Secure Messaging

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citation	Bornstein, Eric Rothman. 2020. Secure Messaging. Bachelor's thesis, Harvard College.
Citable link	https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364683
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Secure Messaging

Eric Bornstein
Advisor: Boaz Barak

in partial fulfillment of the requirements for the degree of
Bachelor of Arts
in the joint subjects of
Computer Science and Mathematics

Harvard University
Cambridge, MA
April 2020

Abstract

This work examines the cryptographic notion of secure two-party communication. It analyzes optimal confidentiality and authenticity in the setting of an insecure channel, insecure user states, and leaky randomness. It presents five constructions of messaging systems in the literature that range in security levels and efficiencies. It compares the schemes' overall advantages and disadvantages, including formally defining their levels of security.

Contents

1 Introduction

We would like to be able to communicate with other people online in a secure way so that no one else could read or alter our messages. Basic encryption techniques can manage this for us, but in general most encryption schemes assume that the private keys will remain private. When messaging is occurring on end-user devices, including cellphones, this assumption may not hold. For example, malware or physical access to the cellphones could be used to reveal any private keys being used for messaging. It's also possible that the software on our devices has bugs, which might leak secret keys. Maybe the data on a phone is being backed up on a server that has a security breach, or maybe the police will gain lawful access to the server. There are many possible ways for the secrets on our phones to be stolen, so we want messaging systems that give us as much security as possible when this happens.

Many recent papers have explored how to achieve security in the face of these concerns. These papers have focused on defining security levels of messaging systems formally and providing constructions that provably meet these definitions. We examine these protocols and their security levels. We compare these different protocols and discuss their relative advantages and disadvantages.

Structure In section 1.1, we give important preliminaries about cryptography that are necessary for understanding this work. In section 1.2, we define notation that we will use throughout this work. In section 2, we define a fully secure messaging scheme. In section 3, we explain some overarching ideas related to achieving security. In section 4, we present the scheme by Jaeger and Stepanovs. In section 5, we discuss security against leaked randomness. In section 6, we discuss the scheme by Jost, Maurer, and Mularczyk. In section 7, we discuss the Signal Protocol and its generalization by Alwen, Coretti, and Dodis. In section 8, we discuss two other schemes. Lastly, in section 9, we compare the schemes and conclude this work.

1.1 Preliminaries

Often in cryptography, when we discuss security, we use the notion of an adversary. We describe people trying to use the cryptographic protocols and an adversary who is trying to accomplish some task like reading a secret message. We let the adversary have certain abilities that a real-world adversary might have like stealing cryptographic keys. We say that a scheme is secure if the adversary cannot accomplish his task even if he uses his abilities. Sometimes the abilities allow the adversary to easily accomplish the task like by using a stolen decryption key to read a private message. So, we say that a scheme is secure only if the adversary cannot accomplish the task unless it is trivially easy. We formally define what constitutes security by using a game, which we denote $\mathbf{G}$. In the game, the adversary is allowed to use oracles which simulate people using the cryptographic system. The game also defines what it means to easily be able to break security.

When we say that the adversary should not be able to do some task, we mean that there should be no possible strategy that the adversary can take to do the task. Unless we are using a scheme as a building block for other schemes, we generally only care whether the

adversary is able to break the security within some reasonable amount of time. Often, instead of worrying about how much time the adversary is given to break the security, we require that the adversary cannot break the security in an amount of time that grows polynomially in the size of the cryptographic keys, which we denote λ. This way, we can pick the size of the cryptographic keys, so that the adversary cannot break the security within any reasonable amount of time.

Similarly, in many cases, the adversary always has a small chance of breaking the security of a cryptographic structure. Again, if we are not using the structure as a building block, we generally only care whether this probability is large. We can define security to mean that the adversary can only break security with probability approaching 0 faster than the inverse of every polynomial in size of the keys. We write $\mathsf{negl}(\lambda)$ to mean any probability negligibly small in λ. Thus, we can make the probability of breaking the security to be as small as we would like by only slightly increasing the key sizes.

When we discuss privacy, our hope is that the adversary is not able to distinguish between encryptions of two messages. Obviously, the adversary can guess which message is encrypted with probability one half. Because of this, we define the advantage of the adversary to be $2\mathsf{Pr}[\mathbf{win}] - 1$ where $\mathsf{Pr}[\mathbf{win}]$ is the probability that the adversary guess correctly. This has the nice property that an adversary guessing randomly has advantage 0 and an adversary that is always correct has advantage 1. Similarly, for games where we do not want the adversary to ever win, we define advantage to be $\mathsf{Pr}[\mathbf{win}]$.

Often proving that a cryptographic protocol is secure is difficult. In fact, (by our definitions) it usually would imply $P \neq NP$, which would solve a famously unsolved problem in computer science. Instead, to prove that a protocol is secure, we prove that if certain building blocks are secure, then our protocol is secure. We then basically hope the building blocks are secure because lots of smart people have spent lots of time trying to break the security and haven't been able to.

To prove that a protocol is secure in this way, we use the notion of reductions. For example, if we want to show that no poly-time adversary can win $\mathbf{G}_0$ with non-negligible probability, we can do it via contradiction. Given $\mathcal{A}_0$ that plays $\mathbf{G}_0$, we build an $\mathcal{A}_1$ that plays $\mathbf{G}_1$. We have $\mathcal{A}_1$ simulate $\mathcal{A}_0$. We want to show that $\mathcal{A}_0$ winning $\mathbf{G}_0$ implies that $\mathcal{A}_1$ wins $\mathbf{G}_1$. So, if no adversary can win $\mathbf{G}_1$ with non-negligible probability, then no adversary can win $\mathbf{G}_0$ with non-negligible probability.

Another type of reduction that we do involves three games. Given an adversary $\mathcal{A}_0$ that plays $\mathbf{G}_0$ or $\mathbf{G}_1$, we construct an adversary $\mathcal{A}'$ that plays $\mathbf{G}'$ by simulating $\mathcal{A}_0$. We then show that if $\mathcal{A}_0$ would have won $\mathbf{G}_0$ but not $\mathbf{G}_1$, then $\mathcal{A}'$ wins $\mathbf{G}'$. This bounds the advantage of $\mathcal{A}_0$ in $\mathbf{G}_0$ by the sum of his advantage in $\mathbf{G}_1$ and the advantage of $\mathcal{A}'$ in $\mathbf{G}'$.

The last major proof method that we use is called a hybrid argument. If we know that $\mathcal{A}$ wins $\mathbf{G}_n$ at least p more often than $\mathbf{G}_0$, then we know that there is some $0 \leq i < n$ such that $\mathcal{A}$ wins $\mathbf{G}_{i+1}$ at least p/n more than he wins $\mathbf{G}_i$. This is proved by a simple averaging argument.

1.2 Notation

We use the convention that cryptographic objects are fully capitalized. For example, a public key encryption scheme is abbreviated PKE. Occasionally we use lower case letters at the start

to represent adjectives. For example, a kuPKE is key-updatable PKE. Security definitions are also fully capitalized, such as CCA, which is chosen ciphertext attack security. If we want to be specific, we might write $\mathsf{PKE.CCA}$ to mean CCA security for a PKE. For algorithms, we capitalize the first letter of each word, so an encryption algorithm is Enc. If we need to specify which encryption algorithm, we write $\mathsf{PKE.Enc}$ to say that it is encryption via the public key encryption scheme. In our security games, we capitalize the first letter of each word except the first of our oracles. For example, send is an oracle in a security game, but Send is an algorithm that a cryptographic object has.

We use $\mathcal{A}$ to represent an adversary. We use $\mathbf{G}$ for a game and $\mathbb{G}$ for a group. When we define algorithms, we use $X(a, b) \to (y, z)$ to mean that running algorithm X on inputs a and b outputs the variables y and z. If we put a dollar sign on top ($\xrightarrow{\$}$), it means that the algorithm is randomized. Similarly, $\leftarrow$ and $\xleftarrow{\$}$ are used in game definitions to mean that the variables on the left are set by the variables or algorithm on the right. We use $x \xleftarrow{\$} \mathcal{R}$ to mean that x is randomness. Similarly, $x \xleftarrow{\$} \mathcal{K}$, means that x is a random key for whatever object it is being used for. We use $X(a; z)$ to denote running algorithm X on input a with randomness z. We use λ to signify the security parameter. We use U to represent a party in the communication, and we use $\overline{\mathsf{U}}$ to represent the other party.

We use $\perp$ as a special character that is different from all strings. It represents a variable that isn't initialized or a failed output from a function. We write $x[\cdot] \leftarrow y$ to mean that x is a dictionary where each value is initialized to y. Similarly, $x[a, \ldots, b] \leftarrow y$ sets the values associated with keys from a to b to the value y in the dictionary x. We use $(a, \cdot) \leftarrow y$ sets a to the first coordinate of y and ignores the second coordinate. Similarly, $\exists(a, \cdot) \in C$ means that there is some tuple in C whose first coordinate is a. If we write $\exists a \in C : X(a)$, it means that there is some element a in C such that $X(a)$ is true. We use $(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_m)$ if $n \leq m$ and $a_i = b_i$ for all $1 \leq i \leq n$. If x is a list, we use $x||b$ to be the list that results from concatenating b to x. We use $\mathcal{H}$ to represent a hash function.

In our security games, the code under the game is run. The rest of the code makes up oracles that the adversary can call. The adversary gets to choose the inputs to the oracles and gets the outputs, but can't get any of the other variables calculated by the oracles. In this way, they are treated as black boxes. In our security games we use **win** and **lose** to mean that the adversary wins or loses. Similarly, **win if** *bool* has the adversary win if *bool* is true. Otherwise, the code is run starting at the next line. We treat a win as outputting 1 and a loss as outputting 0. So, $\mathsf{Pr}[\mathbf{G}(\mathcal{A}) = 1]$ is the probability that $\mathcal{A}$ wins game $\mathbf{G}$. The advantage of an adversary is $\mathsf{Adv}(\mathcal{A})$. We write **Run** $x \xleftarrow{\$} \mathcal{A}$ in our security games to emphasize that the adversary is an algorithm that outputs x. In our security games, **req** means require. So, **req** *bool* makes an oracle return $\perp$ if *bool* is false. If *bool* is true, the oracle continues with the next line of code. In our protocol constructions, **req** *bool* means essentially the same thing. If the protocol is stateful and *bool* is false, then it should return $(\mathsf{st}, \perp)$ where st is the state from before the computation. It might return the state and several coordinates of $\perp$ depending on the definition of the algorithm.

2 Fully Secure Messaging

2.1 Definition

Before we discuss what it means for a messaging system to be secure, we must first define what a messaging system is.

A messaging system or MSG consists of three algorithms:

1. State Initialization: $\mathsf{MSG.Init}(\lambda) \xrightarrow{\$} (\mathsf{st_A}, \mathsf{st_B})$
   This randomized algorithm takes a security parameter λ and randomly creates states for two parties to message each other.

2. Sending: $\mathsf{MSG.Send}(\mathsf{st}, \mathsf{m}) \xrightarrow{\$} (\mathsf{st}^{new}, \mathsf{c})$
   This randomized algorithm takes the state of one party and a message and outputs a new state and ciphertext.

3. Receiving: $\mathsf{MSG.Rcv}(\mathsf{st}, \mathsf{c}) \xrightarrow{\$} (\mathsf{st}^{new}, \mathsf{m})/(\mathsf{st}, \perp)$
   This randomized algorithm takes the state of one party and a ciphertext. It either returns a new state and a message or it returns the old state and $\perp$ signifying that the ciphertext was not valid.

In this system, the states are stored on users' devices. A device updates its state whenever it sends or receives a message. To have any chance of security, Alice and Bob must have some shared piece of information; this is captured by the initialization algorithm. Note, if the parties have shared randomness, they can each run the initialization algorithm and only keep their own state.

If a ciphertext c is generated by a $\mathsf{Send}(\mathsf{st_U}, \mathsf{m})$ execution, then a later $\mathsf{Rcv}(\mathsf{st}_{\overline{\mathsf{U}}}, \mathsf{c})$ should output the message m. Otherwise, the receive algorithm can return $\perp$, which means that the ciphertext either did not come from the other party or isn't the next ciphertext that should be received.

In the schemes that we will consider, Rcv also outputs some information regarding what messages the other party has seen. Specifically, in most schemes we will look at, the $\mathsf{Rcv}(\mathsf{st_U}, \mathsf{c})$ algorithm tells U exactly how many messages $\overline{\mathsf{U}}$ had seen when $\overline{\mathsf{U}}$ sent c.

Some papers include associated data in their definitions of messaging systems. This is public data that is inputted to both Send and Rcv. The Rcv function should only output the message if the associated data matches the associated data used in the Send function to create that ciphertext. From the user's perspective, this is not particularly useful for messaging, but using associated data is very useful for cryptographic building blocks as it is a way to connect the different building blocks.

2.2 Correctness

Now that we have defined what algorithms make up a messaging scheme, we need to define how those functions should interact. The correctness of a MSG essentially is the property that the messaging system works as one would expect. A messaging scheme should follow

$\mathbf{G}_{\mathsf{MSG.CORR}}(\mathcal{A}, \lambda)$:
1: $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}) \xleftarrow{\$} \mathsf{Init}(\lambda)$
2: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
3: $\quad \mathsf{c}_\mathsf{U}[\cdot] \leftarrow \perp$
4: $\quad \mathsf{m}_\mathsf{U}[\cdot] \leftarrow \perp$
5: $\quad r_\mathsf{U} \leftarrow 0$
6: $\quad s_\mathsf{U} \leftarrow 0$
7: **Run** $\mathcal{A}(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B})$

Oracle $\mathsf{send}(\mathsf{U}, \mathsf{m}, z)$:
8: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
9: $(\mathsf{st}_\mathsf{U}, \mathsf{c}) \leftarrow \mathsf{Send}(\mathsf{st}_\mathsf{U}, \mathsf{m}; z)$
10: $\mathsf{m}_\mathsf{U}[s_\mathsf{U}] \leftarrow \mathsf{m}$
11: $\mathsf{c}_\mathsf{U}[s_\mathsf{U}] \leftarrow \mathsf{c}$
12: $s_\mathsf{U} \leftarrow s_\mathsf{U} + 1$
13: **return**

Oracle $\mathsf{receive}(\mathsf{U}, z)$:
14: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
15: **req** $r_\mathsf{U} < s_{\overline{\mathsf{U}}}$
16: $(\mathsf{st}_\mathsf{U}, \mathsf{m}) \leftarrow \mathsf{Rcv}(\mathsf{st}_\mathsf{U}, \mathsf{c}_{\overline{\mathsf{U}}}[r_\mathsf{U}]; z)$
17: **win if** $\mathsf{m} \neq \mathsf{m}_{\overline{\mathsf{U}}}[r_\mathsf{U}]$
18: $r_\mathsf{U} \leftarrow r_\mathsf{U} + 1$
19: **return**

Figure 1: MSG Correctness Game

the general notion that if one party runs $(\mathsf{st}_\mathsf{U}, c) \xleftarrow{\$} \mathsf{Send}(\mathsf{st}_\mathsf{U}, \mathsf{m})$, then the other party should get output m from running $\mathsf{Rcv}(\mathsf{st}_{\overline{\mathsf{U}}}, \mathsf{c})$.

Although it may seem like correctness should be a simple definition along the lines of a regular encryption scheme, it becomes more complicated when we specify in what order Send and Rcv can be called. In particular, for a messaging scheme, we want to allow messages to cross on the wire. This means that Alice and Bob can send messages at the same time and both be able to decrypt the message that the other sends. This means that each party can always send messages without having received all messages from the other party.

There are two definitions of MSG correctness in the literature. The difference is whether or not all ciphertexts need to be received in the order in which they were created. In Signal and its generalization in [1], the MSG is required to be able to receive any ciphertext as soon as it is created. This is called immediate decryption. In the rest of the schemes that we will consider, the MSG is only required to be able to receive messages in order.

Now, we will define correctness without immediate decryption CORR via the game $\mathbf{G}_{\mathsf{MSG.CORR}}$, which is defined in Figure 1. For any λ, we define the advantage of an adversary $\mathcal{A}$ in this game to be $\mathsf{Adv}_{\mathsf{MSG.CORR}}(\mathcal{A}) = \Pr[\mathbf{G}_{\mathsf{MSG.CORR}}(\mathcal{A}, \lambda) = 1]$. In particular, a MSG scheme is correct if $\mathsf{Adv}_{\mathsf{MSG.CORR}}(\mathcal{A}) = 0$ for all (even unbounded) algorithms $\mathcal{A}$.

In the CORR game, the adversary wins if he is able to have the parties disagree on the message associated with any ciphertext. He is able to choose the messages sent, the randomness used, and the order in which the messages are sent and received. The only limitation is that the ciphertexts must be received in the order that the other party created them.

We will discuss the correctness with immediate decryption definition when we discuss the Signal Protocol.

2.3 Security with Protected Randomness

There are two aspects of security for messaging: integrity (or authenticity) and privacy (or confidentiality). Integrity means that Alice should reject any ciphertext that does not come from Bob. Furthermore, the adversary should not be able to cause Alice to skip any of Bob's

messages. For example, if the adversary gives Bob's ith and $i+2$nd messages $(\mathsf{c}_i, \mathsf{c}_{i+2})$ to Alice, she should know that c_{i+2} isn't Bob's $i+1$st message. The same should be true in the other direction. Privacy just means that the adversary should not know what messages Alice or Bob send.

What differentiates secure messaging from regular encryption is that we want to maintain integrity and privacy even if the adversary is able to reveal the states of the devices. This is impossible given that if the adversary knows the state of Alice's phone, he can decrypt Bob's ciphertexts and impersonate Alice. For this reason, we define optimal security to have integrity and privacy whenever it is possible. So, if an adversary has an attack that breaks the integrity or privacy of a secure messaging system, then the same attack would break the integrity or privacy of any correct messaging system.

For now, we will consider the case where the adversary has the power to expose the states of either party and the power to send ciphertexts to either party. Later on, we will consider the scenario in which the adversary has some control over the randomness used by the parties. For now, we will assume that the randomness is protected. In section 5 we discuss the setting where randomness is not protected.

2.3.1 Trivial Attacks

Here we will explain the different attacks that would be possible for any correct messaging system.

1. Distinguishing messages based on size:
   Each messaging system will have some distribution for the size of the encryption of each message. By a counting argument, the distribution of ciphertext sizes cannot be the same for all messages. Thus, some messages can be distinguished with some probability. This is also true of basic encryption. In general, we accept this attack and define security to mean that the adversary cannot distinguish between messages of the same size.

2. Using state exposures to break integrity:
   If the adversary exposes st_U, the state of U, he is able to send messages as if he is U. He does this by running $\mathsf{Send}(\mathsf{st}_\mathsf{U}, \mathsf{m})$. In particular, he is able to send messages that $\overline{\mathsf{U}}$ will think came from U. If this happens, we say that the adversary impersonated U and that the adversary hijacked $\overline{\mathsf{U}}$. This is because now $\overline{\mathsf{U}}$ is essentially using a messaging system with the adversary, meaning the adversary essentially hijacked the messaging system. Once $\overline{\mathsf{U}}$ has received all of the messages that were sent by U before being exposed, the adversary is able to hijack $\overline{\mathsf{U}}$ because the messaging system is correct. However, once, $\overline{\mathsf{U}}$ receives a message sent by U after the exposure, it is no longer trivial to break integrity.

3. Breaking integrity after impersonation:
   When the adversary impersonates U, he hijacks $\overline{\mathsf{U}}$. Thus, $\overline{\mathsf{U}}$ is now participating in the messaging system with the adversary. For this reason, once $\overline{\mathsf{U}}$ is hijacked, the adversary can trivially send more messages that $\overline{\mathsf{U}}$ would accept.

4. Using state exposures to break privacy:
   If the adversary exposes st_{U}, the state of U, he is able to decrypt any ciphertexts that U can decrypt. He does this by running $\mathsf{Rcv}(\mathsf{st}_{\mathsf{U}}, \mathsf{c})$. If at the time of the exposure, U and $\overline{\mathsf{U}}$ were still participating in the same messaging system, then the adversary should be able to decrypt messages sent by $\overline{\mathsf{U}}$. For this attack, we only consider the case where U had not previously been hijacked, and we only consider ciphertexts which will have been sent by $\overline{\mathsf{U}}$ before $\overline{\mathsf{U}}$ is hijacked. In particular, by correctness, the adversary should be able to decrypt all messages that $\overline{\mathsf{U}}$ has already sent except for messages that U has already received. After the exposure, until U sends a new ciphertext and $\overline{\mathsf{U}}$ receives it, the adversary should also be able to decrypt all messages $\overline{\mathsf{U}}$ sends in the future due to correctness.

5. Breaking privacy after impersonation:
   When the adversary impersonates U, he hijacks $\overline{\mathsf{U}}$. Thus, $\overline{\mathsf{U}}$ is now participating in the messaging system with the adversary. For this reason, once $\overline{\mathsf{U}}$ is hijacked, the adversary can trivially read the messages that $\overline{\mathsf{U}}$ sends.

2.3.2 Aspects of Security

Besides the attacks above, the adversary should not be able to break the security or integrity of a secure messaging scheme. We will use certain words to specify different aspects of this security definition. Forward security relates to the privacy and integrity of messages in the past. A scheme is forward secure if exposing the state of U does not help the adversary decrypt the messages that U has already received and does not help the adversary replace messages that U has already sent. Healing relates to the privacy and integrity of future messages. After the adversary exposes the state of U, if after some time the adversary can no longer decrypt messages from $\overline{\mathsf{U}}$ or impersonate U, then we say that the scheme heals. Perfect healing would mean that the adversary cannot break privacy or integrity once $\overline{\mathsf{U}}$ receives a message sent by U after the exposure. Healing can also be known as future security or post-compromise security.

2.3.3 Formal Security Definition

The previous sections should give a clear understanding of what it means for a messaging system to be secure, but we will formally define security via the game $\mathbf{G}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda)$, which is found in Figure 2[1]. We define the advantage of the adversary in this game to be

$$\mathsf{Adv}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda) = 1\right] - 1.$$

We say that a messaging system is SEC secure if for any adversary running in $\mathsf{poly}(\lambda)$ time $\mathsf{Adv}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda)$.

In the game, the adversary has access to four oracles. Running $\mathsf{send}(\mathsf{U}, \mathsf{m})$ causes U to send the message m. Running $\mathsf{receive}(\mathsf{U}, \mathsf{c})$ causes U to try to decrypt c. Except in certain cases, running $\mathsf{expose}(\mathsf{U})$ outputs the state of U. Except in certain cases, running $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ will cause U to send m_b where b is chosen in the initialization phase.

[1] This game definition is organized most like the definition in [7].

$\mathbf{G}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda)$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $(\mathsf{st_A}, \mathsf{st_B}) \xleftarrow{\$} \mathsf{Init}(\lambda)$
3: **for** $\mathsf{U} \in \{\mathsf{A},\mathsf{B}\}$ **do**
4: $\quad \mathsf{c_U}[\cdot] \leftarrow \bot$
5: $\quad r_\mathsf{U} \leftarrow 0$
6: $\quad s_\mathsf{U} \leftarrow 0$
7: $\quad challs_\mathsf{U} \leftarrow \emptyset$
8: $\quad exposed_\mathsf{U} \leftarrow \{-1\}$
9: $\quad hijacked_\mathsf{U} \leftarrow \mathsf{False}$
10: **Run** $b' \xleftarrow{\$} \mathcal{A}(\lambda)$
11: **win if** $b' = b$
12: **lose**

Oracle $\mathsf{send}(\mathsf{U}, \mathsf{m})$:
13: **req** $\mathsf{U} \in \{\mathsf{A},\mathsf{B}\}$
14: $(\mathsf{st_U}, \mathsf{c}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st_U}, \mathsf{m})$
15: **if** $\neg hijacked_\mathsf{U}$ **then**
16: $\quad \mathsf{c_U}[s_\mathsf{U}] \leftarrow \mathsf{c}$
17: $\quad s_\mathsf{U} \leftarrow s_\mathsf{U} + 1$
18: **return** c

Oracle $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$:
19: **req** $\mathsf{U} \in \{\mathsf{A},\mathsf{B}\}$
20: **req** $\neg hijacked_\mathsf{U}$
21: **req** $|\mathsf{m}_0| = |\mathsf{m}_1|$
22: **req** $r_\mathsf{U} > max(exposed_{\overline{\mathsf{U}}})$
23: $challs_\mathsf{U} \leftarrow challs_\mathsf{U} \cup \{s_\mathsf{U}\}$
24: $(\mathsf{st_U}, \mathsf{c}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st_U}, \mathsf{m}_b)$
25: $\mathsf{c_U}[s_\mathsf{U}] \leftarrow \mathsf{c}$
26: $s_\mathsf{U} \leftarrow s_\mathsf{U} + 1$
27: **return** c

Oracle $\mathsf{receive}(\mathsf{U}, \mathsf{c})$:
28: **req** $\mathsf{U} \in \{\mathsf{A},\mathsf{B}\}$
29: $(\mathsf{st_U}, \mathsf{m}) \xleftarrow{\$} \mathsf{Rcv}(\mathsf{st_U}, \mathsf{c})$
30: **req** $\mathsf{m} \neq \bot$
31: **if** $hijacked_\mathsf{U}$ **then**
32: $\quad$ **return** m
33: **if** $\mathsf{c} \neq \mathsf{c}_{\overline{\mathsf{U}}}[r_\mathsf{U}]$ **then**
34: $\quad$ **win if** $r_\mathsf{U} \notin exposed_{\overline{\mathsf{U}}}$
35: $\quad hijacked_\mathsf{U} \leftarrow \mathsf{True}$
36: $\quad$ **return** m
37: $r_\mathsf{U} \leftarrow r_\mathsf{U} + 1$
38: **return**

Oracle $\mathsf{expose}(\mathsf{U})$:
39: **req** $\mathsf{U} \in \{\mathsf{A},\mathsf{B}\}$
40: **if** $hijacked_\mathsf{U}$ **then**
41: $\quad$ **return** $\mathsf{st_U}$
42: **req** $[r_\mathsf{U}, s_{\overline{\mathsf{U}}}) \cap challs_{\overline{\mathsf{U}}} = \emptyset$
43: $exposed_\mathsf{U} \leftarrow exposed_\mathsf{U} \cup \{s_\mathsf{U}\}$
44: **return** $\mathsf{st_U}$

Figure 2: MSG Security Game

The adversary has two ways to win: by breaking integrity or by breaking privacy. He can try to break privacy by figuring out the bit b, which determines whether $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ causes U to send m_0 or m_1. We say that this oracle produces a challenge ciphertext for the adversary. He can break integrity by getting U to accept a message that did not come from $\overline{\mathsf{U}}$ by running $\mathsf{receive}(\mathsf{U}, \mathsf{c})$. Similar to indistinguishability games, the adversary can trivially win this game half of the time by always guessing that b is 0 or 1. This is why we define the advantage to be $2\mathsf{Pr}\left[\mathbf{G}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda) = 1\right] - 1$, which is 0 when the adversary guess randomly and 1 when the adversary is always right.

