lt^*_u \leftarrow lt^*_u + 1$
v17     $\mathrm{P}_u \overset{\cup}{\leftarrow} \{lt^*_u\}$
v18 If $lt^*_u > lt$:
v19     Require $\mathrm{V}_u[lt] \neq \diamond$
v20     Require $\mathrm{V}_u[lt] = (h, \sigma)$
v21 $\mathrm{V}_u[lt] \leftarrow \diamond$
```

Fig. 4. BOOM-signature construction. We use an updatable signature scheme (USS) as building block. Function H is assumed to be a collision-resistant hash function. The *vfy* procedure aborts if parsing fails.

For each user $u \in \{\mathtt{A}, \mathtt{B}\}$ we initialize the signing index lt_u and the verifying index lt^*_u [i01], and the arrays S_u and V_u, which will store information about signed and verified messages, respectively [i02]. We generate pairs of USS signing and verification keys [i03] and initialize the set P_u of messages processed by the current signing key to be empty [i04]. We initialize the accumulated signed transcript AS_u [i05], set the first index [i06] and initialize the accumulated verified transcript av_u [i07]. Finally, we store everything in the users' states [i08].

The *sign* procedure first generates a new USS key pair [s00]. Next, it computes the hash of the message m, the signing index lt_u, the set of processed messages P_u, the verification state, and the array S_u [s01]. It signs the hash with its old key [s02]. It accumulates the new hash and signature in AS_u [s03] and stores the hash, processed set, verification key and signature in S_u [s04]. The signing index, processed set, verification key and array S_u are appended to the signature [s05]. It increments the signing index lt_u [s06] and stores the new signing key along with an empty processed set [s07], before returning the signature [s08].

The *vfy* procedure parses the additional information embedded in the signature [v00] and recomputes the hash [v01]. If the verifying index lt^*_u is less or equal than lt, the verifier will iteratively check signatures until it catches up [v02–v17]. To be concrete, if $lt^*_u = lt$ it will use the current value for the hash and signature [v05] or if $lt^*_u < lt$ it will obtain these values from $\mathrm{S}[lt^*_u]$ [v07]. It will update a copy of its verification key for all indices the signer has processed since generating its signing key [v08–v10] and verify the signature [v11]. Note it uses the transcript for its signed messages to update its verification key, which should match the transcript for the verified messages the signer has used to update its signing key. If signature verification passes, it will replace the verification key [v12]. Note that if USS.*vfy* failed the verification key remains unchanged, as if [v09–v10] were never executed. Next, it accumulates the hash and signature in its verified transcript av_u [v13], stores the hash and signature in $\mathrm{V}_u[lt^*_u]$ for later comparison [v14] and it will update its signing state with av_u [v15]. It increments the index lt^*_u [v16] and add lt^*_u to P_u to indicate it has processed this message into its signing key [v17]. If the verifying index lt^*_u was strictly greater than lt, the verifier will check if an entry exists for this index [v19] and compare whether it is equal to the value of the hash and signature [v20]. At last, the verifier will remove the entry in V_u for index lt to prevent double delivery [v21].

Note for simplicity we omit code lines to 'clean up' variables that are no longer needed. These lines are not required for security, but would help for efficiency. For example, if a party learns its peer has processed signature i, it will no longer have to include the first i entries of S_u in its next signature.

5.2 BOOM-KEM Scheme

In Sect. 5.3 we will use a specialized KEM to achieve confidentiality for our BOOM construction. In this section we describe the inner workings of this cryptographic tool.

SYNTAX. A BOOM-KEM scheme for a key space $\mathcal{K}$ consists of a state space $\mathcal{ST}$, a ciphertext space $\mathcal{C}$, and algorithms *init*, *upd*, *expire*, *enc*, *dec* and the timestamp decoder *ts* that recovers the logical and physical time.

