Symmetric Signcryption and E2EE Group Messaging in Keybase

Joseph Jaeger¹, Akshaya Kumar¹, and Igors Stepanovs²

¹ School of Cybersecurity and Privacy
Georgia Institute of Technology, Atlanta, Georgia, US
{josephjaeger,akshayakumar}@gatech.edu
https://cc.gatech.edu/~josephjaeger/
https://cc.gatech.edu/~akumar805/
² igors.stepanovs@gmail.com
https://igors.org/

23 May 2024

Abstract. We introduce a new cryptographic primitive called symmetric signcryption, which differs from traditional signcryption because the sender and recipient share a secret key. We prove that a natural composition of symmetric encryption and signatures achieves strong notions of security against attackers that can learn and control many keys. We then identify that the core encryption algorithm of the Keybase encrypted messaging protocol can be modeled as a symmetric signcryption scheme. We prove the security of this algorithm, though our proof requires assuming non-standard, brittle security properties of the underlying primitives.
Contents

1 Introduction .................................................................................... 3

2 Preliminaries ................................................................................... 7
  2.1 Standard Primitives .............................................................................. 7
  2.2 Standard Security Notions in a Multi-key Setting.............................................. 8

3 Symmetric Signcryption....................................................................... 10
  3.1 In-Group Unforgeability.......................................................................... 11
  3.2 Out-Group Authenticated Encryption .......................................................... 13

4 Symmetric Signcryption from Encryption and Signatures ........................ 15
  4.1 Sign-then-Encrypt Based Construction ......................................................... 15
  4.2 Encrypt-then-Sign Based Construction ......................................................... 16

5 Modeling Keybase Chat Encryption as Symmetric Signcryption .................. 16
  5.1 Message and Header Encryption Schemes BoxMessage and SealPacket ............ 18

6 Security Analysis of Keybase Chat Encryption ............................................ 20
  6.1 In-Group Unforgeability of BoxMessage and SealPacket ..................................... 20
  6.2 Out-Group AE Security of BoxMessage ....................................................... 21
  6.3 Out-Group AE Security of SealPacket .......................................................... 23
  6.4 Stronger Out-Group AE Security Results....................................................... 27
  6.5 Instantiation and Interpretation ........................................................................ 28

7 Conclusions ..................................................................................... 30

A Proofs for Sign-Then-Encrypt Based Symmetric Signcryption .......................... 33
  A.1 In-Group Unforgeability of StE (Proof of Theorem 1) ...................................... 33
  A.2 Out-group Authenticated Encryption of StE (Proof of Theorem 1) ...................... 34

B Proofs for the In-Group Unforgeability of Keybase........................................ 37
  B.1 In-Group Unforgeability of BoxMessage (Proof of Theorem 2) ......................... 37
  B.2 In-Group Unforgeability of SealPacket (Proof of Theorem 3) ............................. 39

C Analysis and Proofs for the Out-Group AE Security of Keybase ...................... 42
  C.1 M-sparsity of Ed25519 .................................................................................. 42
  C.2 $\Phi_{sp}$ Derives Messages from Hashed NE Keys ........................................... 48
  C.3 KDMAE for Messages Derived from a Hashed Key (Proof of Proposition 1) ....... 51
  C.4 Out-Group AE Security of BoxMessage (Proof of Theorem 4) ......................... 53
  C.5 Out-group AE Security of SealPacket (Proof of Theorem 5) ............................. 57

D Proofs for the Stronger Out-Group AE Security Results ................................. 59
  D.1 Proof for the UV Security of StE (Proof of Proposition 2) ................................. 59
  D.2 Proof for the UV Security of Ed25519 (Proof of Proposition 3) ......................... 60
  D.3 Stronger Out-Group AE Security Results (Proof of Proposition 4) .................... 61

E Other Uses of Symmetric Signcryption in Keybase ........................................ 65
  E.1 Security of SealPacket as a Standalone Encryption Scheme ......................... 65
  E.2 Multi-purpose Scheme SealWhole for Encrypting Long Strings ...................... 66
  E.3 Encryption of Attachments with PostFileAttachment ...................................... 66
1 Introduction

Keybase is a suite of encryption tools. It encompasses a public-key directory, an instant messenger, and a cloud storage service. Keybase was launched in 2014. In February 2020, it reported having accumulated more than 1.1M user accounts [Keyi]. In May 2020, Keybase was acquired by Zoom. At the time, Zoom issued a public statement [Zoo20] saying that the Keybase’s team was meant to play a critical part in building scalable end-to-end encryption for Zoom. The acquisition appears to have put an end to the active development of new Keybase features, but as of February 2024 it keeps receiving regular maintenance updates.

Instant Messaging in Keybase. Keybase implements its own end-to-end encrypted instant messaging protocol. This protocol is designed to support large groups. One-on-one chats are treated as group chats and hence use the same protocol. The protocol also allows to send large files as encrypted attachments in chat. It is impossible to opt out of end-to-end encryption in Keybase. In this work we analyze the security of this protocol.

The Keybase client is open source [Keyc], but the server is not. Our security analysis primarily relies on the source code. Keybase also provides the “Keybase Book” website [Keya] with excellent documentation that explains its cryptographic design. The only prior security analysis of Keybase was done by the NCC Group in 2019 [RPF19], which broadly looked at the security of the entire Keybase ecosystem. In comparison, we provide an in-depth analysis of a single component in Keybase.

Encrypted Group Chats. In this work we consider a setting in which an arbitrary number of users can form a group. All group members share a key for a symmetric encryption scheme. Each instant message within the group is encrypted with this key. Let us use $g$ to denote the identity of a group and $K_g$ to denote the key shared between the members of this group. In Keybase, every member of group $g$ uses the same long-term key $K_g$ to encrypt their outgoing chat messages. Each message is encrypted only once, simultaneously for all recipients. The resulting ciphertext is then broadcast to all members of the group.

The Sender Keys protocol [BCG23,Mar14] can be seen as building on this basic design idea. In Sender Keys, every member of the group owns a distinct symmetric encryption key; they share it with other group members. Each outgoing message is encrypted with the sender’s own key, and the resulting ciphertext is broadcast to the group. Furthermore, each key is used to encrypt only a single message, and immediately afterwards a new key is derived to be used for the next encryption. So every group member tracks every other member’s current encryption key, decrypting each incoming ciphertext with the corresponding sender’s key and subsequently replacing it with an appropriately derived new key. Variants of the Sender Keys protocol are used in the Signal [Mar14], WhatsApp [Wha23], and Matrix [ACDJ22,ADJ23] messengers. In addition, the Messaging Layer Security (MLS) [BBR+23] protocol contains a component called FS-GAEAD [ACDT21] or TreeDEM [WPBB23] that similarly uses a sender’s key to encrypt and broadcast a message (but its overall design significantly differs from design of the Sender Keys protocol).

An encrypted group chat protocol should provide at least confidentiality and integrity of communication, with respect to an attacker that is not a member of the group. In part, this could be achieved by building the protocol from a symmetric encryption scheme that satisfies some notion of authenticated-encryption security. But care is needed to also prevent undesired message replays, reordering, or drops. These requirements are specific to a stateful protocol and do not necessarily follow from properties provided by the underlying stateless scheme.

Sender Authentication in Group Chats. Consider a group chat protocol that is built from a single symmetric encryption scheme where every symmetric key is known to all group members. In such a protocol, group members can impersonate each other. This is true regardless of whether each group uses a single shared encryption key or has each member own a distinct encryption key. To prevent group members from impersonating each other, it is natural to use a digital signature scheme. Let us use $u$ to denote the identity of a user and $sk_u$ to denote this user’s signing key for a digital signature scheme.

What is a sound way to compose a symmetric encryption scheme with a digital signature scheme? Let us consider two sequential compositions of a signing algorithm $\text{Sign}$ with an encryption algorithm $\text{Encrypt}$. We call the resulting schemes $\text{Sign-then-Encrypt}$ and $\text{Encrypt-then-Sign}$, and we show them in Fig. 1. The $\text{Sign-then-Encrypt}$ scheme first signs a message $m$ to obtain its digital signature $s$ and then encrypts $(s,m)$.
to obtain and return a symmetric ciphertext $c$. The **Encrypt-then-Sign** scheme first encrypts $m$ as $c$ and then computes a signature $s$ over $c$; it returns the pair $(c, s)$.

These compositions closely mirror those that are commonly used to build signcryption [ADR02], which is a standard cryptographic primitive that combines digital signatures with public-key encryption [Zhe97], except we replace public-key encryption with symmetric encryption. It is well known that the corresponding compositions for signcryption are not secure in the multi-user setting, unless some effort is taken to bind together the message with the sender and recipient identities [ADR02]. The standard advice is to always sign the recipient’s identity and always encrypt the sender’s identity. Our basic schemes in Fig. 1 would intuitively suffer from similar issues and benefit from similar countermeasures. However, the exact details would depend on what kind of security one expects from these schemes, so we defer this discussion.

The **Sender Keys** protocol [BCG23] prescribes to sign a symmetric ciphertext; and this is indeed done by the **Signal**, **WhatsApp**, and **Matrix** messengers. The MLS protocol [BBR + 23] protocol prescribes encrypting a digital signature with a symmetric encryption scheme. So either protocol can be seen as using some variant of **Encrypt-then-Sign** or **Sign-then-Encrypt** as a subroutine. We will now discuss that Keybase can be seen as extending both of these basic schemes.

**SealPacket**: **Sign-then-Encrypt in Keybase**. Keybase uses a variant of the basic **Sign-then-Encrypt** scheme. It signs the symmetric encryption key along with the plaintext, meaning it signs $(K_g, m)$ instead of just $m$. The resulting scheme is called **SealPacket** and is shown in Fig. 2. In the source code, the decision to sign $K_g$ is explained as follows [Keye]:

> simply using encryption and signing together isn’t good enough ... the inner layer needs to assert something about the outer layer ... a better approach is to mix the outer key into the inner crypto, so that it’s impossible to forget to check it ... That means the inner signing layer needs to assert the encryption key ... We don’t need to worry about whether the signature might leak the encryption key either, because the signature gets encrypted.

Keybase uses **SealPacket** to encrypt the following three types of plaintexts: (1) a metadata header that is automatically created and sent along with every chat message, (2) a file that is sent as an attachment in chat, and (3) an arbitrary string chosen by a chatbot (for secure server-side storage of bot data).

