Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging

Daniel Jost[0000−0002−6562−9665], Ueli Maurer, and Marta Mularczyk*

Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland.
{dajost,maurer,mumarta}@inf.ethz.ch

Abstract. In the era of mass surveillance and information breaches, privacy of Internet communication, and messaging in particular, is a growing concern. As secure messaging protocols are executed on the not-so-secure end-user devices, and because their sessions are long-lived, they aim to guarantee strong security even if secret states and local randomness can be exposed.

The most basic security properties, including forward secrecy, can be achieved using standard techniques such as authenticated encryption. Modern protocols, such as Signal, go one step further and additionally provide the so-called backward secrecy, or healing from state exposures. These additional guarantees come at the price of a moderate efficiency loss (they require public-key primitives).

On the opposite side of the security spectrum are the works by Jaeger and Stepanovs and by Poettering and Rösler, which characterize the optimal security a secure-messaging scheme can achieve. However, their proof-of-concept constructions suffer from an extreme efficiency loss compared to Signal. Moreover, this caveat seems inherent.

This paper explores the area in between: our starting point are the basic, efficient constructions, and then we ask how far we can go towards the optimal security without losing too much efficiency. We present a construction with guarantees much stronger than those achieved by Signal, and slightly weaker than optimal, yet its efficiency is closer to that of Signal (only standard public-key cryptography is used).

On a technical level, achieving optimal guarantees inherently requires key-updating public-key primitives, where the update information is allowed to be public. We consider secret update information instead. Since a state exposure temporally breaks confidentiality, we carefully design such secretly-updatable primitives whose security degrades gracefully if the supposedly secret update information leaks.

1 Introduction and Motivation

1.1 Motivation

The goal of a secure-messaging protocol is to allow two parties, which we from now on call Alice and Bob, to securely exchange messages over asynchronous

* Research was supported by the Zurich Information Security and Privacy Center (ZISC).

communication channels in any arbitrary interleaving, without an adversary being able to read, alter, or inject new messages.

Since mobile devices have become a ubiquitous part of our lives, secure-messaging protocols are almost always run on such end-user devices. It is generally known, however, that such devices are often not very powerful and vulnerable to all kinds of attacks, including viruses which compromise memory contents, corrupted randomness generators, and many more [14, 13]. What makes it even worse is the fact that the sessions are usually long-lived, which requires storing the session-related secret information for long periods of time. In this situation it becomes essential to design protocols that provide some security guarantees even in the setting where the memory contents and intermediate values of computation (including the randomness) can be exposed.

The security guarantee which is easiest to provide is *forward secrecy*, which, in case of an exposure, protects confidentiality of previously exchanged messages. It can be achieved using symmetric primitives, such as stateful authenticated encryption [2].

Further, one can consider *healing* (also known as post-compromise recovery or backward secrecy). Roughly, this means that if after a compromise the parties manage to exchange a couple of messages, then the security is restored.[1] Providing this property was the design goal for some modern protocols, such as OTR [5] and Signal [15]. The price for additional security is a loss of efficiency: in both of the above protocols the parties regularly perform a Diffie-Hellman key exchange (public-key cryptography is necessary for healing). Moreover, the above technique does not achieve optimal post-compromise recovery (in particular, healing takes at least one full round-trip). The actual security achieved by Signal was recently analyzed by Cohn-Gordon et al. [7].

This raises a more conceptual question: what security guarantees of secure messaging are even possible to achieve? This question was first formulated by Bellare et al. [4], who abstract the concept of *ratcheting* and formalize the notions of ratcheted key exchange and communication. However, they only consider a very limited setting, where the exposures only affect the state of one of the parties. More recently, Jaeger and Stepanovs [11], and Poettering and Rösler [16] both formulated the optimal security guarantees achievable by secure messaging. To this end, they start with a utopian definition, which cannot be satisfied by any correct scheme. Then, one by one, they disable all generic attacks, until they end with a formalization for which they can provide a proof-of-concept construction. (One difference between the two formalizations is that [11] considers exposing intermediate values used in the computation, while [16] does not.) The resulting optimal security implies many additional properties, which were not considered before. For example, it requires *post-impersonation security*, which concerns messages sent after an active attack, where the attacker uses an exposed state to impersonate a party (we will say that the partner of the impersonated party is *hijacked*).

[1] Of course, for the healing to take effect, the adversary must remain passive and not immediately use the compromised state to impersonate a party.

Unfortunately, these strong guarantees come at a high price. Both constructions [11, 16] use very inefficient primitives, such as hierarchical identity-based encryption (HIBE) [9, 10]. Moreover, it seems that an optimally-secure protocol would in fact imply HIBE.

This leads to a wide area of mostly unexplored trade-offs with respect to security and efficiency, raising the question how much security can be obtained at what efficiency.

1.2 Contributions

In this work we contribute to a number of steps towards characterizing the area of sub-optimal security. We present an efficient secure-messaging protocol with almost-optimal security in the setting where *both* the memory and the intermediate values used in the computation can be exposed.

Unlike the work on optimal security [11, 16], we start from the basic techniques, and gradually build towards the strongest possible security. Our final construction is based on standard digital signatures and CCA-secure public-key encryption. The ciphertext size is constant, and the size of the secret state grows linearly with the number of messages sent since the last received message (one can prove that the state size cannot be constant). We formalize the precise security guarantees achieved in terms of game-based security definitions.

Intuitively, the almost-optimal security comes short of optimal in that in two specific situations we do not provide post-impersonation security. The first situation concerns exposing the randomness of one of *two* specific messages,[2] and in the second, the secret states of both parties must be exposed at almost the same time. The latter scenario seems rather contrived: if the parties were exposed at *exactly* the same time, then any security would anyway be impossible. However, one could imagine that the adversary suddenly loses access to one of the states, making it possible to restore it. Almost-optimal guarantees mean that the security need not be restored in this case.

It turns out that dealing with exposures of the computation randomness is particularly difficult. For example, certain subtle issues made us rely on a circularly-secure encryption scheme. Hence, we present our overall proof in the random oracle model. We stress, however, that the random oracle assumption is only necessary to provide additional guarantees when the randomness can leak.

1.3 Further Related Work

Most work on secure messaging [4, 11, 16, 8], including this paper, considers the situation where messages can only be decrypted in order (so out-of-order messages must be either buffered or dropped). In a recent work, Alwen, Coretti and Dodis [1] consider a different setting in which it is required that any honestly-generated message can be immediately decrypted. The authors motivate this property by practical aspects, as for example immediate decryption is necessary to prevent

[2] Namely, the messages sent right before or right after an active impersonation attack.

certain denial-of-service attacks. Moreover, immediate decryption is actually achieved by Signal. This setting requires different definitions of both authenticity and correctness. Moreover, requiring the ability to immediately decrypt messages appears to incur a significant hit on the post-impersonation security a protocol can guarantee.

We find it very interesting to analyze the optimal and sub-optimal security guarantees in the setting of [1], and how providing them impacts the efficiency. However, this is not the focus of this work. Note that most practical secure messengers buffer the messages on a central server, so that even if parties are intermittently offline, they receive all their messages once they go online. Hence, not handling out-of-order messages should not significantly affect practicality.

In a recent concurrent and independent work, Durak and Vaudenay [8] also present a very efficient asynchronous communication protocol with sub-optimal security. However, their setting, in contrast to ours, explicitly excludes exposing intermediate values used in computation, in particular, the randomness. Allowing exposure of the randomness seems much closer to reality. Why would we assume that the memory of a device can be insecure, but the sampled randomness is *perfect*? Our construction provides strong security if the randomness fully leaks, while [8] gives no guarantees even if a very small amount of partial information is revealed. In fact, it is not clear how to modify the construction of [8] to work in the setting with randomness exposures. We note that the proof of [8], in contrast to ours, is in the standard model. On the other hand, we only need the random oracle to provide the additional guarantees not considered in [8].

2 Towards Optimal Security Guarantees

In this section we present a high-level overview of the steps that take us from the basic security properties (for example, those provided by Signal) towards the almost-optimal security, which we later implement in our final construction. We stress that all constructions use only standard primitives, such as digital signatures and public-key encryption. The security proofs are in the random oracle model.

2.1 Authentication

We start with the basic idea of using digital signatures and sequence numbers. These simple techniques break down in the presence of state exposures: once a party's signing key is exposed, the adversary can inject messages at any time in the future. To prevent this and guarantee healing in the case where the adversary remains passive, we can use the following idea. Each party samples a fresh signing and verification key with each message, sends along the new (signed) verification key, and stores the fresh signing key to be used for the next message. If either of the parties' state gets exposed, say Alice's, then Eve obtains her current signing key that she can use to impersonate Alice towards Bob at this point in time. If, however, Alice's next message containing a fresh verification key has already been delivered, then the signing key captured by the adversary becomes useless thereby achieving the healing property.

The above technique already allows to achieve quite meaningful guarantees: in fact, it only ignores post-impersonation security. We implement this idea and formalize the security guarantees of the resulting construction in Section 3.

2.2 Confidentiality

Assume now that all communication is authentic, and that none of the parties gets impersonated (that is, assume that the adversary does not inject messages when he is allowed to do so). How can we get forward secrecy and healing?

Forward secrecy itself can be achieved using standard forward-secure authenticated encryption in each direction (this corresponds to Signal's symmetric ratcheting layer). However, this technique provides no healing.

Perfectly Interlocked Communication. The first, basic idea to guarantee healing is to use public-key encryption, with separate keys per direction, and constantly exchange fresh keys. The protocol is sketched in Figure 1. Note that instead of using a PKE scheme, we could also use a KEM scheme and apply the KEM-DEM principle, which is essentially what Signal does for its asymmetric ratcheting layer.

Let us consider the security guarantees offered by this solution. Assume for the moment that Alice and Bob communicate in a completely interlocked manner, i.e., Alice sends one message, Bob replies to that message, and so on. This situation is depicted in Figure 1. Exposing the state of a party, say Alice, right after sending a message ($dk_{\mathsf{A}}^1, ek_{\mathsf{B}}^0$ in the figure) clearly allows to decrypt the next message (m_2), which is unavoidable due to the correctness requirement. However, it no longer affects the confidentiality of any other messages. Further, exposing the state right after receiving a message has absolutely no effect (note that a party can delete its secret key immediately after decrypting, since it will no longer be used). Moreover, exposing the sending or receiving randomness is clearly no worse than exposing both the state right before and after this operation. Hence, our scheme obtains optimal confidentiality guarantees (including forward-secrecy and healing) when the parties communicate in such a turn-by-turn manner.

The Unidirectional Case. The problems with the above approach arise when the communication is not perfectly interlocked. Consider the situation when Alice sends many messages without receiving anything from Bob. The straightforward solution to encrypt all these messages with the same key breaks forward secrecy: Bob can no longer delete his secret key immediately after receiving a message, so exposing his state would expose many messages received by him in the past. This immediately suggests using forward-secure public-key encryption [6], or the closely-related HIBE [9, 10] (as in the works by Jaeger et al. and Poettering et al.). However, we crucially want to avoid using such expensive techniques.

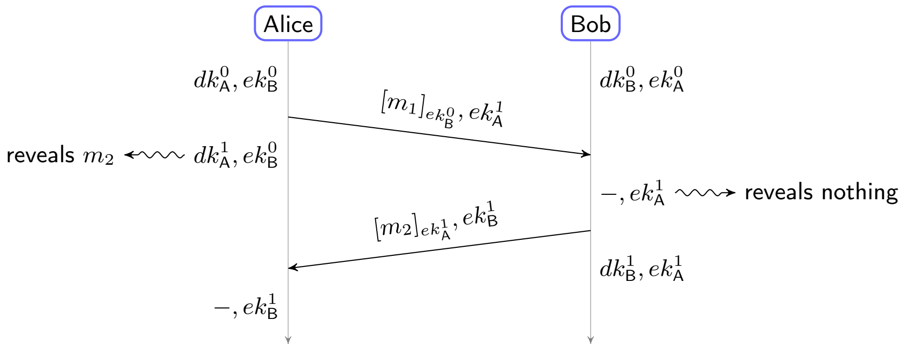


Fig. 1. Constantly exchanging fresh public-keys achieves optimal security when communication is authenticated and in a strict turn-by-turn fashion.