$$\mathit{init} \to \mathcal{ST} \times \mathcal{ST} \qquad \mathit{upd}\langle\mathcal{ST}\rangle \qquad \mathit{expire}\langle\mathcal{ST}\rangle \qquad \mathit{ts}\colon \mathcal{C} \to \mathbb{N} \times \mathbb{N}$$

$$\mathcal{AD} \to enc\langle \mathcal{ST} \rangle \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{AD} \times \mathcal{C} \to dec\langle \mathcal{ST} \rangle \to \mathcal{K}.$$

Construction. Internally our BOOM-KEM construction will invoke the KuKEM primitive introduced in Sect. 4.2, the KeKEM primitive introduced in Sect. 4.3, and a secure KEM combiner K such that if at least one of the input keys is indistinguishable from a uniformly random string of equal length, then so is the output key. In this article we will consider K a random oracle. An implementation could use the CCA secure combiner presented in [16].

We noted both our KuKEM and KeKEM building block can be built generically from hierarchical identity-based encryption (HIBE, [15]). This strong component, while inefficient, should come as no surprise as it has already been proposed by [17] and [24] in the much simpler setting where every message is always delivered, and always in order. Moreover, recent work [6] shows that if an exposure additionally reveals the random coins used for the next *send* operation, the use of KuKEM is required to achieve confidentiality. They hypothesize the same implication holds without revealing the random coins and provide a strong intuition, but a formal proof remains an open problem.

We remark that both our KuKEM and KeKEM can be built from a single HIBE instance if one immediately delegates the master secret key to a 'KuKEM identity' and to a 'KeKEM identity'. We avoid doing so for two reasons. First of all, these primitives correspond to two perpendicular security goals. It is conceptually easier to grasp if we do not intertwine them. Secondly, KeKEM can be built from a forward-secure KEM, which is a simpler primitive than the HIBE-KEM used for KuKEM. Thus it may also be more efficient to separate them.

We provide a construction for a BOOM-KEM in Fig. 5. The construction consists of six procedures: *init*, *enc*, *dec*, *expire*, *upd* and *ts*. A correct decryption procedure *dec* is determined by the encryption procedure: it mirrors the operations in *enc*. As deriving the *dec* procedure is a rather vacuous technical exercise we have omitted it from Fig. 5 to focus on the more interesting cryptographic procedures instead. We have also omitted the *ts* procedure which simply parses the timestamps embedded in each ciphertext. A full reconstruction of all BOOM-KEM procedures is provided in the full version. The construction is quite technical but the general idea is to generate a new KuKEM and a new KeKEM instance with every *enc* invocation for post-compromise security. We update the KeKEM for forward secrecy in physical time, and the KuKEM for forward secrecy in logical time. The *enc* procedure will output a key dependent on the output of the KuKEM encapsulation procedure, the KeKEM encapsulation procedure and the associated data input.

We remark the physical time updates must be a separate primitive as simply updating the KuKEM would render the users out-of-sync. For example, consider the scenario where Alice sends a message, updating her KuKEM. Now physical time advances and both Alice and Bob would update their KuKEM. Finally, Bob receives Alice's message and updates his KuKEM. Clearly the updates have occurred in a different order, hence correctness would fail.