**BoxMessage**: **Encrypt-then-Sign in Keybase**. The **BoxMessage** scheme in Keybase is a variant of the basic **Encrypt-then-Sign** scheme. This scheme, unlike **SealPacket**, is used only for one purpose: to encrypt the body of a chat message. So we denote by $c_{\text{body}}$ the symmetric ciphertext that is created in the inner (encryption) layer of **BoxMessage**. The **BoxMessage** scheme extends the basic **Encrypt-then-Sign** scheme in two ways and is shown in Fig. 3. First, it takes an associated-data field $ad$ and signs $(c_{\text{body}}, ad)$ instead of just $c_{\text{body}}$. Second, rather than use a signature scheme, **BoxMessage** uses **SealPacket** to sign $(c_{\text{body}}, ad)$. 
Keybase uses the auxiliary-data field \( ad \) to authenticate a metadata header for the chat message. This header contains the group’s identity and the sender’s identity among multiple other values. The data in \( ad \) is sent in plain over the network (along with \( c_{body} \)), meaning that \( \text{SealPacket} \) is not meant to provide confidentiality of \( ad \). Indeed, the Keybase documentation explains that \( \text{SealPacket} \) is used to provide the confidentiality of the signature over \( (c_{body}, ad) \) [Keyb]:

The fields in the header aren’t secret from the server, and it actually needs to know several of them … The reason for sign-then-encrypting/signencrypting the header is instead to keep the signature itself private. Even though the server knows who’s talking to whom, because it’s delivering all the messages, it’s better that it can’t prove what it knows.

Interestingly, \( \BoxMessage \) reuses the group’s symmetric encryption key \( K_g \) between its calls to \( \text{Encrypt} \) and \( \text{SealPacket} \). As mentioned above, \( \text{SealPacket} \) will itself first sign \( K_g \) and then run another instance of \( \text{Encrypt} \) with \( K_g \) as the key. In total, the same value of \( K_g \) is therefore used in 3 distinct contexts.

**Symmetric Signcryption.** We define *symmetric signcryption* as a new cryptographic primitive that combines symmetric encryption with digital signatures. We capture the setting where every user owns a signing key pair and in each group, all users share a single symmetric encryption key. The encryption key is long-term, meaning it can be used an arbitrary number of times, simultaneously by all members of the group. This will allow us to formalize and analyze the \( \text{SealPacket} \) and \( \BoxMessage \) schemes.

Note that the use of sender-specific encryption keys in the *Sender Keys* [BCG23] and MLS [BBR+23] protocols can also be captured by symmetric signcryption. Indeed, in either protocol, all symmetric encryption keys can seen as being independently sampled, and each individual key is used only once. This can be thought of as a collection of “one-time-use” symmetric signcryption schemes.

We adapt the standard syntax of (asymmetric) signcryption to suit our setting, defining algorithms \( \text{SigEnc} \) and \( \text{VerDec} \). They both take explicit sender and group identities, nonces, and associated data.

**Security of Symmetric Signcryption.** We define two security notions for symmetric signcryption. The *out-group authenticated encryption* (OAE) security requires confidentiality and integrity of communication against an adversary that does not know the symmetric key of the group it attacks. This is required to hold even against an adversary that can assign to every user an arbitrary (possibly malformed) digital signature key pair. The *in-group unforgeability* (IUF) security requires the unforgeability of messages sent by users whose signing keys are not known to the adversary. This is required to hold even against an adversary that can assign to every group an arbitrary (possibly malformed) symmetric key.

We show how to extend the basic *Sign-then-Encrypt* scheme, by carefully incorporating user and group identifiers, to achieve both of our security notions. We assume strong unforgeability of the underlying digital signature scheme and authenticated-encryption security of the underlying encryption scheme.

**Implementation of \( \text{SealPacket} \) and \( \BoxMessage \).** In the source code of the Keybase client, \( \text{SealPacket} \) is implemented in [Keyf] and \( \BoxMessage \) is implemented in [Keyd]. These schemes are instantiated with the nonce-based authenticated encryption scheme \( \text{XSalsa20-Poly1305} \) [Ber05,Ber08] and the digital signature scheme \( \text{Ed25519} \) [BDL+11,BDL+12]. They also use SHA-512 and SHA-256 which does not significantly affect the design of either scheme so we omit discussing it here, but in the main body of the paper, we attempt to formalize both schemes precisely.

Keybase implements four versions of the \( \BoxMessage \) scheme: V1, V2, V3, and V4. V1 is deprecated; the Keybase client allows to receive but not send messages that use V1. V2 is the default version that
we formalize and analyze in this work. V3 is the same as V2, except it supports exploding messages; the body of an exploding message is encrypted using an ephemeral key instead of $K_g$. V4 is the same as V3, except it makes all group members use a dummy (zero) signing key and instead authenticate messages using pairwise MACs.

**Provable Security Analysis.** We model both BoxMessage and SealPacket as symmetric signcryption schemes and provide formal reductions for their IUF and OAE security. Our analysis is done in a concrete security framework [BDJR97], and in a multi-key setting; we state precise bounds on the advantage of an attacker. The analysis of BoxMessage largely encompasses that of SealPacket, because BoxMessage uses SealPacket in a modular way. So we focus on the analysis of BoxMessage here. The main challenges arise from using $K_g$ in 3 distinct contexts.

First, we aim to show it is hard to switch the context of the XSalsa20-Poly1305 ciphertexts $c_{body}$ and $c_{header}$. Both are encrypted using the same key $K_g$, so there is a risk that an attacker could forge a valid encryption of some body-plaintext $m$ from a known encryption of some header-plaintext $(s, c_{body}, ad)$, or vice versa. To rule out such attacks, we rely on an observation that every application-layer message $m$ that is queried to be encrypted by BoxMessage is encoded in a specific way, whereas every header-plaintext $(s, c_{body}, ad)$ is expected to start with a valid Ed25519 signature. Based on the specification of Ed25519 we show (in the ROM and GGM) that it is hard to cast the encoding used in $m$ as a valid Ed25519 signature (with respect to any verification key of adversary’s choice).

Second, we need to show that XSalsa20-Poly1305 provides authenticated encryption even for certain messages derived from its secret key. This arises because in SealPacket the XSalsa20-Poly1305 key $K_g$ is first signed with Ed25519 and then the resulting signature is encrypted using XSalsa20-Poly1305 under the same key. Here again we rely on the specification of Ed25519. An Ed25519 signature depends on two SHA-512 hash values of the message that is being signed, but it does not depend on the signed message beyond that. We use this (in the ROM) to eliminate the need to consider key-dependent messages and hence only require XSalsa20-Poly1305 to provide the standard notion of authenticated encryption.

We do not know any way to avoid the above analysis. The necessity to use non-standard security notions appears to be inherently implied by the design decisions made in Keybase. This could have been avoided (e.g. with our Sign-then-Encrypt scheme). Overall, our reductions (in the ROM and GGM) rely on the AEAD security of XSalsa20-Poly1305, collision resistance of SHA-256 and SHA-512, and strong unforgeability of Ed25519. We note that Keybase uses the version of Ed25519 that was recently shown to be SUF-CMA secure [BCJZ21,BDD23].

**Limitations of Our Work.** Our analysis of Keybase is intentionally narrow in scope. We perform an in-depth, algorithmic analysis of specific chat components that can be modeled as symmetric signcryption. Other analysis is outside the scope of our work, such as whether these algorithms are secure against timing attacks and whether they provide protection against message replays, reordering, or drops when used within the broader stateful chat protocol. More broadly, our analysis does not explicitly cover many other applications of cryptography in Keybase, including other versions of BoxMessage, encryption of attachments or bot data, the initial key exchange used to agree on group keys, the public-key directory used to share user keys, and the cloud storage service. These applications are important for the overall security of Keybase and have the potential to interplay with each other in subtle ways. For example, user signing keys are used for multiple tasks in Keybase. We believe appropriate context separation is used for these purposes (e.g. all messages signed in SealPacket start with “Keybase-Chat-2”). If not, subtle cross-application attacks may be possible.

**Related Work.** The Hybrid Public-Key Encryption (HPKE) scheme is specified in RFC 9180 [BBLW22]. Alwen, Janneck, Kiltz, and Lipp [AJKL23] analyze the “pre-shared key” modes from RFC 9180. They cast the HPKE$_{AuthPSK}$ mode as an asymmetric signcryption scheme that is augmented with a pre-shared symmetric key, and they define the corresponding security notions. They analyze the security that is achieved by HPKE$_{AuthPSK}$ depending on which combinations of keys are secure. Our definitions are similar in the sense that both works define a signcryption-type primitive that in addition uses a symmetric key. However, the algorithms in [AJKL23] use one more set of keys, and the definitions in [AJKL23] are
stated in the two-user setting. In essence, our primitives are similar in form but are tailored to be used as tools in different settings.

2 Preliminaries

Basic Notation. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$. We denote the empty string by $\varepsilon$. If $x \in \{0, 1\}^*$ is a bit-string then $|x|$ denotes its length, $x[i]$ denotes its $i$-th bit for $1 \leq i \leq |x|$, and $x[i..j] = x[i] \ldots x[j]$ for $1 \leq i \leq j \leq |x|$. By $x \parallel y$ we denote the concatenation of bit-strings $x, y \in \{0, 1\}^*$. We write $(a, b, \ldots)$ to denote an injective encoding of $a, b, \ldots$ into a bit-string that is uniquely decodable, where each of the encoded elements can have an arbitrary data type. If $X$ is a finite set, we let $x \leftarrow X$ denote picking an element of $X$ uniformly at random and assigning it to $x$. If $\text{mem}$ is a table, $\text{mem}[i]$ denotes the element of the table that is indexed by $i$. We let $\bot \notin \{0, 1\}$ be an error code that indicates rejection. Explicitly uninitialized integers are assumed to be initialized to 0, Booleans to false, strings to $\bot$, and sets to the empty set. Uninitialized elements of a table are initialized similarly, based on the type of data they will hold.

Algorithms and Adversaries. By $y \leftarrow A^O(x)$ we denote the randomized execution of algorithm $A$ on input $x$, with oracle access to $O$, producing output $y$. We generalize this to multiple inputs or oracles in the natural way. If multiple oracles are provided, then $A$ is allowed to call them arbitrarily many times, in any order. If $A$ is deterministic, we use $\leftarrow$ instead. Running time is worst case. We let $|A(x)|$ denote the set of all possible outputs of $A$ when invoked with input $x$. The instruction $\text{abort}(x)$ is used to immediately halt an algorithm with output $x$. Adversaries are algorithms. We require that adversaries never pass $\bot$ as input to their oracles.

Security Games and Reductions. We use the code-based game-playing framework of [BR06] for specifying security definitions and writing proofs. We write “require $d$” as shorthand for “If $\neg d$ then return $\bot$”. $\Pr[g]$ denotes the probability that game $g$ returns true. Variables in each game are shared with its oracles. In the security reductions, we omit specifying the running times of the constructed adversaries when they are roughly the same as the running time of the initial adversary. In our security-reduction games, we annotate some lines with comments of the form “// $g_i(i, j)$” to indicate that a particular line belongs only to games $g_i, \ldots, g_{i+1}$; we write $j = \infty$ when the line belongs to all games through the end of the reduction. The lines not annotated with such comments are shared by all of the games that are shown in the particular figure. To determine the difference between $g_{i-1}$ and $g_i$, look for subscripts $[i, j)$ and $[j, k)$.