All other aspects of the game are designed so that the adversary cannot use any trivial attack to win. The variable $challs_\mathsf{U}$ keeps track of which messages sent by U are challenges. The variable $exposed_\mathsf{U}$ keeps track of when U is exposed based on how many messages U had sent. It starts at $\{-1\}$ just so we can easily define a maximum over the set. The last new variable is $hijacked_\mathsf{U}$ which is true if and only if U has been hijacked.

Attack 1 is prohibited by requiring $|\mathsf{m}_0| = |\mathsf{m}_1|$ in chall. In receive, the adversary only wins if $\mathsf{c} \neq \mathsf{c}_{\overline{\mathsf{U}}}[r_\mathsf{U}]$ and $r_\mathsf{U} \notin exposed_{\overline{\mathsf{U}}}$. The first condition is that the ciphertext was not the next ciphertext from U. The second condition implies that the adversary could have used a state exposure to break integrity, which is Attack 2. If he does this, the oracle sets $hijacked_\mathsf{U}$ to be true. In receive, the adversary does not win by causing a hijacked party to accept a message as this would be Attack 3. If U has not been hijacked, then in $\mathsf{expose}(\mathsf{U})$, the adversary is not given the state of U if $[r_\mathsf{U}, s_{\overline{\mathsf{U}}}) \cap challs_{\overline{\mathsf{U}}} \neq \emptyset$. This condition is equivalent to saying that the adversary has already asked for a challenge ciphertext that could be

trivially decrypted if the adversary had st_{U}, which stops Attack 4. Also, $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ requires $r_{\mathsf{U}} > \max(exposed_{\mathsf{U}})$ as otherwise the adversary had already exposed a party's state that could trivially decrypt the challenge ciphertext via Attack 4. Similarly, $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ requires that U is not hijacked as this would allow Attack 5. There are no other restrictions on the adversary's use of the oracles, meaning that any nontrivial attack would constitute a win for the adversary.

3 Insights, Partial Solutions, and Limitations

In this section, we will consider ideas that are prevalent in all of the papers that we will look at. Then, we will present a simple secure construction that does not support messages crossing on the wire.

Public vs. Private Key Cryptography In general, we would prefer to use private key cryptography instead of public key cryptography as it is much faster. However, it is impossible to build a SEC secure MSG without public key cryptography. If a scheme only used private key cryptography and an adversary exposed U, then the adversary will also know the state of $\overline{\mathsf{U}}$. This allows the adversary to read all messages and impersonate either party. For a SEC secure MSG, exposing U should only allow the adversary to read messages sent by $\overline{\mathsf{U}}$ and impersonate U.

Further, public key cryptography is necessary for healing. This is true even for weaker definitions of security. Healing means that after sending enough messages back and forth after an exposure, the states of A and B will become secret. However, with only private key cryptography, after exposing a state, the adversary will be able to read the messages one by one. So, the adversary will be able to figure out the parties' states in the end.

Out of the schemes that we will consider, the ones that use more complicated public key cryptography will be more secure. They will also be slower.

Post-hijack Security One of the aspects of full security that is hardest to achieve is post-hijack security. In fact, only the scheme by Jaeger and Stepanovs has achieved this. In perfect security, if the adversary sends a message to U that isn't from $\overline{\mathsf{U}}$, this should make the state of U useless for communicating with $\overline{\mathsf{U}}$, meaning that exposing U will not help the adversary hijack $\overline{\mathsf{U}}$ or read message from $\overline{\mathsf{U}}$. To achieve this level of security, receiving a message must delete some aspect of the secret keys. However, receiving a message cannot fully delete the secret keys. Consider the case where the adversary exposes U and then has both parties send a message. The message from U should heal the exposure, but if receiving a message deletes all secret keys, then the adversary still knows the state of U. This is because the state he exposed is sufficient for decrypting the message from $\overline{\mathsf{U}}$. Thus, receiving a message must somehow delete/update some part of the secret keys, but it cannot delete the entire secret keys. The challenges are different for decryption keys and signing keys, which we will see later when we discuss the schemes.

Undetected Hijacking If the adversary hijacks U, we do not want U to accept a message from $\overline{\mathsf{U}}$ because then U would not be able to know that the adversary sent him a message.

In other words, the messaging system should not heal from an impersonation. In the real world, if Alice is hijacked, she might realize this if she doesn't receive messages she expected from Bob. We can achieve this piece of security from a collision resistant hash function $\mathcal{H}$. When parties send ciphertext c, they will store $\mathsf{tr}_s \leftarrow \mathcal{H}(\mathsf{c})$. When parties receive and accept ciphertext c, they will store $\mathsf{tr}_r \leftarrow \mathcal{H}(\mathsf{c})$. When a party sends a message, they will include tr_s in the message. When a party receives a ciphertext containing the hash tr'_s, they will only accept this message if $\mathsf{tr}_r = \mathsf{tr}'_s$. Unless the adversary hijacks one of the parties, this will always be true, so this would not break correctness. Also, because the hash function is collision resistant, the adversary would not be able to send either party a ciphertext that has the same hash as any other ciphertext that Alice and Bob have sent. Thus, the adversary could not perform this attack.

Similarly, we can have U include tr_r when he sends a message. It is not as simple for $\overline{\mathsf{U}}$ to check if this is correct because $\overline{\mathsf{U}}$ could have sent many messages that U has not yet received. If $\overline{\mathsf{U}}$ stores the hashes of all the messages he sends, then he could check whether the tr_r that U sent is correct. He would only have to store the hashes for all messages he has sent that have not been acknowledged. Similar to the previous paragraph, this would make it impossible for the adversary to hijack U and then have U send a message that $\overline{\mathsf{U}}$ would accept.

Technically, the constructions should pick a random hash function from a hash family parameterized by the security parameter, so that the probability that an adversary can find a collision is $\mathsf{negl}(\lambda)$.

3.1 Synchronous Solution

In this section, we present versions of a synchronous MSG construction. We give a brief explanation for what level of security these schemes achieve and how they achieve it.

Unidirectional MSG It is actually not difficult to construct a unidirectional MSG, which we call uniMSG, that satisfies the SEC security notion. Note, SEC security for uniMSG is based on the same game, but it only allows the adversary to call send and chall for A and $\mathsf{receive}$ for B. We just need a CCA secure PKE and a strongly unforgeable DS (digital signature scheme / public key signature scheme) with unique signatures. Our construction is in Figure 3.

The construction is very simple. Alice, the sender, generates new key pairs for the PKE and DS. Using her old encryption key, she encrypts the message and new decryption key. She also sends the new verification key. Lastly, she signs everything with her old signing key. For the encryption, she uses the last transcript hash and the new verification key as associated data. She keeps the new encryption and signing keys as her new state. She also calculates the new transcript hash. Bob, the receiver, just verifies the signature, decrypts the message, and stores the new verification key, decryption key, and transcript hash.

To break the authenticity of the messages, the adversary needs the signing key, which he can only get by exposing Alice right before she sends the message. The adversary could also use the ciphertext that Alice produces, but this isn't helpful due to the strong unforgeability of the DS. To break privacy of a ciphertext, the adversary has two options. He can get the decryption key, which is only possible by exposing Bob at some point before he receives the ciphertext. The other way is to have Bob decrypt an injected message. Due to CCA security,

```
uniMSG.Init(λ):
 1: (ek, dk) ←$ PKE.KeyGen(λ)
 2: (sk, vk) ←$ DS.KeyGen(λ)
 3: tr ← ⊥
 4: st_A ← (ek, sk, tr)
 5: st_B ← (dk, vk, tr)
 6: return (st_A, st_B)

uniMSG.Send(st_A, m):
 7: (ek, sk, tr) ← st_A
 8: (ek′, dk′) ←$ PKE.KeyGen(λ)
 9: (sk′, vk′) ←$ DS.KeyGen(λ)
10: ad ← (tr, vk′)
11: c′ ←$ PKE.Enc(ek, (m, dk′), ad)
12: σ ← DS.Sign(sk, (c′, vk′))
13: c ← (c′, vk′, σ)
14: tr′ ← H(c)
15: st_A ← (ek′, sk′, tr′)
16: return (st_A, c)

uniMSG.Rcv(st_B, c):
17: (dk, vk, tr) ← st_B
18: (c′, vk′, σ) ← c
19: req DS.Verify(vk, (c′, vk′), σ)
20: ad ← (tr, vk′)
21: m′ ← PKE.Dec(dk, c′, ad)
22: req m′ ≠ ⊥
23: (m, dk′) ← m′
24: tr′ ← H(c)
25: st_B ← (dk′, vk′, tr′)
26: return (st_B, m)
```

Figure 3: Simple Unidirectional MSG Construction

this is only useful if he can decrypt c' which is the PKE encryption containing the challenge message. He can't get Bob to decrypt this because he can't change vk' (because it is part of the associated data) or σ (because of signature uniqueness). Lastly, we see that hijacking Bob overwrites his whole state, so this scheme has post-hijack security.

Alternating MSG We can take the same idea as unidirectional to create a MSG that alternates who sends a message. We call this an altMSG that is SEC secure. Similarly, SEC security for an altMSG requires that the adversary alternate which party is sending the messages. Our construction is in Figure 4.

```
altMSG.Init(λ):
 1: (ek, dk) ←$ PKE.KeyGen(λ)
 2: (sk, vk) ←$ DS.KeyGen(λ)
 3: tr ← ⊥
 4: st_A ← (ek, sk, tr)
 5: st_B ← (dk, vk, tr)
 6: return (st_A, st_B)

altMSG.Send(st, m):
 7: (ek, sk, tr) ← st
 8: (ek′, dk′) ←$ PKE.KeyGen(λ)
 9: (sk′, vk′) ←$ DS.KeyGen(λ)
10: ad ← (tr, ek′)
11: c′ ←$ PKE.Enc(ek, (m, sk′), ad)
12: σ ← DS.Sign(sk, (c′, ek′))
13: c ← (c′, ek′, σ)
14: tr′ ← H(c)
15: st ← (dk′, vk′, tr′)
16: return (st, c)

altMSG.Rcv(st, c):
17: (dk, vk, tr) ← st
18: (c′, ek′, σ) ← c
19: req DS.Verify(vk, (c′, ek′), σ)
20: ad ← (tr, ek′)
21: m′ ← PKE.Dec(dk, c′, ad)
22: req m′ ≠ ⊥
23: (m, sk′) ← m′
24: tr′ ← H(c)
25: st ← (ek′, sk′, tr′)
26: return (st, m)
```

Figure 4: Simple Alternating MSG Construction

The construction is almost identical to the uniMSG. The sender sends ek' and sk' instead of sending vk' and dk'. The sender still encrypts the private key. Also, the sender keeps vk'

and dk' instead of keeping ek' and sk'.

Security is also clear. To break authentication of a message, the adversary needs the signing key. Again, the ciphertext that the adversary wants to replace is not useful. To do this, the adversary needs to expose the sender before or after he received the signing key. To break privacy, the adversary has the same two options. He can get the decryption key, which is only possible by exposing the receiver right before receiving. He could also get the receiver to decrypt it, but he similarly cannot change vk' or σ. Again, hijacking a party overwrites the entire state.

Synchronous MSG Although uniMSG and altMSG seem natural, we can use this construction idea so long as we know who is going to send the next message. In fact, we can use this construction idea so long as only one party tries to send a message at a time. We call this property synchronicity. Equivalently, this means that no messages can cross on the wire and that each party must receive all outstanding messages before sending one. Our construction is in Figure 5.

---

syncMSG.Init(λ):
1: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
2: $\quad (\mathsf{ek}_\mathsf{U}, \mathsf{dk}_{\overline{\mathsf{U}}}) \xleftarrow{\$} \mathsf{PKE.KeyGen}(\lambda)$
3: $\quad (\mathsf{sk}_\mathsf{U}, \mathsf{vk}_{\overline{\mathsf{U}}}) \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
4: $\mathsf{tr} \leftarrow \bot$
5: $\mathsf{st}_\mathsf{A} \leftarrow (\mathsf{ek}_\mathsf{A}, \mathsf{sk}_\mathsf{A}, \mathsf{dk}_\mathsf{A}, \mathsf{vk}_\mathsf{A}, \mathsf{tr})$
6: $\mathsf{st}_\mathsf{B} \leftarrow (\mathsf{ek}_\mathsf{B}, \mathsf{sk}_\mathsf{B}, \mathsf{dk}_\mathsf{B}, \mathsf{vk}_\mathsf{B}, \mathsf{tr})$
7: **return** $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B})$

syncMSG.Send($\mathsf{st}_\mathsf{U}, \mathsf{m}$):
8: $(\mathsf{ek}, \mathsf{sk}, \mathsf{dk}, \mathsf{vk}, \mathsf{tr}) \leftarrow \mathsf{st}_\mathsf{U}$
9: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
10: $\quad (\mathsf{ek}_\mathsf{U}, \mathsf{dk}_{\overline{\mathsf{U}}}) \xleftarrow{\$} \mathsf{PKE.KeyGen}(\lambda)$
11: $\quad (\mathsf{sk}_\mathsf{U}, \mathsf{vk}_{\overline{\mathsf{U}}}) \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
12: $\mathsf{ad} \leftarrow (\mathsf{tr}, \mathsf{ek}_{\overline{\mathsf{U}}}, \mathsf{vk}_{\overline{\mathsf{U}}})$
13: $\mathsf{c}' \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{ek}, (\mathsf{m}, \mathsf{dk}_{\overline{\mathsf{U}}}, \mathsf{sk}_{\overline{\mathsf{U}}}), \mathsf{ad})$
14: $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{sk}, (\mathsf{c}', \mathsf{ek}_{\overline{\mathsf{U}}}, \mathsf{vk}_{\overline{\mathsf{U}}}))$
15: $\mathsf{c} \leftarrow (\mathsf{c}', \mathsf{ek}_{\overline{\mathsf{U}}}, \mathsf{vk}_{\overline{\mathsf{U}}}, \sigma)$
16: $\mathsf{tr}' \leftarrow \mathcal{H}(\mathsf{c})$
17: $\mathsf{st}_\mathsf{U} \leftarrow (\mathsf{ek}_\mathsf{U}, \mathsf{sk}_\mathsf{U}, \mathsf{dk}_\mathsf{U}, \mathsf{vk}_\mathsf{U}, \mathsf{tr})$
18: **return** $(\mathsf{st}_\mathsf{U}, \mathsf{c})$

syncMSG.Rcv($\mathsf{st}_\mathsf{U}, \mathsf{c}$):
19: $(\mathsf{ek}, \mathsf{sk}, \mathsf{dk}, \mathsf{vk}, \mathsf{tr}) \leftarrow \mathsf{st}_\mathsf{U}$
20: $(\mathsf{c}', \mathsf{ek}', \mathsf{vk}', \sigma) \leftarrow \mathsf{c}$
21: **req** $\mathsf{DS.Verify}(\mathsf{vk}, (\mathsf{c}', \mathsf{ek}', \mathsf{vk}'), \sigma)$
22: $\mathsf{ad} \leftarrow (\mathsf{tr}, \mathsf{ek}', \mathsf{vk}')$
23: $\mathsf{m}' \leftarrow \mathsf{PKE.Dec}(\mathsf{dk}, \mathsf{c}', \mathsf{ad})$
24: **req** $\mathsf{m}' \neq \bot$
25: $(\mathsf{m}, \mathsf{dk}', \mathsf{sk}') \leftarrow \mathsf{m}'$
26: $\mathsf{tr}' \leftarrow \mathcal{H}(\mathsf{c})$
27: $\mathsf{st}_\mathsf{U} \leftarrow (\mathsf{ek}', \mathsf{sk}', \mathsf{dk}', \mathsf{vk}', \mathsf{tr})$
28: **return** $(\mathsf{st}_\mathsf{U}, \mathsf{m})$

Figure 5: Simple Synchronous MSG Construction

The construction essentially just combines the uniMSG and the altMSG. Importantly though, the parties generate two sets of PKE keys and two sets of DS keys. The security proof is clear as well.

So, we see that the only difficulty in achieving SEC security for a full MSG is getting a scheme that works asynchronously. We also note that the schemes in this section lose essentially all of their security if the randomness is leaked.

4 Fully Secure Jaeger-Stepanovs Scheme

The first messaging system in the literature to be SEC secure is by Jaeger and Stepanovs [6]. This scheme also achieves security in the face of randomness manipulation as we will see later on. One downside of this scheme is that sending and receiving messages take a long time.

Sending and receiving n messages can take time $\Theta(n^2)$ if the messages come from the same party. Even when the messages are sent in an alternating pattern, sending and receiving are slow. For comparable implementations, this scheme can take more than 40 times as long to send and receive messages as schemes we will see later on [4].

4.1 Building Blocks

To achieve full security, Jaeger and Stepanovs first introduce asymmetric cryptographic primitives that are updatable. Specifically, they introduce key-updatable public key encryption with associated data and key-updating digital signature schemes. These key-updatable primitives are generalizations of the cryptographic primitives that are not key-updating. These new primitives have the ability to update the public and private keys with any public string. The updated keys should still function as a public and private key pair for use in the underlying cryptographic primitive.

The next two sections will introduce these primitives formally.

4.1.1 Key-Updatable Digital Signature Schemes

Recall that a digital signature scheme DS consists of the following three algorithms.

1. Key Generation: $\mathsf{DS.KeyGen}(\lambda) \xrightarrow{\$} (\mathsf{sk}, \mathsf{vk})$
   This randomized algorithm takes a security parameter and randomly creates a signing key and a verifying key.

2. Signing: $\mathsf{DS.Sign}(\mathsf{sk}, \mathsf{m}) \xrightarrow{\$} \sigma$
   This randomized algorithm takes the signing key and a message and produces a signature for that message.

3. Verifying: $\mathsf{DS.Verify}(\mathsf{vk}, \mathsf{m}, \sigma) \rightarrow \mathsf{True/False}$
   This deterministic algorithm takes a verifying key, a message, and a signature and returns whether the signature is valid.

A key-updatable digital signature scheme kuDS is a generalization of a regular digital signature scheme, which adds the algorithms:

1. Update Signing Key: $\mathsf{kuDS.UpdSK}(\mathsf{sk}, \mathsf{upd}) \xrightarrow{\$} \mathsf{sk}^{new}$
   This randomized algorithm takes a signing key and an update string and produces a new signing key.

2. Update Verifying Key: $\mathsf{kuDS.UpdVK}(\mathsf{vk}, \mathsf{upd}) \rightarrow \mathsf{vk}^{new}$
   This deterministic algorithm takes a verifying key and an update string and produces a new verifying key.

For list of updates $\mathsf{upds} = (\mathsf{upd}_1, \ldots, \mathsf{upd}_n)$, we will also use $\mathsf{UpdSK}(\mathsf{sk}, \mathsf{upds})$ as short hand for updating sk by the elements of upds. Specifically, $\mathsf{sk}_n \xleftarrow{\$} \mathsf{UpdSK}(\mathsf{sk}_0, \mathsf{upds})$ is equivalent to running $\mathsf{sk}_{i+1} \xleftarrow{\$} \mathsf{UpdSK}(\mathsf{sk}_i, \mathsf{upd}_i)$ for i from 0 to $n-1$. We similarly define $\mathsf{UpdVK}(\mathsf{vk}, \mathsf{upds})$.

$\mathbf{G}_{\mathsf{kuDS.UFEXPOS}}(\mathcal{A}, \lambda)$:
1: $(\mathsf{sk}, \mathsf{vk}) \xleftarrow{\$} \mathsf{KeyGen}(\lambda)$
2: $\mathsf{upds} \leftarrow ()$
3: $\mathsf{m}_{sign} \leftarrow \perp$
4: $\sigma_{sign} \leftarrow \perp$
5: $\mathsf{upds}_{sign} \leftarrow \perp$
6: $\mathsf{upds}_{exp} \leftarrow \perp$
7: **Run** $(\mathsf{upds}, \mathsf{m}, \sigma) \xleftarrow{\$} \mathcal{A}(\lambda, \mathsf{vk})$
8: **lose if** $(\mathsf{m}_{sign}, \sigma_{sign}, \mathsf{upds}_{sign}) = (\mathsf{m}, \sigma, \mathsf{upds})$
9: **lose if** $\mathsf{upds}_{exp} \neq \perp \wedge \mathsf{upds}_{exp} \preceq \mathsf{upds}$
10: $\mathsf{vk}_{upd} \leftarrow \mathsf{UpdVK}(\mathsf{vk}, \mathsf{upds})$
11: $b \leftarrow \mathsf{Verify}(\mathsf{vk}, \mathsf{m}, \sigma)$
12: **win if** b
13: **lose**

Oracle $\mathsf{sign}(\mathsf{m})$:
14: **req** $\sigma_{sign} = \perp$
15: $\sigma_{sign} \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, \mathsf{m})$
16: $\mathsf{m}_{sign} \leftarrow \mathsf{m}$
17: $\mathsf{upds}_{sign} \leftarrow \mathsf{upds}$
18: **return** σ_{sign}

Oracle $\mathsf{update}(\mathsf{upd})$:
19: $\mathsf{upds} \leftarrow \mathsf{upds}||\mathsf{upd}$
20: $\mathsf{sk} \xleftarrow{\$} \mathsf{UpdSK}(\mathsf{sk}, \mathsf{upd})$
21: **return**

Oracle $\mathsf{expose}(\mathsf{U})$:
22: **req** $\mathsf{upds}_{exp} = \perp$
23: $\mathsf{upds}_{exp} \leftarrow \mathsf{upds}$
24: **return** sk

Figure 6: kuDS.UFEXPOS Security Game

Correctness The correctness for a kuDS is defined very intuitively. If we update any key pair $(\mathsf{sk}, \mathsf{vk})$ by the same sequence of updates upds, getting $(\mathsf{sk}_{\mathsf{upds}}, \mathsf{vk}_{\mathsf{upds}})$, then $(\mathsf{sk}_{\mathsf{upds}}, \mathsf{vk}_{\mathsf{upds}})$ should function as a valid key pair for the digital signature scheme.

Specifically, for any λ, m and upds, if we run $(\mathsf{sk}, \mathsf{vk}) \xleftarrow{\$} \mathsf{KeyGen}(\lambda)$, $\mathsf{sk}_{\mathsf{upds}} \xleftarrow{\$} \mathsf{UpdSK}(\mathsf{sk}, \mathsf{upds})$, $\mathsf{vk}_{\mathsf{upds}} \xleftarrow{\$} \mathsf{UpdVK}(\mathsf{vk}, \mathsf{upds})$, $\sigma \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_{\mathsf{upds}}, \mathsf{m})$, then $\mathsf{Verify}(\mathsf{vk}_{\mathsf{upds}}, \mathsf{m}, \sigma)$ should return True.

Security There are two aspects of security that we need from the kuDS so that the MSG construction achieves full security.

The first is UNIQ security which says that for any key pair $(\mathsf{sk}, \mathsf{vk})$, message m, and update sequence upds, there should be only one valid signature for m that would be accepted by $\mathsf{vk}_{\mathsf{upds}}$ the output of $\mathsf{UpdVK}(\mathsf{vk}, \mathsf{upds})$. In fact, we only need that an adversary given $(\mathsf{sk}, \mathsf{vk})$ cannot find two signatures for the same message in polynomial time.