We note our security notion implies ciphertexts must contain information about prior ciphertexts. To see that ciphertexts cannot be independent, consider

```
**Proc** $init$
i00 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
i01     $lt_u \leftarrow 0$; $lt^*_u \leftarrow 0$
i02     $ft_u \leftarrow 0$; $pt_u \leftarrow 0$
i03     $\mathrm{AS}_u[\cdot] \leftarrow \bot$
i04     $\mathrm{AR}_u[\cdot] \leftarrow \bot$
i05     $\mathrm{AS}_u[lt_u] \leftarrow H()$
i06     $\mathrm{AR}_u[lt^*_u] \leftarrow H()$
i07     $E_u[\cdot] \leftarrow \bot$
i08     $U_u[\cdot] \leftarrow \bot$
i09     $(E_u[lt_u], \varepsilon^*_{\bar{u}}) \leftarrow \mathrm{ke}.gen(pt_u)$
i10     $(U_u[lt_u], \upsilon^*_{\bar{u}}) \leftarrow \mathrm{ku}.gen$
i11     $\mathrm{KC}_u[\cdot] \leftarrow \bot$
i12     $\mathrm{DK}_u[\cdot] \leftarrow \bot$
i13     $st_u := (\ldots)$
i14 Return $(st_{\mathtt{A}}, st_{\mathtt{B}})$

**Proc** $expire\langle st_u \rangle$
x00 $ft_u \leftarrow ft_u + 1$
x01 For $i \in [lt_u]$:
x02     $\mathrm{ke}.expire\langle E_u[i] \rangle$
```

```
**Proc** $enc\langle st_u \rangle(ad)$
e00 $(k_0, c_0) \leftarrow \mathrm{ke}.enc(\varepsilon^*_u)$
e01 $(k_1, c_1) \leftarrow \mathrm{ku}.enc(\upsilon^*_u)$
e02 $\mathrm{KC}_u[lt_u] \leftarrow (lt^*_u, c_1)$
e03 $(E_u[lt_u], \varepsilon) \leftarrow \mathrm{ke}.gen(pt_u)$
e04 $(U_u[lt_u], \upsilon) \leftarrow \mathrm{ku}.gen$
e05 $\mathrm{ku}.updss\langle U_u[lt_u] \rangle(\mathrm{AR}_u[lt^*_u])$
e06 $c \leftarrow lt_u \parallel pt_u \parallel \varepsilon \parallel \upsilon$
e07 $c \overset{\parallel}{\leftarrow} c_0 \parallel \mathrm{KC}_u[*] \parallel \mathrm{AS}_u[*]$
e08 $adc \leftarrow ad \parallel c$
e09 $k \leftarrow K(k_0, k_1; adc)$
e10 $lt_u \leftarrow lt_u + 1$
e11 $\mathrm{AS}_u[lt_u] \leftarrow H(\mathrm{AS}_u[lt_u - 1] \parallel adc)$
e12 $\mathrm{ku}.updps\langle \upsilon^*_u \rangle(\mathrm{AS}[lt_u])$
e13 Return $(k, c)$

**Proc** $upd\langle st_u \rangle$
u00 $pt_u \leftarrow pt_u + 1$
u01 $\mathrm{ke}.evolveps\langle \epsilon^*_u \rangle$
u02 For $i \in [lt]_u$:
u03     $\mathrm{ke}.evolvess\langle E_u[i] \rangle$
```

Fig. 5. BOOM-KEM construction. Building blocks are a KeKEM, whose algorithms are prefixed with 'ke.', a KuKEM, whose algorithms are prefixed with 'ku.' and a KEM combiner K.

an adversary that exposes Alice and creates two ciphertexts. The adversary will deliver the second ciphertext to Bob, rendering Bob out-of-sync. Now the adversary can challenge Alice, making her send her first ciphertext, and since Bob is out-of-sync, expose Bob. If Bob were able to decrypt any ciphertext with logical index 1, the adversary could now decrypt Alice's challenge ciphertext and win the confidentiality game. Hence, the second ciphertext must 'pin' the first.

We achieve this with the KEM/DEM encryption paradigm. The *enc* procedure will embed past KuKEM ciphertexts in the current ciphertext. When receiving a ciphertext, the *dec* procedure will decapsulate all embedded KuKEM ciphertexts, store the DEM keys and destroy its capability to decapsulate again. Reconsidering our example above, Bob is now only able to decrypt the first ciphertext if it was encrypted with the same DEM key he obtained from the second ciphertext, and Bob has no capability to decapsulate another KuKEM ciphertext. The probability that Alice and the adversary had generated the same KuKEM ciphertext for the first ciphertext is negligible.