Fundamental Lemma of Game Playing. In our proofs, we frequently make use of the Fundamental Lemma of Game Playing [BR06]. Suppose that games $g_i$ and $g_{i+1}$ are syntactically identical except after a Boolean flag bad is set. Then

$$\Pr[g_i] - \Pr[g_{i+1}] \leq \Pr[\text{bad}^i] = \Pr[\text{bad}^{i+1}],$$

where $\Pr[\text{bad}^i]$ denotes the probability of setting the flag bad in game $g^i$.

2.1 Standard Primitives

We will use several standard cryptographic primitives including hash functions, function families, nonce-based encryption schemes, and digital signature schemes.

Hash Functions and Random Oracles. A hash function is a function $H: \{0, 1\}^* \rightarrow \{0, 1\}^{H,\text{ol}}$ for some fixed output length $H,\text{ol} \in \mathbb{N}$. The advantage of an adversary $A_{\text{CR}}$ in breaking the collision resistance of $H$ is defined as $\text{Adv}_{H}^c(A_{\text{CR}}) = \Pr[H(x) = H(y) \wedge x \neq y : (x, y) \leftarrow A_{\text{CR}}]$.
We define every nonce-based encryption scheme \( \text{NE} \) and adversary \( \mathcal{A}_{\text{AEAD}} \). The advantage of adversary \( \mathcal{A}_{\text{AEAD}} \) in breaking the AEAD security of \( \text{NE} \) is defined as

\[
Adv_{\text{NE}}^{\text{AEAD}}(\mathcal{A}_{\text{AEAD}}) = 2 \cdot \Pr[\text{g}_{\text{AEAD}}(\mathcal{A}_{\text{AEAD}})] - 1.
\]

The game samples a challenge bit \( b \), and \( \mathcal{A}_{\text{AEAD}} \) is required to guess it. Adversary \( \mathcal{A}_{\text{AEAD}} \) is given the group oracles \( G \), encryption oracle \( \text{Enc} \), and decryption oracle \( \text{Dec} \). Among the group oracles, \( \text{NewHonGroup} \) creates new groups with honestly generated \( \text{NE} \) keys, \( \text{ExposeGroup} \) reveals the keys of existing groups, and \( \text{NewCorrGroup} \) instantiates new corrupt groups under \( \text{NE} \) keys of \( \mathcal{A}_{\text{AEAD}} \)’s choice. The encryption oracle \( \text{Enc} \) takes a group identifier \( g \), a nonce \( n \), and challenge messages \( m_0, m_1 \), and associated data \( ad \); it returns the encryption of \( m_b \) under the provided parameters. We require \( \text{NE} \) to be nonce-misuse resistant [RS06], meaning that...
no challenge message \(m\) is allowed to be queried across two distinct calls to \(\text{ENC}\) with respect to the same set of \(g, n, ad\). To prevent trivial wins, a corrupt group key can only be used to call \(\text{ENC}\) with \(m_0 = m_1\) and a group key cannot be exposed after it has been used in \(\text{ENC}\) with \(m_0 \neq m_1\). The decryption oracle \(\text{DEC}\) takes \(g, n, c, ad\) as input and decrypts this to the corresponding plaintext \(m\). Following the all-in-one style of [Shr04,RS06], it returns \(\bot\) if \(b = 0\), and it returns \(m\) otherwise. To prevent trivial wins, this oracle never decrypts a ciphertext with an exposed group’s key, and it never decrypts ciphertexts previously produced by \(\text{ENC}\) (with the same \(g, c, ad\)).

\[
\text{Game } g^{\text{AEAD}}(\mathcal{A}_{\text{AEAD}}) \\
\begin{array}{l}
b \leftarrow \{0, 1\}; \quad b' \leftarrow \mathcal{A}_{\text{AEAD}}^{\text{G,Enc,Dec}}, \quad \text{Return } b = b' \\
\text{ENC}(g, n, m_0, m_1, ad) \\
\text{require } K[g] \neq \bot \quad \text{and } |m_0| = |m_1| \\
\text{require } \forall d \in \{0, 1\}, (g, n, m_d, ad) \notin N_d \\
\text{If } m_0 \neq m_1 \text{ then} \\
\quad \text{if group_is_corrupt}[g] \text{ then return } \bot \\
\quad c \leftarrow \text{NE.Enc}[K[g], n, m_0, ad] \\
\quad N_0 \leftarrow N_0 \cup \{(g, n, m_0, ad)\} \\
\quad N_1 \leftarrow N_1 \cup \{(g, n, m_1, ad)\} \\
\quad C \leftarrow C \cup \{(g, n, c, ad)\}; \quad \text{Return } c \\
\text{DEC}(g, n, c, ad) \\
\text{require } K[g] \neq \bot \quad \text{and } \neg \text{group_is_corrupt}[g] \\
\text{require } (g, n, c, ad) \notin C \\
\text{m} \leftarrow \text{NE.Dec}[K[g], n, c, ad] \\
\text{If } b = 0 \text{ then return } \bot \text{ else return } m \\
\end{array}
\]

\[
\text{Game } g^{\text{KR}}(\mathcal{A}_{\text{KR}}) \\
\begin{array}{l}
\mathcal{A}_{\text{KR}}^{\text{G,Enc,Dec,Guess}}, \quad \text{Return win} \\
\text{ENC}(g, n, m, ad) \\
\text{require } K[g] \neq \bot \\
\text{require } (g, n, m, ad) \notin N \\
\text{c} \leftarrow \text{NE.Enc}(K[g], n, m, ad) \\
\text{N} \leftarrow N \cup \{(g, n, m, ad)\}; \quad \text{Return } c \\
\text{DEC}(g, n, c, ad) \\
\text{require } K[g] \neq \bot \\
\text{m} \leftarrow \text{NE.Dec}(K[g], n, c, ad) \\
\text{Return } m \\
\text{Guess}(K) \\
\text{If } \exists y : (K[y] = K \text{ and } \\
\neg \text{group_is_corrupt}[g]) \text{ then} \\
\text{win} \leftarrow \text{true} \\
\end{array}
\]

\[
\text{NewHonGroup}(g) \quad \text{EXPOSEGROUP}(g) \quad \text{NEWCorrGroup}(g, K) \\
\text{require } K[g] = \bot \\
\text{K}[g] \leftrightarrow \{0, 1\}^\text{NE,ki} \\
\text{require } K[g] \neq \bot \quad \text{and } \neg \text{chal}[g] \\
\text{group_is_corrupt}[g] \leftarrow \text{true} \\
\text{Return } K[g] \\
\text{require } K[g] = \bot \\
\text{group_is_corrupt}[g] \leftrightarrow \text{true} \\
\text{K}[g] \leftarrow K
\]

**Fig. 4:** **Left pane:** Game defining authenticated-encryption security of a nonce-based encryption scheme NE. **Right pane:** Game defining key-recovery security of NE. **Bottom pane:** Group oracles \(G = \{\text{NewHonGroup, ExposeGroup, NewCorrGroup}\} \) that are provided to an adversary in either game, except that the **boxed** code only appears in the AEAD security game.

**Key-Recovery Security of NE.** Consider game \(g^{\text{KR}}\) of Fig. 4 for nonce-based encryption scheme NE and adversary \(\mathcal{A}_{\text{KR}}\). The advantage of \(\mathcal{A}_{\text{KR}}\) in breaking the KR security of NE is defined as \(\text{Adv}^{\text{KR}}_{\text{NE}}(\mathcal{A}_{\text{KR}}) = \Pr[g^{\text{KR}}_{\text{NE}}(\mathcal{A}_{\text{KR}})]\). Adversary \(\mathcal{A}_{\text{KR}}\) is given group oracles \(G\), encryption oracle \(\text{ENC}\), decryption oracle \(\text{DEC}\), and key-guessing oracle \(\text{Guess}\). The group oracles \(G\) match those from \(g^{\text{AEAD}}\). The encryption oracle \(\text{ENC}\) takes a group identifier \(g\), a nonce \(n\), a message \(m\), and associated data \(ad\), and returns the encryption of \(m\) under the provided parameters. For consistency with other games, we disallow querying a message \(m\) across two distinct calls to \(\text{ENC}\) with repeated \(g, n\), and \(ad\) values. The decryption oracle \(\text{DEC}\) takes \(g, n, c, ad\) as input and returns the decryption \(m\) of \(c\) under the corresponding parameters. The \(\text{Guess}\) oracle takes key \(K\) as input and sets the win flag if \(K\) is an honest group key. Note that KR security of NE is implied by its AEAD security. Meaning that for any adversary \(\mathcal{A}_{\text{KR}}\) against the KR security of NE, one can build an adversary \(\mathcal{A}_{\text{AEAD}}\) against the AEAD security of NE such that \(\mathcal{A}_{\text{AEAD}}\) wins whenever \(\mathcal{A}_{\text{KR}}\) does. Intuitively, if \(\mathcal{A}_{\text{KR}}\) successfully guesses an honest key then \(\mathcal{A}_{\text{AEAD}}\) can use that key to decrypt challenge ciphertexts previously returned by its \(\text{ENC}\) oracle. The decryption correctness of NE guarantees that \(\mathcal{A}_{\text{AEAD}}\) would be able to deduce the challenge bit.
A symmetric signcryption scheme corrupt users with oracle \text{ExposeUser} reveals signing keys of existing users, and oracle \text{NewCorrUser} instantiates new corrupt users with DS key pairs of \text{SS}’s choice. Adversary \text{SS} is also given signing oracle \text{Sign} that on input \((u, m)\) signs message \(m\) with the signing key of user \(u\). The game requires \text{SS} to forge a signature \(s\) for any user-message pair \((u, m)\) with honest user \(u\) such that \(s\) was not previously returned in response to a \text{Sign}(u, m) query.

### 3 Symmetric Signcryption

In this section, we define syntax and security for multi-user symmetric signcryption. In symmetric signcryption, a user encrypts messages using their signing key and a symmetric key shared by a group of users. We want that nobody outside a group can learn what messages are being encrypted, and nobody at all can forge a message as having come from someone other than themself.