The partial solution offered by Signal is the symmetric ratcheting. In essence, Alice uses the public key once to transmit a fresh shared secret, which can then be used with forward-secure authenticated encryption. However, this solution offers very limited healing guarantees: when Alice's state is exposed, all messages sent by her in the future (or until she receives a new public key from Bob) are exposed. Can we do something better?

The first alternative solution which comes to mind is the following. When encrypting a message, Alice samples a fresh key pair for a public-key encryption scheme, transmits the secret key encrypted along with the message, stores the public key and deletes the secret key. This public key is then used by Alice to send the next message. This approach is depicted in Figure 2. However, this solution totally breaks if the sending randomness does leak. In essence, exposing Alice's randomness causes a large part of Bob's next state to be exposed, hence, we achieve roughly the same guarantees as Signal's symmetric ratcheting.

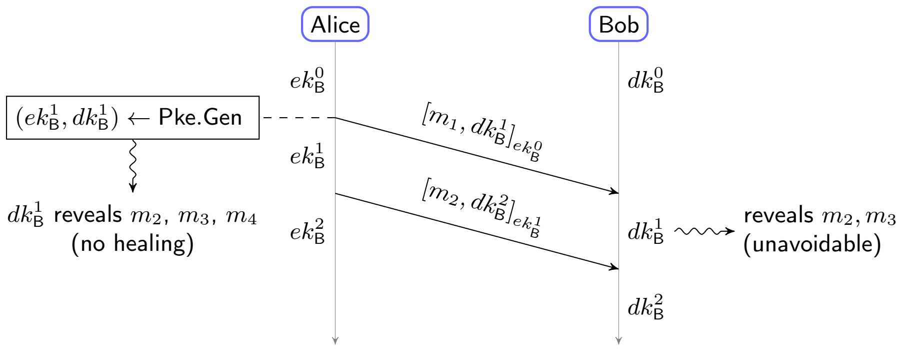


Fig. 2. First attempt to handle asynchronous messages, where one party (here Alice) can send multiple messages in a row. This solution breaks totally when the randomness can leak.

Hence, our approach will make the new decryption key depend on the previous decryption key, and not solely on the update information sent by Alice. We note that, for forward secrecy, we still rely on the update information being transmitted confidentially. This technique achieves optimal security up to impersonation (that is, we get the same guarantees as for simple authentication). The solution is depicted in Figure 3. At a high level, we use the ElGamal encryption, where a key pair of Bob is (b_0, g^{b_0}) for some generator g of a cyclic group. While sending a message, Alice sends a new secret exponent b_1 encrypted under g^{b_0}, the new encryption key is $g^{b_0}g^{b_1}$, and the new decryption key is $b_0 + b_1$.[3] This idea is formalized in Section 4.

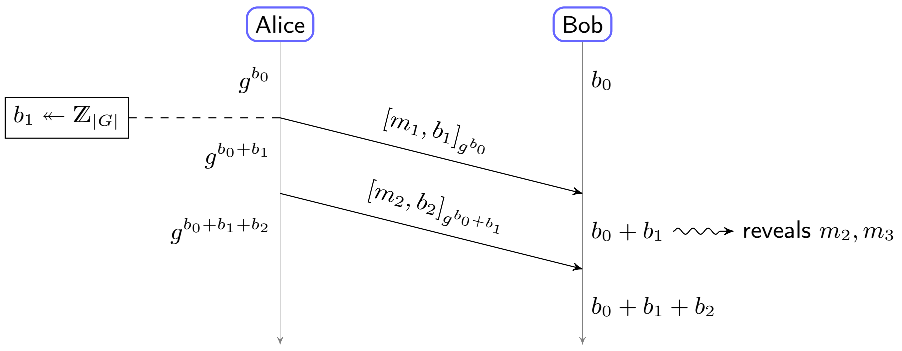


Fig. 3. Second attempt to handle asynchronous messages, where one party (here Alice) can send multiple messages in a row.

2.3 A First Efficient Scheme

Combining the solutions for authentication and confidentiality from the previous subsections already yields a very efficient scheme with meaningful guarantees. Namely, we only give up on the post-impersonation security. That is, we achieve the optimal guarantees up to the event that an adversary uses the exposed state of a party to inject a message to the other party.

One may argue that such a construction is in fact the one that should be used in practice. Indeed, the only guarantees we can hope for after such an active impersonation concern the party that gets impersonated, say Alice, towards the other one, say Bob: Alice should not accept any messages from Bob or the adversary anymore, and the messages she sends should remain confidential. Observe that the former guarantee potentially enables Alice to detect the attack by the lack of replies to her messages. However, providing those guarantees to

[3] Looking ahead, it turns out that in order to prove the security of this construction, we need circular-secure encryption. We achieve this in the random oracle model.

their full extent seems to inherently require very inefficient tools, such as HIBE, in contrast to the quite efficient scheme outlined above.

In the next subsections we make further steps towards our final construction, which provides some, but not all, after-impersonation guarantees, thereby compromising between efficiency and security.

2.4 Post-Impersonation Authentication

Consider the situation where the adversary exposes the state of Alice and uses it to impersonate her towards Bob (that is, he hijacks Bob). Clearly, due to the correctness requirement, the adversary can now send further messages to Bob. For the optimal security, we would require that an adversary cannot make Alice accept any messages from Bob anymore, even given Bob's state exposed at any time after the impersonation.

Note that our simple authentication scheme from Section 2.1 does not achieve this property, as Bob's state contains the signing key at this point. It does not even guarantee that Alice does not accept messages sent by the honest Bob anymore. The latter issue we can easily fix by sending a hash of the communication transcript along with each message. That is, the parties keep a value h (initially 0), which Alice updates as $h \leftarrow \mathrm{Hash}(h \parallel m)$ with every message m she sends, and which Bob updates accordingly with every received message. Moreover, Bob accepts a message only if it is sent together with a matching hash h.

To achieve the stronger guarantee against an adversary obtaining Bob's state, we additionally use ephemeral signing keys. With each message, Alice generates a new signing key, which she securely sends to Bob, and expects Bob to sign his next message with. Intuitively, the adversary's injection "overwrites" this ephemeral key, rendering Bob's state useless. Note that for this to work, we need the last message received by Bob before hijacking to be confidential. This is not the case, for example, if the sending randomness leaks.[4] For this reason, we do not achieve optimal security. In the existing optimal constructions [16, 11] the update information can be public, which, unfortunately, seems to require very strong primitives, such as forward-secure signatures.

2.5 Post-Impersonation Confidentiality

In this section we focus on the case where the adversary impersonates Alice towards Bob (since this is only possible if Alice's state exposed, we now consider her state to be a public value).

Consider once more the two approaches to provide confidentiality in the unidirectional case, presented in Section 2.2 (Figures 2 and 3). Observe that if we assume that the randomness cannot be exposed, then the first solution from Figure 2, where Alice sends (encrypted) a fresh decryption key for Bob, already achieves very good guarantees. In essence, during impersonation the

[4] Note that this also makes the choice of abstraction levels particularly difficult, as we need confidentiality, in order to obtain authentication.

adversary has to choose a new decryption key (consider the adversary sending $[m_3, \bar{dk}^3_{\mathsf{B}}]_{ek^2_{\mathsf{B}}}$ in the figure), which overwrites Bob's state. Hence, the information needed to decrypt the messages sent by Alice from this point on (namely, dk^2_{B}) is lost.[5] In contrast, the second solution from Figure 3 provides no guarantees for post-impersonation messages: after injecting a message and exposing Bob's state, the adversary can easily compute Bob's state from right before the impersonation and use it to decrypt Alice's messages sent after the attack.

While the former idea has been used in [8] to construct an efficient scheme with almost-optimal security for the setting where the randomness generator is perfectly protected, we aim at also providing guarantees in the setting where the randomness can leak. To achieve this, we combine the two approaches, using both updating keys from the latter scheme and ephemeral keys from the former one, in a manner analogous to how we achieved post-impersonation authentication. More concretely, Alice now sends (encrypted), in addition to the exponent, a fresh ephemeral decryption key, and stores the corresponding encryption key, which she uses to additionally encrypt her next message. Now the adversary's injected message causes the ephemeral decryption key of Bob to be overwritten.

As was the case for authentication, this solution does not provide optimal security, since we rely on the fact that the last message, say c, received before impersonation, is confidential. Moreover, in order to achieve confidentiality we also need the message sent by Alice right after c to be confidential.

2.6 The Almost-Optimal Scheme

Using the ideas sketched above, we can construct a scheme with almost-optimal security guarantees. We note that it is still highly non-trivial to properly combine these techniques, so that they work when the messages can be arbitrarily interleaved (so far we only considered certain idealized settings of perfectly interlocked and unidirectional communication).

The difference between our almost-optimal guarantees and the optimal ones [16, 11] is in the imperfection of our post-impersonation security. As explained in the previous subsections, for these additional guarantees we need two messages sent by the impersonated party (Alice above) to remain confidential: the one right before and the one right after the attack. Roughly, these messages are *not* confidential either if the encryption randomness is exposed for one of them, or if the state of the impersonated party is exposed right before receiving the last message before the attack. Note that the latter condition basically means that both parties are exposed at almost the same time. If they were exposed at exactly the same time, any security would anyway be impossible.

In summary, our almost-optimal security seems a very reasonable guarantee in practice.

[5] We can assume that Alice sends this value confidentially. It makes no sense to consider Bob's state being exposed, as this would mean that both parties are exposed at the same time, in which case, clearly, we cannot guarantee any security.

3 Unidirectional Authentication

In this section we formalize the first solution for achieving authentication, sketched informally in Section 2.1. That is, we consider the goal of providing authentication for the communication from a sender (which we call the signer) to a receiver (which we call the verifier) in the presence of an adversary who has full control over the communication channel. Additionally, the adversary has the ability to expose secrets of the communicating parties. In particular, this means that for each party, its internal state and, independently, the randomness it chose during operations may leak.

We first intuitively describe the properties we would like to guarantee:

– As long as the state and sampled randomness of the *signer* are secret, the communication is authenticated (in particular, all sent messages, and only those, can only be received in the correct order). We require that leaking the state or the randomness of the *verifier* has no influence on authentication.
– If the state right before signing the i-th message or the randomness used for this operation is exposed, then the adversary can trivially replace this message by one of her choice. However, we want that if she remains passive (that is, if she delivers sufficiently many messages in order), and if new secrets do not leak, then the security is eventually restored. Concretely, if only the state is exposed, then *only* the i-th message can be replaced, while if the signing randomness is exposed, then only two messages (i and $i+1$) are compromised.

Observe that once the adversary decides to inject a message (while the signer is exposed), security cannot be restored. This is because from this point on, she can send any messages to the verifier by simply executing the protocol. We will say that in such case the adversary *hijacks* the channel, and is now communicating with the verifier.

The above requirements cannot be satisfied by symmetric primitives, because compromising the receiver should have no effect on security. Moreover, in order to protect against deleting and reordering messages, the algorithms need to be stateful. Hence, in the next subsection, we define a new primitive, which we call *key-updating signatures.* At a high level, a key-updating signature scheme is a stateful signature scheme, where the signing key changes with each signing operation, and the verification key changes with each verification. We require that the verification algorithm is deterministic, so that leaking the randomness of the verifier trivially has no effect.