We now discuss the procedures in more detail, starting with *init*. For each user the *init* procedure initializes a sending index lt_u, a receiving index lt^*_u, the first physical time that is still recoverable ft_u and the current physical time pt_u [i01–i02]. It initializes the array AS for the accumulated sent transcript and AR for the accumulated received transcript [i03–i06]. The accumulated transcripts will be used to update the KuKEM states, ensuring the user states diverge when users go out-of-sync. Because ciphertexts may be delivered out-of-order, or not at all, each user will be maintaining several instances of each primitive, ready to decapsulate ciphertexts for any of them. However, it will always encapsulate to the latest one. Hence we initialize storage for multiple secret states, but only one public state, and we store the first KeKEM and KuKEM instance [i07–i10]. Finally, we initialize the array KC to store KuKEM ciphertexts [i11] and the array DK to store DEM keys [i12], as described in the general construction overview.

The *enc* procedure encapsulates keys for both the KeKEM and the KuKEM [e00–e01], and stores the KuKEM ciphertext in KC, along with its receiver index lt^*_u, indicating which public states were used for encapsulation [e02]. Next, it generates a new instance for both the KeKEM and the KuKEM [e03–e04]. It will immediately update the secret state for the KuKEM with the received transcript [e05], as the adversary is allowed unrestricted expose queries if we are out-of-sync. The *enc* procedure combines the KEM ciphertexts into one ciphertext, adds the freshly generated public states, and includes the indices and the sending transcript such that the receiver can correctly update its state [e06–e07]. Subsequently, it uses the KEM-combiner K to produce a key, using the associated data and ciphertext as context [e09]. Finally, it increments the sending index lt_u [e10], accumulates the associated data and ciphertext into its transcript [e11] and updates its public KuKEM state with it [e12].

The *upd* procedure is quite straightforward: it simply updates the public state and evolves the secret states for all its KeKEM instances as physical time advances. Similarly, the *expire* procedure will update all secret states.

Note that for simplicity we have omitted code lines to ‘clean up’ variables that are no longer needed. These lines are not required for security, but would help for efficiency. For example when a user has either received or expired all messages encapsulated for its i-th KeKEM and KuKEM instance, it can drop instance i, as later keys will always be encapsulated to later instances. As another example we remark that, after receiving an acknowledgment from the other user they have received message i, a user would no longer have to embed all their KuKEM ciphertexts for indices less than or equal to i in their current ciphertext.

5.3 BOOM Construction

We first introduce a functional protocol and discuss it in detail before delving into the full BOOM construction that achieves authenticity and confidentiality. The functional protocol consists of all the unmarked code lines in Fig. 6. The protocol has four procedures: the initialization procedure *init*, which initializes the users’ initial states; the sending procedure *send*, which takes a state, associated data and a message, updates the state and outputs a ciphertext; the receiving procedure *recv*, which takes a state, associated data and a ciphertext, updates the state and outputs a message; and the time progression algorithm *tick*, which updates the state.