#### Syntax

A symmetric signcryption scheme \text{SS} specifies algorithms \text{SS.UserKg}, \text{SS.SigEnc}, \text{SS.VerDec}, where \text{SS.VerDec} is deterministic. The syntax used for the algorithms of \text{SS} is given in Fig. 6. Associated to \text{SS} is a group-key length \text{SS.gkl} \in \mathbb{N}, a nonce space \text{SS.NS}, a plaintext space \text{SS.MS} \subseteq \{0, 1\}^*, and an associated-data space \text{SS.AD}. The user’s key generation algorithm \text{SS.UserKg} returns a key pair \((sk, vk)\) where \(sk\) is a signing key and \(vk\) is the corresponding verification key. The signcryption algorithm \text{SS.SigEnc} takes a group’s identifier \(g \in \{0, 1\}^*\) and its symmetric key \(K_g \in \{0, 1\}^{\text{SS.gkl}}\), a sender’s identifier \(u \in \{0, 1\}^*\) and its signing key \(sk_u\), a nonce \(n \in \text{SS.NS}\), a plaintext \(m \in \text{SS.MS}\), and associated data \(ad \in \text{SS.AD}\); it returns a signcryption ciphertext \(c\). The deterministic unsigncryption algorithm \text{SS.VerDec} takes \(g, K_g, u, vk_u, n, c, ad\), where \(vk_u\) is the verification key of the sender \(u\); it returns a plaintext \(m \in \{0, 1\}^* \cup \{\bot\}\), where \(\bot\) indicates a failure to recover a plaintext. We say that \text{SS} is deterministic if \text{SS.SigEnc} is deterministic.

#### Strong Unforgeability of Digital Signatures

Consider game \(g^{\text{SUFCMA}}\) of Fig. 5 for digital signature scheme \text{DS} and adversary \(A_{\text{SUFCMA}}\). The advantage of \(A_{\text{SUFCMA}}\) in breaking the SUFCMA security of \text{DS} is defined as \(\text{Adv}^{\text{SUFCMA}}_{\text{DS}}(A_{\text{SUFCMA}}) = \text{Pr}[g^{\text{SUFCMA}}_{\text{DS}}(A_{\text{SUFCMA}})]\). Adversary \(A_{\text{SUFCMA}}\) is given access to user oracles \(U\), where oracle \text{NewHonUser} creates new users with an honestly generated \text{DS} key pairs, oracle \text{ExposeUser} reveals signing keys of existing users, and oracle \text{NewCorrUser} instantiates new corrupt users with \text{DS} key pairs of \text{SS}’s choice. Adversary \(A_{\text{SUFCMA}}\) is also given signing oracle \text{Sign} that on input \((u, m)\) signs message \(m\) with the signing key of user \(u\). The game requires \(A_{\text{SUFCMA}}\) to forge a signature \(s\) for any user-message pair \((u, m)\) with honest user \(u\) such that \(s\) was not previously returned in response to a \text{Sign}(u, m) query.

![Fig. 5: Game defining strong unforgeability of a digital signature scheme DS, where U = \{NewHonUser, ExposeUser, NewCorrUser\}.](image-url)

![Fig. 6: Syntax of a symmetric signcryption scheme SS.](image-url)
Correctness. The decryption correctness of a symmetric signcryption scheme SS requires that for all \( g \in \{0, 1\}^* \), all \( K_g \in \{0, 1\}^{SS-KG} \), all \( u \in \{0, 1\}^* \), all \( (sk_u, vk_u) \in \{SS.UserKg, all n \in SS.NS, all m \in SS.MS, all ad \in SS.AD, and all c \in SS.SigEnc(g, K_g, u, sk_u, n, m, ad) \) the following holds: \( SS.VerDec(g, K_g, u, vk_u, n, c, ad) = m \)

3.1 In-Group Unforgeability

The strongest variant of in-group unforgeability requires that an attacker cannot modify anything about ciphertexts. We also capture weaker variants. For example, the SealPacket encryption algorithm in Keybase (as defined in Section 5) uses a signing key to bind its ciphertexts to a group’s symmetric key but not to a group’s identifier. So we parameterize our security definition to capture the type of authenticity that is as restrictive as possible except for allowing (what can be described as) cross-group forgeries.

IUF Game: The Basics. Consider game \( G^{IUF} \) of Fig. 7, defined for symmetric signcryption scheme SS, ciphertext-triviality predicate \( pred_{auth}^{trivial} \), and adversary \( A_{IUF} \). The advantage of \( A_{IUF} \) in breaking the IUF security of SS is defined as \( Adv_{SS, pred_{auth}^{trivial}}^{IUF} (A_{IUF}) = Pr_{C \leftarrow SS.SigEnc(g, K_g, u, sk_u, n, m, ad)} [C \leftarrow SS.VerDec(g, K_g, u, vk_u, n, c, ad) = m] \)

IUF Game: User and Group Oracles. Among user oracles, NEWHONUSER creates honest users with honestly generated signing keys, NEWCORRUSER creates corrupt users with malicious signing keys, and EXPOSEUSER exposes the signing keys of existing users. Among group oracles, NEWHONGROUP creates honest groups with honestly sampled symmetric keys, NEWCORRGROUP creates corrupt groups with malicious symmetric keys, and EXPOSEGROUP exposes symmetric keys of existing groups. Oracles NEWHONGROUP and NEWCORRGROUP take as input a set users identifying the new group’s users; the encryption and decryption oracles then disallow queries that match a group to a non-member user. The user and group oracles use tables user_is_corrupt and group_is_corrupt to keep track of the users and groups whose keys are not secure, respectively. The IUF game never checks group_is_corrupt, deliberately giving the adversary full control over group keys.

IUF Game: The SigEnc Oracle. The encryption oracle \( SS.SigEnc \) takes \((g, u, n, m, ad)\) and returns a ciphertext \( c \) that is produced by running \( SS.SigEnc(g, K[g], u, sk[u], n, m, ad) \). Here note that the group and user keys \( K[g] \) and \( sk[u] \) are the only two inputs to \( SS.SigEnc \) that are not directly chosen by the adversary at the moment of querying the SigEnc oracle. At the end of each SigEnc query, the set \( C \) is updated to add the tuple \( ((g, u, n, m, ad), c) \) that can be interpreted as containing the input-output transcript of this query.

IUF Game: The VerDec Oracle. The decryption oracle \( SS.VerDec \) takes \((g, u, n, c, ad)\) and returns the message \( m \) that is recovered by running \( SS.VerDec(g, K[g], u, vk[u], n, c, ad) \). Keys \( K[g], sk[g] \) are the only inputs to \( SS.VerDec \) not directly chosen by the adversary. If \( m \neq \bot \), then the oracle determines if the current Oracle query is a valid forgery and sets the win flag if so. In particular, \( SS.VerDec \) builds the tuple \( z = ((g, u, n, m, ad), c) \) with all input and output values of the current decryption query. It checks \( z \) against the set \( C \) that contains the input-output behavior of all the prior encryption queries. If \( z \) is determined to be trivially obtainable from the information in \( C \), then \( SS.VerDec \) exits early (with \( m \) as its output value); otherwise, it sets the win flag. This check is performed by the ciphertext-triviality predicate \( pred_{auth}^{trivial} \). We will describe the syntax and the sample variants of \( pred_{auth}^{trivial} \) below.

Ciphertext-Triviality Predicates. The IUF game is parameterized by ciphertext-triviality predicate \( pred_{auth}^{trivial} \) (we will also parameterize the OAE game with \( pred_{auth}^{trivial} \)). Predicate \( pred_{auth}^{trivial} \) takes a tuple \( z = ((g, u, n, m, ad), c) \) and a set \( C \) as input, where \( C \) contains tuples of the same format. Here \( z \) describes the input-output values of the current query to \( SS.VerDec \) oracle and each element of \( C \) contains an input-output transcript of a prior SigEnc oracle query. Predicate \( pred_{auth}^{trivial} \) returns true if \( z \) is considered to be trivially forgeable based on the information in \( C \) and false otherwise.

In Fig. 8 we define several ciphertext-triviality predicates. Predicate \( pred_{auth}^{trivial} \) checks if \( z \in C \), capturing the strongest possible level of authenticity. This requires that only prior outputs of SigEnc can be successfully queried to the VerDec oracle; any other successful decryption query causes the adversary to...
### Game $G_{\text{corr}}^{\text{UIF}}$ (A$_{\text{UIF}}$)

\[
\begin{align*}
\text{Return } & \text{win} \\
\text{SIGENC}(g, u, n, m, ad) & \text{require } K[g] \neq \perp \\
& \text{require } sk[u] \neq \perp \text{ and } u \in \text{members}[g] \\
c & \leftarrow SS.SigEnc(g, K[g], u, sk[u], n, m, ad) \\
C & \leftarrow C \cup \{(g, u, n, m, ad), c\} \\
& \text{Return } c \\
\text{VERDEC}(g, u, n, c, ad) & \text{require } K[g] \neq \perp \\
& \text{require } vk[u] \neq \perp \text{ and } u \in \text{members}[g] \\
m & \leftarrow SS.VerDec(g, K[g], u, vk[u], n, c, ad) \\
& \text{If } m = \perp \text{ then return } \perp \\
z & \leftarrow ((g, u, n, m, ad), c) \\
& \text{If } \text{pred}\_\text{auth}(z, C) \text{ then return } m \\
& \text{If } \neg \text{user}\_\text{is}\_\text{corrupt}[u] \text{ then win } \leftarrow \text{true} \\
& \text{Return } m
\end{align*}
\]

### Game $G_{\text{SS pred\_trivial \_func\_out}}^{\text{OAE}}$ (A$_{\text{OAE}}$)

\[
\begin{align*}
b & \leftarrow \{0, 1\}; \quad b' & \leftarrow A_{\text{OAE}}^{\text{U}, \text{G}, \text{SSEnc}, \text{VerDec}} \\
& \text{Return } b = b'
\end{align*}
\]

### Fig. 7: Left pane: Game defining in-group unforgeability IUF of a symmetric signcryption scheme SS with respect to a ciphertext-triviality predicate \text{pred}\_\text{trivial}. Right pane: Game defining out-group authenticated-encryption security OAE of SS with respect to a ciphertext-triviality predicate \text{pred}\_\text{trivial} and an output-guarding function \text{func}\_\text{out}. Bottom pane: User oracles U = \{NewHonUser, ExposeUser, NewCorrUser\} and group oracles G = \{NewHonGroup, ExposeGroup, NewCorrGroup\} that are provided to an adversary in either game, except that the \boxed{boxed} code only appears in the OAE security game.
The user and group oracles in the OAE game are defined in the all-in-one style of [Shr04,Rog04], where an adversary can learn the challenge bit by forging a ciphertext to its decryption oracle. The game is defined as

\[
\text{pred}_{\text{trivial}}^\text{OAE}(z, C) \quad \text{Return } z \in C
\]

\[
\text{pred}_{\text{trivial}}^\text{OAE}(g, u, n, m, ad), c \quad \text{Return } \exists c': ((g, u, n, m, ad), c') \in C
\]

\[
\text{pred}_{\text{except-user}}^\text{OAE}(z, C) \quad \text{Return } \exists g': ((g', u, n, m, ad), c) \in C
\]

\[
\text{pred}_{\text{except-group}}^\text{OAE}(z, C) \quad \text{Return } \exists (g, u, n, m, ad), c \in C
\]

The strongest version of the out-group AE security requires that an attacker outside a chat group can win the IUF game. This predicate can be thought of as making the IUF game capture the “strong” unforgeability of ciphertexts in our group setting. One could capture existential unforgeability by considering the predicate \(\text{pred}_{\text{trivial}}^\text{OAE}\) that does not allow the adversary to win by merely producing new ciphertexts that decrypt to some tuple \((g, u, n, m, ad)\) previously queried to \(\text{SigEnc}\). Predicates \(\text{pred}_{\text{except-group}}^\text{OAE}\) and \(\text{pred}_{\text{except-user}}^\text{OAE}\) capture the authenticity of schemes where a ciphertext encrypting \((g, u, n, m, ad)\) is not bound to the group’s identifier or the user’s identifier, respectively. We use \(\text{pred}_{\text{trivial}}^\text{OAE}\), \(\text{pred}_{\text{except-group}}^\text{OAE}\), and \(\text{pred}_{\text{except-user}}^\text{OAE}\) in our security analysis of Keybase. In this work, we do not use the IUF game – we need it for OAE.