3.1 Key-Updating Signatures

Syntax. A key-updating signature scheme KuSig consists of three polynomial-time algorithms $(\mathsf{KuSig.Gen}, \mathsf{KuSig.Sign}, \mathsf{KuSig.Verify})$. The probabilistic algorithm $\mathsf{KuSig.Gen}$ generates an initial signing key sk and a corresponding verification key vk. Given a message m and sk, the signing algorithm outputs an

Game KuSig-UF

```
Initialization
    $(sk, vk) \leftarrow \mathsf{KuSig.Gen}$
    $s, r \leftarrow 0$
    $\mathcal{B} \leftarrow$ array initialized to $\perp$
    $\mathsf{win} \leftarrow \mathtt{false}$
    $\boxed{\mathsf{lost} \leftarrow \mathtt{false}}$
    $\boxed{\mathsf{Exposed} \leftarrow \emptyset}$
    return $vk$

Oracle Sign
Input: $(m, \mathsf{leak}) \in \mathcal{M} \times \{\mathtt{true}, \mathtt{false}\}$
    $s \leftarrow s + 1$
    $z \twoheadleftarrow \mathcal{R}$
    $(sk, \sigma) \leftarrow \mathsf{KuSig.Sign}(sk, m; z)$
    $\mathcal{B}[s] \leftarrow (m, \sigma)$
    if $\mathsf{leak}$ then
        $\boxed{\mathsf{Exposed} \leftarrow \mathsf{Exposed} \cup \{s - 1, s\}}$
        return $(\sigma, z)$
    else
        return $\sigma$

Oracle Expose
    $\boxed{\mathsf{Exposed} \leftarrow \mathsf{Exposed} \cup \{s\}}$
    return $sk$

Oracle Verify
Input: $(m, \sigma) \in \mathcal{M} \times \Sigma$
    $(vk, v) \leftarrow \mathsf{KuSig.Verify}(vk, m, \sigma)$
    if $v = 0$ then
        return $(0, vk)$
    $r \leftarrow r + 1$
    if $\mathcal{B}[r] \neq (m, \sigma)$ then
        if $r - 1 \in \mathsf{Exposed}$ then
            $\mathsf{lost} \leftarrow \mathsf{lost} \vee \neg\mathsf{win}$
        else
            $\mathsf{win} \leftarrow \mathtt{true}$
    return $(1, vk)$

Finalization
    return $\mathsf{win}$ $\boxed{\wedge \neg\mathsf{lost}}$
```

Fig. 4. The strong unforgeability game for key-updating signatures.

updated signing key and a signature: $(sk', \sigma) \leftarrow \mathsf{KuSig.Sign}(sk, m)$. Similarly, the verification algorithm outputs an updated verification key and the result v of verification: $(vk', v) \leftarrow \mathsf{KuSig.Verify}(vk, m, \sigma)$.

Correctness. Let (sk_0, vk_0) be any output of $\mathsf{KuSig.Gen}$, and let $m_1, \ldots, m_k$ be any sequence of messages. Further, let $(sk_i, \sigma_i) \leftarrow \mathsf{KuSig.Sign}(sk_{i-1}, m_i)$ and $(vk_i, v_i) \leftarrow \mathsf{KuSig.Verify}(vk_{i-1}, m_i, \sigma_i)$ for $i = 1, \ldots, k$. For correctness, we require that $v_i = 1$ for all $i = 1, \ldots, k$.

Security. The security of KuSig is formalized using the game KuSig-UF, described in Figure 4. For simplicity, we define the security in the single-user setting (security in the multi-user setting can be obtained using the standard hybrid argument).

The game interface. The game without the parts of the code marked by boxes defines the interface exposed to the adversary.

At a high level, the adversary wins if he manages to set the internal flag win to $\mathtt{true}$ by providing a message with a forged signature. To this end, he interacts with three oracles: **Sign**, **Verify** and **Expose**. Using the oracle **Sign**, he can obtain signatures and update the secret signing key, using the oracle **Verify**, he can update the verification key (or submit a forgery), and the oracle **Expose** reveals the secret signing key.

A couple of details about the above oracles require further explanation. First, the verification key does not have to be kept secret. Hence, the updated key is always returned by the verification oracle. Second, we extend the signing oracle to additionally allow "insecure" queries. That is, the adversary learns not only the signature, but also the randomness used to generate it.

Disabling trivial attacks. Since the game described above can be trivially won for any scheme, we introduce additional checks (shown in boxes), which disable the trivial "wins".

More precisely, the forgery of a message that will be verified using the key vk, for which the signing key sk was revealed is trivial. The key sk can be exposed either explicitly by calling the oracle **Expose**, or by leaking the signing randomness using the call $\textbf{Sign}(m, \texttt{true})$. To disable this attack, we keep the set $\mathsf{Exposed}$, which, intuitively, keeps track of which messages were signed using an exposed state. Then, in the oracle **Verify**, we check whether the adversary decided to input a trivial forgery (this happens if the index $r-1$ of currently verified message is in $\mathsf{Exposed}$). If so, the game can no longer be won (the variable lost is set to $\texttt{true}$).[6]

Advantage. For an adversary $\mathcal{A}$, let $\mathsf{Adv}^{\mathsf{ku\text{-}suf}}_{\mathsf{KuSig}}(\mathcal{A})$ denote the probability that the game KuSig-UF returns $\texttt{true}$ after interacting with $\mathcal{A}$. We say that a key-updating signature scheme KuSig is KuSig-UF secure if $\mathsf{Adv}^{\mathsf{ku\text{-}suf}}_{\mathsf{KuSig}}(\mathcal{A})$ is negligible for any PPT adversary $\mathcal{A}$.

3.2 Construction

We present a very simple construction of a KuSig, given any one-time signature scheme Sig, existentially-unforgable under chosen-message attack. The construction is depicted in Figure 5. The high-level idea is to generate a new key pair for Sig with each signed message. The message, together with the new verification key and a counter,[7] is then signed using the old signing key, and the new verification key is appended to the signature. The verification algorithm then replaces the old verification key by the one from the verified signature.

Theorem 1. *Let* Sig *be a signature scheme. The construction of Figure 5 is* KuSig-UF *secure, if* Sig *is* 1-SUF-CMA *secure.*

A proof of Theorem 1 is presented in the full version of this work [12].

3.3 Other Definitions of Key-Updating Signatures

Several notions of signatures with evolving keys are considered in the literature. For example, in forward-secure signatures [3] the signing key is periodically updated. However, in such schemes the verification key is fixed. Moreover, the

[6] The adversary knows which states are exposed, and hence can check himself before submitting a forgery attempt, whether this will make him lose the game.

[7] In fact, the counter is not necessary to prove security of the construction, since every message is signed with a different key. However, we find it cleaner to include it.

Construction of KuSig

```
Algorithm KuSig.Gen
    $(sk, vk) \leftarrow \mathsf{Sig.Gen}$
    return $((sk, 0), (vk, 0))$

Algorithm KuSig.Sign
Input: $((sk, s), m; (z_G, z_S)) \in \mathcal{S}_S \times \mathcal{M} \times \mathcal{R}$
    $s \leftarrow s + 1$
    $(sk', vk') \leftarrow \mathsf{Sig.Gen}(z_G)$
    $\sigma \leftarrow \mathsf{Sig.Sign}(sk, vk' \parallel s \parallel m; z_S)$
    return $((sk', s), (\sigma, vk'))$

Algorithm KuSig.Verify
Input: $((vk, r), m, (\sigma, vk')) \in \mathcal{S}_V \times \mathcal{M} \times \Sigma$
    $v \leftarrow \mathsf{Sig.Verify}(vk, vk' \parallel (r + 1) \parallel m, \sigma)$
    if $\neg v$ then
        return $((vk, r), \mathtt{false})$
    else
        return $((vk', r + 1), \mathtt{true})$
```

Fig. 5. The construction of key-updating signatures.

goal of forward secrecy is to protect the past (for signatures, this means that there exists some notion of time and exposing the secret key does not allow to forge signatures for the past time periods). On the other hand, we are interested in protecting the future, that is, the scheme should "heal" after exposure.

The notion closest to our setting is that of key-updateble digital signatures [11]. Here the difference is that their notion provides stronger guarantees (hence, the construction is also less efficient). In particular, in key-updateble digital signatures the signing key can be updated with *any (even adversarially chosen) public* information. In contrast, in our definition the secret key is updated secretly by the signer, and only part of the information used to update it is published as part of the signature.[8]

Relaxing the requirements of key-updateble digital signatures allows us to achieve a very efficient construction ([11] uses rather inefficient forward-secure signatures as a building block). On the other hand, the stronger guarantee seems to be necessary for the optimal security of [11].

4 Unidirectional Confidentiality

In this section we formalize the second solution for achieving confidentiality in the unidirectional setting, where the sender, which we now call the encryptor generates some secret update information and communicates it (encrypted) to the receiver, which we now call the decryptor. In the following, we assume that the secret update information is delivered through an idealized secure channel.

The setting is similar to the one we considered for authentication: the secret states and the randomness of the encryptor and of the decryptor can sometimes be exposed. However, now we also assume that the communication is authenticated.

[8] For example, in our construction the public part of the update is a fresh verification key, and the secret part is the corresponding signing key. This would not satisfy the requirements of [11], since there is no way to update the signing key using only the fresh verification key.

We assume authentication in the sense of Section 3, however, we do not consider hijacking the channel. In this section we give no guarantees if the channel is hijacked.

At a high level, the construction presented in this section should provide the following guarantees:

- Exposing the state of the encryptor should have no influence on confidentiality. Moreover, leaking the encryption randomness reveals only the single message being encrypted.
- Possibility of healing: if at some point in time the encryptor delivers to the decryptor an additional (update) message through some out-of-band secure channel, then any prior exposures of the decryption state should have no influence on the confidentiality of future messages. (Looking ahead, in our overall construction such updates will indeed be sometimes delivered securely.)
- Weak forward secrecy: exposing the decryptor's state should not expose messages sent before the last securely delivered update.

For more intuition about the last two properties, consider Figure 6. The states 1 to 7 correspond to the number of updates applied to encryption or decryption keys. The first two updates are not delivered securely (on the out-of-band channel), but the third one is. Exposing the decryption key at state 5 (after four updates) causes all messages encrypted under the public keys at states 4, 5 and 6 to be exposed. However, the messages encrypted under keys at states 1 to 3 are not affected.

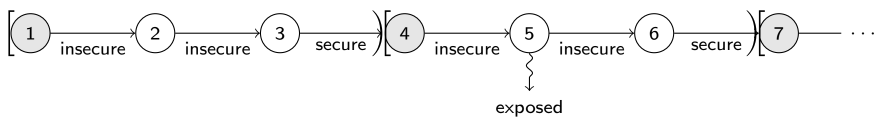


Fig. 6. Intuition behind the confidentiality guarantees.

To formalize the above requirements, we define a new primitive, which we call *secretly key-updatable public-key encryption* (SkuPke).

4.1 Secretly Key-Updatable Public-Key Encryption

At a high level, a secretly key-updatable public-key encryption scheme is a public-key encryption scheme, where both the encryption and the decryption key can be (independently) updated. The information used to update the encryption key can be public (it will be a part of the encryptor's state, whose exposure comes without consequences), while the corresponding update information for the decryption key should be kept secret (this update will be sent through the out-of-band secure channel).

In fact, for our overall scheme we need something a bit stronger: the update information should be generated independently of the encryption or decryption keys. Moreover, the properties of the scheme should be (in a certain sense) preserved even when the same update is applied to many independent key pairs. The reason for these requirements will become more clear in the next section, when we use the secretly key-updatable encryption to construct a scheme for the sesqui-directional setting.

The security definition presented in this section is slightly simplified and it does not consider the above additional guarantees. However, it is sufficient to understand our security goals. In the proof of the overall construction we use the full definition presented in the full version [12], which is mostly a straightforward extension to the multi-instance setting.

Syntax. Formally, a secretly key-updatable public-key encryption scheme SkuPke consists of six polynomial-time algorithms (SkuPke.Gen, SkuPke.Enc, SkuPke.Dec, SkuPke.UpdateGen, SkuPke.UpdateEk, SkuPke.UpdateDk). The probabilistic algorithm SkuPke.Gen generates an initial encryption key ek and a corresponding decryption key dk. Then, the probabilistic encryption algorithm can be used to encrypt a message m as $c \leftarrow \mathsf{SkuPke.Enc}(ek, m)$, while the deterministic decryption algorithm decrypts the message: $m \leftarrow \mathsf{SkuPke.Dec}(dk, c)$.

Furthermore, the probabilistic algorithm SkuPke.UpdateGen generates public update information u_e and the corresponding secret update information u_d, as $(u_e, u_d) \leftarrow \mathsf{SkuPke.UpdateGen}$. The former can then be used to update an encryption key $ek' \leftarrow \mathsf{SkuPke.UpdateEk}(u_e, ek)$, while the latter can be used to update the corresponding decryption key $dk' \leftarrow \mathsf{SkuPke.UpdateDk}(u_d, dk)$.