For each user u, the *init* procedure initializes the logical time lt_u and lt^*_u [i03], the physical time pt_u [i04], the set of received indices RI_u [i04], the set of received timestamps RT_u [i05], the set of received acknowledgments RA_u [i05],

```
  Proc $\mathit{init}$
○ i00 $(st_{\mathtt{A}}^{\mathrm{BS}}, st_{\mathtt{B}}^{\mathrm{BS}}) \leftarrow \mathrm{BS}.\mathit{init}$
● i01 $(st_{\mathtt{A}}^{\mathrm{BK}}, st_{\mathtt{B}}^{\mathrm{BK}}) \leftarrow \mathrm{BK}.\mathit{init}$
  i02 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
  i03     $lt_u \leftarrow 0$; $lt_u^* \leftarrow 0$
  i04     $pt_u \leftarrow 0$; $\mathrm{RI}_u \leftarrow \emptyset$
  i05     $\mathrm{RT}_u \leftarrow \emptyset$; $\mathrm{RA}_u \leftarrow \emptyset$
  i06     $\mathrm{HS}_u[\cdot] \leftarrow \bot$; $\mathrm{HR}_u[\cdot] \leftarrow \bot$
  i07     $st_u := (\ldots)$
  i08 Return $(st_{\mathtt{A}}, st_{\mathtt{B}})$

  Proc $\mathit{send}\langle st_u\rangle(ad, m)$
  s00 $\mathrm{ctx} \leftarrow lt_u \parallel pt_u \parallel \mathrm{HS}_u[*] \parallel \mathrm{RI}_u$
○ s01 $(sk, vk) \leftarrow \mathrm{OTS}.\mathit{gen}$
○ s02 $\sigma_1 \leftarrow \mathrm{BS}.\mathit{sign}\langle st_u^{\mathrm{BS}}\rangle(vk)$
○ s03 $\mathrm{ctx} \overset{\parallel}{\leftarrow} vk \parallel \sigma_1$
● s04 $(k, c') \leftarrow \mathrm{BK}.\mathit{enc}\langle st_u^{\mathrm{BK}}\rangle(vk)$
● s05 $\mathrm{ctx} \overset{\parallel}{\leftarrow} c'$
● s06 $m \leftarrow \mathrm{E}.\mathit{enc}(k, m)$
  s07 $c \leftarrow \mathrm{ctx} \parallel m$
○ s08 $\sigma_2 \leftarrow \mathrm{OTS}.\mathit{sign}(sk, ad \parallel c)$
○ s09 $c \overset{\parallel}{\leftarrow} \sigma_2$
  s10 $\mathrm{HS}_u[lt_u] \leftarrow H(ad \parallel c)$
  s11 $lt_u \leftarrow lt_u + 1$
  s12 Return $c$

  Proc $\mathit{ts}(c)$
  t00 Parse $lt_u \parallel pt_u \parallel \ldots \leftarrow c$
  t01 Return $(lt_u, pt_u)$
```

```
  Proc $\mathit{tick}\langle st_u\rangle$
  u00 $pt_u \leftarrow pt_u + 1$
● u01 $\mathrm{BK}.\mathit{upd}\langle st_u^{\mathrm{BK}}\rangle$
● u02 If $\Delta(0, pt_u) > \delta$: $\mathrm{BK}.\mathit{expire}\langle st_u^{\mathrm{BK}}\rangle$

  Proc $\mathit{recv}\langle st_u\rangle(ad, c)$
  r00 $h \leftarrow H(ad \parallel c)$
○ r01 $c \parallel \sigma_2 \leftarrow c$
  r02 Parse $\mathrm{ctx} \parallel m \leftarrow c$
● r03 Parse $\mathrm{ctx} \parallel c' \leftarrow \mathrm{ctx}$
○ r04 Parse $\mathrm{ctx} \parallel vk \parallel \sigma_1 \leftarrow \mathrm{ctx}$
  r05 Parse $lt \parallel pt \parallel \mathrm{HS}[*] \parallel R \leftarrow \mathrm{ctx}$
○ r06 $\mathrm{BS}.\mathit{vfy}\langle st_u^{\mathrm{BS}}\rangle(vk, \sigma_1)$
○ r07 $\mathrm{OTS}.\mathit{vfy}(vk, ad \parallel c, \sigma_2)$
  r08 Require $lt \notin \mathrm{RI}_u$
  r09 Require $\Delta(pt, pt_u) \leq \delta$
  r10 Require $\mathrm{RT}_u \cup \{(lt, pt)\}$ monotone
  r11 Require $R \subseteq [\![lt_u]\!]$
  r12 While $lt_u^* \leq lt$:
  r13     If $lt_u^* < lt$: $\mathrm{HR}_u[lt_u^*] \leftarrow \mathrm{HS}[lt_u^*]$
  r14     Else: $\mathrm{HR}_u[lt_u^*] \leftarrow h$
  r15     $lt_u^* \leftarrow lt_u^* + 1$
  r16 If $lt_u^* > lt$: Require $\mathrm{HR}_u[lt] = h$
  r17 $\mathrm{RI}_u \overset{\cup}{\leftarrow} \{lt\}$
  r18 $\mathrm{RT}_u \overset{\cup}{\leftarrow} \{(lt, pt)\}$
  r19 $\mathrm{RA}_u \overset{\cup}{\leftarrow} R$
● r20 $k \leftarrow \mathrm{BK}.\mathit{dec}\langle st_u^{\mathrm{BK}}\rangle(vk, c')$
● r21 $m \leftarrow \mathrm{E}.\mathit{dec}(k, m)$
  r22 Return $(\mathrm{RA}_u, m)$
```

Fig. 6. The functional construction consists of the unmarked lines. The authentic construction adds the lines marked with ○. The BOOM construction consists of all lines. BS is the BOOM-signature scheme construction in Fig. 4, BK is the BOOM-KEM construction in Fig. 5, OTS is a (one-time) signature scheme, and E is a symmetric encryption scheme.