The second is UFEXPOS, which stands for unforgeability with exposure and one signature. The formal definition is based on the game $\mathbf{G}_{\mathsf{kuDS.UFEXPOS}}$ found in Figure 6. A kuDS scheme is UFEXPOS secure if for all $\mathcal{A}$

$$\mathsf{Adv}_{\mathsf{kuDS.UFEXPOS}}(\mathcal{A}, \lambda) = \Pr\left[\mathbf{G}_{\mathsf{kuDS.UFEXPOS}}(\mathcal{A}, \lambda) = 1\right] = \mathsf{negl}(\lambda).$$

In the game, the adversary is given a verification key vk. He tries to forge a signature for vk or vk updated by some upds. The adversary is allowed one signature by the corresponding sk updated by upds_{sign}. He is also allowed to see the sk after it is updated by upds_{exp} with $\mathsf{upds}_{sign} \preceq \mathsf{upds}_{exp} \not\preceq \mathsf{upds}$.

Construction Jaeger and Stepanovs construct a kuDS from a forward secure key-evolving signature scheme with unique signatures found in [2], which is the same as a kuDS except

that it only allows the keys to be updated with the empty string ϵ.

Essentially, a kuDS signing key is based on a key-evolving signing key sk. When we want to update sk with upd, we just use sk to sign upd and we update sk by the key-evolving scheme. A kuDS signature is just the signatures for the updates and then using the updated signing key to sign the message.

Based on this construction, the key lengths, signature lengths, and runtimes for signing and verifying will be linear in the number of updates.

To show that this is secure, we only need to show that winning $\mathbf{G}_{\mathsf{kuDS.UFEXPOS}}$ reduces to winning the security game for the key-evolving scheme. This is done by showing that if the adversary forges a signature in $\mathbf{G}_{\mathsf{kuDS.UFEXPOS}}$, then he must have either forged a signature for one of the updates or for the message itself in the key-evolving scheme.

4.1.2 Key-Updatable Public Key Encryption with Associated Data

The definition of key-updatable public key encryption with associated data is analogous to that of key-updatable digital signature schemes. Recall that a public key encryption scheme PKE consists of the following three algorithms.

1. Key Generation: $\mathsf{PKE.KeyGen}(\lambda) \xrightarrow{\$} (\mathsf{ek}, \mathsf{dk})$
   This randomized algorithm takes a security parameter and randomly creates an encryption key and a decryption key.

2. Encryption: $\mathsf{PKE.Enc}(\mathsf{ek}, \mathsf{m}, \mathsf{ad}) \xrightarrow{\$} \mathsf{c}$
   This randomized algorithm takes the encryption key, a message, and associated data and produces a ciphertext for that message.

3. Decryption: $\mathsf{PKE.Dec}(\mathsf{dk}, \mathsf{c}, \mathsf{ad}) \rightarrow \mathsf{m}/\perp$
   This deterministic algorithm takes a decryption key, a ciphertext, and associated data and returns a message or reports failure with $\perp$.

A key-updatable digital signature scheme kuPKE is a generalization of a regular public key encryption scheme, which adds the algorithms:

1. Update Encryption Key: $\mathsf{kuPKE.UpdEK}(\mathsf{ek}, \mathsf{upd}) \rightarrow \mathsf{ek}^{new}$
   This deterministic algorithm takes an encryption key and an update string and produces a new encryption key.

2. Update Decryption Key: $\mathsf{kuPKE.UpdDK}(\mathsf{dk}, \mathsf{upd}) \xrightarrow{\$} \mathsf{dk}^{new}$
   This randomized algorithm takes a decryption key and an update string and produces a new decryption key.

For list of updates $\mathsf{upds} = (\mathsf{upd}_1, \ldots, \mathsf{upd}_n)$, we will also use $\mathsf{UpdEK}(\mathsf{ek}, \mathsf{upds})$ as short hand for updating ek by the elements of upds. Specifically, $\mathsf{ek}_n \leftarrow \mathsf{UpdEK}(\mathsf{ek}_0, \mathsf{upds})$ is equivalent to running $\mathsf{ek}_{i+1} \leftarrow \mathsf{UpdEK}(\mathsf{ek}_i, \mathsf{upd}_i)$ for i from 0 to $n-1$. We similarly define $\mathsf{UpdDK}(\mathsf{dk}, \mathsf{upds})$.

$\mathbf{G}_{\mathsf{kuPKE.CCAEXP}}(\mathcal{A}, \lambda)$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $(\mathsf{ek}, \mathsf{dk}) \xleftarrow{\$} \mathsf{KeyGen}(\lambda)$
3: $\mathsf{upds}_{ek} \leftarrow ()$
4: $\mathsf{upds}_{dk} \leftarrow ()$
5: $\mathsf{upds}_{exp} \leftarrow \perp$
6: $challs = \emptyset$
7: **Run** $b' \xleftarrow{\$} \mathcal{A}(\lambda, \mathsf{ek})$
8: **win if** $b' = b$
9: **lose**

Oracle $\mathsf{updateEK}(\mathsf{upd})$:
10: $\mathsf{upds}_{ek} \leftarrow \mathsf{upds}_{ek} || \mathsf{upd}$
11: $\mathsf{ek} \leftarrow \mathsf{UpdEK}(\mathsf{dk}, \mathsf{upd})$
12: **return**

Oracle $\mathsf{updateDK}(\mathsf{upd})$:
13: $\mathsf{upds}_{dk} \leftarrow \mathsf{upds}_{dk} || \mathsf{upd}$
14: $\mathsf{dk} \xleftarrow{\$} \mathsf{UpdDK}(\mathsf{dk}, \mathsf{upd})$
15: **return**

Oracle $\mathsf{chall}(\mathsf{m}_0, \mathsf{m}_1, \mathsf{ad})$:
16: **req** $\mathsf{upds}_{exp} = \perp \vee \mathsf{upds}_{exp} \not\preceq \mathsf{upds}_{ek}$
17: $\mathsf{c} \xleftarrow{\$} \mathsf{Enc}(\mathsf{ek}, \mathsf{m}_b, \mathsf{ad})$
18: $challs \leftarrow challs \cup \{(\mathsf{upds}_{ek}, \mathsf{c}, \mathsf{ad})\}$
19: **return** c

Oracle $\mathsf{decrypt}(\mathsf{c}, \mathsf{ad})$:
20: **req** $(\mathsf{upds}_{dk}, \mathsf{c}, \mathsf{ad}) \notin challs$
21: $\mathsf{m} \xleftarrow{\$} \mathsf{Dec}(\mathsf{dk}, \mathsf{c}, \mathsf{ad})$
22: **return** m

Oracle $\mathsf{expose}(\mathsf{U})$:
23: **req** $\mathsf{upds}_{exp} = \perp$
24: **req** $\nexists(\mathsf{upds}, \mathsf{c}, \mathsf{ad}) \in challs : \mathsf{upds}_{dk} \preceq \mathsf{upds}$
25: $\mathsf{upds}_{exp} \leftarrow \mathsf{upds}_{dk}$
26: **return** dk

Figure 7: kuPKE.CCAEXP Security Game

Correctness The correctness for a kuPKE is defined very intuitively. If we update any key pair $(\mathsf{ek}, \mathsf{dk})$ by the same sequence of updates upds, getting $(\mathsf{ek}_{\mathsf{upds}}, \mathsf{dk}_{\mathsf{upds}})$, then $(\mathsf{ek}_{\mathsf{upds}}, \mathsf{dk}_{\mathsf{upds}})$ should function as a valid key pair for the public key encryption scheme.

Specifically, for any λ, m, ad and upds, if we run $(\mathsf{ek}, \mathsf{dk}) \xleftarrow{\$} \mathsf{KeyGen}(\lambda)$, $\mathsf{ek}_{\mathsf{upds}} \leftarrow \mathsf{UpdEK}(\mathsf{ek}, \mathsf{upds})$, $\mathsf{dk}_{\mathsf{upds}} \xleftarrow{\$} \mathsf{UpdDK}(\mathsf{dk}, \mathsf{upds})$, $\mathsf{c} \xleftarrow{\$} \mathsf{Enc}(\mathsf{ek}_{\mathsf{upds}}, \mathsf{m}, \mathsf{ad})$, then $\mathsf{Dec}(\mathsf{dk}_{\mathsf{upds}}, \mathsf{c}, \mathsf{ad})$ should return m.

Security The security that we will need from the kuPKE scheme is chosen ciphertext attack with exposure CCAEXP. This is similar to chosen ciphertext attack security, but we allow the adversary to receive multiple challenge ciphertexts from updated versions of the same encryption key. We also let the adversary expose an updated version of the decryption key so long as that does not allow the adversary to trivially win the security game.

We formally define this security based on $\mathbf{G}_{\mathsf{kuPKE.CCAEXP}}$ found in Figure 7. Specifically, a kuPKE scheme is CCAEXP secure if for all $\mathcal{A}$

$$\mathsf{Adv}_{\mathsf{kuPKE.CCAEXP}}(\mathcal{A}, \lambda) = 2\Pr\left[\mathbf{G}_{\mathsf{kuPKE.CCAEXP}}(\mathcal{A}, \lambda) = 1\right] - 1 = \mathsf{negl}(\lambda).$$

Construction Jaeger and Stepanovs construct a kuPKE from a HIBE, which we will define briefly. First, an Identity-based Encryption scheme is a way to allow a group of identities to essentially share a public encryption key. There is one secret holder who generates a secret key and a public key. The secret holder can then generate a decryption key for each

person based on their identity. Then, people can encrypt to any specific person using only the original public key and the person's identity. The security notion for an Identity-based Encryption scheme is that the encryption is secure so long as the person's decryption key and the original secret key are secret. A Hierarchical Identity-based Encryption (HIBE) scheme is a generalization of an Identity-based Encryption scheme where identities take the form of nodes in a tree. Any person can take his decryption key and create a decryption key for all his children (sub-identities) in the identity tree

Now we discuss how to construct a kuPKE from a HIBE. To update a decryption key dk with update string upd, we just calculate the decryption key for the sub-identity associated with string upd in the HIBE. Note, to do this construction, the HIBE must support identities at unbounded depth in the tree.

Based on this construction while using the HIBE of [5], the key lengths, ciphertext lengths, and runtimes for encryption and decryption will be linear in the number of updates.

The security of the kuPKE scheme reduces very straightforwardly to the security of the HIBE.

4.2 Construction

The MSG construction of Jaeger and Stepanovs can be found in Figure 8. It uses a kuDS, a kuPKE, and a collision resistant hash function $\mathcal{H}$.

In the construction, each party stores a signing key, a verifying key, an encryption key, and a list of decryption keys. They also store tr_r the hash of the last message they received and tr_s a list of hashes of messages they have sent. They also store s, r, s^{ack}, which are the number of messages sent, the number of messages received, and the number of messages that they have sent that have been acknowledged.

Of course, the encryption keys are used to encrypt the messages they send, the decryption keys are used to decrypt ciphertexts, the signing keys are used to sign messages, and verifying keys are used to verify those signatures. The last hash of the transcript is also sent just like in Section 3. The interesting aspects of the construction are when the keys are generated and how they are updated.

Whenever a party sends a message, they will generate keys for the kuDS and the kuPKE. They will send the public keys vk, ek to the other party and keep the private keys sk, dk. They will forget the old signing key that they had, but will keep all decryption keys that they have. They cannot get rid of the old decryption keys in case the other party sends them a message before receiving the new encryption key. Also, before encrypting the messages, they will update the encryption key with hashes of sent ciphertexts, so that the encryption key will match the decryption key that the other party will use[2]. Also, when they use the kuPKE, they will use $s, r, s^{ack}, \mathsf{tr}_r$, the most recent element of tr_s, and the new public keys as associated data, which they also send.

When a party receives a message, they check that the included transcripts are correct. Before using the verifying key, they update it using tr_s to match the signing key that the

[2]In successive calls to Send, a party will update the same encryption key with the same updates. The second call will have one more update at the end. This is not actually necessary, but makes the code more readable. In particular, any time we update the encryption key, we can store the updated version and forget the old versions.

MSG.Init(λ):
1: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
2: $\quad \mathsf{dk_U}[\cdot] \leftarrow \bot$
3: $\quad (\mathsf{ek}_{\overline{\mathsf{U}}}, \mathsf{dk_U}[0]) \xleftarrow{\$} \mathsf{kuPKE.KeyGen}(\lambda)$
4: $\quad (\mathsf{sk}_{\overline{\mathsf{U}}}, \mathsf{vk_U}) \xleftarrow{\$} \mathsf{kuDS.KeyGen}(\lambda)$
5: $\mathsf{tr}_s[\cdot] \leftarrow \bot$
6: $\mathsf{tr}_s[0] \leftarrow \bot$
7: $\mathsf{tr}_r \leftarrow \bot$
8: $(s, r, s^{ack}) \leftarrow (0, 0, 0)$
9: $\mathsf{st_A} \leftarrow (s, r, s^{ack}, \mathsf{ek_A}, \mathsf{dk_A}, \mathsf{sk_A}, \mathsf{vk_A}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
10: $\mathsf{st_B} \leftarrow (s, r, s^{ack}, \mathsf{ek_B}, \mathsf{dk_B}, \mathsf{sk_B}, \mathsf{vk_B}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
11: **return** $(\mathsf{st_A}, \mathsf{st_B})$

MSG.Send(st, m):
12: $(s, r, s^{ack}, \mathsf{ek}, \mathsf{dk}, \mathsf{sk}, \mathsf{vk}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
13: $s \leftarrow s + 1$
14: $(\mathsf{ek}', \mathsf{dk}[s]) \xleftarrow{\$} \mathsf{kuPKE.KeyGen}(\lambda)$
15: $(\mathsf{sk}', \mathsf{vk}') \xleftarrow{\$} \mathsf{kuDS.KeyGen}(\lambda)$
16: $\mathsf{ad} \leftarrow (s, r, \mathsf{vk}', \mathsf{ek}', \mathsf{tr}_r, \mathsf{tr}_s[s-1])$
17: $\mathsf{ek}_{upd} \leftarrow \mathsf{kuPKE.UpdEK}(\mathsf{ek}, \mathsf{tr}_s[s^{ack}+1, \ldots, s-1])$
18: $c_{pke} \xleftarrow{\$} \mathsf{kuPKE.Enc}(\mathsf{ek}_{upd}, \mathsf{m}, \mathsf{ad})$
19: $c' \leftarrow (c_{pke}, \mathsf{ad})$
20: $\sigma \xleftarrow{\$} \mathsf{kuDS.Sign}(\mathsf{sk}, \mathsf{c}')$
21: $\mathsf{c} \leftarrow (\mathsf{c}', \sigma)$
22: $\mathsf{tr}_s[s] \leftarrow \mathcal{H}(c)$
23: $\mathsf{st} \leftarrow (s, r, s^{ack}, \mathsf{ek}, \mathsf{dk}, \mathsf{sk}', \mathsf{vk}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
24: **return** $(\mathsf{st}, \mathsf{c})$

MSG.Rcv(st, c):
25: $(s, r, s^{ack}, \mathsf{ek}, \mathsf{dk}, \mathsf{sk}, \mathsf{vk}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
26: $(\mathsf{c}', \sigma) \leftarrow c$
27: $(\mathsf{c}_{pke}, \mathsf{ad}) \leftarrow c'$
28: $(s', r', \mathsf{vk}', \mathsf{ek}', \mathsf{tr}'_r, \mathsf{tr}'_s) \leftarrow \mathsf{ad}$
29: **req** $s' = r + 1 \wedge \mathsf{tr}'^r = \mathsf{tr}^s[r'] \wedge \mathsf{tr}'^s = \mathsf{tr}^r$
30: $\mathsf{vk}_{upd} \leftarrow \mathsf{kuDS.UpdVK}(\mathsf{vk}, \mathsf{tr}_s[s^{ack}+1, \ldots, r'])$
31: $b \leftarrow \mathsf{kuDS.Verify}(\mathsf{vk}_{upd}, \mathsf{c}', \sigma)$
32: **req** b
33: $r \leftarrow r + 1$
34: $s^{ack} \leftarrow r'$
35: $\mathsf{m} \leftarrow \mathsf{kuPKE.Dec}(\mathsf{dk}[s^{ack}], \mathsf{c}_{pke}, \mathsf{ad})$
36: $\mathsf{tr}_s[0, \ldots, s^{ack}-1] \leftarrow \bot$
37: $\mathsf{dk}[0, \ldots, s^{ack}-1] \leftarrow \bot$
38: $\mathsf{tr}_r \leftarrow \mathcal{H}(\mathsf{c})$
39: $\mathsf{sk}_{upd} \xleftarrow{\$} \mathsf{kuDS.UpdSK}(\mathsf{sk}, \mathsf{tr}_r)$
40: **for** $i \in [s^{ack}, s]$ **do**
41: $\quad \mathsf{dk}[i] \xleftarrow{\$} \mathsf{kuPKE.UpdDK}(\mathsf{dk}[i], \mathsf{tr}_r)$
42: $\mathsf{st} \leftarrow (s, r, s^{ack}, \mathsf{ek}', \mathsf{dk}, \mathsf{sk}_{upd}, \mathsf{vk}', \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
43: **return** $(\mathsf{st}, \mathsf{m})$

Figure 8: Jaeger and Stepanovs MSG Construction

other party used. After decrypting the message, they will update their signing key and all their decryption keys with the hash of the current ciphertext. They also replace their old encryption key and verifying key with the new ones that they received. Lastly, they can delete any decryption keys and hashes of sent messages that they will not need anymore. This happens if the other party acknowledges a ciphertext that came after the ciphertext associated with the decryption keys and hashes.

Correctness We note that whenever a party is receiving a ciphertext from the other party, they will use a verifying key and decryption key that are associated with the signing key and encryption key that the other party used. Also, both parties will have updated their keys with the same updates, which are hashes of the transcript. Also, both parties calculate the transcripts in the same way. Thus, the correctness of the MSG is easily reduced to the correctness of the underlying kuDS and kuPKE.

Inefficiencies Although this scheme is SEC secure, it is quite inefficient. Notice that each party has to store decryption keys $\mathsf{dk}[s^{ack}, \ldots, s-1]$ and hashes of the sending transcript $\mathsf{tr}_s[s^{ack}, \ldots, s-1]$. They are storing $s - s^{ack}$ keys and transcripts, which is the number of messages that a party U has sent that U does not know whether $\overline{\mathsf{U}}$ has received. To make this even worse, each decryption key is updated every time a party receives a message, and the decryption key length grows linearly with the number of updates. So, if Alice sends n messages and Bob sends m and then they receive the other party's messages, the states will grow on the order of $\Theta(nm)$, which is very bad. This is linear in the number of times that messages cross on the wire.

Runtime is also inefficient when the parties are not communicating using alternation. In the sending algorithm, encryption takes time linear in the number of messages sent without response. In the receiving algorithm, decryption takes time linear in the size of the decryption key, which is also the number of messages the other party sent without response. So, sending and receiving take time linear in the number of messages sent without response. If one party sends n messages in a row, it takes $\Omega(n^2)$ time to send and receive all the ciphertexts.

Note that signing does not take constant time, but takes time linear in the number of messages received since sending a message. Verifying takes time linear in the number of sent messages that the other party is currently acknowledging. So, signing and verifying take amortized constant time. This means the overall runtime is $\Theta(n^2)$ to send and receive n messages in a row.

4.3 Proof of security

We claim that if the kuDS is UNIQ secure and $\mathsf{UFEXPOS}$ secure, the kuPKE is CCAEXP secure, and the hash function $\mathcal{H}$ is collision resistant, then the MSG construction in Figure 8 is SEC secure.

We prove this using game transitions. Let $\mathcal{A}$ be an algorithm that plays $\mathbf{G}_{\mathsf{MSG.SEC}}$. Taking the results of each transition and the final reduction gets $\mathsf{Adv}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda)$.

Transition 1 Let $\mathbf{G}_1$ be the same as $\mathbf{G}_{\mathsf{MSG.SEC}}$ except that we store all ciphertexts outputted by send or used as inputs to $\mathsf{receive}$ into a set called cs. If there ever are $\mathsf{c}, \mathsf{c}' \in cs$ where $\mathsf{c} \neq \mathsf{c}'$ and $\mathcal{H}(\mathsf{c}) = \mathcal{H}(\mathsf{c}')$, then the game immediately ends with a loss.

Now, we can create $\mathcal{A}_{\mathcal{H}}$ that simulates $\mathcal{A}$. We keep track of cs. If there ever are $\mathsf{c}, \mathsf{c}' \in cs$ where $\mathsf{c} \neq \mathsf{c}'$ and $\mathcal{H}(\mathsf{c}) = \mathcal{H}(\mathsf{c}')$, then $\mathcal{A}_{\mathcal{H}}$ will output this to win the collision resistance game, $\mathbf{G}_{\mathcal{H}}$. Note that $\mathcal{A}_{\mathcal{H}}$ runs in about the same amount of time as $\mathcal{A}$.

Because $\mathbf{G}_{\mathsf{MSG.SEC}}$ and $\mathbf{G}_1$ only differ when there is a hash collision, we have

$$\mathsf{Adv}_{\mathsf{MSG.SEC}}(\mathcal{A}, \lambda) - \mathsf{Adv}_1(\mathcal{A}, \lambda) \leq 2\mathsf{Adv}_{\mathcal{H}}(\mathcal{A}_{\mathcal{H}}, \lambda) = \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}_1(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}_1(\mathcal{A}, \lambda) = 1\right] - 1$ and $\mathsf{Adv}_{\mathcal{H}}(\mathcal{A}_{\mathcal{H}}, \lambda) = \mathsf{Pr}\left[\mathbf{G}_{\mathcal{H}}(\mathcal{A}_{\mathcal{H}}, \lambda)\right]$. The last step is due to the collision resistance of $\mathcal{H}$. We multiplied $\mathsf{Adv}_{\mathcal{H}}(\mathcal{A}_{\mathcal{H}}, \lambda)$ by 2 because the $\mathsf{Adv}_{\mathsf{MSG.SEC}}$ and Adv_1 are defined by 2 multiplied by the probability of winning the game.

The purpose of this transition is to ensure that there are no transcript hashes tr_r or tr_s that are equal but are hashes of different ciphertexts. The reason for this transition is that it will ensure that keys are updated with different strings. This way, in future transitions we can ensure that we can simulate the security games for kuDS and kuPKE appropriately.

Transition 2 Let $\mathbf{G}_2$ be the same as $\mathbf{G}_1$, except that the adversary loses if he calls $\mathsf{receive}(\mathsf{U}, (\mathsf{c}, \sigma))$ if $\exists i, (\mathsf{c}, \sigma') \in \mathsf{c}_{\overline{\mathsf{U}}}[i]$ with $\sigma \neq \sigma'$ and U accepts the ciphertext. This means that the adversary loses if he ever finds a second signature for a message that $\overline{\mathsf{U}}$ signed.

We can create $\mathcal{A}_{\mathsf{UNIQ}}$ that simulates $\mathcal{A}$ to play $\mathbf{G}_{\mathsf{UNIQ}}$. Given a $(\mathsf{sk}, \mathsf{vk})$ pair, we can simulate $\mathbf{G}_1$ and use $(\mathsf{sk}, \mathsf{vk})$ as one of the key pairs created in a random send call. If the adversary ever calls $\mathsf{receive}(\mathsf{U}, (\mathsf{c}, \sigma))$ where σ is the second valid signature and σ' was already found to be valid, then $\mathcal{A}_{\mathsf{UNIQ}}$ wins with outputs $\mathsf{c}, \sigma, \sigma'$ and the appropriate updates. Note that $\mathcal{A}_{\mathsf{UNIQ}}$ runs in about the same amount of time as $\mathcal{A}$.

If there are q calls to send, then there are $q + 2$ pairs of signing and verifying keys, so if $\mathcal{A}$ finds a second signature for a key pair, it has a $\frac{1}{q+2}$ probability of being the one we want. Because $\mathbf{G}_1$ and $\mathbf{G}_2$ only differ when the adversary finds a second signature, we have

$$\mathsf{Adv}_1(\mathcal{A}, \lambda) - \mathsf{Adv}_2(\mathcal{A}, \lambda) \leq 2(q+2)\mathsf{Adv}_{\mathsf{UNIQ}}(\mathcal{A}_{\mathsf{UNIQ}}, \lambda) = 2(q+2)\mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}_2(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}_2(\mathcal{A}, \lambda) = 1\right] - 1$ and $\mathsf{Adv}_{\mathsf{UNIQ}}(\mathcal{A}_{\mathsf{UNIQ}}, \lambda) = \mathsf{Pr}\left[\mathbf{G}_{\mathsf{UNIQ}}(\mathcal{A}_{\mathsf{UNIQ}}, \lambda)\right]$. The last step is due to UNIQ security of kuDS.

Transition 3 Let $\mathbf{G}_3$ be the same as $\mathbf{G}_2$, except that the win condition from $\mathsf{receive}$ becomes a loss. This is the case of the adversary successfully hijacking a party when he shouldn't be able to.