Correctness. Let (ek_0, dk_0) be the output of SkuPke.Gen, and let $(ue_1, ud_1), \ldots,$ (ue_k, ud_k) be any sequence of outputs of SkuPke.UpdateGen. For $i = 1 \ldots k$, let $ek_i \leftarrow \mathsf{SkuPke.UpdateEk}(ue_i, e_{i-1})$ and $dk_i \leftarrow \mathsf{SkuPke.UpdateDk}(ud_i, d_{i-1})$. A SkuPke is called correct, if $\mathsf{SkuPke.Dec}(dk_k, \mathsf{SkuPke.Enc}(ek_k, m)) = m$ for any message m with probability 1.

Security. Figure 7 presents the single-instance security game for a SkuPke scheme, which we describe in the following paragraphs.

The game interface. The interface exposed to the adversary is defined via the part of the code not marked by boxes.

We extend the standard notion of IND-CPA for public-key encryption, where the adversary gets to see the initial encryption key ek and has access to a left-or-right **Challenge** oracle. Furthermore, the adversary can generate new update information by calling the oracle **UpdateGen**, and later apply the generated updates to the encryption and decryption key, by calling, respectively, the oracles **UpdateEk** and **UpdateDk**. In our setting the adversary is allowed to expose the randomness and the state of parties. The encryption state is considered

Game SkuPke-CPA

```
Initialization
  $b \twoheadleftarrow \{0,1\}$
  $\mathcal{U}_e, \mathcal{U}_d \leftarrow$ array initialized to $\bot$
  $\mathsf{ind}, \mathsf{ind}_d, \mathsf{ind}_e \leftarrow 1$
  $\boxed{\mathsf{NLeak} \leftarrow \{1\}}$
  $\boxed{\mathsf{Chal} \leftarrow \emptyset}$
  $\mathsf{exp} \leftarrow -1$
  $ek, dk \leftarrow \mathsf{SkuPke.Gen}$
  return $ek$
```

```
Oracle UpdateGen
Input: $z \in \mathcal{R} \cup \{\bot\}$
  $\mathsf{ind} \leftarrow \mathsf{ind} + 1$
  if $z = \bot$ then
    $(\mathcal{U}_e[\mathsf{ind}], \mathcal{U}_d[\mathsf{ind}]) \leftarrow \mathsf{SkuPke.UpdateGen}$
    $\boxed{\mathsf{NLeak} \leftarrow \mathsf{NLeak} \cup \{\mathsf{ind}\}}$
    return $\mathcal{U}_e[\mathsf{ind}]$
  else
    $(\mathcal{U}_e[\mathsf{ind}], \mathcal{U}_d[\mathsf{ind}]) \leftarrow \mathsf{SkuPke.UpdateGen}(z)$
    return $(\mathcal{U}_e[\mathsf{ind}], \mathcal{U}_d[\mathsf{ind}])$
```

```
Oracle Challenge
Input: $(m_0, m_1, i) \in \mathcal{M}^2 \times (\mathbb{N} \setminus \{0,1\})$
  $\boxed{\mathsf{nc}_1 \leftarrow \mathsf{exp} \geq \mathsf{ind}_e \wedge (\mathsf{ind}_e, \mathsf{exp}] \cap \mathsf{NLeak} = \emptyset)}$
  $\boxed{\mathsf{nc}_2 \leftarrow \mathsf{exp} < \mathsf{ind}_e \wedge (\mathsf{exp}, \mathsf{ind}_e] \cap \mathsf{NLeak} = \emptyset)}$
  if $|m_0| \neq |m_1| \vee i > \mathsf{ind} \boxed{\vee\ \mathsf{nc}_1 \vee \mathsf{nc}_2}$ then
    return $\bot$
  $\boxed{\mathsf{Chal} \leftarrow \mathsf{Chal} \cup \{\mathsf{ind}_e\}}$
  $c \leftarrow \mathsf{SkuPke.Enc}(ek, m_b \parallel \mathcal{U}_d[i])$
  return $c$
```

```
Oracle UpdateEk
  if $\mathsf{ind}_e \geq \mathsf{ind}$ then
    return $\bot$
  $\mathsf{ind}_e \leftarrow \mathsf{ind}_e + 1$
  $ek \leftarrow \mathsf{SkuPke.UpdateEk}(\mathcal{U}_e[\mathsf{ind}_e], ek)$
  return $ek$
```

```
Oracle Expose
  if $\mathsf{exp} \geq 0$ then
    return $\bot$
  $\mathsf{ne}_1 \leftarrow \exists c \in \mathsf{Chal}\ (c \geq \mathsf{ind}_d] \wedge (\mathsf{ind}_d, c] \cap \mathsf{NLeak} = \emptyset)$
  $\mathsf{ne}_2 \leftarrow \exists c \in \mathsf{Chal}\ (c < \mathsf{ind}_d] \wedge (c, \mathsf{ind}_d] \cap \mathsf{NLeak} = \emptyset)$
  if $\mathsf{ne}_1 \vee \mathsf{ne}_2$ then
    return $\bot$
  $\mathsf{exp} \leftarrow \mathsf{ind}_d$
  return $dk$
```

```
Oracle UpdateDk
  if $\mathsf{ind}_d \geq \mathsf{ind}$ then
    return $\bot$
  $\mathsf{ind}_d \leftarrow \mathsf{ind}_d + 1$
  $dk \leftarrow \mathsf{SkuPke.UpdateDk}(\mathcal{U}_d[\mathsf{ind}_d], dk)$
```

```
Finalization
Input: $d \in \{0,1\}$
  return $(d = b)$
```

Fig. 7. The single-instance confidentiality game for secretly key-updatable encryption.

public information, hence, the key *ek* and the public update $\mathcal{U}_e[\mathsf{ind}]$ are always returned by the corresponding oracles. The decryption key *dk* can be revealed by calling the **Expose** oracle[9], and the secret decryption updates — by setting the randomness for the oracle **UpdateGen**. Finally, the **Challenge** oracle encrypts the message together with the previously generated secret update information, chosen by the adversary (recall the idea sketched in Section 2.2).

Disabling trivial attacks. In essence, in the presence of exposures, it is not possible to protect the confidentiality of all messages. As already explained, we allow an exposure of the secret key to compromise secrecy of all messages sent between two consecutive secure updates. Hence, the game keeps track of the following events: generating a secure update (the set NLeak), exposing the secret key (the variable exp), and asking for a challenge ciphertext (the set Chal). Then, the adversary is not allowed to ask for a challenge generated using the encryption

[9] For technical reasons, we only allow one query to the **Expose** oracle.

Construction of SkuPke

```
Algorithm SkuPke.Gen
    $x \twoheadleftarrow \mathbb{Z}_q$
    return $(g^x, x)$

Algorithm SkuPke.UpdateGen
    $x \twoheadleftarrow \mathbb{Z}_q$
    return $(g^x, x)$

Algorithm SkuPke.UpdateEk
Input: $(u_e, ek)$
    return $ek \cdot u_e$

Algorithm SkuPke.UpdateDk
Input: $(u_d, dk)$
    return $(dk + u_d) \mod q$

Algorithm SkuPke.Enc
Input: $(ek, m)$
    $r \twoheadleftarrow \mathbb{Z}_q$
    return $(g^r, \mathrm{Hash}_{|m|}(ek^r) \oplus m)$

Algorithm SkuPke.Dec
Input: $(dk, (c_1, c_2))$
    return $\mathrm{Hash}_{|c_2|}(c_1^{dk}) \oplus c_2$
```

Fig. 8. The construction of secretly key-updatable encryption.

key, corresponding to a decryption key, which is in the "exposed" interval (that is, if all updates between the decryption key and the exposed state are insecure). An analogous condition is checked by the **Expose oracle**.

Advantage. Recall that in this section we present the single-instance security game, but in the proofs later we need the multi-instance version SkuPke-MI-CPA defined in the full version [12]. Hence, we define security using the multi-instance game. For an adversary $\mathcal{A}$, let $\mathsf{Adv}^{\mathsf{sku\text{-}cpa}}_{\mathsf{SkuPke}}(\mathcal{A}) := 2\Pr[\mathcal{A}^{\text{SkuPke-MI-CPA}} \Rightarrow \mathtt{true}] - 1$, where $\Pr[\mathcal{A}^{\text{SkuPke-MI-CPA}} \Rightarrow \mathtt{true}]$ denotes the probability that the game SkuPke-MI-CPA returns `true` after interacting with an adversary $\mathcal{A}$. We say that a secretly key-updatable encryption scheme scheme SkuPke is SkuPke-MI-CPA secure if $\mathsf{Adv}^{\mathsf{sku\text{-}cpa}}_{\mathsf{SkuPke}}(\mathcal{A})$ is negligible for any PPT adversary $\mathcal{A}$.

4.2 Construction

We present an efficient construction of SkuPke, based on the ElGamal cryptosystem. At a high level, the key generation, encryption and decryption algorithms are the same as in the ElGamal encryption scheme. To generate the update information, we generate a new ElGamal key pair, and set the public and private update to, respectively, the new public and private ElGamal keys. To update the encryption key, we multiply the two ElGamal public keys, while to update the decryption key, we add the ElGamal secret keys. Finally, in order to deal with encrypting previously generated update information, we need the hash function $\mathrm{Hash}_l(\cdot)$, where l is the output length.

The construction is defined in Figure 8. We let G be a group of prime order q, generated by g. These parameters are implicitly passed to all algorithms.

A proof of the following theorem is presented in the full version [12].

Theorem 2. *The construction of Figure 8 is* SkuPke-MI-CPA *secure in the random oracle model, if CDH is hard.*

5 Sesquidirectional Confidentiality

The goal of this section is to define additional confidentiality guarantees in the setting where also an authenticated back channel from the decryptor to the encryptor exists (but we still focus only the properties of the unidirectional from the encryptor to the decryptor). That is, we assume a perfectly-authenticated back channel and a forward channel, authenticated in the sense of Section 3 (in particular, we allow hijacking the decryptor).

It turns out that in this setting we can formalize all confidentiality properties needed for our overall construction of a secure channel. Intuitively, the properties we consider include forward secrecy, post-hijack security, and healing through the back channel.

Forward secrecy. Exposing the decryptor's state should not expose messages which he already received.

Post-hijack guarantees. Ideally, we would like to guarantee that if the communication to the decryptor is hijacked, then all messages sent by the encryptor *after* hijacking are secret, even if the decryptor's state is exposed (note that these messages cannot be read by the decryptor, since the adversary caused his state to be "out-of-sync"). However, this guarantee turns out to be extremely strong, and seems to inherently require HIBE. Hence, we relax it by giving up on the secrecy of post-hijack messages in the following case: a message is sent insecurely (for example, because the encryption randomness is exposed), the adversary *immediately* hijacks the communication, and at some later time the decryptor's state is exposed. We stress that the situation seems rather contrived, as explained in the introduction.

Healing through the back channel. Intuitively, the decryptor will update his state and send the corresponding update information on the back channel. Once the encryptor uses this information to update his state, the parties heal from past exposures. At a high level, this means that we require the following additional guarantees:

- Healing: messages sent after the update information is delivered are secret, irrespective of any exposures of the decryptor's state, which happened before the update was generated.
- Correctness: in the situation where the messages on the back channel are delayed, it should still be possible to read the messages from the forward channel. That is, it should be possible to use a decryption key after i updates to decrypt messages encrypted using an "old" encryption key after $j < i$ updates.

Challenges. It turns out that the setting with both the back channel, and the possibility of hijacking, is extremely subtle. For example, one may be tempted to use an encryption scheme which itself updates keys and provides some form of forward secrecy, and then simply send on the back channel a fresh key pair for that scheme. With this solution, in order to provide correctness, every generated secret key would have to be stored until a ciphertext for a newer key arrives. Unfortunately, this simple solution does not work. Consider the following situation: the encryptor sends two messages, one before and one after receiving an update on the back channel, and these messages are delayed. Then, the adversary hijacks the decryptor by injecting an encryption under the *older* of the two keys. However, if now the decryptor's state is exposed, then the adversary will learn the message encrypted with the *new* key (which breaks the post-hijack guarantees we wish to provide). Hence, it is necessary that receiving a message updates all decryption keys, also those for future messages. Intuitively, this is why we require that the same update for SkuPke can be applied to many keys.