and the arrays of hashed sent ciphertexts HS_u and hashed received ciphertexts HR_u [i06]. The *tick* procedure increments the user's physical time pt_u [u00].

The *send* procedure takes associated data ad and message m as input. It creates context ctx which includes the user's current time (lt_u, pt_u), the hashes of previously sent ciphertexts $\mathrm{HS}_u[*]$ and the set of received indices RI_u [s00]. The context ctx together with the message m will form the ciphertext c [s07]. Finally, it stores the hash $H(ad \parallel c)$ of the associated data and the ciphertext [s10], increments the logical time lt_u [s11] and returns the ciphertext [s12].

The *recv* procedure first hashes the ciphertext [r00] and subsequently parses it to obtain the message m [r02] and the context variables lt, pt, $\mathrm{HS}[*]$ and R [r05]. Now, recall that 'Require C' is short for 'If $\neg C$: Abort'. Thus the *recv* procedure performs four sanity checks to guarantee functionality. (1) A ciphertext has not yet been received for this logical time lt [r08]. (2) The ciphertext is fresh, that is the Δ difference between its physical creation time pt and the user's time pt_u is 'small' [r09]. (3) Time is monotonic: a message that is newer in logical time must be newer in physical time [r10]. (4) Only sent messages can be acknowledged [r11]: Bob cannot acknowledge having received a message that Alice never sent. Next, *recv* handles the out-of-order delivery. While u's receiving index lt_u^* is smaller or equal than lt, it will iteratively update its array HR_u with the received hashes that it obtains from $\mathrm{HS}[*]$ or the current ciphertext itself [r12–r15]. If u's receiving index is greater than lt, it will require the hash h of the current ciphertext is equal to the stored value for that index $\mathrm{HR}_u[lt]$ [r16]. Finally, it will update its set of received indices RI_u [r17], its set of received timestamps RT_u [r18], its set of received acknowledgments RA_u [r19], and return (RA_u, m) [r22].

We extend the functional protocol to an authentication protocol by including the lines marked with $\circ$. The *init* procedure now initializes a BOOM-signature scheme BS [i00]. The *send* procedure generates a fresh one-time signature key pair (sk, vk) [s01], and calls BS.*sign* to obtain a signature σ_1 on the verification key vk [s02]. We add vk and σ_1 to the context ctx [s03]. We use the signing key sk to sign the associated data ad and ciphertext c [s022], and append the signature σ_2 to c [s09]. The *recv* procedure will parse the newly added signatures and verification key [r01,r04]. It will first verify the signature on the verification key vk by calling BS.*vfy* [r06]. Then it uses vk to verify the signature on the associated data and ciphertext [r062].