We can create $\mathcal{A}_{\mathsf{UFEXPOS}}$ that simulates $\mathcal{A}$ to play $\mathbf{G}_{\mathsf{UFEXPOS}}$. Given a vk, we can simulate $\mathbf{G}_2$ and act as if vk was generated in a random send call. When the secret key associated with vk is supposed to be updated, $\mathcal{A}_{\mathsf{UFEXPOS}}$ will call $\mathsf{update}(\mathsf{tr}_r)$. When the signing key is meant to be used, $\mathcal{A}_{\mathsf{UFEXPOS}}$ will call the sign oracle. Lastly, if $\mathcal{A}$ tries to expose the party's state that contains the signing key or an updated version of it, then $\mathcal{A}_{\mathsf{UFEXPOS}}$ gets the signing key using the expose oracle. If $\mathcal{A}$ ever hits the hijacking win condition of $\mathbf{G}_2$, then $\mathcal{A}_{\mathsf{UFEXPOS}}$ outputs the message and signature to win $\mathbf{G}_{\mathsf{kuDS.UFEXPOS}}$. Because we have ensured that no two distinct ciphertexts hash to the same value (via transition 1), we can guarantee that $\mathcal{A}_{\mathsf{UFEXPOS}}$ does not produce a trivial signature. Note that $\mathcal{A}_{\mathsf{UFEXPOS}}$ runs in about the same amount of time as $\mathcal{A}$.

If there are q calls to send, then there are $q+2$ pairs of signing and verifying keys, so if $\mathcal{A}$ forges a signature for a key pair, it has a $\frac{1}{q+2}$ probability of being the one we want. Because $\mathbf{G}_2$ and $\mathbf{G}_3$ only differ when the adversary forges a signature, we have

$$\mathsf{Adv}_2(\mathcal{A}, \lambda) - \mathsf{Adv}_3(\mathcal{A}, \lambda) = (q+2)\mathsf{Adv}_{\mathsf{UFEXPOS}}(\mathcal{A}_{\mathsf{UFEXPOS}}, \lambda) = 2(q+2)\mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}_3(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}_3(\mathcal{A}, \lambda) = 1\right] - 1$. The last step is due to $\mathsf{UFEXPOS}$ security of kuDS.

Final Reduction In this last step, we will relate $\mathsf{Adv}_3(\mathcal{A}, \lambda)$ directly to $\mathsf{Adv}_{\mathsf{CCAEXP}}(\mathcal{A}_{\mathsf{CCAEXP}}, \lambda)$ for a $\mathcal{A}_{\mathsf{CCAEXP}}$ that we will define. The general idea is that for input ek, $\mathcal{A}_{\mathsf{CCAEXP}}$ will simulate $\mathcal{A}$ playing $\mathbf{G}_3$ and act as if ek was generated in a random send call.

When the decryption key associated with ek is supposed to be updated, $\mathcal{A}_{\mathsf{CCAEXP}}$ will call $\mathsf{updateDK}$. When the encryption key is supposed to be updated, $\mathcal{A}_{\mathsf{CCAEXP}}$ will call $\mathsf{updateEK}$. As stated in the construction section, whenever we update the encryption key, we will not need to encrypt with the old key, so this does not cause problems.

Whenever $\mathcal{A}$ calls $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ when U is encrypting with ek, $\mathcal{A}_{\mathsf{CCAEXP}}$ will call $\mathsf{chall}(\mathsf{m}_0, \mathsf{m}_1, \mathsf{ad})$ where ad is the associated data that U should use to encrypt.

When the decryption key is meant to be used, $\mathcal{A}_{\mathsf{CCAEXP}}$ will call the $\mathsf{decrypt}$ oracle except for challenge ciphertexts. Because $\mathcal{A}$ never receives the decryption of challenge ciphertexts, $\mathcal{A}_{\mathsf{CCAEXP}}$ does not actually need to know the decryptions of the challenge ciphertexts.

Lastly, if $\mathcal{A}$ tries to expose the party's state that contains the decryption key or an updated version of it, then $\mathcal{A}_{\mathsf{CCAEXP}}$ gets the decryption key using the expose oracle.

When $\mathcal{A}$ outputs b', $\mathcal{A}_{\mathsf{CCAEXP}}$ also outputs b'. If the challenge oracle in $\mathbf{G}_{\mathsf{CCAEXP}}$ outputs encryptions of m_b, then, in the simulation of $\mathbf{G}_3$, the challenge oracle will also output encryptions of m_b (when encrypting with ek or an updated version). Thus, $\mathcal{A}$ wins the simulated $\mathbf{G}_3$ if and only if $\mathcal{A}_{\mathsf{CCAEXP}}$ wins $\mathbf{G}_{\mathsf{CCAEXP}}$. Because we have ensured that no two distinct ciphertexts hash to the same value (via transition 1), we can guarantee that $\mathcal{A}_{\mathsf{CCAEXP}}$ is not able to trivially decrypt a challenge ciphertext. Note that $\mathcal{A}_{\mathsf{CCAEXP}}$ runs in about the same amount of time as $\mathcal{A}$.

The problem with this reduction so far is that $\mathcal{A}_{\mathsf{CCAEXP}}$ does not know that challenge bit b, so he can't actually simulate $\mathbf{G}_3$ when the challenge should be encrypted with a different encryption key. What he does instead is treat $b = 0$ when encrypting with keys created before ek and $b = 1$ when encrypting with keys created after ek. Let ek be the mth generated encryption key. Notice that $b = 1, m = i$ produces the same simulation as if $b = 0, m = i - 1$. We can then use a hybrid argument to see that

$$\mathsf{Adv}_3(\mathcal{A}, \lambda) = (q+3)\mathsf{Adv}_{\mathsf{CCAEXP}}(\mathcal{A}_{\mathsf{CCAEXP}}, \lambda) = \mathsf{negl}(\lambda)$$

where the last step is due to CCAEXP of kuPKE.

To see the hybrid argument, let $\mathbf{G}'_3$ be the simulated version of the game. Then

$$\begin{aligned}
\mathsf{Adv}_3(\mathcal{A}, \lambda) &= \mathsf{Pr}\left[\mathbf{G}_3(\mathcal{A}, \lambda) = 1 | b = 1\right] - \mathsf{Pr}\left[\mathbf{G}_3(\mathcal{A}, \lambda) = 0 | b = 0\right] && (1)\\
&= \mathsf{Pr}\left[\mathbf{G}'_3(\mathcal{A}, \lambda) = 1 | b = 1, m = 0\right] - \mathsf{Pr}\left[\mathbf{G}'_3(\mathcal{A}, \lambda) = 0 | b = 0, m = q+2\right] && (2)\\
&= \sum_{i=0}^{q+2} \mathsf{Pr}\left[\mathbf{G}'_3(\mathcal{A}, \lambda) = 1 | b = 1, m = i\right] - \mathsf{Pr}\left[\mathbf{G}'_3(\mathcal{A}, \lambda) = 0 | b = 0, m = i\right] && (3)\\
&= (q+3)\mathsf{Pr}\left[\mathbf{G}_{\mathsf{CCAEXP}}(\mathcal{A}_{\mathsf{CCAEXP}}, \lambda) = 1 | b = 1\right] - \mathsf{Pr}\left[\mathbf{G}_{\mathsf{CCAEXP}}(\mathcal{A}_{\mathsf{CCAEXP}}, \lambda) = 0 | b = 0\right] && (4)\\
&= (q+3)\mathsf{Adv}_{\mathsf{CCAEXP}}(\mathcal{A}_{\mathsf{CCAEXP}}, \lambda). && (5)
\end{aligned}$$

Here, the third line is a telescoping sum, which uses the property that $b = 1, m = i$ produces the same simulation as if $b = 0, m = i - 1$. The fourth line uses the fact that $\mathbf{G}_{\mathsf{CCAEXP}}(\mathcal{A}_{\mathsf{CCAEXP}}, \lambda)$ is $\mathbf{G}'_3(\mathcal{A}, \lambda)$ for a random choice of m.

5 Unprotected Randomness Security

Up until this point we have considered the setting where the adversary has the ability to expose the states of the parties and to control the messaging channel. This means the adversary can choose when the parties send and receive messages and what messages and

ciphertexts they send and receive. In particular, the adversary can send a ciphertext to U that was not created by $\overline{\mathsf{U}}$.

Now, we will consider the setting where the adversary also has the ability to manipulate the randomness used by the parties. This is motivated by real attacks on randomness generators such as in [8].

When we say that the adversary has the ability to expose the state of a party, we have subtly assumed that that adversary can only expose the state of a party in between computations. In real life, it might be possible for an adversary to expose the state of a party while the party is performing a computation. As stated in [6], we can model this ability by giving the adversary the ability to expose the state of the party before the computation and also leak the randomness that the party uses during the computation. Clearly, having the state before the computation and the randomness of the computation is sufficient for calculating the state at any time during the computation.

The ability to expose a party during a computation only motivates the ability to leak randomness when the adversary already knows the state before the computation. In fact, by slightly adjusting any scheme, we can make this the only case in which exposing randomness poses any threat. The adjustment is that we add randomness storage.

Randomness Storage In the adjusted scheme, each party will store some randomness r_1 as part of their state. Then, when they are supposed to use randomness r_2 for a computation, the party can combine r_1 and r_2 to get pseudorandomness r_1' and r_2'. The party can then use r_2' for the computation and store r_1'. By combining the randomness properly, the adversary cannot know r_1' or r_2' unless he knows both r_1 and r_2. This is exactly the purpose of the $\mathsf{PRFPRNG}$ used by the Generalized Signal Protocol which we will see later.

In practice, MSG designers will need to decide whether the risk of an attack on the randomness generator is worth the time it takes to create the pseudorandomness.

Note, if a scheme does not implement this randomness storage, leaking the randomness may expose parts of the state of the party from before the computation.

Optimal Security under Leaked Randomness As mentioned previously, the ability to leak randomness should only impact security if the party's state was compromised before performing the computation. If the party's state was already compromised, leaking the randomness exposes the state after the computation. If the party is running $\mathsf{Send}(\mathsf{st}, \mathsf{m})$, then m will also be exposed. Nothing else should be exposed. In particular, the other party's state should not be compromised.

Security under Chosen Randomness We can motivate allowing the adversary to choose the randomness in a similar way. We would still want security even if the adversary is able to choose what value the randomness generator outputs. Also, if the adversary gets access to one of the parties' devices, he may be able to cause the device to performa a computation using certain randomness.

For the most part, optimal security under chosen randomness is the same as optimal security under leaked randomness. They differ on the notion of post-hijack security. When the adversary hijacks U, this should cause the state of U to be out of sync with $\overline{\mathsf{U}}$. Then,

exposing the state of U should not help the adversary read messages from $\overline{\mathsf{U}}$ or hijack $\overline{\mathsf{U}}$. However, if the adversary can choose the randomness that $\overline{\mathsf{U}}$ will use, he can hijack U and keep U and $\overline{\mathsf{U}}$ in sync. This should only work if the adversary uses ciphertext c to hijack U and then chooses randomness so that $\overline{\mathsf{U}}$ also sends c. Essentially, the adversary hijacks U by predicting what message $\overline{\mathsf{U}}$ will send. For real world security, this doesn't matter.

In general, it seems that a MSG that is secure when randomness leaks should be secure under chosen randomness unless there are certain choices of randomness that are just bad. If there are bad choices of randomness, a scheme could likely be adjusted to use different randomness instead of the bad choice of randomness.

Security of JS scheme under Unprotected Randomness If we add in randomness storage, the scheme by Jaeger and Stepanovs is optimally secure in the setting of leaked randomness and chosen randomness. Informally, we see this because a call to Send does not create any secrets that the other party will store. To formally show that the scheme is secure even when the adversary can leak or choose the randomness, one can just go through almost the same steps as the original security proof.

6 Jost-Maurer-Mularczyk Scheme

In [7], Jost, Maurer, and Mularczyk try to use simple public key cryptography methods to build a MSG that is as secure as possible. Their scheme is very similar to the one of Jaeger and Stepanovs. The big difference in construction is that while Jaeger and Stepanovs use versions of PKE and DS that can be updated with public information, Jost, Maurer, and Mularczyk use a version of PKE that can only be securely updated with secret information. Jost, Maurer, and Mularczyk also employ ephemeral (one-time use) keys for PKE and DS. In the end, they claim to create an MSG that almost has SEC security and almost is secure against chosen randomness. This claim is incorrect, but we show how to fix their scheme to achieve it. Then, the scheme's only non-trivial security flaw is its post-hijack security. In particular, the scheme loses security if the messages sent by U directly before or after the adversary hijacks $\overline{\mathsf{U}}$ are compromised.

6.1 Secretly Key-Updatable Public Key Encryption

This building block is similar to the key-updatable public key encryption (kuPKE) of Jaeger and Stepanovs, but the updates to the decryption keys are expected to be secret. So, it is called a secretly key-updatable public key encryption scheme (skuPKE). It also generalizes the PKE, so it uses the same $\mathsf{Init}, \mathsf{Enc}$, and Dec algorithms[3]. It also has the following three algorithms.

1. Update Generation: $\mathsf{skuPKE.UpdGen}() \xrightarrow{\$} (\mathsf{upd}_e, \mathsf{upd}_d)$
   This randomized algorithm creates an update that can be used for encryption keys and

[3]These won't need to use associated data, so we will exclude associated data from the arguments to Enc and Dec

a corresponding update that can be used for decryption keys[4].

2. Update Encryption Key: $\mathsf{skuPKE.UpdEK}(\mathsf{ek}, \mathsf{upd}_e) \to \mathsf{ek}^{new}$
   This deterministic algorithm takes an encryption key and an update and produces a new encryption key.

3. Update Decryption Key: $\mathsf{skuPKE.UpdDK}(\mathsf{dk}, \mathsf{upd}_d) \to \mathsf{dk}^{new}$
   This deterministic algorithm takes a decryption key and an update and produces a new decryption key.

Correctness Correctness for a skuPKE is defined intuitively. If we take a key pair $(\mathsf{ek}, \mathsf{dk})$ and update them both by corresponding updates, then they should still be a valid key pair.

Formally, if we run $(\mathsf{ek}_0, \mathsf{dk}_0) \xleftarrow{\$} \mathsf{Init}(\lambda)$ and $(\mathsf{upd}_e^i, \mathsf{upd}_d^i) \xleftarrow{\$} \mathsf{UpdGen}()$, $\mathsf{ek}_i \leftarrow \mathsf{UpdEK}(\mathsf{ek}_{i-1}, \mathsf{upd}_e^i)$, $\mathsf{dk}_i \leftarrow \mathsf{UpdDK}(\mathsf{dk}_{i-1}, \mathsf{upd}_d^i)$ for i from 1 to k, then with probability 1, running, $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{ek}_k, \mathsf{m})$ and $\mathsf{m}' \leftarrow \mathsf{Dec}(\mathsf{dk}_k, \mathsf{c})$ gets $\mathsf{m} = \mathsf{m}'$.

Security In some ways a skuPKE is inherently weaker than a kuPKE because we expect the skuPKE updates to be secret. If we update a decryption key dk and then let the adversary expose the new decryption key dk', we only need dk to be secure if the update was secret. For a kuPKE, dk would have been secure even if the update was public. However, making the updates secret allows the opportunity to heal from exposures, which we want.

Consider taking a decryption key dk_0 and set $\mathsf{dk}_i \leftarrow \mathsf{UpdDK}(\mathsf{dk}_{i-1}, \mathsf{upd}_d^i)$ for i from 1 to k. If upd_d^a and upd_d^b are secure for $a < b$, then we expect dk_i to be secure for $i \in [a, b)$ so long as none of those decryption keys are exposed. We also want to think of dk_0 as being generated by a secret update if it was initialized with secret randomness.

For kuPKE, there is no problem in updating different keys with the same update information because the update information is public. We want the same guarantees with skuPKE. We do require that all the decryption keys are updated at the same time with the same updates and that if the adversary exposes one decryption key, he exposes them all. We also allow the different decryption keys to be generated at different times.

Given that we would like to use the skuPKE to encrypt the updates to the skuPKE, we would want security even when the encryption keys are used to encrypt the updates. This is circular security of a sort. Also, it turns out that we should only allow the adversary to expose the decryption keys once. This is so that we can prove the construction is secure according to this definition. This will turn out to be fine for our overall security

We formally define this security based on $\mathbf{G}_{\mathsf{skuPKE.CPAEXP}}$ found in Figure 9. Specifically, a skuPKE scheme is CPAEXP secure if

$$\mathsf{Adv}_{\mathsf{skuPKE.CPAEXP}}(\mathcal{A}, \lambda) = 2\Pr\left[\mathbf{G}_{\mathsf{skuPKE.CPAEXP}}(\mathcal{A}, \lambda) = 1\right] - 1 = \mathsf{negl}(\lambda).$$

Note, here the security prohibits chosen plaintext attacks whereas kuPKE security prohibits chosen ciphertext attacks.

[4]This algorithm is based on the structure of the encryption and decryption keys. We will end up using El Gamal encryption, so this algorithm depends on the group we choose. We write this algorithm as if there are no arguments just to simplify exposition.

$\mathbf{G}_{\mathsf{kuPKE.CCAEXP}}(\mathcal{A}, \lambda)$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $n \leftarrow 0$
3: $\mathsf{upd}_e[\cdot] \leftarrow \perp$
4: $\mathsf{upd}_d[\cdot] \leftarrow \perp$
5: $\mathsf{ek}[\cdot] \leftarrow \perp$
6: $\mathsf{dk}[\cdot] \leftarrow \perp$
7: $i_{gen}, i_d \leftarrow (1,1)$
8: $i_e[\cdot] \leftarrow 1$
9: $secure \leftarrow \{1\}$
10: $challs \leftarrow \emptyset$
11: $exp \leftarrow -1$
12: **Run** $b' \xleftarrow{\$} \mathcal{A}(\lambda)$
13: **win if** $b' = b$
14: **lose**

Oracle $\mathsf{newKey}()$:
15: $n \leftarrow n + 1$
16: $(\mathsf{ek}[n], \mathsf{dk}[n]) \leftarrow \mathsf{KeyGen}(\lambda)$
17: $i_e[n] \leftarrow i_{gen}$
18: **return**

Oracle $\mathsf{updGen}(z)$:
19: $i_{gen} \leftarrow i_{gen} + 1$
20: **if** $z = \perp$ **then**
21: $\quad(\mathsf{upd}_e[i_{gen}], \mathsf{upd}_d[i_{gen}]) \xleftarrow{\$} \mathsf{UpdGen}()$
22: $\quad secure \leftarrow secure \cup \{i_{gen}\}$
23: **else**
24: $\quad(\mathsf{upd}_e[i_{gen}], \mathsf{upd}_d[i_{gen}]) \leftarrow \mathsf{UpdGen}(; z)$
25: **return** $\mathsf{upd}_e[i_{gen}]$

Oracle $\mathsf{updateEK}(j)$:
26: **req** $j \in [1, n]$
27: **req** $i_e[j] < i_{gen}$
28: $i_e[j] \leftarrow i_e[j] + 1$
29: $\mathsf{ek}[j] \leftarrow \mathsf{UpdEK}(\mathsf{ek}[j], \mathsf{upd}_e[i_e[j]])$
30: **return**

Oracle $\mathsf{updateDK}()$:
31: **req** $i_d < i_{gen}$
32: $i_d \leftarrow i_d + 1$
33: **for** $j \in [1, n]$ **do**
34: $\quad\mathsf{dk}[j] \leftarrow \mathsf{UpdDK}(\mathsf{dk}[j], \mathsf{upd}_e[i_d])$
35: **return**

Oracle $\mathsf{chall}(j, \mathsf{m}_0, \mathsf{m}_1, i)$:
36: **req** $j \in [1, n]$
37: **req** $|\mathsf{m}_0| = |\mathsf{m}_1|$
38: **req** $i \in [2, i_{gen}]$
39: **req** $\neg(exp \geq i_d \wedge (i_e[j], exp] \cap secure = \emptyset)$
40: **req** $\neg(exp \leq i_d \wedge (exp, i_e[j]] \cap secure = \emptyset)$
41: $challs \leftarrow challs \cup \{i_e[j]\}$
42: $\mathsf{c} \xleftarrow{\$} \mathsf{skuPKE}(\mathsf{ek}[j], \mathsf{m}_b || \mathsf{upd}_d[i])$
43: **return** c

Oracle $\mathsf{expose}()$:
44: **req** $exp = -1$
45: **req** $\nexists c \in challs : c \geq i_d \wedge (i_d, c] \cap secure = \emptyset$
46: **req** $\nexists c \in challs : c \leq i_d \wedge (c, i_d] \cap secure = \emptyset$
47: **return** dk

Figure 9: skuPKE.CPAEXP Security Game

In the construction, we keep track of how many times each encryption key has been updated with i_e. Also, i_{gen}, i_d are the number of updates that have been generated and the number of updates to the decryption keys. The set *secure* is the set of updates that are secret.

Construction Our construction of a skuPKE is quite simple. It is just El Gamal encryption. Updates are of the form (g^a, a). A decryption key b is updated to $b + a$. An encryption key g^b is updated to $g^a g^b$. Recall, we have $(g^r, \mathcal{H}(\mathsf{ek}^r) \oplus \mathsf{m}) \leftarrow \mathsf{Enc}(\mathsf{ek}, \mathsf{m}; r)$ for $r \in \mathbb{Z}_p$ where p is the size of the underlying group. Also, $\mathcal{H}(h^{\mathsf{dk}}) \oplus c \leftarrow \mathsf{Dec}(\mathsf{dk}, (h, c))$.

Proof of Security This construction is secure in the random oracle model if the Computational Diffie-Hellman assumption is true. We give the abbreviated proof here. For an $\mathcal{A}$ that wins that skuPKE game, we want to create an $\mathcal{A}_{CDH}$ that can compute g^{ab} from $A = g^a$ and $B = g^b$

A standard hybrid argument shows that if $\mathcal{A}$ can win the skuPKE game, then there is some i, where $\mathcal{A}$ would win the game even if the jth challenge encrypted m_0 for $j < i$, m_1 for $j > i$, and m_b for $j = i$.

Then, $\mathcal{A}_{CDH}$ simulates this new game. First, $\mathcal{A}_{CDH}$ guesses that the last secure update before the ith challenge will be the jth. Instead of making the jth secure update (g^r, r), $\mathcal{A}_{CDH}$ will make it $(g^{r_1} A, r_1)$. Then, the $j + 1$st secure update will be $(g^{r_2} A^{-1}, r_2)$ so that the decryption keys match the encryption keys after the $j + 1$st secure update. It is possible to guess that there will be no secure updates before the ith challenge. In this case, we just multiply A to the encryption keys until the first secure update is $(g^{r_2} A^{-1}, r_2)$. Then, $\mathcal{A}$ will ask for the ith challenge with encryption key $g^x A$. When, this happens, $\mathcal{A}_{CDH}$ gives the ciphertext (B, r_3) for random r_3 with $|r_3| = |\mathsf{m}_0|$.

The only way for $\mathcal{A}$ to win this simulation is to ask for a hash of B^{xa}. If he didn't ask for that hash, then the hash is equally likely to be $r_3 \oplus \mathsf{m}_0$ and $r_3 \oplus \mathsf{m}_1$. So, if $\mathcal{A}$ asks for hashes of $(h_1, \ldots, h_q)$, $\mathcal{A}_{CDH}$ can output $h_k B^{-x}$ for random k. If $\mathcal{A}$ ever asks for a hash of B^{xa}, which he does with non-negligible probability, then $\mathcal{A}_{CDH}$ has a $1/q$ chance of outputting g^{ab}.

6.2 Sesquidirectional Privacy

In this section, we will discuss how Jost, Maurer, and Mularczyk use the skuPKE to achieve privacy. To do this, we will focus on how they try to maintain privacy of messages from Alice to Bob. They do this via a scheme called a healable and key updating public key encryption scheme (hkuPKE). This is essentially just an MSG where messages are only sent from Alice to Bob. Bob sends ciphertexts to Alice only to heal from exposures. We want the hkuPKE to be private so long as the ciphertexts from Bob to Alice are authenticated. We call this sesquidirectional because sesqui is Latin for one and a half. The half represents the ciphertexts from Bob that do not contain messages.

Construction The construction can be found in Figure 10. It uses a CCA secure PKE and the CPAEXP secure skuPKE.

The general idea is that we will use the skuPKE in essentially the same way that Jaeger and Stepanovs use the kuPKE. Instead of updating with the transcript. The sender generates an update and sends it encrypted. If we didn't use the PKE, this is exactly what our construction is. However, this would provide no post-hijack security. If the adversary hijacks Bob and then exposes Bob, the adversary can figure out the decryption keys from before the hijack. This is because when Bob receives the injected message, he will update his decryption keys with the adversarially chosen updates.