5.1 Healable And Key-Updating Public-Key Encryption

To formalize the requirements sketched above, we define healable and key-updating public-key encryption (HkuPke). In a nutshell, a HkuPke scheme is a *stateful* public-key encryption scheme with additional algorithms used to generate and apply updates, sent on the back channel.

Syntax. A healable and key-updating public-key encryption scheme HkuPke consists of five polynomial-time algorithms (HkuPke.Gen, HkuPke.Enc, HkuPke.Dec, HkuPke.BcUpEk, HkuPke.BcUpDk).

The probabilistic algorithm HkuPke.Gen generates an initial encryption key ek and a corresponding decryption key dk. Encryption and decryption algorithms are stateful. Moreover, for reasons which will become clear in the overall construction of a secure channel, they take as input additional data, which need not be kept secret.[10] Formally, we have $(ek', c) \leftarrow \mathsf{HkuPke.Enc}(ek, m, ad)$ and $(dk', m) \leftarrow \mathsf{HkuPke.Dec}(dk, c, m)$, where ek' and dk' are the updated keys and ad is the additional data. The additional two algorithms are used to handle healing through the back channel: the operation $(dk', upd) \leftarrow \mathsf{HkuPke.BcUpDk}(dk)$ outputs the updated decryption key dk' and the information upd, which will be sent on the back channel. Then, the encryption key can be updated by executing $ek' \leftarrow \mathsf{HkuPke.BcUpEk}(ek, upd)$.

Correctness. Intuitively, we require that if all ciphertexts are decrypted in the order of encryption, and if the additional data used for decryption matches that used for encryption, then they decrypt to the correct messages. Moreover, decryption must also work if the keys are updated in the meantime, that is, if an arbitrary sequence of HkuPke.BcUpDk calls is performed and the ciphertext is generated at a point where only a prefix of the resulting update information has

[10] Roughly, the additional data is needed to provide post-hijack security of the final construction: changing the additional data means that the adversary decided to hijack the channel, hence, the decryption key should be updated.

Game HkuPke-CPA

```
Initialization
    $b \twoheadleftarrow \{0,1\}$
    $(ek, dk) \leftarrow \mathsf{HkuPke.Gen}$
    $s, r, i, j \leftarrow 0$
    $\boxed{\mathsf{exposed} \leftarrow -1}$
    $\boxed{\mathsf{hijacked} \leftarrow \mathtt{false}}$
    $\boxed{\mathsf{Challenges} \leftarrow \emptyset}$
    $\mathcal{B}, \mathcal{U} \leftarrow$ array initialized to $\perp$
    return $ek$

Oracle Encrypt
Input: $(m, ad) \in \mathcal{M} \times \mathcal{AD}$
    $s \leftarrow s + 1$
    $z \twoheadleftarrow \mathcal{R}$
    $(ek, c) \leftarrow \mathsf{HkuPke.Enc}(ek, m, ad; z)$
    $\mathcal{B}[s] \leftarrow (c, ad)$
    return $(ek, c, z)$

Oracle BcUpdateEk
    if $j = i$ then
        return $\perp$
    $j \leftarrow j + 1$
    $ek \leftarrow \mathsf{HkuPke.BcUpEk}(ek, \mathcal{U}[j])$

Oracle Challenge
Input: $(m_0, m_1, ad) \in \mathcal{M}^2 \times \mathcal{AD}$
    $\boxed{\textbf{if } |m_0| \neq |m_1| \textbf{ then}}$
    $\boxed{\quad \textbf{return } \perp}$
    $\boxed{\textbf{if } j \leq \mathsf{exposed} \textbf{ then}}$
    $\boxed{\quad \textbf{return } \perp}$
    $(ek, c, z) \leftarrow \textbf{Encrypt}(m_b, ad)$
    $\boxed{\mathsf{Challenges} \leftarrow \mathsf{Challenges} \cup \{s\}}$
    return $(ek, c)$

Oracle Decrypt
Input: $(c, ad) \in \mathcal{C} \times \mathcal{AD}$
    $(dk, m) \leftarrow \mathsf{HkuPke.Dec}(dk, c, ad)$
    if $m = \perp$ then
        return $\perp$
    $\boxed{\textbf{if } \mathsf{hijacked} \vee (c, ad) \neq \mathcal{B}[r+1] \textbf{ then}}$
    $\boxed{\quad \mathsf{hijacked} \leftarrow \mathtt{true}}$
    $\boxed{\quad \textbf{return } m}$
    $\boxed{\textbf{else}}$
        $r \leftarrow r + 1$
        return $\perp$

Oracle BcUpdateDk
Input: $\mathsf{leak} \in \{\mathtt{true}, \mathtt{false}\}$
    $\boxed{\textbf{if } \neg\mathsf{hijacked} \textbf{ then}}$
        $i \leftarrow i + 1$
        $z \twoheadleftarrow \mathcal{R}$
        $(dk, \mathcal{U}[i]) \leftarrow \mathsf{HkuPke.BcUpDk}(dk; z)$
        if $\mathsf{leak}$ then
            $\mathsf{exposed} \leftarrow i$
            return $(\mathcal{U}[i], z)$
        else
            return $\mathcal{U}[i]$

Oracle Expose
    $\boxed{\mathsf{vuln}_1 \leftarrow r \notin \mathsf{Challenges}}$
    $\boxed{\mathsf{vuln}_2 \leftarrow r + 1 \leq s \wedge r + 1 \notin \mathsf{Challenges}}$
    $\boxed{\textbf{if } \mathsf{hijacked} \wedge \neg\mathsf{vuln}_1 \wedge \neg\mathsf{vuln}_2 \textbf{ then}}$
        return $dk$
    $\boxed{\textbf{else if } \forall e \in (r, s] \; e \notin \mathsf{Challenges} \textbf{ then}}$
    $\boxed{\quad \mathsf{exposed} \leftarrow i}$
    $\boxed{\quad \textbf{return } dk}$
    $\boxed{\textbf{else}}$
    $\boxed{\quad \textbf{return } \perp}$

Finalization
Input: $d \in \{0,1\}$
    return $(d = b)$
```

Fig. 9. The confidentiality game for healable and key-updating encryption.

been applied to the encryption key using $\mathsf{HkuPke.BcUpEk}$. A formal definition of correctness is given in the full version [12].

Security. The security of HkuPke is formalized using the game HkuPke-CPA, described in Figure 9. Similarly to the unidirectional case, we extend the IND-CPA game.

The interface. Consider the (insecure) variant of our game without the parts of the code marked in boxes. As in the IND-CPA game, the adversary gets to see the encryption key ek and has access to a left-or-right **Challenge** oracle. Since HkuPke schemes are stateful, we additionally allow the adversary to update the decryption key through the calls to the **Decrypt** oracle (which for now only returns $\perp$). The encryption key is updated using the calls to the **Encrypt** oracle (where the encrypted message is known) and to the **Challenge** oracle.

Furthermore, in our setting the adversary is allowed to expose the randomness and the state. To expose the state (that is, the decryption key), he can query the **Expose** oracle. To expose the randomness of any randomized oracle, he can set the input flag leak to $\mathtt{true}$.

Finally, the adversary can access two oracles corresponding to the back channel: the oracle **BcUpdateDk** executes the algorithm $\mathsf{HkuPke.BcUpDk}$ and returns the update information to the adversary (this corresponds to sending on the back channel), and the oracle **BcUpdateEk executes** $\mathsf{HkuPke.BcUpEk}$ with the next generated update (since the channel is authenticated, the adversary has no influence on which update is applied).

Disabling trivial attacks. Observe that certain attacks are disabled by the construction itself. For example, the randomness used to encrypt a challenge ciphertext cannot be exposed.

Furthermore, the game can be trivially won if the adversary asks for a challenge ciphertext and, before calling Decrypt with this ciphertext, exposes the decryption key (by correctness, the exposed key can be used to decrypt the challenge). We disallow this by keeping track of when the adversary queried a challenge in the set $\mathsf{Challenges}$, and adding corresponding checks in the **Expose** oracle. Similarly, in the **Challenge** oracle we return $\perp$ whenever the decryption key corresponding to the current encryption key is known to the adversary. Finally, the decryptor can be hijacked, which the game marks by setting $\mathsf{hijacked}$ to $\mathtt{true}$. Once this happens, the **Decrypt** oracle "opens up" and returns the decrypted message.

Moreover, ideally, exposing the secret key after hijacking would not reveal anything about the messages (the adversary gets to call Expose "for free", without setting $\mathsf{exposed}$). However, as already mentioned, we relax slightly the security. In particular, exposing is free only when hijacking did not occur immediately after leaking encryption randomness. This is checked using the conditions vuln_1 and vuln_2.

Advantage. In the following, let $\mathsf{Adv}^{\mathsf{hku\text{-}cpa}}_{\mathsf{HkuPke}}(\mathcal{A}) := 2\Pr[\mathcal{A}^{\text{HkuPke-CPA}} \Rightarrow \mathtt{true}] - 1$, where $\Pr[\mathcal{A}^{\text{HkuPke-CPA}} \Rightarrow \mathtt{true}]$ denotes the probability that the game HkuPke-CPA returns $\mathtt{true}$ after interacting with an adversary $\mathcal{A}$. We say that a healable and key-updating encryption scheme scheme HkuPke is HkuPke-CPA secure if $\mathsf{Adv}^{\mathsf{hku\text{-}cpa}}_{\mathsf{HkuPke}}(\mathcal{A})$ is negligible for any PPT adversary $\mathcal{A}$.

5.2 Construction

To construct a HkuPke scheme, we require two primitives: a secretly key-updatable encryption scheme SkuPke from Section 4, and an IND-CCA2 secure public-key encryption scheme *with associated data* PkeAd. Intuitively, the latter primitive is a public-key encryption scheme, which additionally takes into account non-secret associated data, such that the decryption succeeds if and only if the associated