3.2 Out-Group Authenticated Encryption

The strongest version of the out-group AE security requires that an attacker outside a chat group can neither learn any information about the exchanged messages nor modify the exchanged ciphertexts in any way. We also capture weaker variants of this security notion. For example, the SealPacket encryption algorithm (as defined in Section 5) does not use a group’s symmetric key to explicitly bind its ciphertexts to a user’s signing key or a user’s identifier when used in isolation. So we capture a variant of out-group AE security that is as restrictive as possible except for allowing an attacker to violate the sender’s authenticity within any particular group.

OAE Game: The Basics. Consider game \(g_{\text{OAE}}\) of Fig. 7 for symmetric signcryption scheme \(SS\), ciphertext-triviality predicate \(\text{pred}_{\text{trivial}}\), output-guarding function \(\text{func}_{\text{out}}\), and adversary \(A_{\text{OAE}}\). The advantage in breaking the OAE security of \(SS\) is defined as

\[
\text{Adv}^\text{OAE}_{SS,\text{pred}_{\text{trivial}},\text{func}_{\text{out}}}(A_{\text{OAE}}) = \text{Pr}[g_{\text{OAE}}(\text{Adv}^\text{OAE}_{SS,\text{pred}_{\text{trivial}},\text{func}_{\text{out}}}(A_{\text{OAE}}))]
\]

Adversary \(A_{\text{OAE}}\) is given access to user and group oracles \(U\) and \(G\) and to the encryption and decryption oracles \(\text{SigEnc}\) and \(\text{VerDec}\). The goal of the adversary is to guess the challenge bit \(b\). Our security game is defined in the all-in-one style of [Shr04,Rog04], where an adversary can learn the challenge bit by foraging a ciphertext to its decryption oracle.

OAE Game: User and Group Oracles. The user and group oracles in the OAE game are defined as in the IUF game, except it does not allow calling the EXPOSEGROUP oracle to expose the key of a group that was previously used for a left-or-right challenge-encryption query (as explained below). The OAE game never checks the contents of \texttt{user_is_corrupt}, deliberately giving the adversary full control over user keys.

OAE Game: The SigEnc Oracle. The encryption oracle \(\text{SigEnc}\) takes \((g, u, n, m_0, m_1, ad)\) and returns a ciphertext \(c\) by running \(SS,\text{SigEnc}(g, K[g], u, sk[u], n, m_b, ad)\). The group and user keys \(K[g]\) and \(sk[u]\) are the only inputs to \(SS,\text{SigEnc}\) not directly chosen by the adversary querying the \(\text{SigEnc}\) oracle (and the encrypted message \(m_b\) depends on the challenge bit). The \(\text{SigEnc}\) query requires that \(m_0 = m_1\) and will only use insecure group keys for non-challenge encryptions (i.e. for \(m_0 = m_1\)). This \(\text{SigEnc}\) oracle captures nonce-misuse resistance [RS06], using the sets \(N_d\) to prevent trivial wins. At the end of \(\text{SigEnc}\) queries, the set \(C\) is updated to add the tuple \(((g, u, n, m_b, ad), c)\), and the set \(Q\) is updated to add the tuple \(((g, u, n, m_0, m_1, ad), c)\). Here the \(Q\) set can contain the input-output “transcript” of \(\text{SigEnc}\) queries from the adversary’s point of view, whereas the set \(C\) is more informative because it contains the message that was actually encrypted. We will explain the purpose of these sets below.
**OAE Game:** The **VerDec Oracle**. The decryption oracle VerDec takes \((g, u, n, c, ad)\) and returns the message \(m\) output by SS.VerDec\((g, K[g], u, vk[u], n, c, ad)\). Keys \(K[g], sk[g]\) are the only inputs to SS.VerDec not directly chosen by the adversary querying the VerDec oracle. The VerDec oracle disallows queries with corrupt group keys; if an adversary knows a group’s key then it can decrypt ciphertexts for the group on its own. If SS.VerDec recovers a non-⊥ message \(m\) and the end of the VerDec oracle is reached, then the challenge bit is meant to be revealed through returning \(m\) if \(b = 1\) and ⊥ otherwise. However, this intuition is not precise; it depends on how VerDec responds to queries that are identified as being trivially forgeable. Similarly to how trivial forgeries were handled in the IUF game, here VerDec builds \(z = ((g, u, n, m, ad), c)\) and uses a ciphertext-triviality predicate \(\text{pred}_{\text{trivial}}\) to check \(z\) against the set \(C\) from SigEnc. If \(z\) is considered not trivially obtainable from the information in \(C\), then VerDec proceeds to its last instruction that returns ⊥ or \(m\) depending on the challenge bit. Otherwise, VerDec should return an output that does not depend on the challenge bit to prevent trivial wins. Such an output is produced by the output-guarding function \(\text{func}_{\text{out}}(z, Q)\). We now describe the syntax and variants of \(\text{func}_{\text{out}}\).

**Output-Guarding Functions.** The OAE game can be parameterized by different choices of an output-guarding function \(\text{func}_{\text{out}}\). We define \(\text{func}_{\text{out}}\) to take a tuple \(z = ((g, u, n, m, ad), c)\) and a set \(Q\) as input, where \(Q\) contains tuples with the format \(((g, u, n, m_0, m_1, ad), c)\). Here \(z\) describes the input-output values of a single query to the VerDec oracle, and each element of \(C\) specifies the input-output of a prior SigEnc oracle query. At a high level, \(z\) contains the message \(m\) that was recovered during an ongoing VerDec call, and \(m\) is the only value in \(z\), \(Q\) not necessarily known by the adversary. One might want to define VerDec to return \(m\) whenever the input is identified as a trivial forgery, but \(m\) could potentially trivially reveal the challenge bit. So one could roughly think of \(\text{func}_{\text{out}}\) as the function that should enable VerDec to return \(m\) when possible. However, it should determine – from \(z\) and \(Q\) – if \(m\) would trivially help the adversary win and then “guard” VerDec against returning this \(m\).

\[
\begin{align*}
\text{func}_{\text{out}}^\dagger(z, Q) & \quad \text{func}_{\text{out}}^\text{silence-with-}m_1[\text{pred}_{\text{trivial}}](z, Q) \\
\text{Return } ∪ & \quad \text{For each } ((g, u, n, m_0, m_1, ad), c) \in Q \text{ do} \\
& \quad \text{If } m_0 ≠ m_1 \text{ then} \\
& \quad \quad \text{If } \text{pred}_{\text{trivial}}(z, \{(g, u, n, m_0, ad), c\}) \text{ then return } m_1 \\
& \quad \quad \text{If } \text{pred}_{\text{trivial}}(z, \{(g, u, n, m_1, ad), c\}) \text{ then return } m_1 \\
& \quad ((g, u, n, m, ad), c) ← z; \text{ Return } m
\end{align*}
\]

Fig. 9: Sample output-guarding functions \(\text{func}_{\text{out}}^\dagger\) and \(\text{func}_{\text{out}}^\text{silence-with-}m_1[\text{pred}_{\text{trivial}}]\). Function \(\text{func}_{\text{out}}^\text{silence-with-}m_1\) is parameterized by a ciphertext-triviality predicate \(\text{pred}_{\text{trivial}}\).

In **Fig. 9** we define two output-guarding functions. The function \(\text{func}_{\text{out}}^\dagger\) always returns ⊥. This provides no useful information to the adversary and so captures a comparatively weaker security notion. The function \(\text{func}_{\text{out}}^\text{silence-with-}m_1[\text{pred}_{\text{trivial}}]\) is parameterized by an arbitrary ciphertext-triviality predicate \(\text{pred}_{\text{trivial}}\) and captures the following logic. For every element in \(Q\) that describes a challenge encryption (i.e. \(m_0 ≠ m_1\)) performed by SigEnc, this function checks whether \(z\) is trivially forgeable based on the information that the adversary could have learned from the corresponding response. This is checked if \(z\) would be trivially forgeable for both choices of \(b ∈ \{0, 1\}\) or only for only one choice of \(b\). The output-guarding function returns \(m_1\) when this condition passes. If no element of \(Q\) triggered the above, then the output-guarding function returns the \(m\) contained in \(z\), i.e. the actual message recovered in VerDec.

**The Use of \(\text{func}_{\text{out}}^\text{silence-with-}m_1[\text{pred}_{\text{trivial}}]\) in Our Work.** We target \(\text{func}_{\text{out}}^\text{silence-with-}m_1[\text{pred}_{\text{trivial}}]\) as the output-guarding function that provides the strongest possible security guarantees for the schemes that we analyze in this work. For every \(\text{pred}_{\text{trivial}}\) we use, \(\text{pred}_{\text{trivial}}(z, \{(g, u, n, m^*, ad), c\})\) can only be true when \(z\) contains \(m^*\). So for elements of \(Q\) with \(m_0 ≠ m_1\), only one of the two if conditions can pass, meaning it is necessary to silence the output. Otherwise, the adversary can trivially win the game by building \(z\), \(Q\) and evaluating \(\text{pred}_{\text{trivial}}\) to distinguish between \(b = 0\) or \(b = 1\). (This attack assumes the adversary can always compute
It would not suffice to use \( \text{we can build an adversary defined in Fig. 8. Let from some } \) Theorem 1. The results are in Appendix A. Note that the same result can also be proved for the signcryption scheme as defined in Fig. 10, with group-key length associated-data space \( \text{Construction 1. Our StE-based symmetric signcryption scheme achieves } \) 4.1 Sign-then-Encrypt Based Construction

Our StE-based symmetric signcryption scheme StE is shown in Fig. 10. Of it, we prove the following results.