To achieve post-hijack security, some aspect of Bob's decryption key must be deleted when he receives a message. Jost, Maurer, and Mularczyk use an additional layer of PKE. Because each key will be used at most once, we call them ephemeral. Whenever a party sends a message, they generate a new PKE key pair. Bob sends the encryption key to Alice when he sends a ciphertext. When Alice sends a message, she sends Bob randomness z to be used in PKE.Init so Bob can get the decryption key, This ensures that the decryption key is unrelated to other keys. Bob stores all the decryption keys he makes, until he receives a decryption key from Alice. When he receives a key from Alice, that is the only key that he stores. Alice uses the matching encryption key.

Security For security of hkuPKE, we give the adversary Alice's state and he can control the messages sent to Bob. He is also able to leak Alice's randomness and expose Bob's state. We also let him leak Bob's randomness and treat this like an exposure of his state from after the computation. Once Bob is hijacked, we do not allow ciphertexts to be sent by Bob.

The messages from Alice to Bob should be secure so long as the randomness is not leaked and either the skuPKE decryption key or the ephemeral PKE decryption key are secure. Also, we claim that the skuPKE is fully secure excluding post-hijack security. If the

hkuPKE.Init(λ):
$s, r, s^{ask} \leftarrow (0, 0, 0)$
$\mathsf{dk}[\cdot] \leftarrow \perp$
$\mathsf{dk}_{eph}[\cdot] \leftarrow \perp$
$\mathsf{upd}_e[\cdot] \leftarrow \perp$
$(\mathsf{ek}, \mathsf{dk}[0]) \xleftarrow{\$} \mathsf{skuPKE.KeyGen}(\lambda)$
$(\mathsf{ek}_{eph}, \mathsf{dk}_{eph}[0]) \xleftarrow{\$} \mathsf{PKE.KeyGen}(\lambda)$
$\mathsf{tr}_s, \mathsf{tr}_r \leftarrow (\perp, \perp)$
$\mathsf{st_A} \leftarrow (s, r, \mathsf{ek}, \mathsf{ek}_{eph}, \mathsf{tr}_s, \mathsf{upd}_e, \lambda)$
$\mathsf{st_B} \leftarrow (s, r, s^{ack}, \mathsf{dk}, \mathsf{dk}_{eph}, \mathsf{tr}_r, \lambda)$
return $(\mathsf{st_A}, \mathsf{st_B})$

hkuPKE.SendA(st, m, ad):
$(s, r, \mathsf{ek}, \mathsf{ek}_{eph}, \mathsf{tr}_s, \mathsf{upd}_e, \lambda) \leftarrow \mathsf{st}$
$s \leftarrow s + 1$
$(\mathsf{upd}_e[s], \mathsf{upd}_d) \xleftarrow{\$} \mathsf{skuPKE.UpdGen}()$
$z \xleftarrow{\$} \mathcal{R}$
$\mathsf{ek}_{upd} \leftarrow \mathsf{skuPKE.UpdEK}(\mathsf{ek}, \mathsf{upd}_e[s^{ack}+1, \ldots, s-1])$
$\mathsf{c}_{pke} \xleftarrow{\$} \mathsf{skuPKE.Enc}(\mathsf{ek}_{upd}, (\mathsf{m}, \mathsf{upd}_d, z))$
$\mathsf{c}_{eph} \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{ek}_{eph}, \mathsf{c}_{pke}, \mathsf{ad})$
$\mathsf{c} \leftarrow (\mathsf{c}_{eph}, r)$
$(\mathsf{ek}'_{eph}, \cdot) \leftarrow \mathsf{PKE.Init}(\lambda; \mathcal{H}(\mathsf{tr}_s || z))$
$\mathsf{tr}_s \leftarrow \mathcal{H}(\mathsf{c})$
$\mathsf{st} \leftarrow (s, r, \mathsf{ek}, \mathsf{ek}'_{eph}, \mathsf{tr}_s, \mathsf{upd}_e, \lambda)$
return $(\mathsf{st}, \mathsf{c})$

hkuPKE.SendB(st):
$(s, r, s^{ack}, \mathsf{dk}, \mathsf{dk}_{eph}, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
$s \leftarrow s + 1$
$(\mathsf{ek}, \mathsf{dk}[s]) \xleftarrow{\$} \mathsf{skuPKE.KeyGen}(\lambda)$
$(\mathsf{ek}_{eph}, \mathsf{dk}_{eph}[s]) \xleftarrow{\$} \mathsf{PKE.KeyGen}(\lambda)$
$\mathsf{c} \leftarrow (\mathsf{ek}, \mathsf{ek}_{eph}, s)$
$\mathsf{st} \leftarrow (s, r, s^{ack}, \mathsf{dk}, \mathsf{dk}_{eph}, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
return $(\mathsf{st}, \mathsf{c})$

hkuPKE.RcvA(st, c):
$(s, r, \mathsf{ek}, \mathsf{ek}_{eph}, \mathsf{tr}_s, \mathsf{upd}_e, \lambda) \leftarrow \mathsf{st}$
$(\mathsf{ek}', \mathsf{ek}'_{eph}, r') \leftarrow \mathsf{c}$
if $r' \geq s$ **then**
    $\mathsf{ek}_{eph} \leftarrow \mathsf{ek}'_{eph}$
$\mathsf{ek} \leftarrow \mathsf{ek}'$
$r \leftarrow r + 1$
$\mathsf{st} \leftarrow (s, r, \mathsf{ek}, \mathsf{ek}_{eph}, \mathsf{tr}_s, \mathsf{upd}_e, \lambda)$
return st

hkuPKE.RcvB(st, c, ad):
$(s, r, s^{ack}, \mathsf{dk}, \mathsf{dk}_{eph}, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
$(\mathsf{c}_{eph}, r') \leftarrow \mathsf{c}$
req $s^{ack} \leq r' \leq s$
$\mathsf{c}_{pke} \leftarrow \mathsf{PKE.Dec}(\mathsf{dk}_{eph}[r'], \mathsf{c}, \mathsf{ad})$
req $\mathsf{c}_{pke} \neq \perp$
$\mathsf{m}' \leftarrow \mathsf{skuPKE.Dec}(\mathsf{dk}[r'], \mathsf{c}_{pke})$
req $\mathsf{m}' \neq \perp$
$(\mathsf{m}, \mathsf{upd}_d, z) \leftarrow \mathsf{m}'$
$r \leftarrow r + 1$
$s^{ack} \leftarrow r'$
$\mathsf{m} \leftarrow \mathsf{kuPKE.Dec}(\mathsf{dk}[s^{ack}], \mathsf{c}_{pke})$
$\mathsf{dk}[0, \ldots, s^{ack} - 1] \leftarrow \perp$
$(\cdot, \mathsf{dk}'_{eph}) \leftarrow \mathsf{PKE.Init}(\lambda; \mathcal{H}(\mathsf{tr} || z))$
for $i \in [s^{ack}, s]$ **do**
    $\mathsf{dk}[i] \xleftarrow{\$} \mathsf{skuPKE.UpdDK}(\mathsf{dk}[i], \mathsf{upd}_d)$
    $\mathsf{dk}_{eph}[i] \xleftarrow{\$} \mathsf{dk}'_{eph}$
$\mathsf{st} \leftarrow (s, r, s^{ack}, \mathsf{dk}, \mathsf{dk}_{eph}, \mathsf{tr}_r, \lambda)$
$\mathsf{tr}_r \leftarrow \mathcal{H}(\mathsf{c})$
return $(\mathsf{st}, \mathsf{m})$

Figure 10: Jost, Maurer, and Mularczyk Sesquidirectional Unauthenticated hkuPKE Construction

adversary exposes Bob, the adversary will still be able to decrypt all the messages sent by Alice until Alice receives a new ciphertext from Bob.

Now, we consider when the ephemeral PKE decryption key is secure. Consider the case where Bob has received r messages from Alice before being hijacked. If the rth message that Alice sent was secure (meaning the adversary cannot trivially decrypt it and the randomness was not leaked), then hijacking Bob will overwrite the ephemeral decryption key for Alice's $r+1$st message. If the $r+1$st message from Alice is also sent securely, the skuPKE encryption key will be updated secretly and the new ephemeral PKE decryption key will be secure as well. So, if the rth and $r+1$st messages sent by Alice are secure, exposing Bob after hijacking

should not allow the adversary to break privacy. If either of the messages are not secure, exposing Bob after hijacking is just as bad as exposing him before hijacking him.

Proof of Security We only give a brief idea of the security proof here. See [7] for the full proof. Let $\mathcal{A}$ be an adversary that wins the hkuPKE game.

We want to first show that the messages before hijacking are secure. We do this by reducing to the security of the skuPKE. Essentially, each time the adversary knows Bob's state and makes him send a message, there is a new instance of the skuPKE game. By a hybrid argument, there is some i where $\mathcal{A}$ would still win the game even if in the $j < i$ skuPKE game instance all challenges were encryptions of m_0, in the $j > i$ skuPKE game instance all challenges were encryptions of m_1, and only in the ith instance of a skuPKE game were the challenges correct.

Specifically, $\mathcal{A}_1$ plays the skuPKE game and uses it to simulate this changed hkuPKE game. When $\mathcal{A}$ has Alice send a message, $\mathcal{A}_1$ calls updGen and updateEK. Until Bob is exposed, when Bob sends a message, $\mathcal{A}_1$ calls newKey. When Bob receives a message, $\mathcal{A}_1$ calls updateDK. When Bob is exposed, $\mathcal{A}_1$ calls expose. Note that $\mathcal{A}_1$ will only need to call expose once because he will be able to figure out Bob's state. Lastly, when Alice sends a challenge message, $\mathcal{A}_1$ calls chall.

Because the skuPKE is secure, we have shown that the hkuPKE is secure until hijacking. Now, we want to show that unless the message before or after hijacking is compromised, then there still is post-hijack privacy. Essentially, if the messages are secure, we can replace the randomness used to create the ephemeral decryption keys from the ciphertext with other randomness. The adversary cannot notice this due to the privacy we have shown. Lastly, we can then reduce to the security of the ephemeral encryption.

6.3 Construction

At this point, we could use the hkuPKE of Jost, Maurer, and Mularczyk with the authentication methods of Jaeger and Stepanovs to get an MSG that is fully secure except for post-hijack privacy. Jost, Maurer, and Mularczyk instead propose a method of authentication without using kuDS. They just sign with two DS keys: $\mathsf{sk}, \mathsf{sk}^{eph}$. When a party sends a message, they also generate new values of sk and sk^{eph}. They also generate an ephemeral signing key for the other party and send it decrypted. When a party receives a message, they will make sk^{eph} the new ephemeral signing key. Essentially, sk is only changed when a party sends a message, but sk^{eph} is changed every time a message is sent or received. The goal of overwriting sk^{eph} even when receiving a message is to provide some post-hijack authentication. Lastly, the parties keep track of the corresponding verification keys and use them to receive a ciphertext. The goal of overwriting sk and sk^{eph} when sending a message is to allow for healing. The full construction is found in Figure 11

6.4 Security

First, we consider what authenticity the signatures in the MSG provide. For an adversary to impersonate U, the adversary needs the right value of sk, which he can only get by exposing U before U sends another message or leaking the randomness of the original message. However,

$\mathsf{MSG.Init}(\lambda)$:
1: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
2: $(\mathsf{st}^e_{\mathsf{U}}, \mathsf{st}^d_{\overline{\mathsf{U}}}) \xleftarrow{\$} \mathsf{hkuPKE.Init}(\lambda)$
3: $(\mathsf{sk}_{\overline{\mathsf{U}}}, \mathsf{vk}_{\mathsf{U}}) \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
4: $\mathsf{vk}^{eph}_{\mathsf{U}}[\cdot] \leftarrow \perp$
5: $(\mathsf{sk}^{eph}_{\overline{\mathsf{U}}}, \mathsf{vk}^{eph}_{\mathsf{U}}[-1]) \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
6: $\mathsf{tr}_s[\cdot] \leftarrow \perp$
7: $\mathsf{tr}_s[0] \leftarrow 0$
8: $\mathsf{tr}_r \leftarrow 0$
9: $(s, r, s^{ack}) \leftarrow (0, 0, 0)$
10: $\mathsf{st}_{\mathsf{A}} \leftarrow (s, r, s^{ack}, \mathsf{st}^e_{\mathsf{A}}, \mathsf{st}^d_{\mathsf{A}}, \mathsf{sk}_{\mathsf{A}}, \mathsf{vk}_{\mathsf{A}}, \mathsf{sk}^{eph}_{\mathsf{A}}, \mathsf{vk}^{eph}_{\mathsf{A}}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
11: $\mathsf{st}_{\mathsf{B}} \leftarrow (s, r, s^{ack}, \mathsf{st}^e_{\mathsf{B}}, \mathsf{st}^d_{\mathsf{B}}, \mathsf{sk}_{\mathsf{B}}, \mathsf{vk}_{\mathsf{B}}, \mathsf{sk}^{eph}_{\mathsf{B}}, \mathsf{vk}^{eph}_{\mathsf{B}}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
12: **return** $(\mathsf{st}_{\mathsf{A}}, \mathsf{st}_{\mathsf{B}})$

$\mathsf{MSG.Send}(\mathsf{st}, \mathsf{m})$:
13: $(s, r, s^{ack}, \mathsf{st}^e, \mathsf{st}^d, \mathsf{sk}, \mathsf{vk}, \mathsf{sk}^{eph}, \mathsf{vk}^{eph}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
14: $s \leftarrow s + 1$
15: $(\mathsf{sk}^{eph}_1, \mathsf{vk}^{eph}_1) \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
16: $(\mathsf{sk}^{eph}_2, \mathsf{vk}^{eph}_2) \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
17: $(\mathsf{sk}', \mathsf{vk}') \xleftarrow{\$} \mathsf{DS.KeyGen}(\lambda)$
18: $(\mathsf{st}^d, \mathsf{c}_1) \leftarrow \mathsf{hkuPKE.SendB}(\mathsf{st}^d)$
19: $\mathsf{ad} \leftarrow (s, r, \mathsf{c}_1, \mathsf{vk}', \mathsf{vk}^{eph}_2, \mathsf{tr}_r, \mathsf{tr}_s[s-1])$
20: $(\mathsf{st}^e, \mathsf{c}_2) \leftarrow \mathsf{hkuPKE.SendA}(\mathsf{st}_e, (\mathsf{m}, \mathsf{sk}^{eph}_1), \mathsf{ad})$
21: $\mathsf{c}' \leftarrow (\mathsf{c}_2, \mathsf{ad})$
22: $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{sk}, \mathsf{c}')$
23: $\sigma^{eph} \leftarrow \mathsf{DS.Sign}(\mathsf{sk}^{eph}, \mathsf{c}')$
24: $\mathsf{c} \leftarrow (\mathsf{c}', \sigma, \sigma^{eph})$
25: $\mathsf{vk}^{eph}[s] \leftarrow \mathsf{vk}^{eph}_1$
26: $\mathsf{tr}_s[s] \leftarrow \mathcal{H}(c)$
27: $\mathsf{st} \leftarrow (s, r, s^{ack}, \mathsf{st}^e, \mathsf{st}^d, \mathsf{sk}', \mathsf{vk}, \mathsf{sk}^{eph}_2, \mathsf{vk}^{eph}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
28: **return** $(\mathsf{st}, \mathsf{c})$

$\mathsf{MSG.Rcv}(\mathsf{st}, \mathsf{c})$:
29: $(s, r, s^{ack}, \mathsf{st}^e, \mathsf{st}^d, \mathsf{sk}, \mathsf{vk}, \mathsf{sk}^{eph}, \mathsf{vk}^{eph}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda) \leftarrow \mathsf{st}$
30: $(\mathsf{c}', \sigma, \sigma^{eph}) \leftarrow \mathsf{c}$
31: $(\mathsf{c}_2, \mathsf{ad}) \leftarrow c'$
32: $(s', r', \mathsf{c}_1, \mathsf{vk}', \mathsf{vk}'^{eph}, \mathsf{tr}'_r, \mathsf{tr}'_s) \leftarrow \mathsf{ad}$
33: **req** $s' = r + 1 \wedge \mathsf{tr}'^r = \mathsf{tr}^s[r'] \wedge \mathsf{tr}'^s = \mathsf{tr}^r$
34: **if** $r' > s^{ack}$ **then**
35: $\mathsf{vk}_1 \leftarrow \mathsf{vk}^{eph}[r']$
36: **else**
37: $\mathsf{vk}_1 \leftarrow \mathsf{vk}^{eph}[-1]$
38: $b^{eph} \leftarrow \mathsf{DS.Verify}(\mathsf{vk}_1, \mathsf{c}', \sigma^{eph})$
39: $b \leftarrow \mathsf{DS.Verify}(\mathsf{vk}, \mathsf{c}', \sigma)$
40: **req** $b \wedge b^{eph}$
41: $\mathsf{vk}^{eph}[-1] \leftarrow \mathsf{vk}'^{eph}$
42: $r \leftarrow r + 1$
43: $s^{ack} \leftarrow r'$
44: $(\mathsf{st}^d, \mathsf{m}') \leftarrow \mathsf{hkuPKE.RcvB}(\mathsf{st}^d, \mathsf{c}_2, \mathsf{ad})$
45: $(\mathsf{m}, \mathsf{sk}'^{eph}) \leftarrow \mathsf{m}'$
46: $\mathsf{st}^e \leftarrow \mathsf{hkuPKE.RcvA}(\mathsf{st}^e, \mathsf{c}_1)$
47: $\mathsf{tr}_s[0, \ldots, s^{ack} - 1] \leftarrow \perp$
48: $\mathsf{vk}^{eph}[0, \ldots, s^{ack} - 1] \leftarrow \perp$
49: $\mathsf{tr}_r \leftarrow \mathcal{H}(\mathsf{c})$
50: $\mathsf{st} \leftarrow (s, r, s^{ack}, \mathsf{st}^e, \mathsf{st}^d, \mathsf{sk}, \mathsf{vk}', \mathsf{sk}'^{eph}, \mathsf{vk}^{eph}, \mathsf{tr}_s, \mathsf{tr}_r, \lambda)$
51: **return** $(\mathsf{st}, \mathsf{m})$

Figure 11: Jost, Maurer, and Mularczyk MSG Construction

exposing U after hijacking U will still allow the adversary to get sk. The adversary also needs to get a value of sk^{eph} that U would use.

In [7], Jost, Maurer, Mularczyk make the false claim that if the messages from $\overline{\mathsf{U}}$ directly before and after hijacking U are secure, then post-hijack authentication is perfect. However, this is untrue. Although sk^{eph} is overwritten with each received ciphertext, $\overline{\mathsf{U}}$ only expects U to use the most recent value of sk^{eph}. This means that even if $\overline{\mathsf{U}}$ sends some secure values of sk^{eph}, anytime the sending randomness of $\overline{\mathsf{U}}$ leaks, the adversary gets a value of sk^{eph} that he can use. So, this construction only achieves post-hijack security if U sends a message before being exposed (to delete sk) or if $\overline{\mathsf{U}}$ never leaks sending randomness.

The problem with their strategy is that in their construction receiving a message essentially changes the requirements for signing the next message. So, a secure scheme needs the signing key to be updated. In other words, when receiving a message, the signing key needs to be used and then deleted.

With this notion in mind, the logical construction uses sk^{eph} to sign the transcript hash of each message that a party receives. Whenever a party sends a message, they include the signatures of the transcript hashes. The downside of this scheme is that ciphertexts grow linearly in the number of ciphertexts a party is acknowledging having received. On the plus side, this is amortized constant.

We can consider the security of this new signing scheme, which is clearly at least as secure as the Jost-Maurer-Mularczyk scheme. Now, to impersonate U, the adversary needs to sign with all sk^{eph} keys that $\overline{\mathsf{U}}$ has sent. So, if the adversary hijacks U there is some sk^{eph} that is deleted. The adversary will only be able to break post-hijack authentication if the sk^{eph} that is deleted was created with leaked randomness. Thus, this security scheme achieves the security that Jost, Maurer, and Mularczyk claim.

We will not formally prove the security level of this scheme (neither the original scheme or the fixed scheme). All that needs to be done is to show that the authentication and privacy aspects interact well. The general idea is that if U is not hijacked, then privacy of messages sent by U reduces directly to hkuPKE privacy. If U is hijacked, we don't require any privacy of messages sent by U.

7 Generalized Signal Protocol

No treatment of secure messaging can be complete without discussing the Signal Protocol. This secure messaging protocol is used by Signal, WhatsApp, Facebook Messenger, Skype, and Google Allo [1]. Importantly, Facebook Messenger and Google Allo only use the Signal Protocol if the user enables Secret Conversation mode or Incognito mode. The Signal Protocol has two major advantages over the previous messaging schemes. It is faster than the other schemes because it uses private key cryptography to encrypt and sign messages. Also, Signal allows the parties to receive messages out of order and still be able to decrypt them. This is called immediate decryption or message loss resilience.

When Signal was originally designed, there was a consensus that it was secure. However, at the time it was created, there was no formal analysis regarding its security in the literature. Its design combines different cryptographic building blocks each with different security aspects so that the overall scheme has all of the elements of security. In [1], Alwen, Coretti, and Dodis provide a formal analysis of Signal's structure. They specify each of the building blocks that Signal uses. By generalizing each of the building blocks, Alwen, Coretti, and Dodis create a Generalized Signal Protocol, which follows the same structure as Signal but can be built using different constructions of the building blocks. One benefit of this treatment is that Signal is not post-quantum secure, but by changing the construction of one building block, we can construct a quantum secure variant.

Because Signal was not designed to achieve a specific security notion and was instead designed starting with the building blocks, it's most natural to start studying Signal by looking at the building blocks as defined in [1].

Signal uses three building blocks that we will discuss soon. They are a continuous key agreement CKA, a forward-secure authenticated encryption with associated data FSAEAD, and a cross between a pseudorandom function and a pseudorandom number generator with input called a $\mathsf{PRFPRNG}$.

Essentially, a CKA is used between the parties to generate keys in a synchronous way. This requires the parties to take turns generating new keys. The PRFPRNG incorporates the new keys with the previous ones to create a pseudorandom key. A new FSAEAD instance is used to encrypt consecutive messages sent by the same party for each pseudorandom key.

Before we discuss the building blocks and construction, we will define immediate decryption.

7.1 Immediate Decryption and Message Loss Resilience

As we have seen, immediate decryption is a property that says that messages can be received by U immediately after $\overline{\mathsf{U}}$ sends them. This also means that the messages can be received in any order. This is an extension of correctness. Correctness with immediate decryption idCORR is defined via the game $\mathbf{G}_{\mathsf{MSG.idCORR}}$ in Figure 12. Any adversary should not be able to win the game with any probability. Note that we also require that Rcv is able to determine the index of any ciphertext that it receives.

$\mathbf{G}_{\mathsf{MSG.idCORR}}(\mathcal{A}, \lambda)$:
1: $(\mathsf{st_A}, \mathsf{st_B}) \xleftarrow{\$} \mathsf{Init}(\lambda)$
2: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
3: $\quad \mathsf{c_U}[\cdot] \leftarrow \bot$
4: $\quad \mathsf{m_U}[\cdot] \leftarrow \bot$
5: $\quad received_{\mathsf{U}} \leftarrow \emptyset$
6: $\quad s_{\mathsf{U}} \leftarrow 0$
7: **Run** $\mathcal{A}(\mathsf{st_A}, \mathsf{st_B})$

Oracle $\mathsf{send}(\mathsf{U}, \mathsf{m}, z)$:
8: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
9: $(\mathsf{st_U}, \mathsf{c}) \leftarrow \mathsf{Send}(\mathsf{st_U}, \mathsf{m}; z)$
10: $\mathsf{m_U}[s_{\mathsf{U}}] \leftarrow \mathsf{m}$
11: $\mathsf{c_U}[s_{\mathsf{U}}] \leftarrow \mathsf{c}$
12: $s_{\mathsf{U}} \leftarrow s_{\mathsf{U}} + 1$
13: **return**

Oracle $\mathsf{receive}(\mathsf{U}, i, z)$:
14: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
15: **req** $i < s_{\overline{\mathsf{U}}}$
16: **req** $i \notin received_{\mathsf{U}}$
17: $(\mathsf{st_U}, \mathsf{m}, i') \leftarrow \mathsf{Rcv}(\mathsf{st_U}, \mathsf{c}_{\overline{\mathsf{U}}}[i]; z)$
18: **win if** $i \neq i'$
19: **win if** $\mathsf{m} \neq \mathsf{m}_{\overline{\mathsf{U}}}[i]$
20: $received_{\mathsf{U}} \leftarrow received_{\mathsf{U}} \setminus \{i\}$
21: **return**

Figure 12: MSG with Immediate Decryption Correctness Game

We say that a MSG that is idCORR correct also has message loss resilience. This means that the parties can continue to communicate even if a ciphertext is lost. The previous schemes do not have message loss resilience. If a ciphertext is lost in those schemes, the parties will not be able to communicate. Technically, in the previous schemes, if a message sent by U is lost, $\overline{\mathsf{U}}$ will still be able to message U, but U will not be able to message $\overline{\mathsf{U}}$. For message loss resilience, we also expect that the messaging system will maintain security according to some definition.