Construction of HkuPke

```
Algorithm HkuPke.Gen
  $DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, U_e \leftarrow$ array initialized to $\perp$
  $(ek^{\mathsf{upd}}, DK^{\mathsf{upd}}[0]) \leftarrow \mathsf{SkuPke.Gen}$
  $(ek^{\mathsf{eph}}, DK^{\mathsf{eph}}[0]) \leftarrow \mathsf{PkeAd.Gen}$
  $s, r, i, j, tr_s, tr_r \leftarrow 0$
  $i_{\mathsf{ack}} \leftarrow -1$
  return $((ek^{\mathsf{upd}}, ek^{\mathsf{eph}}, s, j, U_e, tr_s),$
      $(DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, r, i, i_{\mathsf{ack}}, tr_r))$
```

```
Algorithm HkuPke.Enc
Input: $((ek^{\mathsf{upd}}, ek^{\mathsf{eph}}, s, j, U_e, tr_s), m, ad;$
        $(z_1, \ldots, z_4)) \in \mathcal{EK} \times \mathcal{M} \times \mathcal{AD} \times \mathcal{R}$
  $s \leftarrow s + 1$
  $(U_e[s], u_d) \leftarrow \mathsf{SkuPke.UpdateGen}(z_1)$
  $\hat{c} \leftarrow \mathsf{SkuPke.Enc}(ek^{\mathsf{upd}}, (m, u_d, z_2); z_3)$
  $c \leftarrow \mathsf{PkeAd.Enc}(ek^{\mathsf{eph}}, \hat{c}, ad; z_4)$
  $tr_s \leftarrow \mathrm{Hash}(tr_s \parallel (c, j, ad))$
  $ek^{\mathsf{upd}} \leftarrow \mathsf{SkuPke.UpdateEk}(U_e[s], ek^{\mathsf{upd}})$
  $(ek^{\mathsf{eph}}, \_) \leftarrow \mathsf{PkeAd.Gen}(\mathrm{Hash}(tr_s \parallel z_2))$
  return $((ek^{\mathsf{upd}}, ek^{\mathsf{eph}}, s, j, U_e, tr_s), (c, j))$
```

```
Algorithm HkuPke.BcUpDk
Input: $((DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, r, i, i_{\mathsf{ack}}, tr_r);$
        $(z_1, z_2)) \in \mathcal{DK} \times \mathcal{R}$
  $i \leftarrow i + 1$
  $(\hat{ek}^{\mathsf{upd}}, \hat{dk}^{\mathsf{upd}}) \leftarrow \mathsf{SkuPke.Gen}(z_1)$
  $(\hat{ek}^{\mathsf{eph}}, \hat{dk}^{\mathsf{eph}}) \leftarrow \mathsf{PkeAd.Gen}(z_2)$
  $DK^{\mathsf{upd}}[i] \leftarrow \hat{dk}^{\mathsf{upd}}$
  $DK^{\mathsf{eph}}[i] \leftarrow \hat{dk}^{\mathsf{eph}}$
  return $((DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, r, i, i_{\mathsf{ack}}, tr_r),$
      $(\hat{ek}^{\mathsf{upd}}, \hat{ek}^{\mathsf{eph}}, r))$
```

```
Algorithm HkuPke.BcUpEk
Input: $((ek^{\mathsf{upd}}, ek^{\mathsf{eph}}, s, j, U_e, tr_s),$
        $(\hat{ek}^{\mathsf{upd}}, \hat{dk}^{\mathsf{eph}}, r^{\mathsf{msg}})) \in \mathcal{EK} \times \mathcal{U}$
  if $r^{\mathsf{msg}} \geq s$ then
    $ek^{\mathsf{eph}} \leftarrow \hat{ek}^{\mathsf{eph}}$
  $ek^{\mathsf{upd}} \leftarrow \hat{ek}^{\mathsf{upd}}$
  for $\ell \leftarrow (r^{\mathsf{msg}} + 1), \ldots, s$ do
    $ek^{\mathsf{upd}} \leftarrow \mathsf{SkuPke.UpdateEk}(U_e[\ell], ek^{\mathsf{upd}})$
  return $(ek^{\mathsf{upd}}, ek^{\mathsf{eph}}, s, j + 1, U_e, tr_s)$
```

```
Algorithm HkuPke.Dec
Input: $((DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, r, i, i_{\mathsf{ack}}, tr_r), (c, i_{\mathsf{msg}}), ad) \in \mathcal{DK} \times \mathcal{C} \times \mathcal{AD}$
  if $i_{\mathsf{msg}} \geq i_{\mathsf{ack}} \wedge i_{\mathsf{msg}} > i$ then
    $\hat{c} \leftarrow \mathsf{PkeAd.Dec}(DK^{\mathsf{eph}}[i_{\mathsf{msg}}], c, ad)$
    if $\hat{c} \neq \perp$ then
      $\hat{m} \leftarrow \mathsf{SkuPke.Dec}(DK^{\mathsf{upd}}[i_{\mathsf{msg}}], \hat{c})$
      if $\hat{m} \in \mathcal{M} \times \mathsf{SkuPke}.\mathcal{U} \times \mathsf{PkeAd}.\mathcal{DK}$ then
        $(m, u_d, z) \leftarrow \hat{m}$
        $tr_r \leftarrow \mathrm{Hash}(tr_r \parallel (c, i_{\mathsf{msg}}, ad))$
        $(\_, \hat{dk}^{\mathsf{eph}}) \leftarrow \mathsf{PkeAd.Gen}(\mathrm{Hash}(tr_r \parallel z_2))$
        for $\ell \leftarrow 1 \ldots i$ do
          if $\ell < i_{\mathsf{msg}}$ then
            $DK^{\mathsf{eph}}[\ell] \leftarrow \perp$
            $DK^{\mathsf{upd}}[\ell] \leftarrow \perp$
          else
            $DK^{\mathsf{eph}}[\ell] \leftarrow \hat{dk}^{\mathsf{eph}}$
            $DK^{\mathsf{upd}}[\ell] \leftarrow \mathsf{SkuPke.UpdateDk}(u_d, DK^{\mathsf{upd}}[\ell])$
        return $((DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, r + 1, i, i_{\mathsf{msg}}), m)$
  return $((DK^{\mathsf{upd}}, DK^{\mathsf{eph}}, r, i, i_{\mathsf{ack}}, tr_r), \perp)$
```

Fig. 10. The construction of healable and key-updating encryption.

data has not been modified. A bit more formally, in the corresponding security game the decryption oracle is only blinded if the adversary requests to decrypt the challenge ciphertext together with the associated data provided with the challenge. It will decrypt the challenge for any other associated data. A formal description of this notion, together with a simple construction in the random oracle model, is presented in the full version [12].

At the core of our construction, in order to encrypt a message m, we generate an update u_e, d_d for an SkuPke scheme and encrypt the secret update information

u_d together with m. This update information is then used during decryption to update the secret key.

Unfortunately, this simple solution has a few problems. First, we need the guarantee that after the decryptor is hijacked, his state cannot be used to decrypt messages encrypted afterwards. We achieve this by adding a second layer of encryption, using a PkeAd. We generate a new key pair during every encryption, and send the new decryption key along with m and u_d, and store the corresponding encryption key for the next encryption operation. The decryptor will use his current such key to decrypt the message and then completely overwrite it with the new one he just received. Therefore, we call those keys "ephemeral". The basic idea is of course that during the hijacking, the adversary has to provide a different ciphertext containing a new ephemeral key, which will then be useless for him when exposing the receiver afterwards. In order to make this idea sound, we have to ensure that this key is not only different from the previous one, but unrelated. To achieve this, we actually do not send the new encryption key directly, but send a random value z instead and then generate the key pairs using $\mathrm{Hash}(tr \parallel z)$ as randomness. Here tr stands for a hash chain of ciphertexts and associated data sent/received so far, including the current one. Overall, an encryption of m is $\mathsf{PkeAd.Enc}(ek^{\mathsf{eph}}, \mathsf{SkuPke.Enc}(ek^{\mathsf{upd}}, (m, u_d, z_2)), ad)$, for some associated data ad.

Second, we need to provide healing guarantees through the back channel. This is achieved by generating fresh key pairs for both, the updating and the ephemeral, encryption schemes. For correctness, the encryptor however might have to ignore the new ephemeral key, if he detects that it will be overwritten by one of his updates in the meantime. He can detect this by the decryptor explicitly acknowleding the number of messages he received so far as part of the information transmitted on the backward-channel.

Third, observe that for correctness, the decryptor needs to store all decryption keys generated during the back-channel healing, until he receives a ciphertext for a newer key (consider the back-channel messages being delayed). In order to still guarantee post-hijack security, we apply the SkuPke update u_d to *all* secret keys he still stores. This also implies that the encryptor has to store the corresponding public update information and apply them the the new key he obtains from the backward-channel, if necessary.

Theorem 3. *Let* SkuPke *be a secretly key-updatable encryption scheme, and let* PkeAd *be an encryption scheme with associated data. The scheme of Figure 10 is* HkuPke-CPA *secure in the random oracle model, if the* SkuPke *scheme is* SkuPke-MI-CPA *secure, and the* PkeAd *is* IND-CCA2-AD *secure.*

A proof of Theorem 3 is presented in the full version of this work [12].

6 Overall Security

So far, we have constructed two intermediate primitives that will help us build a secure messaging protocol. First, we showed a unidrectional authentication scheme that provides healing after exposure of the signer's state. Second, we introduced a sesqui-directional confidentiality scheme that achieves forward secrecy, healing after the exposure of the receiver's state, and it also provides post-hijack confidentiality.

The missing piece, except showing that the schemes can be securely plugged together, is post-hijack authentication: with the unidirectional authentication scheme we introduced, exposing a hijacked party's secret state allows an attacker to forge signatures that are still accepted by the other party. This is not only undesirable in practice (the parties lose the chance of detecting the hijack), but it actually undermines post-hijack confidentiality as well. More specifically, an attacker might trick the so far uncompromised party into switching over to adversarially chosen "newer" encryption key, hence becoming a man-in-the-middle after the fact.

In contrast to confidentiality, one obtains healing of authentication in the unidirectional setting, but post-hijack security requires some form of bidirectional communication: receiving a message must irreversibly destroy the signing key. Generally, we could now follow the approach we took when dealing with the confidentiality and define a sesqui-directional authentication game. We refrain from doing so, as we believe that this does not simplify the exposition. As the reader will see later, our solution for achieving post-hijack authentication guarantees requires that the update information on the backward-channel is transmitted confidentially. This breaks the separation between authentication and confidentiality. More concretely, in order for a sesqui-directional authentication game to serve as a useful intermediate abstraction on which one could then build upon, it would now have to model the partial confidential channel of HkuPke in sufficient details. Therefore, we avoid such an intermediate step, and build our overall secure messaging scheme directly. First, however, we formalize the precise level of security we actually want to achieve.

6.1 Almost-Optimal Security of Secure Messaging

Syntax. A *secure messaging scheme* SecMsg consists of the following triple of polynomial-time algorithms $(\mathsf{SecMsg.Init}, \mathsf{SecMsg.Send}, \mathsf{SecMsg.Receive})$. The probabilistic algorithm $\mathsf{SecMsg.Init}$ generates an initial pair of states st_{A} and st_{B} for Alice and Bob, respectively. Given a message m and a state st_{u} of a party, the probabilistic sending algorithm outputs an updated state and a ciphertext c: $(st_{\mathsf{u}}, c) \leftarrow \mathsf{SecMsg.Send}(st_{\mathsf{u}}, m; z)$. Analogously, given a state and a ciphertext, the receiving algorithms outputs an updated state and a message m: $(st_{\mathsf{u}}, m) \leftarrow \mathsf{SecMsg.Send}(st_{\mathsf{u}}, c)$.

Correctness. Correctness of a secure messaging scheme SecMsg requires that if all sent ciphertext are received in order (per direction), then they decrypt to the correct message. More formally, we say the scheme is correct if no adversary can win the correctness game SecMsg-Corr, depicted in Figure 11, with non-negligible probability. For simplicity, we usually consider perfect correctness, i.e., even an unbounded adversary must have probability zero in winning the game.

Game SecMsg-Corr

```
Initialization
  $(st_{\mathsf{A}}, st_{\mathsf{B}}) \leftarrow \mathsf{SecMsg.Init}$
  $\mathsf{win} \leftarrow \mathtt{false}$
  for $\mathsf{u} \in \{\mathsf{A}, \mathsf{B}\}$ do
      $\mathcal{B}_{\mathsf{u}\to\bar{\mathsf{u}}}, \mathcal{M}_{\mathsf{u}\to\bar{\mathsf{u}}} \leftarrow$ array initialized to $\bot$
      $s_{\mathsf{u}}, r_{\mathsf{u}} \leftarrow 0$

Oracle Send
Input: $(\mathsf{u}, m) \in \{\mathtt{A}, \mathtt{B}\} \times \mathcal{M}$
  $s_{\mathsf{u}} \leftarrow s_{\mathsf{u}} + 1$
  $z \twoheadleftarrow \mathcal{R}$
  $(st_{\mathsf{u}}, c) \leftarrow \mathsf{SecMsg.Send}(st_{\mathsf{u}}, m; z)$
  $\mathcal{B}_{\mathsf{u}\to\bar{\mathsf{u}}}[s_{\mathsf{u}}] \leftarrow c$
  $\mathcal{M}_{\mathsf{u}\to\bar{\mathsf{u}}}[s_{\mathsf{u}}] \leftarrow m$
  return $(c, z)$

Oracle Receive
Input: $\mathsf{u} \in \{\mathtt{A}, \mathtt{B}\}$
  if $r_{\mathsf{u}} \geq s_{\bar{\mathsf{u}}}$ then
      return $\bot$
  $r_{\mathsf{u}} \leftarrow r_{\mathsf{u}} + 1$
  $(st_{\mathsf{u}}, m) \leftarrow \mathsf{SecMsg.Receive}(st_{\mathsf{u}}, \mathcal{B}_{\bar{\mathsf{u}}\to\mathsf{u}}[r_{\mathsf{u}}])$
  if $m \neq \mathcal{M}_{\bar{\mathsf{u}}\to\mathsf{u}}[r_{\mathsf{u}}]$ then
      $\mathsf{win} \leftarrow \mathtt{true}$
  return $m$

Finalization
  return $\mathsf{win}$
```

Fig. 11. The correctness game for a secure messaging scheme.

Security. The security of SecMsg is formalized using the game SecMsg-Sec, described in Figure 12.