**Construction 1.** Let DS be a digital signature scheme. Let NE be a nonce-based encryption scheme with associated-data space \( \text{NE.AD} = \{0,1\}^* \). Then StE = SIGN-THEN-ENCRYPT-SS[DS,NE] is the symmetric signcryption scheme as defined in Fig. 10, with group-key length \( \text{StE.gkl = NE.kl} \), nonce space \( \text{StE.NS} = \{0,1\}^{\text{NE.nl}} \), message space \( \text{StE.MS} = \{0,1\}^* \), and associated-data space \( \text{StE.AD} = \{0,1\}^* \).

**Theorem 1.** Let \( \text{StE = SIGN-THEN-ENCRYPT-SS[DS,NE]} \) be the symmetric signcryption scheme built from some DS, NE as specified in Construction 1. Let \( \text{pred}_{\text{trivial}} \) be the ciphertext-triviality predicate as defined in Fig. 8. Let \( \text{Ad} \) be any adversary against the IUF security of StE with respect to \( \text{pred}_{\text{trivial}} \). Then we can build an adversary \( \text{Ad}_{\text{SUFCMA}} \) against the SUFCMA security of DS such that

\[
\text{Adv}_{\text{StE-pred}_{\text{trivial}}} (\text{Ad}) \leq \text{Adv}_{\text{DS-SUFCMA}} (\text{Ad}_{\text{SUFCMA}}).
\]
Let $\func{out}_{\text{out}}$ be the output-guarding function as defined in Fig. 9. Let $A_{\text{OAE}}$ be any adversary against the OAE security of $\text{StE}$ with respect to $\text{pred}_{\text{trivial}}$ and $\func{out}_{\text{out}}$. Then we can build adversary $A_{\text{AEAD}}$ against the AEAD security of $\text{NE}$ such that
\[
\text{Adv}_{\text{StE}, \text{pred}_{\text{trivial}}, \func{out}_{\text{out}}}(A_{\text{OAE}}) \leq \text{Adv}_{\text{AEAD}}(A_{\text{AEAD}}).
\]

4.2 Encrypt-then-Sign Based Construction

We now present the formal definition of our $\text{EtS}$ construction. We omit proving the security of this construction. The IUF security proof for $\text{EtS}$ would follow similar steps as the one for $\text{StE}$. The OAE security proof would require us to introduce an additional assumption about the signature scheme.

<table>
<thead>
<tr>
<th>$\text{EtS.UserKg}$</th>
<th>$\text{EtS.VerDec}(g, K_g, u, v_ku, n, c, ad)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(sk, vk) \leftarrow \text{DS.Kg} ; \text{Return} (sk, vk)$</td>
<td>$(c_{\text{NE}}, s) \leftarrow \langle c \rangle$</td>
</tr>
<tr>
<td>$\text{EtS.SigEnc}(g, K_g, u, s_ku, n, m, ad)$</td>
<td>$m_s \leftarrow \langle g, n, c_{\text{NE}}, ad \rangle$</td>
</tr>
<tr>
<td>$\text{ad}_{\text{NE}} \leftarrow \langle u, ad \rangle$</td>
<td>If $\neg \text{DS.Ver}(v_ku, m_s)$ then return $\bot$</td>
</tr>
<tr>
<td>$c_{\text{NE}} \leftarrow \text{NE.Enc}(K_g, n, m, \text{ad}_{\text{NE}})$</td>
<td>$\text{ad}_{\text{NE}} \leftarrow \langle u, ad \rangle$</td>
</tr>
<tr>
<td>$m \leftarrow \text{NE.Dec}(K_g, n, c_{\text{NE}}, \text{ad}_{\text{NE}})$</td>
<td>$m \leftarrow \langle g, n, \text{cNE}, ad \rangle$</td>
</tr>
<tr>
<td>$s \leftarrow \text{DS.Sig}(sk_u, m_s) ; \text{Return} (c_{\text{NE}}, s)$</td>
<td>$\text{Return} m$</td>
</tr>
</tbody>
</table>

Fig. 11: Symmetric signcryption scheme $\text{EtS} = \text{ENCRYPT-THEN-SIGN-SS[NE, DS]}$.

Construction 2. Let $\text{NE}$ be a nonce-based encryption scheme with associated-data space $\text{NE.AD} = \{0, 1\}^*$. Let $\text{DS}$ be a digital signature scheme. Then $\text{EtS} = \text{ENCRYPT-THEN-SIGN-SS[NE, DS]}$ is the symmetric signcryption scheme as defined in Fig. 11, with group-key length $\text{EtS.gkl} = \text{NE.kl}$, nonce space $\text{EtS.NS} = \{0, 1\}^{\text{NE.kl}}$, message space $\text{EtS.MS} = \{0, 1\}^*$, and associated-data space $\text{EtS.AD} = \{0, 1\}^*$.

OAE Security of $\text{EtS}$ Requires Unique Signatures. At the beginning of this section, we claimed that proving the OAE security of $\text{EtS}$ with respect to $\text{pred}_{\text{trivial}}$ and $\func{out}_{\text{out}}$ would require the assumption that the signature scheme produces unique signatures. Here we provide the intuition for why this assumption is needed. Consider a digital signature scheme that does not produce unique signatures. Then the adversary could expose an honest user in the OAE security game (or create a corrupt user with the same signing key as that of another honest user), and use their signing key to produce a new signature over an existing, honestly generated ciphertext. This attack can only be mitigated if $\text{DS}$ produces unique signatures. As digital signature schemes standardized in NIST’s FIPS 186-5 standard [CMRR23] (including the RSA DSA, ECDSA, and EDDSA) do not guarantee unique signatures, we do not analyze the OAE security of the $\text{EtS}$ construction in this work.

5 Modeling Keybase Chat Encryption as Symmetric Signcryption

We analyze the security of the cryptographic algorithm $\text{BoxMessage}$ that Keybase uses to encrypt and authenticate chat messages from a sender to a group. $\text{BoxMessage}$ combines multiple cryptographic primitives to offer end-to-end encrypted messaging. In particular it uses $\text{XSalsa20-Poly1305}$, $\text{SHA-256}$, $\text{SHA-512}$, and $\text{Ed25519}$ as building blocks. Within $\text{BoxMessage}$, the $\text{SealPacket}$ subroutine encrypts and authenticates message headers. We show the pseudocode for these algorithms in Fig. 12 and visual depictions of their data flows in Fig. 13. We omit the decryption algorithms $\text{BoxMessage.VerDec}$ and $\text{SealPacket.VerDec}$ from Fig. 12 as Keybase’s implementation of these algorithms follows naturally from the corresponding
The `BoxMessage.SigEnc(g, K_g, u, sk_u, n, m_body, ad)` algorithm accepts the following inputs – group’s identifier `g`, symmetric group key `K_g`, sender identifier `u`, sender signing key `sk_u`, nonce `n = (n_body, n_header)`, message `m_body`, and associated data `ad`. It performs the following steps. First it calls `XSalsa20-Poly1305.Enc(K_g, n_body, m_body)` to encrypt `n_body` using key `K_g` and nonce `n_body`, and obtains the ciphertext `c_body`. It builds header plaintext `m_header` as `(ad, u, g, h_body)` (a unique encoding of `ad`, `u`, `g`, and hash `h_body = SHA-256(n_body || c_body)`). It then invokes `SealPacket.SigEnc(g, K_g, u, sk_u, n_header, m_header, ε)` to encrypt `m_header` using `sk_u`, `K_g`, and `n_header`, and obtains the ciphertext `c_header`. Finally, it returns `(c_body, c_header)`. To decrypt ciphertext `(c_body, c_header)`, the algorithm `BoxMessage.VerDec` (not shown) ensures that `c_header` decrypts into the header plaintext `m_header` that is equal to the unique string `(ad, u, g, h_body)` composed from the inputs of `BoxMessage.VerDec`. In Keybase, the sender identifier `u` is their username and the group identifier `g` is constructed canonically from the usernames of the group members.

Fig. 12: The `BoxMessage` and `SealPacket` algorithms used in Keybase for encrypting chat messages from a user to a group. Here `g` is the group’s identifier, `K_g` is the symmetric key shared by all group members, `u` is the sender’s identifier, and `sk_u` is the sender’s signing key.

Fig. 13: The `BoxMessage`/`BM` (Left) and `SealPacket`/`SP` (Right) algorithms used to encrypt chat messages and message headers in Keybase. We omit the inputs `g`, `u`, and `ad` from `SP` as it does not use `g` or `u` and `ad = ε` always holds.

SigEnc algorithms. We define the `VerDec` algorithms explicitly in our formalization of `BoxMessage` and `SealPacket`.

To formalize the security of `BoxMessage`, it is crucial to first identify the formal primitive underlying this algorithm and the security goals it aims to achieve. None of the existing primitives in literature seem to aptly model this object, but it is naturally captured by the symmetric signcryption primitive that we defined in Section 3. Similarly, `SealPacket` can also be modeled as a symmetric signcryption scheme from which `BoxMessage` is built. In this section, we present modular constructions that cast `BoxMessage` and `SealPacket` as symmetric signcryption schemes. We first provide a general overview of the two algorithms.