7.2 Continuous Key Agreement

As mentioned earlier, building a messaging system is difficult because each party can send multiple messages in a row and because the messages that the parties send can cross on the wire. A continuous key agreement is similar to a messaging scheme without these difficulties. A continuous key agreement is a system so that two parties can agree on a sequence of keys by taking turns sending each other ciphertexts. Also, the continuous key exchange will have

no authentication security. It will only provide indistinguishability guarantees. This is the only part of the Signal Protocol that uses public key cryptography.

A continuous key agreement or CKA consists of three algorithms:

1. State Initialization: $\mathsf{CKA.Init}(\lambda) \xrightarrow{\$} (\mathsf{st_A}, \mathsf{st_B})$
   This randomized algorithm takes a security parameter λ and randomly creates states for the two parties to use.

2. Sending: $\mathsf{CKA.Send}(\mathsf{st}) \xrightarrow{\$} (\mathsf{st}^{new}, \mathsf{c}, \mathsf{k})/(\mathsf{st}, \perp, \perp)$
   This randomized algorithm takes the state of one party and outputs a new state, a ciphertext, and a random key. If st cannot be used for sending, then this returns $(\mathsf{st}, \perp, \perp)$.

3. Receiving: $\mathsf{CKA.Rcv}(\mathsf{st}, \mathsf{c}) \xrightarrow{\$} (\mathsf{st}^{new}, \mathsf{k})/(\mathsf{st}, \perp)$
   This deterministic algorithm takes the state of one party and a ciphertext. It either returns a new state and a key or it returns the old state and $\perp$ signifying that the ciphertext was not valid. If st cannot be used for receiving, then this returns $(\mathsf{st}, \perp)$.

7.2.1 Correctness

In the MSG, a CKA is used for the parties to share keys in a synchronous way. This means that the parties are sending ciphertexts in an alternating pattern. For this reason a state st can only be used for either sending or receiving.

So, the definition of correctness for a CKA only requires the parties to send ciphertexts in an alternating pattern. We require that any time a party calls Send and gets ciphertext c and key k that the other party will receive k when they call Rcv with ciphertext c.

Formally, if for $(\mathsf{st}_0, \mathsf{st}_1) \xleftarrow{\$} \mathsf{CKA.Init}(\lambda)$, then running $(\mathsf{st}_{i+2}, \mathsf{c}_i, \mathsf{k}_i) \xleftarrow{\$} \mathsf{CKA.Send}(\mathsf{st}_i), (\mathsf{st}_{i+1}, \mathsf{k}'_i) \xleftarrow{\$} \mathsf{CKA.Rcv}(\mathsf{st}_{i+1}, \mathsf{c}_i)$ for i from 0 to $n = \mathsf{poly}(\lambda)$ should have $\mathsf{k}_j = \mathsf{k}'_j$ for all $j \in [0, n]$. Note, the even states st_{2a} will all belong to Alice, and the odd states st_{2a+1} will all belong to Bob.

7.2.2 Security

In this section, we discuss security of a CKA. Note that the notion of a CKA is based on the construction of Signal. Because of this, understanding the construction of a CKA helps to understand the notion of security.

The general notion of security for a CKA is that the adversary should not be able to figure out what keys the parties agree on. We would also like this to be true if the adversary is able to control the randomness that the parties use and expose the states of the party. However, we do not let the adversary change the ciphertexts.

Like in previous schemes, there are trivial attacks that the adversary can perform, which are allowed in this definition of security. If the adversary exposes the state of one of the parties, he is able to read any ciphertext that the party can read.

There are two aspects of security that are suboptimal. We will parameterize the security by Δ_{CKA}. Revealing the state of either party should not help the adversary know the keys associated with ciphertexts that were sent and received Δ_{CKA} before the most recent

$\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{CKA.INDEXP}}(\mathcal{A}, \lambda)$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $(\mathsf{st}_\mathsf{A}, \mathsf{st}_\mathsf{B}) \xleftarrow{\$} \mathsf{Init}(\lambda)$
3: $i_{send} \leftarrow 0$
4: $i_{rcv} \leftarrow 0$
5: $\mathsf{c}_{next} \leftarrow \bot$
6: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
7: $\quad challs_\mathsf{U} \leftarrow \emptyset$
8: $\quad exposed_\mathsf{U} \leftarrow \emptyset$
9: $\mathsf{U} \leftarrow \mathsf{A}$
10: **Run** $b' \xleftarrow{\$} \mathcal{A}(\lambda)$
11: **win if** $b' = b$
12: **lose**

Oracle send():
13: **req** $i_{send} = i_{rcv}$
14: $(\mathsf{st}_\mathsf{U}, \mathsf{c}, \mathsf{k}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st}_\mathsf{U})$
15: $\mathsf{c}_{next} \leftarrow \mathsf{c}$
16: $\mathsf{U} \leftarrow \overline{\mathsf{U}}$
17: $i_{send} \leftarrow i_{send} + 1$
18: **return** (c, k)

Oracle send(r):
19: **req** $i_{send} = i_{rcv}$
20: $(\mathsf{c}, \mathsf{k}) \leftarrow \mathsf{send}(; r)$
21: $exposed_\mathsf{A} \leftarrow exposed_\mathsf{A} \cup \{i_{send}\}$
22: $exposed_\mathsf{B} \leftarrow exposed_\mathsf{B} \cup \{i_{send}\}$
23: **return** (c, k)

Oracle chall():
24: **req** $i_{send} = i_{rcv}$
25: **req** $i_{send} - 1 \notin exposed_{\overline{\mathsf{U}}}$
26: $(\mathsf{st}_\mathsf{U}, \mathsf{c}, \mathsf{k}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st}_\mathsf{U})$
27: $\mathsf{c}_{next} \leftarrow \mathsf{c}$
28: $challs_\mathsf{U} \leftarrow challs_\mathsf{U} \cup \{i_{send}\}$
29: $\mathsf{U} \leftarrow \overline{\mathsf{U}}$
30: $i_{send} \leftarrow i_{send} + 1$
31: **if** $b = 0$ **then**
32: $\quad$ **return** (c, k)
33: $\mathsf{k}' \xleftarrow{\$} \mathcal{K}$
34: **return** $(\mathsf{c}, \mathsf{k}')$

Oracle receive():
35: **req** $i_{send} = i_{rcv} + 1$
36: $(\mathsf{st}_\mathsf{U}, \mathsf{k}) \leftarrow \mathsf{Rcv}(\mathsf{st}_\mathsf{U}, \mathsf{c}_{next})$
37: $i_{rcv} \leftarrow i_{rcv} + 1$
38: **return**

Oracle expose(U_{exp}):
39: **req** $\mathsf{U}_{exp} \in \{A, B\}$
40: **req** $i_{rcv} \geq \max(challs_\mathsf{A}) + \Delta_{\mathsf{CKA}}$
41: **req** $i_{rcv} \geq \max(challs_\mathsf{B}) + \Delta_{\mathsf{CKA}}$
42: **if** $\overline{\mathsf{U}} = \mathsf{U}_{exp}$ **then**
43: $\quad exposed_{\mathsf{U}_{exp}} \leftarrow exposed_{\mathsf{U}_{exp}} \cup \{i_{send}\}$
44: **return** $\mathsf{st}_{\mathsf{U}_{exp}}$

Figure 13: CKA Security Game

one. Note that $\Delta_{\mathsf{CKA}} = 0$ is the full security case. Secondly, if the adversary chooses the randomness to be used by U in Send, then we are okay if the adversary is able to figure out the state of U after sending the message and the state of $\overline{\mathsf{U}}$ after receiving the message.

The security notion that we have for CKA is indistinguishability under exposures INDEXP. We formally define this security via the game $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{CKA.INDEXP}}(\mathcal{A}, \lambda)$, which is found in Figure 13. This security definition is parameterized by an integer Δ_{CKA}. We define the advantage of the adversary in this game to be

$$\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{\mathsf{CKA.INDEXP}}(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{CKA.INDEXP}}(\mathcal{A}, \lambda) = 1\right] - 1.$$

We say that a CKA is INDEXP secure for Δ_{CKA} if for any adversary running in $\mathsf{poly}(\lambda)$ time $\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{\mathsf{CKA.INDEXP}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda)$.

In the game, the adversary is given oracles that allow him to cause either party to send or receive ciphertexts, but he needs to follow the alternating pattern of which party is sending messages. When he uses the send oracle, he gets the resulting key. He can also ask for a challenge ciphertext. He is given either the correct key or a random key. He wins if he can figure out whether he is given the correct keys or random keys. He is also allowed to expose the states of the parties.

He is not allowed to create a challenge if there is an exposed state that he can use to trivially decrypt the challenge ciphertext. We also let the adversary choose the randomness used in a send message, but if the adversary chooses the randomness of a message, we call the states of both parties exposed. Note, this means that there are nontrivial attacks that we are forbidding, making the security notion weaker.

Lastly, we forbid the adversary from exposing the state of either party if they have gotten (either through sending or receiving) fewer than Δ_{CKA} keys since getting a challenge key.

Many of the variables used in the game are similar to those in the MSG security game. We also use $i_{send}, i_{rcv}, \mathsf{U}$ as index and to enforce the alternating pattern of the key agreement. We have i_{send}, i_{rcv} counting the number of messages that have been sent and received. Also, U keeps track of who is the next party to act.

7.2.3 Constructions

We will first look at a simple method to use a KEM to construct a CKA with $\Delta_{\mathsf{CKA}} = 0$. Then, we will look at the El Gamal KEM and show a way to make the CKA ciphertexts half as long. We do this by using a KEM ciphertext as a public key. However, this will create a CKA with $\Delta_{\mathsf{CKA}} = 1$. Alwen, Coretti, and Dodis show that a similar trick can be applied to the Learning with Errors (LWE) based Frodo KEM found in [3].

CKA from KEM This is a very simple method of turning a KEM into a CKA. The CKA is initialized by setting Alice's state to be a KEM public key and Bob's state to be the corresponding private key. To send a message, the sender uses their public KEM key to encapsulate a key. They also generate a new KEM key pair. The sender sends the KEM ciphertext and the new public key. The new state of the sender is the generated secret key. The receiver just decapsulates the ciphertext and saves the KEM public key as the new state. This construction is shown in Figure 14. Note, the Send algorithm takes a public key as input, and the Rcv algorithm takes a secret key as input.

$\mathsf{CKA.Init}(\lambda)$:
1: $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KEM.KeyGen}(\lambda)$
2: **return** $(\mathsf{pk}, \mathsf{sk})$

$\mathsf{CKA.Send}(\mathsf{pk})$:
3: $(\mathsf{c}', \mathsf{k}) \xleftarrow{\$} \mathsf{KEM.Enc}(\mathsf{pk})$
4: $(\mathsf{pk}', \mathsf{sk}') \xleftarrow{\$} \mathsf{KEM.KeyGen}(\lambda)$
5: $\mathsf{c} \leftarrow (\mathsf{c}', \mathsf{pk}')$
6: **return** $(\mathsf{sk}', \mathsf{c}, \mathsf{k})$

$\mathsf{CKA.Rcv}(\mathsf{sk}, \mathsf{c})$:
7: $(\mathsf{c}', \mathsf{pk}) \leftarrow \mathsf{c}$
8: $\mathsf{k} \leftarrow \mathsf{KEM.Dec}(\mathsf{sk}, \mathsf{c}')$
9: **if** $\mathsf{k} = \perp$ **then**
10: **return** $(\mathsf{sk}, \perp)$
11: **return** $(\mathsf{pk}, \mathsf{k})$

Figure 14: Simple CKA Construction from KEM

The proof that this CKA is INDEXP secure for $\Delta_{\mathsf{CKA}} = 0$ reduces almost directly to the KEM indistinguishability security. Winning $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{CKA.INDEXP}}$ is essentially distinguishing the true encapsulated key from randomness given a KEM public key and the ciphertext. The rest of the proof follows from a hybrid argument.

7.2.4 Efficient Communication CKA for El Gamal KEM

Recall that the El Gamal KEM is defined over some group $\mathbb{G} = \langle g \rangle$ for some group element g with $|\mathbb{G}| = p$. A public and secret key pair is (g^x, x). Encapsulation takes the public key

$h = g^x$ and random $r \in \mathbb{Z}_p$ and outputs key h^r and ciphertext g^r. Decapsulation takes secret key x and ciphertext f and outputs key f^x.

Instead of sending the KEM ciphertext and a new KEM public key, the Signal protocol makes the new public key the same as the ciphertext. This is possible because the sender knows the exponent of the ciphertext. This makes the ciphertext half as long. See Figure 15 for the construction.

CKA.Init(λ):
1: pick $\mathbb{G} = \langle g \rangle$ of size $p \approx 2^\lambda$
2: $r \xleftarrow{\$} \mathbb{Z}_p$
3: $\mathsf{pk} \leftarrow g^r$
4: $\mathsf{sk} \leftarrow r$
5: **return** $(\mathsf{pk}, \mathsf{sk})$

CKA.Send(pk):
6: $h \leftarrow \mathsf{pk}$
7: $r \xleftarrow{\$} \mathbb{Z}_p$
8: $\mathsf{sk} \leftarrow r$
9: $\mathsf{c} \leftarrow g^r$
10: $\mathsf{k} \leftarrow h^r$
11: **return** $(\mathsf{sk}, \mathsf{c}, \mathsf{k})$

CKA.Rcv(sk, c):
12: $x \leftarrow \mathsf{sk}$
13: $\mathsf{pk} \leftarrow \mathsf{c}$
14: $\mathsf{k} \leftarrow \mathsf{c}^x$
15: **return** $(\mathsf{pk}, \mathsf{k})$

Figure 15: Communication Efficient CKA construction from El Gamal KEM

The downside of this change is that the sender must keep the randomness r used to encapsulate the message. He will get rid of this information after receiving the next message. For this reason, this CKA construction is INDEXP secure for $\Delta_{\mathsf{CKA}} = 1$ if the El Gamal KEM is secure. The El Gamal KEM is secure if the Decisional Diffie-Hellman problem is hard over the group $\mathbb{G}$.

7.3 Forward Secure Authenticated Encryption with Associated Data

As mentioned earlier, one of the challenges of building a messaging system is allowing communication in both directions. A forward secure authenticated encryption with associated data is essentially a unidirectional messaging scheme with a weaker definition of security. This must also satisfy immediate decryption.

In the Signal construction, whenever a party U creates a key with the CKA, a new key will be created using the PRFPRNG. Then, this key will be used to initialize a forward secure authenticated encryption with associated data so that U can send encrypted messages to $\overline{\mathsf{U}}$. The forward secure authenticated encryption with associated data uses no randomness except for the key generated by the PRFPRNG.

A forward secure authenticated encryption with associated data or FSAEAD consists of three deterministic algorithms:

1. State Initialization: $\mathsf{FSAEAD.Init}(\mathsf{k}) \to (\mathsf{st_S}, \mathsf{st_R})$
   This deterministic algorithm takes a key k and creates a state for the sender $\mathsf{st_S}$ and a state for the receiver $\mathsf{st_R}$.

2. Sending: $\mathsf{FSAEAD.Send}(\mathsf{st_S}, \mathsf{m}, \mathsf{ad}) \to (\mathsf{st}_\mathsf{S}^{new}, \mathsf{c})$
   This deterministic algorithm takes the sender's state, a message, and associated data and outputs a new state and a ciphertext.

3. Receiving: $\mathsf{FSAEAD.Rcv}(\mathsf{st_R}, \mathsf{c}, \mathsf{ad}) \to (\mathsf{st}_\mathsf{R}^{new}, \mathsf{m}, i)/(\mathsf{st}, \perp, \perp)$
   This deterministic algorithm takes the receiver's state, a ciphertext, and associated data. It either returns a new state, a message, and an index or it returns the old state followed by two $\perp$ signifying that the ciphertext was not valid. The index should be the index of c in the list of ciphertexts generated by the sender.

7.3.1 Correctness

In a FSAEAD, all ciphertexts sent should be decrypted to the correct message. Also, the FSAEAD should satisfy immediate decryption, meaning the receiver should be able to receive the messages in any order and know what that order is.

Formally, if $(\mathsf{st_S}, \mathsf{st_R}) \leftarrow \mathsf{FSAEAD.Init}(\mathsf{k})$ and we run $(\mathsf{st_S}, \mathsf{c}_j) \leftarrow \mathsf{FSAEAD.Send}(\mathsf{st_S}, \mathsf{m}_j, \mathsf{ad}_j)$ for $j = 0$ to $n = \mathsf{poly}(|k|)$, then, for any permutation π on 0 to n, running $(\mathsf{st_R}, \mathsf{m}'_{\pi(j)}, i_{\pi(j)}) \leftarrow \mathsf{FSAEAD.Rcv}(\mathsf{st_R}, \mathsf{c}_{\pi(j)}, \mathsf{ad}_{\pi(j)})$ for $i = 0$ to $n = \mathsf{poly}(|k|)$ should make $i_j = j$ and $\mathsf{m}'_j = \mathsf{m}_j$ for all j. This should hold for any values of m_j and ad_j.

7.3.2 Security

There are two aspects of security that we expect from a FSAEAD. First, is that the adversary should not be able to forge a message or break the privacy of the scheme. We would like this even if the adversary can expose the state of either party. However, our notion of security allows for trivial attacks and non-trivial attacks. We allow the adversary to break authentication and privacy of any future messages given the state of the sender or the receiver.

We formally define SEC security for a FSAEAD via the game $\mathbf{G}_{\mathsf{FSAEAD.SEC}}(\mathcal{A}, \lambda)$, which is found in Figure 16. We define the advantage of the adversary in this game to be

$$\mathsf{Adv}_{\mathsf{FSAEAD.SEC}}(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}_{\mathsf{FSAEAD.SEC}}(\mathcal{A}, \lambda) = 1\right] - 1.$$

We say that a FSAEAD is SEC secure if for any adversary running in $\mathsf{poly}(\lambda)$ time $\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{\mathsf{FSAEAD.SEC}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda)$. Note, the key space is chosen based on λ.

In the game, the adversary is given oracles that allow him to cause the sender to send a message and the receiver to receive any message. When he uses the send oracle, he gets the resulting key. He can also ask for a challenge ciphertext, which encrypts one of two messages based on the challenge bit b. He wins if he can figure out b. He is also allowed to expose the states of the parties.

He is not allowed to create a challenge if he has exposed either party. He is also not allowed to expose the receiver if the receiver has yet to receive all of the challenge ciphertexts. Note, there are nontrivial attacks that we are forbidding, making the security notion weaker.

Many of the variables used in the game are similar to previous security games. We also use i to count the number of sent messages. We have $exposed_\mathsf{R}$ be a boolean for whether the receiver has been exposed. We use $exposed_\mathsf{S}$ to be the number of messages the sender sent before being exposed for the first time.

$\mathbf{G}_{\mathsf{FSAEAD.SEC}}(\mathcal{A}, \lambda)$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $k \leftarrow \mathcal{K}$
3: $(\mathsf{st_S}, \mathsf{st_R}) \xleftarrow{\$} \mathsf{Init}(k)$
4: $\mathsf{c}[\cdot] \leftarrow \perp$
5: $\mathsf{m}[\cdot] \leftarrow \perp$
6: $\mathsf{ad}[\cdot] \leftarrow \perp$
7: $i \leftarrow 0$
8: $challs \leftarrow \emptyset$
9: $received \leftarrow \emptyset$
10: $exposed_{\mathsf{S}} \leftarrow \infty$
11: $exposed_{\mathsf{R}} \leftarrow \mathsf{False}$
12: **Run** $b' \xleftarrow{\$} \mathcal{A}(\lambda)$
13: **win if** $b' = b$
14: **lose**

Oracle $\mathsf{send}(\mathsf{m}, \mathsf{ad})$:
15: $(\mathsf{st_S}, \mathsf{c}) \leftarrow \mathsf{Send}(\mathsf{st_S}, \mathsf{m}, \mathsf{ad})$
16: $\mathsf{c}[i] \leftarrow \mathsf{c}$
17: $\mathsf{m}[i] \leftarrow \mathsf{m}$
18: $\mathsf{ad}[i] \leftarrow \mathsf{ad}$
19: $i \leftarrow i + 1$
20: **return** c

Oracle $\mathsf{chall}(\mathsf{m}_0, \mathsf{m}_1, \mathsf{ad})$:
21: **req** $exposed_{\mathsf{S}} = \infty$
22: **req** $\neg exposed_{\mathsf{R}}$
23: **req** $|\mathsf{m}_0| = |\mathsf{m}_1|$
24: $challs \leftarrow challs \cup \{i\}$
25: $\mathsf{c} \leftarrow \mathsf{send}(\mathsf{m}_b, \mathsf{ad})$
26: **return** c

Oracle $\mathsf{receive}(\mathsf{U}, \mathsf{c})$:
27: **req** $\neg exposed_{\mathsf{R}}$
28: $(\mathsf{st_R}, \mathsf{m}, i') \leftarrow \mathsf{Rcv}(\mathsf{st_R}, \mathsf{c}, \mathsf{ad})$
29: **req** $\mathsf{m} \neq \perp$
30: **win if** $i' \in received$
31: **win if** $exposed_{\mathsf{S}} > i' \wedge (\mathsf{m}, \mathsf{c}, \mathsf{ad}) \neq (\mathsf{m}[i'], \mathsf{c}[i'], \mathsf{ad}[i'])$
32: $received \leftarrow received \cup \{i'\}$
33: **if** $i' \in challs$ **then**
34: **return** $\perp$
35: **return** m

Oracle $\mathsf{exposeR}()$:
36: **req** $challs \subset received$
37: $exposed_{\mathsf{R}} \leftarrow \mathsf{True}$
38: **return** $\mathsf{st_R}$

Oracle $\mathsf{exposeS}(\mathsf{U})$:
39: $exposed_{\mathsf{S}} \leftarrow \min(exposed_{\mathsf{S}}, i)$
40: **return** $\mathsf{st_U}$

Figure 16: FSAEAD Security Game

7.3.3 Construction

Signal's construction of a FSAEAD is very simple. It uses a PRG and an AEAD (authenticated encryption with associated data). The sender's state and the receiver's state are initialized to the same key. To send a message, the sender uses the PRG to get an AEAD key and another key. The sender uses the AEAD key to send a message to the receiver, and the state becomes the other key. The sender also sends the index of the message. The receiver can use his PRG to calculate the same key that the sender used and can then decrypt the message. He then deletes the AEAD key. We can see that the receiver can decrypt the messages in any order. The scheme can be found in Figure 17[5]

It is not difficult to see that this scheme achieves SEC security for FSAEAD. In fact, the definition of FSAEAD.SEC is built to match the security achieved by this scheme. First, we can reduce $\mathbf{G}_{\mathsf{FSAEAD.SEC}}$ to a version where PRG is random (so long as the adversary doesn't know the input). Distinguishing these versions would break the PRG. An adversary winning this game can be turned into an adversary winning the AEAD game by a hybrid argument.

[5]Care must be taken when implementing this. The runtime of receiving is linear in the highest index of a message. If an adversary can cause someone to receive a message where the index is very high, this would cause the receiver to spend a lot of time before he can check whether the ciphertext is valid. A practical implementation of this might assume that the number of messages skipped will be small, or a practical implementation of this might also include a signature of the message using a signing key that does not change, so that the adversary cannot make the receiver try to receive a message with high index.

FSAEAD.Init(k):
1: $i \leftarrow 0$
2: $\mathsf{k}_{\mathsf{AEAD}}[\cdot] \leftarrow \perp$
3: $\mathsf{st}_{\mathsf{S}} \leftarrow (\mathsf{k}, i)$
4: $\mathsf{st}_{\mathsf{R}} \leftarrow (\mathsf{k}, \mathsf{k}_{\mathsf{AEAD}}, i)$
5: **return** $(\mathsf{st}_{\mathsf{S}}, \mathsf{st}_{\mathsf{R}})$

FSAEAD.Send($\mathsf{st}_{\mathsf{S}}, \mathsf{m}, \mathsf{ad}$):
6: $(\mathsf{k}, i) \leftarrow \mathsf{st}_{\mathsf{S}}$
7: $(\mathsf{k}_{\mathsf{AEAD}}, \mathsf{k}') \leftarrow \mathsf{PRG}(\mathsf{k})$
8: $\mathsf{c}' \leftarrow \mathsf{AEAD.Enc}(\mathsf{k}_{\mathsf{AEAD}}, \mathsf{m}, \mathsf{ad})$
9: $\mathsf{c} \leftarrow (\mathsf{c}', i)$
10: $\mathsf{st}_{\mathsf{S}} \leftarrow (\mathsf{k}', i+1)$
11: **return** $(\mathsf{st}_{\mathsf{S}}, \mathsf{c})$

FSAEAD.Rcv($\mathsf{st}_{\mathsf{R}}, \mathsf{c}, \mathsf{ad}$):
12: $(\mathsf{k}, \mathsf{k}_{\mathsf{AEAD}}, i) \leftarrow \mathsf{st}_{\mathsf{R}}$
13: $(\mathsf{c}', j) \leftarrow \mathsf{c}$
14: **req** $j \geq i \vee \mathsf{k}_{\mathsf{AEAD}}[j] \neq \perp$
15: **while** $j \geq i$ **do**
16: $\quad (\mathsf{k}_{\mathsf{AEAD}}[i], \mathsf{k}) \leftarrow \mathsf{PRG}(\mathsf{k})$
17: $\quad i \leftarrow i + 1$
18: $\mathsf{m} \leftarrow \mathsf{AEAD.Dec}(\mathsf{k}_{\mathsf{AEAD}}[j], \mathsf{c}, \mathsf{ad})$
19: **req** $\mathsf{m} \neq \perp$
20: $\mathsf{k}_{\mathsf{AEAD}}[j] \leftarrow \perp$
21: $\mathsf{st}_{\mathsf{R}} \leftarrow (\mathsf{k}, \mathsf{k}_{\mathsf{AEAD}}, i)$
22: **return** $(\mathsf{st}_{\mathsf{R}}, \mathsf{m}, i)$

Figure 17: FSAEAD Construction of Signal

7.4 PRFPRNG

A PRFPRNG is a cross between a pseudorandom function and a pseudorandom number generator with input. This will be made clear in the security section. This is used to turn the randomness from the CKA into pseudorandomness that can be securely used by the FSAEAD.