In general, the game composes the aspects of the security game for key-updating signature scheme KuSig-UF, depicted in Figure 4 on Page 11, with the sesqui-directional confidentiality game HkuPke-CPA, depicted in Figure 9 on Page 20. Nevertheless, there are a few noteworthy points:

– The game can be won in two ways: either by guessing the bit b, i.e., breaking confidentiality, or by setting the flag win to `true`, i.e., being able to inject messages when not permitted by an appropriate state exposure. Note that in contrast to the unidirectional authentication game, the game still has to continue after a permitted injection, hence no lost flag exists, as we want to guarantee post-hijack security.
– In contrast to the sesqui-directional confidentiality game, the **Send** oracle takes an additional flag as input modeling whether the randomness used during this operations leaks or not. This allows us to capture that a message might not remain confidential because the receivers decryption key has been exposed, yet it contributes to the healing of the reverse direction (which is not the case if the freshly sampled secret key already leaks again).
– Observe that r_{u} stops increasing the moment the user u is hijacked. Hence, whenever $\mathsf{hijacked}_{\mathsf{u}}$ is `true`, r_{u} corresponds to the number of messages he received before.
– The two flags vuln_1 and vuln_2 correspond to the two situations in which we cannot guarantee proper post-hijack security. First, vuln_1 corresponds to the situation that the last message from $\bar{\mathsf{u}}$ to u before u got hijacked was not transmitted confidentiality. This can have two reasons: either the randomness of the encryption of $\bar{\mathsf{u}}$ leaked, or u has been exposed just before receiving that message. Observe that in order to hijack u right after that message, the state

of $\bar{\mathsf{u}}$ needs to be exposed right after sending that message. So in a model where randomness does not leak, vuln_1 implies that both parties' state have been compromised almost at the same time. Secondly, vuln_2 implies that the next message by $\bar{\mathsf{u}}$ was not sent securely either.

Game SecMsg-Sec

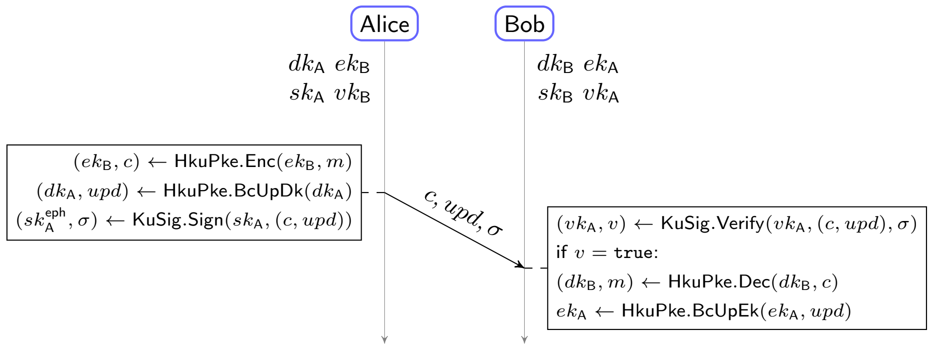
```
Initialization
  $(st_A, st_B) \leftarrow$ SecMsg.Init
  $b \twoheadleftarrow \{0,1\}$
  $\mathsf{win} \leftarrow \mathtt{false}$
  for $\mathsf{u} \in \{\mathsf{A},\mathsf{B}\}$ do
      $\mathcal{B}_{\mathsf{u}\to\bar{\mathsf{u}}} \leftarrow$ array initialized to $\bot$
      $s_\mathsf{u}, r_\mathsf{u} \leftarrow 0$
      $\mathsf{hijacked}_\mathsf{u} \leftarrow \mathtt{false}$
      $\mathsf{Exposed}_\mathsf{u} \leftarrow \{-1\}$
      $\mathsf{Challenges}_{\mathsf{u}\to\bar{\mathsf{u}}} \leftarrow \emptyset$

Oracle Send
Input: $(\mathsf{u}, m, \mathsf{leak})$
            $\in \{\mathtt{A},\mathtt{B}\} \times \mathcal{M} \times \{\mathtt{true},\mathtt{false}\}$
  $s_\mathsf{u} \leftarrow s_\mathsf{u} + 1$
  $z \twoheadleftarrow \mathcal{R}$
  $(st_\mathsf{u}, c) \leftarrow$ SecMsg.Send$(st_\mathsf{u}, m; z)$
  if $\neg\mathsf{hijacked}_\mathsf{u}$ then
      if leak then
          $\mathsf{Exposed}_\mathsf{u} \leftarrow \mathsf{Exposed}_\mathsf{u} \cup \{s_\mathsf{u}\}$
      $\mathcal{B}_{\mathsf{u}\to\bar{\mathsf{u}}}[s_\mathsf{u}] \leftarrow c$
  if leak then
      return $(c, z)$
  else
      return $c$

Oracle Challenge
Input: $(\mathsf{u}, m_0, m_1) \in \{\mathtt{A},\mathtt{B}\} \times \mathcal{M}^2$
  if $|m_0| \neq |m_1| \vee \mathsf{hijacked}_\mathsf{u}$
        $\vee r_\mathsf{u} \leq \max(\mathsf{Exposed}_{\bar{\mathsf{u}}})$ then
      return $\bot$
  $c \leftarrow$ Send$(\mathsf{u}, m_b, \mathtt{false})$
  $\mathsf{Challenges}_{\mathsf{u}\to\bar{\mathsf{u}}} \leftarrow \mathsf{Challenges}_{\mathsf{u}\to\bar{\mathsf{u}}} \cup \{s_\mathsf{u}\}$
  return $c$
```

```
Oracle Receive
Input: $(\mathsf{u}, c) \in \{\mathtt{A},\mathtt{B}\} \times \mathcal{C}$
  $(st_\mathsf{u}, m) \leftarrow$ SecMsg.Receive$(st_\mathsf{u}, c)$
  if $m = \bot$ then
      return $\bot$
  if $\mathsf{hijacked}_\mathsf{u}$ then
      return $m$
  else if $c \neq \mathcal{B}_{\bar{\mathsf{u}}\to\mathsf{u}}[r_\mathsf{u} + 1]$ then
      if $r_\mathsf{u} \in \mathsf{Exposed}_{\bar{\mathsf{u}}}$ then
          $\mathsf{hijacked}_\mathsf{u} \leftarrow \mathtt{true}$
      else
          $\mathsf{win} \leftarrow \mathtt{true}$
      return $m$
  else
      $r_\mathsf{u} \leftarrow r_\mathsf{u} + 1$
      return $\bot$

Oracle Expose
Input: $\mathsf{u} \in \{\mathtt{A},\mathtt{B}\}$
  $\boxed{\begin{array}{l} \mathsf{vuln}_1 \leftarrow r_\mathsf{u} \notin \mathsf{Challenges}_{\bar{\mathsf{u}}\to\mathsf{u}} \\ \mathsf{vuln}_2 \leftarrow (r_\mathsf{u}{+}1 \leq s_\mathsf{u}) \wedge r_\mathsf{u}{+}1 \notin \mathsf{Challenges}_{\bar{\mathsf{u}}\to\mathsf{u}} \end{array}}$
  if $\mathsf{hijacked}_\mathsf{u}$ $\boxed{\wedge\neg\mathsf{vuln}_1 \wedge \neg\mathsf{vuln}_2}$ then
      return $st_\mathsf{u}$
  else if $\forall i \in (r_\mathsf{u}, s_{\bar{\mathsf{u}}}]\ \ i \notin \mathsf{Challenges}_{\bar{\mathsf{u}}\to\mathsf{u}}$ then
      if $\mathsf{hijacked}_\mathsf{u}$ then
          $\mathsf{Exposed}_\mathsf{u} \leftarrow \mathsf{Exposed}_\mathsf{u} \cup \{s_u, \ldots, \infty\}$
      else
          $\mathsf{Exposed}_\mathsf{u} \leftarrow \mathsf{Exposed}_\mathsf{u} \cup \{s_u\}$
      return $st_\mathsf{u}$
  else
      return $\bot$

Finalization
Input: $d \in \{0,1\}$
  return $(d = b) \vee \mathsf{win}$
```

Fig. 12. The game formalizing almost-optimal security of a secure messaging scheme. The $\boxed{\text{solid boxes}}$ indicate the differences in comparison to the game with optimal security.

6.2 Construction

Our basic scheme. As the first step, consider a simplified version of our scheme depicted in Figure 13. This construction works by appropriatly combinding one instance of our unidirectional key-updating signature scheme SkuSig, and one instance of our healable and key-updating confidentiality scheme HkuPke, per direction.



Alice
Bob
dk_A ek_B
sk_A vk_B
dk_B ek_A
sk_B vk_A
$(ek_B, c) \leftarrow \mathsf{HkuPke.Enc}(ek_B, m)$
$(dk_A, upd) \leftarrow \mathsf{HkuPke.BcUpDk}(dk_A)$
$(sk_A^{\mathsf{eph}}, \sigma) \leftarrow \mathsf{KuSig.Sign}(sk_A, (c, upd))$
c, upd, σ
$(vk_A, v) \leftarrow \mathsf{KuSig.Verify}(vk_A, (c, upd), \sigma)$
if $v = \mathtt{true}$:
$(dk_B, m) \leftarrow \mathsf{HkuPke.Dec}(dk_B, c)$
$ek_A \leftarrow \mathsf{HkuPke.BcUpEk}(ek_A, upd)$

Fig. 13. The scheme obtained by plugging our HkuPke and our SkuSig schemes together. Note how the keys are only used for the corresponding direction, except the update information for the encryption key of our sesqui-directional confidentiality scheme, which is sent along the message.

Adding post-hijack authenticity. The scheme depicted in Figure 13 does not provide any post-hijack authenticity, which we now add.

Observe that in order to achieve such a guarantee, we have to resort to sesqui-directional techniques, i.e., we have to send some update information on the channel from u to $\bar{\mathsf{u}}$ that affects the signing key for the other direction. Given that this update information must "destroy" the signing key in case of a hijack, we will use the following simple trick: the update information is simply a fresh signing key under which the other party has to sign, whenever he acknowledges the receipt of this message. Note that the signer only has to keep the latest such signing key he received, and can securely delete all previous ones. Hence, whenever he gets hijacked, the signing key that he previously stored, and that he needs to sign his next message, gets irretrievably overwritten. This, of course, requires that those signing keys are transmitted securely, and hence will be included in the encryption in our overall scheme. However, the technique as described so far does not heal properly. In order to restore the healing property, we will simply ratchet this key as well in the usual manner: whenever we use it, we sample a fresh signing key and send the verification key along. In short, the additional signature will be produced with the following key:

– If we acknowledge a fresh message, i.e., we received a message since last sending one, we use the signing key included in that message (only the last one in case we received multiple messages).
– Otherwise, we use the one we generated during sending the last message.

To further strengthen post-hijack security, the parties also include a hash of the communication transcript in each signature. This ensures that even if the deciding message has not been transferred confidentially, at least the receiver will not accept any messages sent by the honest but hijacked sender. A summary of the additional signatures, the key handling, and the transcript involved in the communication form Alice to Bob is shown in Figure 14. Of course, the actual scheme is symmetric and these additional signatures will be applied by both parties. See Figure 15 for the full description of our overall scheme.