It may appear peculiar not to sign the ciphertext directly with the BOOM-signature. However, this design decision is made to simplify the confidentiality construction. If we sign the ciphertext directly, the adversary could expose the user to obtain its signing key and generate a new signature for the ciphertext. Indeed, this would not break authenticity as the forgery is trivial. Nonetheless, if the adversary submits the ciphertext to the Recv oracle with a different signature, the oracle will decrypt and return the (challenge) message. Now, because the one-time signature key pair is generated during the *send* procedure, it cannot be exposed. Thus, if the adversary succeeds in creating a valid but different signature, this would break the strong unforgeability property.

This brings us to the lines marked with $\bullet$. Including these lines provides confidentiality, resulting in our BOOM protocol. The *init* procedure now also initializes a BOOM-KEM BK [i01]. The *send* procedure provides the BOOM-KEM with the verification key as context when requesting (k, c') [s04], appends c' to the context [s05], and uses k to encrypt the message [s06].

The adversary could have exposed the sender's state and created a (trivial) forgery by generating its own one-time signature pair. The Recv oracle would accept the ciphertext and attempt to decrypt it. Therefore, it is critical for confidentiality that the key derivation is dependent on the verification key [s04]. The *recv* procedure parses the newly added c' [r03] and inputs it, along with vk, to BK.*dec* to retrieve k [r20]. Subsequently, *recv* uses k to decrypt m [r20rypt].

The *tick* procedure now calls BK.*upd* [u01] because its state must advance over time, even when no messages are exchanged, to achieve forward secrecy in physical time. Once time has advanced δ times it will start calling BK.*expire* [u02] to indicate we no longer desire to be able to decrypt 'old' messages. Neither of these procedures require the physical time as input because they advance linearly over time, with the expire procedure lagging behind the update procedure. This completes the description of our BOOM protocol in Fig. 6.

Our construction provides authenticity and confidentiality. The proofs are in the full version.

Theorem 1. *Let π be the BOOM construction in Fig. 6, let* AUTH *be the authenticity game in Fig. 2 that calls π's procedures in its oracles, let H be a perfectly collision resistant hash function, and let $\mathcal{A}$ be an adversary that makes at most q_s* Send *queries. Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that*

$$\mathbf{Adv}_{\pi}^{\mathrm{AUTH}}(\mathcal{A}) \leq q_s \cdot \left(\mathbf{Adv}_{\mathrm{OTS}}^{\mathrm{SUF}}(\mathcal{A}') + \mathbf{Adv}_{\mathrm{USS}}^{\mathrm{AUTH}}(\mathcal{A}')\right).$$

Theorem 2. *Let π be the BOOM construction in Fig. 6, let* CONF *be the confidentiality game in Fig. 3 that calls π's procedures in its oracles and let $\mathcal{A}$ be an adversary that makes at most q_c* Chal *queries and ϵ the probability that the adversary successfully computes a pre-image of the random oracle. Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that $\mathbf{Adv}_{\pi}^{\mathrm{CONF}}(\mathcal{A}) \leq$*

$$2q_c\left(\mathbf{Adv}_{\mathrm{keKEM}}^{\mathrm{CONF}}(\mathcal{A}') + \mathbf{Adv}_{\mathrm{kuKEM}}^{\mathrm{CONF}}(\mathcal{A}') + \mathbf{Adv}_{\mathrm{E}}^{\mathrm{CONF}}(\mathcal{A}')\right) + \mathbf{Adv}_{\pi}^{\mathrm{AUTH}}(\mathcal{A}') + \epsilon.$$

6 Conclusion

After ACD [2] observed that research on secure messaging protocols routinely only considers settings with a guaranteed in-order delivery of messages, while most real-world protocols like Signal are actually designed for out-of-order delivery, we reassess the model and construction of ACD and argue that the intuitive notion of forward secrecy is not provided. We identify that the reason for this is the lack of modeling of physical time, which is required to express that ciphertexts may time out and expire. We hence develop new security models for the out-of-order delivery setting with immediate decryption. Our model incorporates the concept of physical clocks and implements a maximally strong corruption model. We finally design a proof-of-concept protocol that provably satisfies it.