A PRFPRNG consists of two deterministic algorithms:

1. State Initialization: $\mathsf{PRFPRNG.Init}(\mathsf{k}) \to \mathsf{st}$
   This deterministic algorithm takes a key k and creates a state for the PRFPRNG.

2. Update: $\mathsf{PRFPRNG.Upd}(\mathsf{st}, \mathsf{k}) \to (\mathsf{st}^{new}, r)$
   This deterministic algorithm takes the state and an input k. It outputs a new state and some pseudorandomness r.

Just like a pseudorandom function or a pseudorandom number generator with input, there is no notion of correctness.

7.4.1 Security

Here, we will be specific in how a PRFPRNG is meant to be a cross between a pseudorandom function and a pseudorandom number generator. If st is random and has high entropy, then $\mathsf{Upd}(\mathsf{st}, \cdot)$ should behave like a pseudorandom function. Like a pseudorandom number generator with input, if k is random and has high entropy, $\mathsf{Upd}(\mathsf{st}, \mathsf{k})$ should be indistinguishable from random for any st.

7.4.2 Construction

Note, a random function is secure in this way. Signal uses a function called HKDF, which is also assumed to be secure in this way. The function HKDF was introduced in [9] as a key derivation function that does not rely too heavily on cryptographic hash functions.

Another possible construction involves a pseudorandom permutation Π and a pseudorandomness generator G. The output of $\mathsf{Upd}(\mathsf{st}, \mathsf{k})$ is $G(\Pi_{\mathsf{st}}(\mathsf{k}))$. We see that for a random state st, $\Pi_{\mathsf{st}}(\mathsf{k})$ is indistinguishable from random, so $G(\Pi_{\mathsf{st}}(\mathsf{k}))$ is also indistinguishable from random. Similarly, for any st and random k, $\Pi_{\mathsf{st}}(\mathsf{k})$ is indistinguishable from random, so $G(\Pi_{\mathsf{st}}(\mathsf{k}))$ is too.

7.5 Construction

The foundation of the generalized Signal protocol is the CKA. This allows the two parties to privately agree on keys in an alternating way. Both parties can then use a PRFPRNG and a shared state to generate a new key, which is then used for unidirectional messaging from one party to another. The formal implementation is found in Figure 18.

Every key generated by the CKA starts what we will call an epoch. Both parties will keep track of what epoch they are in using the variable t, which is the number of CKA keys they have seen. Both parties start in epoch 0. Whenever Alice sends her first message, she will move to epoch 1 and run CKA.Send. If U is in epoch t and receives a message from $\overline{\mathsf{U}}$ where $\overline{\mathsf{U}}$ is in epoch $t+1$, then U will run CKA.Rcv and move to epoch $t+1$. The next time that U sends a message, she will run CKA.Send and move to epoch $t+2$. So, Alice will only send messages during odd epochs, and Bob will only send messages during even epochs. Also, note that Alice and Bob can be in different epochs, but the epochs can differ by at most one.

Whenever U starts a new epoch, they will store $\mathsf{c}^{\mathsf{CKA}}$, the ciphertext outputted by CKA.Send, index of the epoch t, and the number of messages they sent in epoch $t-2$, which was the party's last sending epoch. They will store this as c_{epoch}, which they will include in all of their messages this epoch. This is because they need to send this in each message until U knows that $\overline{\mathsf{U}}$ has received c_{epoch}. At this point, $\overline{\mathsf{U}}$ has already started a new epoch, and it is time for U to start a new epoch. In the algorithm, if c_{epoch} is $\perp$, it signifies that it is time for U to start a new epoch.

The parties also store st^p, which is the state of the PRFPRNG. When the parties move to the next epoch, they run PRFPRNG.Upd on the key generated by the CKA to get a new key. This key is then used to initialize a FSAEAD. Each party will use their most recent initialized FSAEAD to send their messages, whether or not they just started a new epoch. Also, they will use c_{epoch} as the associated data.

When U is receiving a ciphertext, the ciphertext will tell U which epoch it is from. Then, U will use the FSAEAD state from that epoch to decrypt the message. Once U has received all the messages from an epoch, U will delete the FSAEAD state from that epoch. To do this, U stores $r[\cdot], r_{max}[\cdot]$, which are the number of messages received in an epoch and the number of messages in the epoch. The latter is found in c'_{epoch}, which is sent by $\overline{\mathsf{U}}$. The receiving algorithm outputs the epoch of the message, its index in the epoch and the message itself. This can easily be turned into the overall index of the message.

The initialization algorithm initializes the CKA, PRFPRNG, and the FSAEAD of the first epoch. We use $\mathcal{K}^p, \mathcal{K}^f$ to be the set of possible keys for the PRFPRNG and FSAEAD, which are based on λ.

MSG.Init(λ):
1: $\mathsf{st}_{\mathsf{S}}^{\mathsf{A}} \leftarrow \bot$
2: $\mathsf{st}_{\mathsf{R}}^{\mathsf{A}}[\cdot] \leftarrow \bot$
3: $\mathsf{st}_{\mathsf{R}}^{\mathsf{B}}[\cdot] \leftarrow \bot$
4: $r_{max}[\cdot] \leftarrow \infty$
5: $r[\cdot] \leftarrow 0$
6: $t \leftarrow 0$
7: $s \leftarrow 0$
8: $\mathsf{c}_{epoch}^{\mathsf{A}} \leftarrow \bot$
9: $\mathsf{c}_{epoch}^{\mathsf{B}} \leftarrow (\bot, 0, 0)$
10: $(\mathsf{st}_{\mathsf{A}}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{B}}^{\mathsf{CKA}}) \leftarrow \mathsf{CKA.Init}(\lambda)$
11: $\mathsf{k}^p \xleftarrow{\$} \mathcal{K}^p$
12: $\mathsf{st}^p \leftarrow \mathsf{PRFPRNG.Init}(\mathsf{k}^p)$
13: $\mathsf{k}^f \leftarrow \mathcal{K}^f$
14: $(\mathsf{st}_{\mathsf{S}}^{\mathsf{B}}, \mathsf{st}_{\mathsf{R}}^{\mathsf{A}}[0]) \leftarrow \mathsf{FSAEAD.Init}(\mathsf{k}^f)$
15: $\mathsf{st}_{\mathsf{A}} \leftarrow (\mathsf{st}_{\mathsf{A}}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{S}}^{\mathsf{A}}, \mathsf{st}_{\mathsf{R}}^{\mathsf{A}}, \mathsf{st}^p, t, s, r, r_{max}, \mathsf{c}_{epoch}^{\mathsf{A}})$
16: $\mathsf{st}_{\mathsf{B}} \leftarrow (\mathsf{st}_{\mathsf{B}}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{S}}^{\mathsf{B}}, \mathsf{st}_{\mathsf{R}}^{\mathsf{B}}, \mathsf{st}^p, t, s, r, r_{max}, \mathsf{c}_{epoch}^{\mathsf{B}})$
17: **return** $(\mathsf{st}_{\mathsf{A}}, \mathsf{st}_{\mathsf{B}})$

MSG.Send(st, m):
18: $(\mathsf{st}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{S}}, \mathsf{st}_{\mathsf{R}}, \mathsf{st}^p, t, s, r, r_{max}, \mathsf{c}_{epoch}) \leftarrow \mathsf{st}$
19: **if** $\mathsf{c}_{epoch} = \bot$ **then**
20: $\quad t \leftarrow t + 1$
21: $\quad (\mathsf{st}^{\mathsf{CKA}}, \mathsf{c}^{\mathsf{CKA}}, \mathsf{k}^{\mathsf{CKA}}) \xleftarrow{\$} \mathsf{CKA.Send}(\mathsf{st}^{\mathsf{CKA}})$
22: $\quad \mathsf{c}_{epoch} \leftarrow (\mathsf{c}^{\mathsf{CKA}}, t, s)$
23: $\quad s \leftarrow 0$
24: $\quad (\mathsf{st}^p, \mathsf{k}) \leftarrow \mathsf{PRFPRNG.Upd}(\mathsf{st}^p, \mathsf{k}^{\mathsf{CKA}})$
25: $\quad \mathsf{st}_{\mathsf{S}} \leftarrow \mathsf{FSAEAD.Init}(\mathsf{k})$
26: $(\mathsf{st}_{\mathsf{S}}, \mathsf{c}') \leftarrow \mathsf{FSAEAD.Send}(\mathsf{st}_{\mathsf{S}}, \mathsf{m}, \mathsf{c}_{epoch})$
27: $\mathsf{c} \leftarrow (\mathsf{c}', \mathsf{c}_{epoch})$
28: $s \leftarrow s + 1$
29: $\mathsf{st} \leftarrow (\mathsf{st}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{S}}, \mathsf{st}_{\mathsf{R}}, \mathsf{st}^p, t, s, r, r_{max}, \mathsf{c}_{epoch})$
30: **return** $(\mathsf{st}, \mathsf{c})$

MSG.Rcv(st, c):
31: $(\mathsf{st}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{S}}, \mathsf{st}_{\mathsf{R}}, \mathsf{st}^p, t, s, r, r_{max}, \mathsf{c}_{epoch}) \leftarrow \mathsf{st}$
32: $(\mathsf{c}', \mathsf{c}'_{epoch}) \leftarrow \mathsf{c}$
33: $(\mathsf{c}^{\mathsf{CKA}}, t', s') \leftarrow \mathsf{c}'_{epoch}$
34: **req** $\mathsf{st}_{\mathsf{R}}[t'] \neq \bot \vee (t' = t + 1 \wedge \mathsf{c}_{epoch} \neq \bot)$
35: **if** $t' = t + 1$ **then**
36: $\quad t \leftarrow t + 1$
37: $\quad r_{max}[t-2] \leftarrow s'$
38: $\quad (\mathsf{st}^{\mathsf{CKA}}, \mathsf{k}) \xleftarrow{\$} \mathsf{CKA.Rcv}(\mathsf{st}^{\mathsf{CKA}}, \mathsf{c}^{\mathsf{CKA}})$
39: $\quad (\mathsf{st}^p, \mathsf{k}) \leftarrow \mathsf{PRFPRNG.Upd}(\mathsf{st}^p, \mathsf{k}^{\mathsf{CKA}})$
40: $\quad (\cdot, \mathsf{st}_{\mathsf{R}}[t]) \leftarrow \mathsf{FSAEAD.Init}(\mathsf{k})$
41: $\quad \mathsf{st}_{\mathsf{S}} \leftarrow \bot$
42: $\quad \mathsf{c}_{epoch} \leftarrow \bot$
43: $(\mathsf{st}_{\mathsf{R}}[t'], \mathsf{m}, i) \leftarrow \mathsf{FSAEAD.Rcv}(\mathsf{st}_{\mathsf{R}}[t'], \mathsf{c}', \mathsf{c}'_{epoch})$
44: **req** $\mathsf{m} \neq \bot$
45: **req** $i \leq r_{max}[t']$
46: $r[t'] \leftarrow r[t'] + 1$
47: **if** $r[t'] = r_{max}[t']$ **then**
48: $\quad \mathsf{st}_{\mathsf{R}}[t'] \leftarrow \bot$
49: $\quad r[t'] \leftarrow \bot$
50: $\quad r_{max}[t'] \leftarrow \bot$
51: $\mathsf{st} \leftarrow (\mathsf{st}^{\mathsf{CKA}}, \mathsf{st}_{\mathsf{S}}, \mathsf{st}_{\mathsf{R}}, \mathsf{st}^p, t, s, r, r_{max}, \mathsf{c}_{epoch})$
52: **return** (m, t', i)

Figure 18: Generalized Signal Construction

7.5.1 Differences with Signal Protocol

This is essentially how Signal works. There are minor differences in how Signal indexes the epochs and FSAEAD keys. Also, the Signal Protocol performs some computations earlier than in Figure 18. Also, Signal Protocol initializes the keys slightly differently because in practice there is no shared randomness when initializing the messaging scheme. Overall, there aren't any major differences.

7.6 Security of Generalized Signal Protocol

In this section, we will discuss the notion of security for the Generalized Signal Protocol. We use the term idSEC for this security to emphasize that the security is weaker due to achieving immediate decryption. However, we do not claim that this is optimal. In fact, this is not optimal. Also, note that because immediate decryption allows for receiving messages in any order, the security notion gets complicated.

Security of Generalized Signal has essentially two differences from optimal security SEC of a MSG. First, it takes slightly longer for Generalized Signal to heal from state exposure. Secondly (and more importantly), exposing the state of U can allow an adversary to read ciphertexts sent by U and send messages to U. Optimal security only allows the adversary to read ciphertexts sent by $\overline{\mathsf{U}}$ and send messages to $\overline{\mathsf{U}}$.

We formally define idSEC parametrized by $\Delta_{\mathsf{CKA}} \in \{0,1\}$ via the complicated game $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}(\mathcal{A},\lambda)$, which is found in Figure 19. We define the advantage of the adversary in this game to be

$$\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}(\mathcal{A},\lambda) = 2\mathsf{Pr}\left[\mathbf{G}_{\mathsf{MSG.SEC}}(\mathcal{A},\lambda)=1\right]-1.$$

We say that a messaging system is idSEC secure for Δ_{CKA} if for any adversary running in $\mathsf{poly}(\lambda)$ time $\mathsf{Adv}_{\mathsf{MSG.idSEC}}(\mathcal{A},\lambda) = \mathsf{negl}(\lambda)$.

The game $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}(\mathcal{A},\lambda)$ is quite complicated and is mainly provided for completeness's sake. A slightly weaker, but far simpler security notion for this construction is defined in [1].

The game has the same oracles as the other MSG security games. Also, the adversary can win the idSEC game in the same ways[6]. He can guess the challenge bit b or he can break authenticity when he shouldn't be able to. The difference between this game and $\mathbf{G}_{\mathsf{MSG.SEC}}$ is when we allow the adversary to be able to break authenticity and privacy. Almost all of these differences can be found in the expose oracle.

Now, we will discuss *comp*, which is the set of epoch index pairs for which the adversary is able to decrypt the corresponding ciphertext. The adversary is not allowed to issue a challenge for any epoch and index in the set. Also, the adversary is not allowed to expose a state that would compromise any already issued challenge. A state is only compromised during the expose oracle. When U is exposed, any ciphertexts that $\overline{\mathsf{U}}$ has sent and U has not received are compromised. Also, all future (meaning U has not seen them) ciphertexts from the current epoch are compromised. If U is the sender in the current epoch, all ciphertexts in the next epoch are compromised. If U is the sender in this epoch and $\Delta_{\mathsf{CKA}} = 1$, and the previous epoch was compromised at any point, then all ciphertexts in the current epoch are compromised.

Now, we will discuss *repl*, which is the set of epoch index pairs for which the adversary can impersonate the sender. The adversary does not win if he is able to impersonate the sender for any epoch index pair in *repl*. Note that $comp \subset repl$. All messages from the next epoch are also replaceable. By replaceable, we mean that the adversary can make the receiver accept a different message. If U is the sender in the current epoch t_{U}, all ciphertexts in epoch $t_{\mathsf{U}}+2$ are compromised. If U is the sender in this epoch and $\Delta_{\mathsf{CKA}} = 1$, and the previous epoch was compromised at any point, then all ciphertexts (that haven't been

[6]The adversary can also win by getting a party to accept multiple messages corresponding to the same epoch and index. This can be easily avoided when constructing the MSG.

$\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}(\mathcal{A}, \lambda)$:
1: $b \xleftarrow{\$} \{0,1\}$
2: $(\mathsf{st_A}, \mathsf{st_B}) \xleftarrow{\$} \mathsf{Init}(\lambda)$
3: $comp \leftarrow \emptyset$
4: $repl \leftarrow \emptyset$
5: $challs \leftarrow \emptyset$
6: **for** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$ **do**
7: $\quad t_{\mathsf{U}} \leftarrow 0$
8: $\quad i_{\mathsf{U}} \leftarrow 0$
9: $\quad \mathsf{c_U}[\cdot, \cdot] \leftarrow \perp$
10: $\quad sent_{\mathsf{U}} \leftarrow \emptyset$
11: $\quad received_{\mathsf{U}} \leftarrow \emptyset$
12: $\quad hijacked_{\mathsf{U}} \leftarrow \mathsf{False}$
13: **Run** $b' \xleftarrow{\$} \mathcal{A}(\lambda)$
14: **win if** $b' = b$
15: **lose**

Oracle $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$:
16: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
17: **req** $|\mathsf{m}_0| = |\mathsf{m}_1|$
18: **req** $\neg hijacked_{\mathsf{U}}$
19: **if** $i_{\mathsf{U}} = 0 \wedge \neg(t_{\mathsf{U}} = 0 \wedge \mathsf{U} = \mathsf{B})$ **then**
20: $\quad$ **req** $(t_{\mathsf{U}} + 1, i_{\mathsf{U}}) \notin comp$
21: $\quad t_{\mathsf{U}} \leftarrow t_{\mathsf{U}} + 1$
22: **else**
23: $\quad$ **req** $(t_{\mathsf{U}}, i_{\mathsf{U}}) \notin comp$
24: $challs \leftarrow challs \cup \{t_{\mathsf{U}}, i_{\mathsf{U}}\}$
25: $(\mathsf{st_U}, \mathsf{c}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st_U}, \mathsf{m}_b)$
26: $\mathsf{c_U}[t_{\mathsf{U}}, i_{\mathsf{U}}] \leftarrow \mathsf{c}$
27: $sent_{\mathsf{U}} \leftarrow sent_{\mathsf{U}} \cup \{(t_{\mathsf{U}}, i_{\mathsf{U}})\}$
28: $i_{\mathsf{U}} \leftarrow i_{\mathsf{U}} + 1$
29: **return** c

Oracle $\mathsf{send}(\mathsf{U}, \mathsf{m})$:
30: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
31: **if** $i_{\mathsf{U}} = 0 \wedge \neg(t_{\mathsf{U}} = 0 \wedge \mathsf{U} = \mathsf{B})$ **then**
32: $\quad t_{\mathsf{U}} \leftarrow t_{\mathsf{U}} + 1$
33: $(\mathsf{st_U}, \mathsf{c}) \xleftarrow{\$} \mathsf{Send}(\mathsf{st_U}, \mathsf{m})$
34: $\mathsf{c_U}[t_{\mathsf{U}}, i_{\mathsf{U}}] \leftarrow \mathsf{c}$
35: $sent_{\mathsf{U}} \leftarrow sent_{\mathsf{U}} \cup \{(t_{\mathsf{U}}, i_{\mathsf{U}})\}$
36: $i_{\mathsf{U}} \leftarrow i_{\mathsf{U}} + 1$
37: **return** c

Oracle $\mathsf{receive}(\mathsf{U}, \mathsf{c})$:
38: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
39: $(\mathsf{st_U}, \mathsf{m}, t, i) \xleftarrow{\$} \mathsf{Rcv}(\mathsf{st_U}, \mathsf{c})$
40: **req** $\mathsf{m} \neq \perp$
41: **win if** $(t, i) \in received_{\mathsf{U}}$
42: $received_{\mathsf{U}} \leftarrow received_{\mathsf{U}} \cup \{t, i\}$
43: **if** $(t, i) \in challs$ **then**
44: $\quad \mathsf{m} \leftarrow \perp$
45: **if** $\mathsf{c} \neq \mathsf{c}_{\overline{\mathsf{U}}}[t, i]$ **then**
46: $\quad$ **win if** $(t, i) \notin repl \wedge \neg hijacked_{\mathsf{U}}$
47: $\quad hijacked_{\mathsf{U}} \leftarrow (t > t_{\mathsf{U}})$
48: **else if** $t = t_{\mathsf{U}} + 1$ **then**
49: $\quad repl \leftarrow repl \setminus \{(t_{\mathsf{U}}, j) : c[t_{\mathsf{U}}, j] = \perp\}$
50: $\quad$ **if** $(t, i) \notin comp$ **then**
51: $\quad\quad repl \leftarrow repl \setminus \{(t, j) : j \in \mathbb{N}\}$
52: $t_{\mathsf{U}} \leftarrow \max(t_{\mathsf{U}}, t)$
53: **if** $\mathsf{c} \neq \mathsf{c}_{\overline{\mathsf{U}}}[t, i]$ **then**
54: $\quad$ **return** (t, i, m)
55: **return**

Oracle $\mathsf{expose}(\mathsf{U})$:
56: **req** $\mathsf{U} \in \{\mathsf{A}, \mathsf{B}\}$
57: $newComp \leftarrow sent_{\overline{\mathsf{U}}}$
58: $newComp \leftarrow newComp \cup \{(t_{\mathsf{U}}, i) : i \geq i_{\mathsf{U}}\}$
59: $newComp \leftarrow newComp \setminus received_{\mathsf{U}}$
60: **if** $i_{\mathsf{U}} \neq 0$ **then**
61: $\quad newComp \leftarrow newComp \cup \{(t_{\mathsf{U}} + 1, i) : i \in \mathbb{N}\}$
62: $\quad$ **if** $\Delta_{\mathsf{CKA}} = 1 \wedge \exists (t_{\mathsf{U}} - 1, \cdot) \in comp$ **then**
63: $\quad\quad newComp \leftarrow newComp \cup \{(t_{\mathsf{U}}, i) : i \in \mathbb{N}\}$
64: **req** $challs \cap newComp = \emptyset$
65: $comp \leftarrow comp \cup newComp$
66: $newRepl \leftarrow sent_{\overline{\mathsf{U}}}$
67: $newRepl \leftarrow newRepl \cup \{(t_{\mathsf{U}}, i) : i \geq i_{\mathsf{U}}\}$
68: $newRepl \leftarrow newRepl \cup \{(t_{\mathsf{U}} + 1, i) : i \geq i_{\mathsf{U}}\}$
69: **if** $i_{\mathsf{U}} \neq 0$ **then**
70: $\quad newRepl \leftarrow newRepl \cup \{(t_{\mathsf{U}} + 2, i) : i \in \mathbb{N}\}$
71: $\quad$ **if** $\Delta_{\mathsf{CKA}} = 1 \wedge \exists (t_{\mathsf{U}} - 1, \cdot) \in comp$ **then**
72: $\quad\quad newRepl \leftarrow newRepl \cup \{(t_{\mathsf{U}}, i) : i \in \mathbb{N}\}$
73: $newRepl \leftarrow newRepl \setminus received_{\mathsf{U}}$
74: $newRepl \leftarrow newRepl \setminus received_{\overline{\mathsf{U}}}$
75: $repl_{\mathsf{U}} \leftarrow repl \cup newRepl$
76: **return** $\mathsf{st_U}$

Figure 19: Generalized Signal Security Game

received) in the current epoch are replaceable. Note that sometimes an epoch is replaceable, but not compromised. This is when the adversary needs to call CKA.Send with a party's state to inject a message. If the adversary lets the real message be transmitted, the new epoch will use a key from the CKA that the adversary does not know about.

If we are willing to specify a weaker security, we can simplify the security notion. The adversary cannot impersonate U for epoch t index i even if he exposes both parties in epoch $t - 2 - \Delta_{\mathsf{CKA}}$ and exposes U in epoch $t + 1$ or after sending the i index message in epoch t. Similarly, he can't break privacy for a message sent in epoch t even if he exposes both parties in epoch $t - 2 - \Delta_{\mathsf{CKA}}$ and exposes both parties after they've sent or received the message. This is actually how idSEC is defined in [1]. We see that in $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}(\mathcal{A}, \lambda)$ at most $2 + \Delta_{\mathsf{CKA}}$ epochs are compromised from any call to expose.

In addition to the differences mentioned above, the notion of hijacking is different in this setting than the non-immediate decryption setting. There are two types of injected ciphertexts: ones that start new epochs and ones that don't. Ciphertexts that don't start new epochs essentially have no impact in the game (as long as it isn't a win). If the adversary injects a ciphertext that does start a new epoch for a party, we say that the party is hijacked.