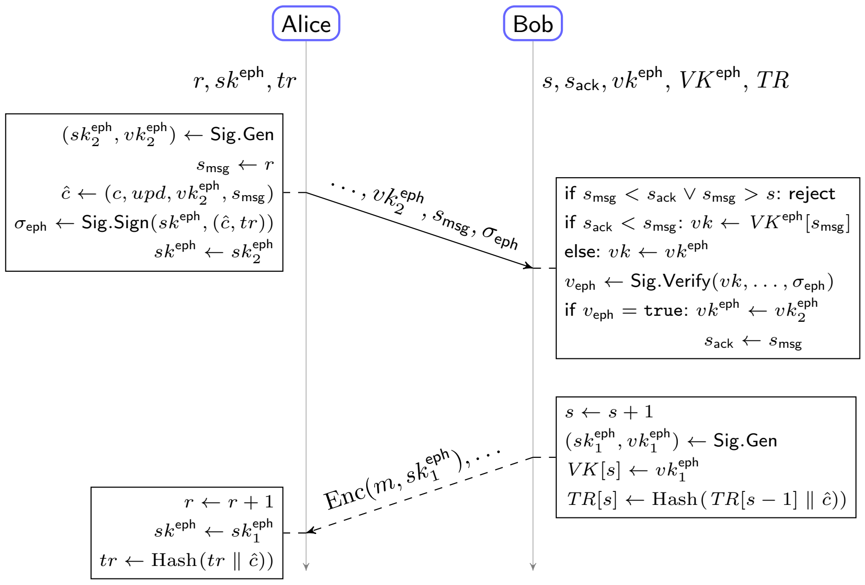

Alice
Bob
r, sk^{eph}, tr
$s, s_{\mathsf{ack}}, vk^{\mathsf{eph}}, VK^{\mathsf{eph}}, TR$
$(sk_2^{\mathsf{eph}}, vk_2^{\mathsf{eph}}) \leftarrow \mathsf{Sig.Gen}$
$s_{\mathsf{msg}} \leftarrow r$
$\hat{c} \leftarrow (c, upd, vk_2^{\mathsf{eph}}, s_{\mathsf{msg}})$
$\sigma_{\mathsf{eph}} \leftarrow \mathsf{Sig.Sign}(sk^{\mathsf{eph}}, (\hat{c}, tr))$
$sk^{\mathsf{eph}} \leftarrow sk_2^{\mathsf{eph}}$
$\ldots, vk_2^{\mathsf{eph}}, s_{\mathsf{msg}}, \sigma_{\mathsf{eph}}$
if $s_{\mathsf{msg}} < s_{\mathsf{ack}} \vee s_{\mathsf{msg}} > s$: reject
if $s_{\mathsf{ack}} < s_{\mathsf{msg}}$: $vk \leftarrow VK^{\mathsf{eph}}[s_{\mathsf{msg}}]$
else: $vk \leftarrow vk^{\mathsf{eph}}$
$v_{\mathsf{eph}} \leftarrow \mathsf{Sig.Verify}(vk, \ldots, \sigma_{\mathsf{eph}})$
if $v_{\mathsf{eph}} = \mathtt{true}$: $vk^{\mathsf{eph}} \leftarrow vk_2^{\mathsf{eph}}$
$s_{\mathsf{ack}} \leftarrow s_{\mathsf{msg}}$
$s \leftarrow s + 1$
$(sk_1^{\mathsf{eph}}, vk_1^{\mathsf{eph}}) \leftarrow \mathsf{Sig.Gen}$
$VK[s] \leftarrow vk_1^{\mathsf{eph}}$
$TR[s] \leftarrow \mathrm{Hash}(TR[s-1] \parallel \hat{c}))$
$\mathrm{Enc}(m, sk_1^{\mathsf{eph}}), \ldots$
$r \leftarrow r + 1$
$sk^{\mathsf{eph}} \leftarrow sk_1^{\mathsf{eph}}$
$tr \leftarrow \mathrm{Hash}(tr \parallel \hat{c}))$

Fig. 14. Handling of the additional signature keys for the communication from Alice to Bob. Each message additionally includes an index s_{msg}, indicating the number of messages Alice received so far, which allows Bob to look up the corresponding verification key. Moreover, they also maintain include a hash of the transcript in each signature.

Theorem 4. *Let* HkuPke *be a healable and key-updating encryption scheme, let* KuSig *be a key-updating signature scheme, and let* Sig *be a signature scheme. The scheme* SecChan *of Figure 15 is* SecMsg-Sec *secure, if* HkuPke *scheme is* HkuPke-CPA *secure,* KuSig *is* KuSig-UF *secure, and* Sig *is* 1-SUF-CMA *secure.*

A proof of Theorem 4 can be found in the full version [12].

References

[1] Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the signal protocol. In: Advances in Cryptology — EUROCRYPT 2019. Springer Berlin Heidelberg, Berlin, Heidelberg (2019), to appear

[2] Bellare, M., Kohno, T., Namprempre, C.: Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. ACM Trans. Inf. Syst. Secur. **7**(2), 206–241 (May 2004)

[3] Bellare, M., Miner, S.K.: A forward-secure digital signature scheme. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 431–448. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)

[4] Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Advances in Cryptology – CRYPTO 2017. pp. 619–650 (2017)

[5] Borisov, N., Goldberg, I., Brewer, E.: Off-the-record communication, or, why not to use pgp. In: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society. pp. 77–84. WPES '04, ACM, New York, NY, USA (2004)

[6] Canetti, R., Halevi, S., Katz, J.: A forward-secure public-key encryption scheme. In: Biham, E. (ed.) Advances in Cryptology — EUROCRYPT 2003. pp. 255–271. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[7] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A Formal Security Analysis of the Signal Messaging Protocol. 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017 pp. 451–466 (2017)

[8] Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement without key-update primitives. Cryptology ePrint Archive, Report 2018/889 (2018), `https://eprint.iacr.org/2018/889`

[9] Gentry, C., Silverberg, A.: Hierarchical id-based cryptography. In: Zheng, Y. (ed.) Advances in Cryptology — ASIACRYPT 2002. pp. 548–566. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)

[10] Horwitz, J., Lynn, B.: Toward hierarchical identity-based encryption. In: Knudsen, L.R. (ed.) Advances in Cryptology — EUROCRYPT 2002. pp. 466–481. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)

[11] Jaeger, J., Stepanovs, I.: Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 33–62. Springer (2018)

[12] Jost, D., Maurer, U., Mularczyk, M.: Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging. Cryptology ePrint Archive, Report 2018/954 (2018), `https://eprint.iacr.org/2018/954`, (full version of this paper)

[13] Kaplan, D., Kedmi, S., Hay, R., Dayan, A.: Attacking the linux prng on android: Weaknesses in seeding of entropic pools and low boot-time entropy. In: Proceedings of the 8th USENIX Conference on Offensive Technologies. pp. 14–14. WOOT'14, USENIX Association, Berkeley, CA, USA (2014)

[14] Li, Y., Shen, T., Sun, X., Pan, X., Mao, B.: Detection, classification and characterization of android malware using api data dependency. In: Thuraisingham, B., Wang, X., Yegneswaran, V. (eds.) Security and Privacy in Communication Networks. pp. 23–40. Springer International Publishing, Cham (2015)

[15] Open Whisper Systems. Signal protocol library for java/android. GitHub repository (2017), `https://github.com/WhisperSystems/libsignal-protocol-java`, accessed: 2018-10-01

[16] Poettering, Bertram and Rösler, Paul: Towards Bidirectional Ratcheted Key Exchange. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 3–32. Springer International Publishing, Cham (2018)

Construction SecChan of SecMsg

```
Algorithm SecMsg.Init
  for $\mathsf{u} \in \{\mathsf{A}, \mathsf{B}\}$ do
      $(ek_\mathsf{u}, dk_\mathsf{u}) \leftarrow \mathsf{HkuPke.Gen}$
      $(sk_\mathsf{u}^\mathsf{upd}, vk_\mathsf{u}^\mathsf{upd}) \leftarrow \mathsf{KuSig.Gen}$
      $(sk_\mathsf{u}^\mathsf{eph}, vk_\mathsf{u}^\mathsf{eph}) \leftarrow \mathsf{Sig.Gen}$
  for $\mathsf{u} \in \{\mathsf{A}, \mathsf{B}\}$ do
      $st_\mathsf{u} \leftarrow (0, 0, 0, dk_\mathsf{u}, ek_{\bar{\mathsf{u}}}, sk_\mathsf{u}^\mathsf{upd}, vk_{\bar{\mathsf{u}}}^\mathsf{upd}, sk_\mathsf{u}^\mathsf{eph}, vk_{\bar{\mathsf{u}}}^\mathsf{eph}, [\,], 0, [\,])$
  return $(st_\mathsf{A}, st_\mathsf{B})$
```

```
Algorithm SecMsg.Send
Input: $(st, m; z) \in \mathcal{S} \times \mathcal{M} \times \mathcal{R}$
  $(r, s, s_\mathsf{ack}, dk, ek, sk^\mathsf{upd}, vk^\mathsf{upd}, sk^\mathsf{eph}, vk^\mathsf{eph}, VK^\mathsf{eph}, tr, TR) \leftarrow st$
  $(sk_1^\mathsf{eph}, vk_1^\mathsf{eph}) \leftarrow \mathsf{Sig.Gen}(z_1)$          ▷ The key pair for the backwards channel.
  $(sk_2^\mathsf{eph}, vk_2^\mathsf{eph}) \leftarrow \mathsf{Sig.Gen}(z_2)$           ▷ The key pair for the forwards channel.

  ▷ Encrypt.
  $(dk, upd) \leftarrow \mathsf{HkuPke.BcUpDk}(dk; z_3)$
  $(ek, c) \leftarrow \mathsf{HkuPke.Enc}(ek, (m, sk_1^\mathsf{eph}), (upd, vk_2^\mathsf{eph}, r); z_4)$

  ▷ Sign.
  $\hat{c} \leftarrow (c, upd, vk_2^\mathsf{eph}, r)$
  $(sk^\mathsf{upd}, \sigma_\mathsf{upd}) \leftarrow \mathsf{KuSig.Sign}(sk^\mathsf{upd}, (\hat{c}, tr); z_5)$
  $\sigma_\mathsf{eph} \leftarrow \mathsf{Sig.Sign}(sk^\mathsf{eph}, (\hat{c}, tr); z_6)$

  ▷ Update the state.
  $s \leftarrow s + 1$
  $VK[s] \leftarrow vk_1^\mathsf{eph}$
  $TR[s] \leftarrow \mathrm{Hash}(TR[s-1] \parallel \hat{c})$
  $st \leftarrow (r, s, s_\mathsf{ack}, dk, ek, sk^\mathsf{upd}, vk^\mathsf{upd}, sk_2^\mathsf{eph}, vk^\mathsf{eph}, VK^\mathsf{eph}, tr, TR)$
  return $(st, (\hat{c}, \sigma_\mathsf{upd}, \sigma_\mathsf{eph}))$
```

```
Algorithm SecMsg.Receive
Input: $(st, (\hat{c}, \sigma_\mathsf{upd}, \sigma_\mathsf{eph})) \in \mathcal{S} \times \mathcal{C}$
  $(r, s, s_\mathsf{ack}, dk, ek, sk^\mathsf{upd}, vk^\mathsf{upd}, sk^\mathsf{eph}, vk^\mathsf{eph}, VK^\mathsf{eph}, tr, TR) \leftarrow st$
  $(c, upd, vk_\mathsf{msg}^\mathsf{eph}, s_\mathsf{msg}) \leftarrow \hat{c}$
  $v \leftarrow \mathtt{false}$
  if $s_\mathsf{ack} \leq s_\mathsf{msg} \leq s$ then
      if $s_\mathsf{msg} > s_\mathsf{ack}$ then
          $vk \leftarrow VK^\mathsf{eph}[s_\mathsf{msg}]$
      else
          $vk \leftarrow vk^\mathsf{eph}$
      $v_\mathsf{eph} \leftarrow \mathsf{Sig.Verify}(vk, (\hat{c}, TR[s_\mathsf{msg}]), \sigma_\mathsf{eph})$
      $(vk^\mathsf{upd}, v_\mathsf{upd}) \leftarrow \mathsf{KuSig.Verify}(vk^\mathsf{upd}, \hat{c}, \sigma_\mathsf{upd})$
      $v \leftarrow v_\mathsf{eph} \wedge v_\mathsf{upd}$
  if $v$ then
      $ek \leftarrow \mathsf{HkuPke.BcUpEk}(ek, upd)$
      $(dk, (m, sk_\mathsf{msg}^\mathsf{eph})) \leftarrow \mathsf{HkuPke.Dec}(dk, c, (upd, vk_\mathsf{msg}^\mathsf{eph}, s_\mathsf{msg}))$
      $r \leftarrow r + 1$
      $tr \leftarrow \mathrm{Hash}(tr \parallel \hat{c})$
      $st \leftarrow (r, s, s_\mathsf{msg}, dk, ek, sk^\mathsf{upd}, vk^\mathsf{upd}, sk_\mathsf{msg}^\mathsf{eph}, vk_\mathsf{msg}^\mathsf{eph}, VK^\mathsf{eph}, tr, TR)$
      return $(st, m)$
  else
      return $(st, \perp)$
```

Fig. 15. The construction of an almost-optimally secure messaging scheme.