The `BoxMessage` Chat-Encryption Algorithm. The `BoxMessage.SigEnc` algorithm accepts the following inputs – group’s identifier `g`, symmetric group key `K_g`, sender identifier `u`, sender signing key `sk_u`, nonce `n = (n_body, n_header)`, message `m_body`, and associated data `ad`. It performs the following steps. First it calls `XSalsa20-Poly1305.Enc(K_g, n_body, m_body)` to encrypt `n_body` using key `K_g` and nonce `n_body`, and obtains the ciphertext `c_body`. It builds header plaintext `m_header` as `(ad, u, g, h_body)` (a unique encoding of `ad`, `u`, `g`, and hash `h_body = SHA-256(n_body || c_body)`). It then invokes `SealPacket.SigEnc(g, K_g, u, sk_u, n_header, m_header, ε)` to encrypt `m_header` using `sk_u`, `K_g`, and `n_header`, and obtains the ciphertext `c_header`. Finally, it returns `(c_body, c_header)`. To decrypt ciphertext `(c_body, c_header)`, the algorithm `BoxMessage.VerDec` (not shown) ensures that `c_header` decrypts into the header plaintext `m_header` that is equal to the unique string `(ad, u, g, h_body)` composed from the inputs of `BoxMessage.VerDec`. In Keybase, the sender identifier `u` is their username and the group identifier `g` is constructed canonically from the usernames of the group members.
The SealPacket Header-Encryption Algorithm. The SealPacket algorithm accepts the same inputs as BoxMessage, except it does not take associated data $ad$ as input. We set $\text{SealPacket.AD} = \{\varepsilon\}$, meaning $ad = \varepsilon$ is always true. When $\text{SealPacket.SigEnc}$ is called from $\text{BoxMessage.SigEnc}$, it encrypts chat headers. To encrypt $m$ with nonce $n$, it starts by hashing $m$ to obtain $h = \text{SHA-512}(m)$. Then it builds an input $m_s$ to the Ed25519 signature scheme by concatenating the prefix string “Keybase-Chat-2” with the unique encoding $(K_g, n, h)$ of $K_g$, $n$, and $h$. It invokes $\text{Ed25519.Sig}$ to produce a signature $s$ over $m_s$ using the signing key $sk_u$. Finally it calls $\text{XSalsa20-Poly1305.Enc}$ to encrypt $m_c = s || m$ using the key $K_g$ and nonce $n$, and obtains the ciphertext $c$ which is returned. To decryption ciphertext $c$, the $\text{SealPacket.VerDec}$ algorithm (not shown) first recovers $m_c$ from $c$ and then parses $m_c$ to obtain $s || m$. Note that $m_c$ can be unambiguously parsed into $s || m$ because $\text{Ed25519}$ produces fixed-length signatures. Then $\text{SealPacket.VerDec}$ reconstructs $m_s$ and ensures that $s$ verifies as a valid signature for $m_s$ under the sender’s public key $vk_u$. We study the security of SealPacket in the context of the BoxMessage algorithm, but this is not the only context in which Keybase uses SealPacket. It is also used independently for the encryption of long strings and attachments. In Appendix E we detail other uses of SealPacket in Keybase.

Analysis Challenges. The descriptions of BoxMessage and SealPacket that we have given so far already present the following challenges in their analysis.

Key Reuse in BoxMessage. The same symmetric key $K_g$ is used in BoxMessage and SealPacket. This violates the principle of key separation, which says that one should always use distinct keys for distinct algorithms and modes of operation. Without context separation, this potentially allows an attacker to forward ciphertexts produced by one algorithm to another. There is no explicit context separation, so our analysis will “extract” separation by making assumptions of $\text{Ed25519}$ and using low-level details of how messages are encoded.

Cyclic Key Dependency in SealPacket. The message $m_s$ signed in SealPacket is derived from the symmetric group key $K_g$ which is also used to encrypt the signature. This produces what is known as an “encryption cycle”, a generalization of encrypting one’s own key [BRS03]. Standard AEAD security does not guarantee security when messages being encrypted depend on the key used for encryption. We use an extension of AEAD security allowing key-dependent messages and prove (in the random oracle model) that $\text{XSalsa20-Poly1305}$ achieves it for the particular key-dependent messages required.

Lack of Group/User Binding in SealPacket. By looking at the SealPacket algorithm in Fig. 12 we can see that the inputs $u$ and $g$ are never used by the algorithm. This means that a SealPacket ciphertext does not, in general, bind to the group’s or user’s identifiers. This could potentially allow a malicious user to impersonate another group member. When SealPacket is used within BoxMessage, it is always invoked on a message that contains the group’s and the user’s identifier, so the lack of group/user binding in SealPacket is not consequential there.

Nonce Repetition in Keybase. $\text{XSalsa20-Poly1305}$ is not secure when nonces repeat so our security analysis disallows nonce repetition between BoxMessage and/or SealPacket. The Keybase implementation uses uniformly random nonces, making collisions highly unlikely. Moreover, our results show that BoxMessage is robust to accidental non-uniformity in randomness as long nonces do not repeat.

We highlight the consequences of repetition across either of $n_{\text{body}}$ or $n_{\text{header}}$. The $\text{XSalsa20-Poly1305}$ authenticated encryption scheme combines the $\text{XSalsa20}$ stream cipher and the $\text{Poly1305}$ one-time message authentication code. The $\text{XSalsa20}$ stream cipher XORs plaintexts with a keystream that is deterministically derived from its key and nonce. So nonce repetition allows learning the XOR of the underlying messages. This is not an issue for twice repetition of $n_{\text{header}}$, the header plaintext $n_{\text{header}}$, is sent in the clear with ciphertexts in Keybase (a detail omitted from our abstraction), but reveals confidential information if the repetition involves a $n_{\text{body}}$. Moreover, the $\text{Poly1305}$ key is extracted from the first 32 keystream bytes. Nonce repetition results in key reuse, allowing an attacker to create forgeries for this key.

5.1 Message and Header Encryption Schemes BoxMessage and SealPacket

Message Encryption Scheme BM. Our modular symmetric signcryption construction BM models the BoxMessage chat-encryption algorithm as follows.
Our modular symmetric signcryption construction

![Fig.14: Symmetric signcryption scheme](image)

construction

![Fig.15: Symmetric signcryption scheme](image)

as the key for both

models the

Construction 3. Let \( M \subseteq \{0, 1\}^\star \). Let \( NE \) be a nonce-based encryption scheme. Let \( H \) be a hash function. Let \( SP \) be a deterministic symmetric signcryption scheme. Then \( \text{BM} = \text{BOX-MESSAGE-SS}[M, NE, H, SP] \) is the deterministic symmetric signcryption scheme as defined in Fig. 14, with message space \( \text{BM.MS} = M \) and associated-data space \( \text{BM.AD} = \{0, 1\}^\star \). We require the following. The group key taken by \( \text{BM} \) is used as the key for both \( NE \) and \( SP \), so \( \text{BM.gkl} = \text{NE.kl} = \text{SP.gkl} \). The nonce taken by \( \text{BM} \) is a pair containing a separate nonce for each of \( NE \) and \( SP \), so \( \text{BM.NS} = \{0, 1\}^{NE.nl} \times \text{SP.NS} \).

**HEADER ENCRYPTION SCHEME SP.** Our modular symmetric signcryption construction \( SP \) models the header-encryption algorithm \( \text{SealPacket} \) as follows.
Construction 4. Let \( H \) be a hash function. Let \( DS \) be a deterministic digital signature scheme. Let \( NE \) be a nonce-based encryption scheme. Then \( SP = SEAL-PACKET-SS[H, DS, NE] \) is the deterministic symmetric signcryption scheme as defined in Fig. 15, with group-key length \( SP.gkl = NE.kl \), nonce space \( SP.NS = \{0,1\}^{NE.m} \), message space \( SP.MS = \{0,1\}^* \), and associated-data space \( SP.AD = \{\varepsilon\} \).

6 Security Analysis of Keybase Chat Encryption

In this section we analyze the security of the symmetric signcryption schemes \( BM \) and \( SP \) defined in Section 5. In Section 6.1, we show the in-group unforgeability of \( BM \) and \( SP \). In Sections 6.2 and 6.3, we show the out-group AE security of \( BM \) and \( SP \). This requires us to introduce two weaker variants of the OAE security notion, one each for \( BM \) and \( SP \), by relaxing the level of nonce-misuse requirements of the OAE game defined in Fig. 7. The \( SP \) analysis requires two new security notions, \( M \)-sparsity for digital signature schemes and authenticated encryption for key-dependent messages for nonce-based encryption schemes.

6.1 In-Group Unforgeability of BoxMessage and SealPacket

In-Group Unforgeability of BoxMessage. We reduce the in-group unforgeability of the symmetric signcryption scheme \( BM = BOX-MESSAGE-SS[M, NE, H, SP] \) to the security of its underlying primitives. In particular, we reduce the IUF security of \( BM = BOX-MESSAGE-SS[M, NE, H, SP] \) to the IUF security of \( SP \) and the collision resistance of \( H \).

A BM ciphertext is a pair \((c_{\text{body}}, c_{\text{header}})\) comprising an \( NE \) ciphertext \( c_{\text{body}} \) and an \( SP \) ciphertext \( c_{\text{header}} \), which encrypts \((ad, u, g, h_{\text{body}})\). The adversary's objective is to forge a BM ciphertext by either forging \( c_{\text{body}} \) or \( c_{\text{header}} \). The adversary can use a corrupt group key \( K_g \), so \( c_{\text{body}} \) ciphertexts are easily forged. However, this does not suffice to produce a BM forgery because \( c_{\text{header}} \) encrypts the hash of \( c_{\text{body}} \). Therefore, it would need to forge a corresponding \( c_{\text{header}} \) ciphertext. The IUF security of \( SP \) prevents the adversary from forging \( c_{\text{header}} \) ciphertexts. As a result, the adversary can only reuse honestly generated \( c_{\text{header}} \) from its prior queries to SigEnc in its forgery attempts. Since an honest \( c_{\text{header}} \) effectively commits to \( ad, u, g, h_{\text{body}} \), and \( m_{\text{header}} \), using an old \( c_{\text{header}} \) to construct a new BM ciphertext requires finding a new \( NE \) nonce-ciphertext pair that hashes to the same \( h_{\text{body}} \) under \( H \). Collision resistance of \( H \) prevents this.

Theorem 2. Let \( BM = BOX-MESSAGE-SS[M, NE, H, SP] \) be the symmetric signcryption scheme built from some \( M, NE, H, SP \) as specified in Construction 3. Let \( \text{pred}_{\text{trivial}}^{\text{uf}} \) and \( \text{pred}_{\text{trivial}}^{\text{uf-exception-group}} \) be the ciphertext-triviality predicates as defined in Fig. 8. Let \( A_{\text{IUF-of-BM}} \) be any adversary against the IUF security of \( BM \) with respect to \( \text{pred}_{\text{trivial}}^{\text{uf}} \). Then we can build adversaries \( A_{\text{IUF-of-SP}} \) and \( A_{\text{CR}} \) such that

\[
\text{Adv}_{BM,\text{pred}_{\text{trivial}}^{\text{uf}}}(A_{\text{IUF-of-BM}}) \leq \text{Adv}_{SP,\text{pred}_{\text{trivial}}^{\text{uf-exception-group}}}(A_{\text{IUF-of-SP}}) + \text{Adv}_{H}^{\text{CR}}(A_{\text{CR}}).
\]

The formal proof of Theorem 2 is in Appendix B.1.

In-Group Unforgeability of SealPacket. In-group unforgeability of the symmetric signcryption scheme \( SP = SEAL-PACKET-SS[H, DS, NE] \) reduces to the security of DS and \( H \). We parameterize the IUF security of \( SP \) to aim for a relaxed version of strong unforgeability because \( SP \) ciphertexts do not directly depend on the group's identifier \( g \) (even though it depends on the group key \( K_g \)).