The adversary is able hijack the party in the same sense as before causing the party to essentially now be in a conversation with the adversary. The party is still able to read any previous messages from the other party. However, the adversary is also able to keep the two parties in the conversation together. He can do this by using the same randomness that the other party used to start the same epoch. Then, it will be just as if the adversary injected a ciphertext that did not start a new epoch. In particular, the adversary is able to send a message to a party that isn't noticed as an injection.

7.6.1 Modular Security Notions

This complicated security notion can be broken down into two slightly simpler notions of security: authenticity and privacy. Then, we will show that if a MSG scheme is authentic and private according to these definitions, then it is idSEC. We prove this using game transitions. Both games are just defined in relation to $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}$. In both games, the adversary has to provide an epoch t^* that it will attack. The adversary also has to provide an epoch t^*_L, which is the last compromised epoch. Let $\mathcal{A}$ be an algorithm that plays $\mathbf{G}_{\mathsf{MSG.idSEC}}$.

In the authenticity game $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{auth}$, the adversary no longer gets to use the chall oracle. He also no longer wins by guessing b. He can only win by injecting a message in epoch t^* that he shouldn't be allowed to inject. The same rules for compromising messages apply, but he is also not allowed to compromise epochs between $t^*_L + 1$ and t^*.

In the privacy game $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{priv}$, the adversary cannot win by injecting a message. Also, he can only create a challenge ciphertext in epoch t^*. The same rules for compromising messages apply, but he is also not allowed to compromise epochs between $t^*_L + 1$ and t^*.

Transition 1 Let $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_1$ be the same as $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}$ except that the adversary cannot win by injection.

Now, we can create $\mathcal{A}_{auth}$ to play $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{auth}$ that simulates $\mathcal{A}$. First, $\mathcal{A}_{auth}$ picks random epochs $t^*_L < t^*$. Then, $\mathcal{A}_{auth}$ will pick a random challenge bit b, and simulate $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}$ using the oracles of $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{auth}$. If $\mathcal{A}$ calls $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$, then $\mathcal{A}_{auth}$ will call $\mathsf{send}(\mathsf{U}, \mathsf{m}_b)$. If $\mathcal{A}$ would win via injection in epoch t, then $\mathcal{A}_{auth}$ uses the injection ciphertext to win $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{auth}$. Note that $\mathcal{A}_{auth}$ runs in about the same amount of time as $\mathcal{A}$.

Because $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}$ and $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_1$ only differ when there is a successful injection, we have

$$\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{\mathsf{MSG.idSEC}}(\mathcal{A}, \lambda) - \mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_1(\mathcal{A}, \lambda) \leq 2q^2_{epochs}\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{auth}(\mathcal{A}_{auth}, \lambda)$$

where $\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_1(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}^{\Delta_{\mathsf{CKA}}}_1(\mathcal{A}, \lambda) = 1\right] - 1$ and $\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{auth}(\mathcal{A}_{auth}, \lambda) = \mathsf{Pr}\left[\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{auth}(\mathcal{A}_{auth}, \lambda)\right]$. Here q_{epochs} is the number of epochs. The multiplier of q^2_{epochs} is because t^* and t^*_L each independently have at least $1/q_{epochs}$ probability of being the epoch that is attacked by $\mathcal{A}$ and the epoch of last compromisation.

Final Reduction In the second step, we will relate $\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_1(\mathcal{A}, \lambda)$ directly to $\mathsf{Adv}^{\Delta_{\mathsf{CKA}}}_{priv}(\mathcal{A}_{priv}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{priv}(\mathcal{A}, \lambda) = 1\right] - 1$ for a $\mathcal{A}_{priv}$ that we will define.

We can create $\mathcal{A}_{priv}$ to play $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{priv}$ that simulates $\mathcal{A}$. First, $\mathcal{A}_{priv}$ picks random epochs $t^*_L < t^*$. Then, $\mathcal{A}_{priv}$ will simulate $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_1$ using the oracles of $\mathbf{G}^{\Delta_{\mathsf{CKA}}}_{priv}$. If $\mathcal{A}$ calls $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$

in epoch t^*, then $\mathcal{A}_{priv}$ will call $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$. If $\mathcal{A}$ calls $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ in epoch $t' < t^*$, then $\mathcal{A}_{priv}$ will call $\mathsf{send}(\mathsf{U}, \mathsf{m}_0)$. If $\mathcal{A}$ calls $\mathsf{chall}(\mathsf{U}, \mathsf{m}_0, \mathsf{m}_1)$ in epoch $t' > t^*$, then $\mathcal{A}_{priv}$ will call $\mathsf{send}(\mathsf{U}, \mathsf{m}_1)$. Lastly, $\mathcal{A}_{priv}$ outputs the same guess as $\mathcal{A}$. Note that $\mathcal{A}_{priv}$ runs in about the same amount of time as $\mathcal{A}$.

We can use a hybrid argument to see that

$$\mathsf{Adv}_{1}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) = q_{epochs}^2 \mathsf{Adv}_{priv}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}_{priv}, \lambda)$$

We have q_{epochs}^2 because there is a $1/q_{epoch}$ probability of picking t_L^* correctly, and the other factor of q_{epoch} comes from the hybrid argument.

7.7 Proof of Security

We claim that if the CKA is INDEXP secure for Δ_{CKA}, the FSAEAD is FSAEAD.SEC secure, and the PRFPRNG is secure, then the MSG construction in Figure 18 is idSEC secure for Δ_{CKA}.

We do this by proving that the construction is authentic and private based on the definition in the previous section. Both properties are proved in almost the identical way, so we will only walk through the privacy proof.

We prove this using game transitions. Let $\mathcal{A}$ be an algorithm that plays $\mathbf{G}_{priv}^{\Delta_{\mathsf{CKA}}}$ from the previous section. Taking the results of each transition and the final reduction gets $\mathsf{Adv}_{priv}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) = \mathsf{negl}(\lambda)$.

Transition 1 Let $\mathbf{G}_1^{\Delta_{\mathsf{CKA}}}$ be the same as $\mathbf{G}_{priv}^{\Delta_{\mathsf{CKA}}}$ except that keys outputted by the CKA in epochs $t_L^* + 1$ to $t^* - 1$ are replaced with random keys[7].

Now, we can create $\mathcal{A}_{\mathsf{CKA}}$ to play $\mathbf{G}_{\mathsf{CKA}}^{\Delta_{\mathsf{CKA}}}$ that simulates $\mathcal{A}$. First, $\mathcal{A}_{\mathsf{CKA}}$ chooses a challenge bit b. Then, he uses the CKA instance to build the MSG construction using the oracles of $\mathbf{G}_{\mathsf{CKA}}^{\Delta_{\mathsf{CKA}}}$ whenever needed to simulate the MSG. However, for the epochs of $t_L^* + 1$ to $t^* - 1$, $\mathcal{A}_{\mathsf{CKA}}$ will call chall to get either the real CKA key or a random key.

Notice, that if chall outputs the real key, $\mathcal{A}_{\mathsf{CKA}}$ will perfectly simulate $\mathbf{G}_{priv}^{\Delta_{\mathsf{CKA}}}$ and if chall outputs a random key, $\mathcal{A}_{\mathsf{CKA}}$ will perfectly simulate $\mathbf{G}_1^{\Delta_{\mathsf{CKA}}}$. We have $\mathcal{A}_{\mathsf{CKA}}$ guess that chall outputs a real key if $\mathcal{A}$ correctly guesses the challenge bit that $\mathcal{A}_{\mathsf{CKA}}$ chose.

So,

$$\mathsf{Adv}_{priv}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) - \mathsf{Adv}_{1}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) \leq 2\mathsf{Adv}_{\mathsf{CKA}}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}_{\mathsf{CKA}}, \lambda) = \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}_{1}^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) = 2\mathsf{Pr}\left[\mathbf{G}_1^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) = 1\right] - 1$.

Transition 2 Let $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$ be the same as $\mathbf{G}_1^{\Delta_{\mathsf{CKA}}}$ except that the outputs of PRFPRNG.Upd in epochs $t_L^* + 1$ to $t^* - 1$ are replaced with randomness.

We use the same idea as the previous transition. If $\mathcal{A}$ can distinguish between $\mathbf{G}_1^{\Delta_{\mathsf{CKA}}}$ and $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$, then we can find $\mathcal{A}_{\mathsf{PRFPRNG}}$ that can distinguish the PRFPRNG from random. Just as

[7]Technically, we should require that $\mathcal{A}$ should say during which epoch he will next compromise the party after, either t^* or $t^* + 1$. Allowing for this will reduce the advantage of $\mathcal{A}_{\mathsf{CKA}}$ by a factor of 2. We ignore this as it complicates the exposition and does not affect the overall proof.

before, $\mathcal{A}_{\mathsf{PRFPRNG}}$ simulates the games. If $\mathcal{A}_{\mathsf{PRFPRNG}}$ uses the true output of the PRFPRNG during epochs $t_L^* + 1$ and $t^* - 1$, then the simulation is $\mathbf{G}_1^{\Delta_{\mathsf{CKA}}}$. If $\mathcal{A}_{\mathsf{PRFPRNG}}$ uses randomness instead of the output, then the simulation is $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$.

So,

$$\mathsf{Adv}_1^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) - \mathsf{Adv}_2^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) \leq \mathsf{Adv}_{\mathsf{PRFPRNG}}(\mathcal{A}_{\mathsf{PRFPRNG}}, \lambda) = \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}_{\mathsf{PRFPRNG}}$ is defined appropriately in relation to $\mathbf{G}_{\mathsf{PRFPRNG}}$, which we didn't explicitly define.

Final Reduction In the last step, we directly relate $\mathsf{Adv}_2^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda)$ to $\mathsf{Adv}_{\mathsf{FSAEAD}}(\mathcal{A}_{\mathsf{FSAEAD}}, \lambda)$ for a $\mathcal{A}_{\mathsf{FSAEAD}}$ that we will define.

We can create $\mathcal{A}_{\mathsf{FSAEAD}}$ to simulate $\mathcal{A}$ playing $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$. However, $\mathcal{A}_{\mathsf{FSAEAD}}$ will treat the FSAEAD instance of epoch t^* as if it is the instance in $\mathbf{G}_{\mathsf{FSAEAD}}$, using oracles to simulate $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$. Lastly, $\mathcal{A}_{\mathsf{FSAEAD}}$ guesses the same challenge bit as $\mathcal{A}$.

We can see that $\mathcal{A}_{\mathsf{FSAEAD}}$ perfectly simulates $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$ and that the challenge bit in the simulated $\mathbf{G}_2^{\Delta_{\mathsf{CKA}}}$ is the same as the challenge bit in $\mathbf{G}_{\mathsf{FSAEAD}}$. So,

$$\mathsf{Adv}_2^{\Delta_{\mathsf{CKA}}}(\mathcal{A}, \lambda) = \mathsf{Adv}_{\mathsf{FSAEAD}}(\mathcal{A}_{\mathsf{FSAEAD}}, \lambda) = \mathsf{negl}(\lambda).$$

7.8 Security with Unprotected Randomness

Now, we will discuss how letting the adversary choose the randomness affects the security. First, we see that parties only use randomness if they are starting a new epoch in Send. So, for all other computations, letting the adversary choose the randomness does nothing because there is no randomness.

When U is starting a new epoch, the randomness is only used for the CKA. Based on our security definitions, revealing the randomness lets the adversary know the CKA key and the CKA states of the sender and the receiver after he receives the CKA ciphertext. However, for $\Delta_{\mathsf{CKA}} \in \{0, 1\}$, the receiver's state (after receiving) will be public anyway. This means that letting the adversary choose the randomness is no worse than exposing the state of U after the computation and exposing the message. In fact, the CKA key and state are only useful to the adversary if he knows (or will know) the state of the PRFPRNG because the FSAEAD uses keys generated by the PRFPRNG.

By the same logic as before, if we implement the randomness storage technique in section 5, choosing the randomness is useless unless the adversary knows the state of U before the computation. In that case, allowing the adversary to choose the randomness only exposes the state of U from after the computation and the message being sent.

In some sense, by implementing randomness storage, the Generalized Signal Protocol perfectly handles chosen randomness attacks because only the state after computation and the message are exposed. However, exposing the state of a party allows the adversary to perform more attacks against the Generalized Signal Protocol than against the other MSG schemes.

8 Other Schemes

In this section we will discuss two other schemes in the literature: the Poettering-Rösler scheme [11] and the Durak-Vaudenay scheme [4]. Neither scheme supports immediate decryption. Also, neither is secure to randomness leakage. They both consider the notion of key-exchange instead of messaging. The difference is that Send does not take a message as an input, but it additionally outputs a random key. Also, Rcv returns the key instead of a message. Correctness of a key exchange states that the two parties should agree on the key. Instead of being a MSG, Poettering and Rösler call their construction a Bidirectional Ratcheted Key-Exchange BRKE[8].

We can see that a BRKE and a MSG are essentially equivalent. If we make the messages random and use them as the keys, a MSG becomes a BRKE. We can use the BRKE keys to encrypt messages with a CCA secure private key encryption scheme to construct a MSG.

One difference between the security games for MSG and the security games for BRKE is how the challenges work. In the security games for BRKE, the adversary can cause parties to send ciphertexts. He can also ask for the key associated with a ciphertext. Lastly, he can ask for a challenge relating to a ciphertext in which he is given either the correct key or a random key.

8.1 Poettering-Rösler Scheme

The Poettering-Rösler scheme [11] was introduced at the same time as the Jaeger-Stepanovs scheme, and it is often stated to be fully secure, such as in [4, 7]. However, this scheme does not have the same SEC level of security as the Jaeger-Stepanovs scheme even in the case of protected randomness.

Security Definition As mentioned before, in the security game, the adversary gets to challenge ciphertexts and gets the real key or a random key. Here, he can ask for the key that either the sender or receiver got from the ciphertext, but he can't challenge the same ciphertext with the purpose of trivially figuring out the challenge bit.

Besides the randomness leakage and how challenges work, the only difference of this security definition and optimal SEC security is that Poettering and Rösler do not consider authenticity in their definition. The only way for the adversary to win the security game is to figure out the challenge bit when he shouldn't be able to. This does imply some notion of authenticity. If the adversary injects a ciphertext to U, he can challenge the ciphertext that he injected. Thus, if he knows the key associated with his own ciphertext he wins the game. However, if the adversary exposes U and then hijacks U, the adversary is trivially able to read his own ciphertext, meaning the adversary does not win. This means that the security definition allows for the possibility that exposing U is enough to hijack U.

[8] A ratchet is a physical device that allows motion in one direction but not in the other direction. The reason why these schemes are called ratcheted key-exchanges is because from the current state, we can get the future keys, but we can't go backwards and get the keys from the past.

Construction Here we give a brief sketch of the construction. In [11], to construct BRKE, they first start with Sesquidirectional Ratcheted Key-Exchange SRKE where only the messages from Alice to Bob create keys. They build their SRKE from a key-updatable key encapsulation mechanism kuKEM. The kuKEM is very similar to the kuPKE from [6] and is also constructed from HIBE. This is used by Alice to send Bob keys. These keys are then used as input to a hash function which produces shared randomness between Alice and Bob. When Bob sends a message to Alice, he sends a new kuKEM encryption key and a new DS verification key. He signs his message with his old signing key. It's important to note that one can get Alice's state from Bob's state. They then construct the BRKE by just combining two instances of SRKE.

The runtime of the algorithms, state sizes, and ciphertext sizes are asymptotically equivalent to the Jaeger-Stepanovs scheme.

Attack In this section we discuss a non-trivial attack against the construction. To impersonate U, the adversary needs the signing key of U, which is computed by the last message that U sent. So, the adversary must expose U. This would usually allow the adversary to impersonate U. However, if the adversary hijacks U and then exposes U, based on the construction, the adversary is still able to get the signing key. In a fully secure scheme, this would not be the case.

The full attack starts by exposing $\overline{\mathsf{U}}$. Then, the adversary hijacks U and exposes U. Then, the adversary hijacks $\overline{\mathsf{U}}$. The adversary needs to impersonate U for both instances of the SRKE. Because Alice's state can be derived from Bob's state in the SRKE, having exposed $\overline{\mathsf{U}}$ is enough to impersonate U for the SRKE that has keys generated from U sending the message. To impersonate U in the second SRKE, the signing key is the only secret needed.

This section shows that the scheme does not achieve post-hijack authentication. It also shows that the scheme does achieve authentication up until hijacking.

8.2 Durak-Vaudenay Scheme

The scheme by Durak and Vaudenay [4] is notable in that it is the only scheme with amortized constant runtime that doesn't have immediate decryption. It is however weaker than all other schemes excluding the Generalized Signal Protocol. Essentially, its security states that privacy and authenticity are maintained until either party is hijacked.

In their construction, each party keeps a list of sending states, which are pairs of encapsulation keys and signing keys. They also keep a list of receiving states which are decapsulation keys and verification keys. Whenever U sends a message, he also creates a sending state and sends the corresponding receiving state to $\overline{\mathsf{U}}$. He creates another pair of states, keeping the receiving state of this one and sending the sending state. To provide the authentication and privacy, he uses all of the sending states he already had. Each one is used to securely encapsulate some key, and the ciphertext of each is used as associated data for the encapsulation of the next. The final key outputted in the scheme is the XOR of the individually encapsulated keys. Lastly, he deletes the sending states he used. Similarly, to receive a message, a party uses all of the receiving states.

9 Conclusion

With five different schemes in the literature, the natural question to ask is which scheme should you use. As a user, this is an easy question to answer because of these protocols only Signal is widely implemented. As a designer, it is a harder question to answer. It is clear that the Poettering-Rösler scheme is never the best choice because it is worse than the Jaeger-Stepanovs scheme in every way. Besides that scheme, the other four have their own comparative advantages and disadvantages. Each of the schemes might be the right choice depending on the needs of a system.

The first question a system designer should ask is whether or not a system needs immediate decryption as only the Generalized Signal Protocol achieves that. If there is a chance that a message might be lost, only a scheme with message loss resilience would suffice. In fact, for the other four schemes, messages that were generated after a lost message are indistinguishable from messages an adversary could produce even without having compromised either state. This means that even the possibility of receiving messages out of order makes the other four schemes unusable as is.

If immediate decryption is not necessary, other major differences between the schemes are the runtimes, ciphertext lengths, and state sizes. In the Generalized Signal Protocol all three of these are constant size, except that that state size grows linearly in the number of lost messages. This last piece cannot be compared to the other protocols because they can't withstand losing a message. In these categories, the next best is the Durak-Vaudenay scheme, which has amortized constant size ciphertexts and runtimes. However, the runtimes, ciphertexts sizes, and state sizes can all be linear in the number of messages sent without reply. The next best is the Jost-Maurer-Mularczyk scheme, which has constant ciphertext size, but has state size and runtime that grow linearly in the number of messages sent without reply. The worst is the Jaeger-Stepanovs scheme, which has ciphertexts and runtime that grow linearly and state size that can even grow to quadratic size in the number of messages. Not only does this scheme have bad asymptotic performance, but it is very slow even when the messages are alternating according to [4]. Depending on the needs of the system, the schemes with worse asymptotics might not be possible to use.

A common argument made in the papers with worse asymptotics is that in most messaging schemes the messages are almost alternated. In other words, the claim is that most people do not send that many messages without getting a response. While this may be true in general, there are certainly cases where one might expect someone to send many messages without expecting a response. Also, the papers with worse security (and better asymptotics) will heal from exposures faster when the messages nearly alternate. This means that alternating communication improves all the schemes.

The last category on which to compare the schemes is security. The schemes with worse asymptotics have better security. The Jaeger-Stepanovs scheme has optimal security even in the case of leaked or chosen randomness (with a minor change). The scheme with the next best security is the Jost-Maurer-Mularczyk scheme. Its only security flaw is that if an adversary hijacks U and then exposes U, he can impersonate U or read messages from $\overline{\mathsf{U}}$ if $\overline{\mathsf{U}}$ leaks randomness or if U had been exposed previously. However, if U is hijacked and then sends a message before being exposed, there is no security flaw. Essentially, the adversary is

able to become a middleman if he can expose the states of both parties at nearly the same time or if he can make a U leak randomness and expose $\overline{\mathsf{U}}$ after hijacking $\overline{\mathsf{U}}$. A middleman attack is a very harmful attack where the adversary acts as an intermediary between Alice and Bob. Alice and Bob think they are communicating with each other, but the adversary intercepts every message from U, reads it, and then encrypts it again, and sends it to $\overline{\mathsf{U}}$.

The Durak-Vaudenay scheme has this vulnerability and another one. If U leaks his sending randomness, the states of both parties are exposed. This allows the adversary to read messages or hijack either party. He can even become a middleman by just leaking the randomness of one sending operation.

The Generalized Signal protocol is the least secure. Exposing the state of a party often exposes the state of the other party. This means that exposing one party's state allows reading all messages. It also allows the adversary to become a middleman. If we added a layer of public key encryption and signing where keys remained the same for an epoch, we could make the Generalized Signal Protocol more secure to middleman attacks. Its security to middleman attacks would fall somewhere between that of the Durak-Vaudenay scheme and the Jost-Maurer-Mularczyk scheme. Even with this change, the Generalized Signal Protocol heals more slowly from exposures compared to the other schemes. Adding this layer of public key cryptography would slow the protocol, but it would not change the asymptotics of the protocol.

It is worth noting that if the adversary cannot expose the state of either party and cannot leak randomness, then the schemes would all be perfectly secure. It's also worth noting that an optimally secure messaging system is not perfect; exposing U will always let the adversary impersonate U and read messages from $\overline{\mathsf{U}}$. So, to compare the security of the different schemes, we must account for the baseline level of security.

For example, to compare the security of the Jaeger-Stepanovs scheme to the Jost-Maurer-Mularczyk scheme, we want to know whether it is easier for the adversary to expose U, leak sending randomness of U, hijack $\overline{\mathsf{U}}$, and then expose $\overline{\mathsf{U}}$ compared to just exposing both parties at the same time. If it is much easier, then we would want to use the optimally secure Jaeger-Stepanovs scheme. Consider a phishing attack. If the adversary sends $\overline{\mathsf{U}}$ a message impersonating U and is able to get $\overline{\mathsf{U}}$ to somehow reveal his state, all the adversary needs to do is just expose U and leak his sending randomness. If the adversary accomplishes this phishing attack, he can become a middleman.

Along the same lines, this argument shows that the security of the MSG is less important for a system designer than making sure that it is difficult for an adversary to be able expose either party or leak randomness. On the flip side, if the rest of the system is less secure, the security of the MSG becomes more important.

In terms of security, it is also worth noting that the Jost-Maurer-Mularczyk scheme is only secure in the random oracle model due to the El Gamal encryption. The Jaeger-Stepanovs scheme is also only secure in the random oracle model due to the construction of the kuDS. The Durak-Vaudenay scheme and Generalized Signal Protocol are secure in the standard model. Similarly, because of El Gamal encryption and the construction of the kuDS, the Jost-Maurer-Mularczyk scheme and the Jaeger-Stepanovs scheme are both not secure against a quantum computing adversary. The Durak-Vaudenay scheme and Generalized Signal Protocol can use building blocks that keep the same level of security even against a quantum computing adversary.

All of the schemes that we have looked at consider secure messaging between two people each on one device, but in the real world communication is often more complicated. We would like a protocol for group messaging with provable security, but there are no such schemes in the literature. Signal (the company) treats a group messaging scheme as just several pairwise messaging schemes between each of the group participants [10]. This certainly has security, but it would be interesting to formally analyze what level of security it achieves. In particular, their method allows different members of the group to have different views of what messages have been sent. It would also be interesting to know if there are group messaging schemes that are more secure or more efficient (but with a reasonable level of security).

Another interesting generalization is the case where each user has multiple devices that they would like to message from. One way to implement this would be to treat the set of devices as a group and use group messaging. Another method is to treat one device as the main device for each person. Then, each device can securely communicate with the main device, and the main devices can securely communicate with each other. This has the advantage of fewer pairwise messaging schemes, but it has the disadvantage that the main device must be online to communicate from any device.

References

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *EUROCRYPT*, 2018.

[2] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 431–448, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[3] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *Proc. 23rd ACM Conference on Computer and Communications Security (CCS) 2016*. ACM, October 2016.

[4] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security*, pages 343–362, Cham, 2019. Springer International Publishing.

[5] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. Cryptology ePrint Archive, Report 2002/056, 2002. `https://eprint.iacr.org/2002/056`.

[6] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. Cryptology ePrint Archive, Report 2018/553, 2018. `https://eprint.iacr.org/2018/553`.

[7] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *Advances*

in Cryptology – EUROCRYPT 2019, pages 159–188, Cham, 2019. Springer International Publishing.

[8] David Kaplan, Sagi Kedmi, Roee Hay, and Avi Dayan. Attacking the linux PRNG on android: Weaknesses in seeding of entropic pools and low boot-time entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, August 2014. USENIX Association.

[9] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. Cryptology ePrint Archive, Report 2010/264, 2010. `https://eprint.iacr.org/2010/264`.

[10] Moxie Marlinspike. Private group messaging. `https://signal.org/blog/private-groups/`, 2014. Accessed: 2020-04-01.

[11] Bertram Poettering and Paul Rösler. Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296, 2018. `https://eprint.iacr.org/2018/296`.