References

1. Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: CoCoA: Concurrent continuous group key agreement. In: EUROCRYPT 2022 (2022)
2. Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2_5

3. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56784-2_9
4. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1463–1483. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3484820
5. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg (Nov 2020). https://doi.org/10.1007/978-3-030-64378-2_10
6. Balli, F., Rösler, P., Vaudenay, S.: Determining the core primitive for optimally secure ratcheting. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part III. LNCS, vol. 12493, pp. 621–650. Springer, Heidelberg (Dec 2020). https://doi.org/10.1007/978-3-030-64840-4_21
7. Bellare, M., Kohno, T., Namprempre, C.: Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In: Atluri, V. (ed.) ACM CCS 2002. pp. 1–11. ACM Press (Nov 2002). https://doi.org/10.1145/586110.586112
8. Bellare, M., Miner, S.K.: A forward-secure digital signature scheme. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_28
9. Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the Signal Double Ratchet algorithm. In: CRYPTO. Lecture Notes in Computer Science, vol. (to appear), pp. 1–30. Springer (2022)
10. Caforio, A., Durak, F.B., Vaudenay, S.: On-demand ratcheting with security awareness. Cryptology ePrint Archive, Report 2019/965 (2019), https://eprint.iacr.org/2019/965
11. Canetti, R., Halevi, S., Katz, J.: A forward-secure public-key encryption scheme. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 255–271. Springer, Heidelberg (May 2003). https://doi.org/10.1007/3-540-39200-9_16
12. Canetti, R., Jain, P., Swanberg, M., Varia, M.: Universally composable end-to-end secure messaging. In: CRYPTO. Lecture Notes in Computer Science, vol. (to appear), pp. 1–30. Springer (2022)
13. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 451–466 (2017)
14. Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 19. LNCS, vol. 11689, pp. 343–362. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26834-3_20
15. Gentry, C., Silverberg, A.: Hierarchical ID-based cryptography. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 548–566. Springer, Heidelberg (Dec 2002). https://doi.org/10.1007/3-540-36178-2_34
16. Giacon, F., Heuer, F., Poettering, B.: KEM combiners. In: Abdalla, M., Dahab, R. (eds.) PKC 2018, Part I. LNCS, vol. 10769, pp. 190–218. Springer, Heidelberg (Mar 2018). https://doi.org/10.1007/978-3-319-76578-5_7
17. Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 33–62. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_2

18. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2_6
19. Jost, D., Maurer, U., Mularczyk, M.: A unified and composable take on ratcheting. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019, Part II. LNCS, vol. 11892, pp. 180–210. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-36033-7_7
20. Li, C., Palanisamy, B.: Timed-release of self-emerging data using distributed hash tables. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 2344–2351 (2017)
21. Liu, J., Jager, T., Kakvi, S.A., Warinschi, B.: How to build time-lock encryption. Designs, Codes and Cryptography **86**(11), 2549–2586 (2018)
22. Marlinspike, M., Perrin, T.: The Double Ratchet Algorithm (November 2016), https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf
23. Marson, G.A., Poettering, B.: Security notions for bidirectional channels. IACR Trans. Symm. Cryptol. **2017**(1), 405–426 (2017). https://doi.org/10.13154/tosc.v2017.i1.405-426
24. Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 3–32. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_1
25. Schwenk, J.: Modelling time for authenticated key exchange protocols. In: Kutylowski, M., Vaidya, J. (eds.) ESORICS 2014, Part II. LNCS, vol. 8713, pp. 277–294. Springer, Heidelberg (Sep 2014). https://doi.org/10.1007/978-3-319-11212-1_16
26. Yan, H., Vaudenay, S.: Symmetric asynchronous ratcheted communication with associated data. In: Aoki, K., Kanaoka, A. (eds.) IWSEC 20. LNCS, vol. 12231, pp. 184–204. Springer, Heidelberg (Sep 2020). https://doi.org/10.1007/978-3-030-58208-1_11