An SP ciphertext encrypts \( s \parallel m \) under \( K_g \). The adversary can use a corrupt \( K_g \), but forging an SP ciphertext still requires the signature \( s \). So the adversary must either forge a new signature or reuse an honest signature from a prior SigEnc query. The SUFCA security of DS prevents the former. An honest signature \( s \) is computed over “Keybase-Chat-2” \( \langle K_g, n, h \rangle \) where \( h \) is the hash of the message \( m \). Hence reusing an honest signature could use a new SP ciphertext that encrypts \( s \parallel m \) with \( K_g, n \), but the tidiness of \( NE \) prevents this. So reusing an honest signature requires finding a new message that hashes to the same \( h \) under \( H \). Collision resistance of \( H \) prevents this.
Fig. 16: A summary of the reductions that we provide for the wOAE security of SP and the bwOAE security of BM.

**Theorem 3.** Let $SP = SEAL-PACKET-SS[H, DS, NE]$ be the symmetric signcryption scheme built from some $H$, $DS$, and $NE$ as specified in Construction 4. Let $pred_{\text{trivial}}$ be the ciphertext-triviality predicate as defined in Fig. 8. Let $A_{\text{IUF-of-SP}}$ be any adversary against the IUF security of SP with respect to $pred_{\text{trivial}}$. Then we can build adversaries $A_{\text{SUFCMA}}$ and $A_{\text{CR}}$ such that

$$Adv^\text{IUF}_{SP,pred_{\text{trivial}}}(A_{\text{IUF-of-SP}}) \leq Adv^\text{SUFCMA}_{\text{DS}}(A_{\text{SUFCMA}}) + Adv^\text{CR}_H(A_{\text{CR}}).$$

The formal proof of Theorem 3 is in Appendix B.2.

### 6.2 Out-Group AE Security of BoxMessage

Out-group AE security of $BM = BOX-MESSAGE-SS[\mathcal{M}, NE, H, SP]$ reduces to the security of its underlying primitives as summarized by the rightmost arrows of Fig. 16. At a high level, we show that BM achieves a variant of OAE security (bwOAE) if SP achieves another variant of OAE security (wOAE) and $H$ is collision-resistant. Because $NE = XSalsa20-Poly1305$ in Keybase (which is not nonce-misuse resistant), both variants disallow nonce repetition.

**Intuition.** An BM ciphertext is a pair $(c_{body}, c_{header})$ consisting of an NE ciphertext $c_{body}$ and an SP ciphertext $c_{header}$. One way the adversary could learn the challenge bit is by querying its VERDEC oracle on a forged BM ciphertext that decrypts successfully. In order to accomplish that, the adversary must forge the underlying SP ciphertext $c_{header}$ or reuse an honestly generated $c_{header}$. The former is prevented by the out-group AE security of SP. The latter is prevented by the collision resistance of $H$ because of the following. An honestly generated $c_{header}$ effectively commits to $ad$, $u$, $g$, $h_{body}$, and $h_{header}$. In order to reuse $c_{header}$, an adversary must find a new NE nonce-ciphertext pair that hashes to $h_{body}$, hence producing a collision. It follows that the VERDEC oracle is essentially useless to the adversary; it can only serve to decrypt non-challenge ciphertexts that were previously returned by SigEnc. So it remains to show that the adversary cannot learn the challenge bit solely based on the BM ciphertexts that it receives from SigEnc. For any ciphertext $(c_{body}, c_{header})$ returned by SigEnc, the SP ciphertext $c_{header}$ encrypts a hash of $c_{body}$ but otherwise does not depend on the challenge bit. So the adversary gains no advantage from observing $c_{header}$. Finally, the AEAD security of NE guarantees that $c_{body}$ does not reveal the challenge bit.

Because the header encryption scheme SP and the body encryption scheme NE use the same symmetric key $K_g$, we require integrity of SP ciphertexts produced using $K_g$ hold even when the adversary can obtain other NE encryptions under the same key. Similarly, the NE ciphertexts generated using the symmetric key $K_g$ should be indistinguishable even when the adversary can obtain SP encryptions and decryptions under the same key. We introduce a variant of the OAE game in Definition 3 to capture these joint requirements.

**Restrictions on Nonce Misuse in BM and SP.** We now define new variants of out-group AE security for our analysis of Keybase. The BM and SP schemes in Keybase are built from the authenticated encryption scheme XSalsa20-Poly1305 that is not nonce-misuse resistant so we modify the OAE game to disallow nonce repetition. We start with wOAE security for SP.
Definition 1. Let SS be a symmetric signcryption scheme. Consider the OAE security game for SS of Fig. 7 (w.r.t. any $\text{pred}_{\text{trivial}}^{\text{sec}}, \text{func}_{\text{out}}^{\text{sec}}$). We define a new variant of this game as follows. The instruction preventing nonce misuse

\[
\text{require } \forall d \in \{0, 1\}, (g, u, n, m_d, ad) \notin N_d \quad \text{is replaced with} \quad \text{require } (g, n) \notin N.
\]

In addition, the instructions updating the nonce set

\[
N_0 \leftarrow N_0 \cup \{(g, u, n, m_0, ad)\} \quad \text{are replaced with} \quad N \leftarrow N \cup \{(g, n)\}.
\]

We denote the resulting game (and security notion) by $\text{bwOAE}$. It is a weak variant of OAE that does not require nonce-misuse resistance. We define an adversary’s advantage in breaking the $\text{bwOAE}$ security of SS in the natural way.

Now we define $\text{bwOAE}$ security for BM. The nonce of BM is a pair of two separate nonces $n = (n_{\text{body}}, n_{\text{header}})$. The $\text{bwOAE}$ security game independently applies the group-nonce uniqueness condition introduced in Definition 1 to each of $(g, n_{\text{body}})$ and $(g, n_{\text{header}})$, and it also requires that $n_{\text{body}} \neq n_{\text{header}}$. This is necessary because BM calls $\text{NE.Enc}$ on $(g, n_{\text{body}})$, and SP calls $\text{NE.Enc}$ on $(g, n_{\text{header}})$. In Keybase both NE schemes are $\text{XSalsa20-Poly1305}$ using the same key.

Definition 2. Let $\mathcal{X}, \mathcal{Y}$ be any sets. Let SS be a symmetric signcryption scheme with the nonce space $SS.NS = \mathcal{X} \times \mathcal{Y}$. Consider the OAE security game for SS of Fig. 7 (w.r.t. any $\text{pred}_{\text{trivial}}^{\text{sec}}, \text{func}_{\text{out}}^{\text{sec}}$). We define a new variant of this game as follows. The instruction preventing nonce misuse

\[
\text{require } \forall d, (g, u, n, m_d, ad) \notin N_d \quad \text{is replaced with} \quad (n_{\text{body}}, n_{\text{header}}) \leftarrow n
\]

\[
\text{If } n_{\text{body}} = n_{\text{header}} \text{ then return } \bot
\]

\[
\text{If } (g, n_{\text{body}}) \in N \text{ then return } \bot
\]

\[
\text{If } (g, n_{\text{header}}) \in N \text{ then return } \bot.
\]

In addition, the instructions updating the nonce set

\[
N_0 \leftarrow N_0 \cup \{(g, u, n, m_0, ad)\} \quad \text{are replaced with} \quad N \leftarrow N \cup \{(g, n_{\text{header}})\}
\]

\[
N_1 \leftarrow N_1 \cup \{(g, u, n, m_1, ad)\} \quad \text{are replaced with} \quad N \leftarrow N \cup \{(g, n_{\text{body}})\}.
\]

We denote the resulting game (and security notion) by $\text{bwOAE}$. Beyond being defined for SS with a bipartite nonce space, this variant of OAE is weak in that it does not require nonce-misuse resistance. We define an adversary’s advantage in breaking the $\text{bwOAE}$ security of SS in the natural way.

The Joint Security Required of SP and NE. Here we define the security notion required from SP when it is used in the presence of arbitrary NE encryptions under the same symmetric group keys that are used by SP. We call this notion $\text{wOAE[Enc][M, NE]}$. It is a parameterized version of the $\text{wOAE}$ game defined in Definition 1. We use it for our analysis of the $\text{bwOAE}$ security of BM.

At the start of this section, we discussed that the security reduction for BM intuitively requires that it is hard to forge an SP ciphertext (without knowing the corresponding group key $K_y$) in the presence of NE encryptions. Our definition of $\text{wOAE[Enc][M, NE]}$ captures this by providing the adversary access to an NE encryption oracle ENC in addition to the SigEnc and VerDec oracles in the out-group AE security game of SP. We stress that proving the security of BM does not, in principle, require us to provide the SigEnc oracle to the adversary. We choose to require this stronger level of security from SP because of the following reasons. On the one hand, in Section 5 we explained why it is beneficial to prove that SP satisfies a strong security notion, going beyond what is required by BM. On the other hand, this stronger security notion that we require from SP will not come at the cost of introducing additional assumptions or achieving looser concrete-security bounds in our analysis of BM.

Definition 3. Let SS be a symmetric signcryption scheme. Let $\mathcal{M} \subseteq \{0, 1\}^*$. Let NE be a nonce-based encryption scheme. Consider the $\text{wOAE}$ security game for SS as defined in Definition 1 (w.r.t. any $\text{pred}_{\text{trivial}}^{\text{sec}}, \text{func}_{\text{out}}^{\text{sec}}$). We define a variant of this game by adding an oracle that is defined as follows.
ENC[M, NE](g, mbody, mbody, 0, mbody, 1)
require K[g] ≠ ⊥ and mbody, 0 = mbody, 1
require (g, mbody) ≠ N and mbody, 0, mbody, 1 ∈ M
If mbody, 0 ≠ mbody, 1 then
  If group_is_corrupt[g] then return ⊥
  char[g] ← true
  cbody ← NE.Enc(K[g], mbody, mbody, b)
  N ← N ∪ {(g, mbody)} ; Return cbody

It shares set N, bit b, and the tables K, group_is_corrupt, char with the rest of the security game. We denote the resulting game (and security notion) by wOAE[ENC[M, NE]]. It simultaneously requires out-group AE security of SS (without nonce repetition) and an IND-style security of NE. We define an adversary’s advantage in breaking this security notion in the natural way.

Note that we require the messages that the adversary queries to the Enc oracle to be in M. Intuitively, in our security analysis of BM, an adversary will only be able to obtain NE encryptions of messages in the message space of BM. So in the security reduction for BM we will use M = BM.MS.

**OUT-GROUP AE SECURITY OF BoxMessage.** We prove bwOAE security of BM.

**Theorem 4.** Let BM = BOX-MESSAGE-SS[M, NE, H, SP] be the symmetric signcryption scheme built from some M, NE, H, SP as specified in Construction 3. Let pred_{trivial} and pred_{trivial-exception-user} be the ciphertext-triviality predicates as defined in Fig. 8. Let func_{out} be the output-guarding functions as defined in Fig. 9. Let wOAE[ENC[M, NE]] be the security notion as defined in Definition 3. Let A_{bwOAE-of-BM} be any adversary against the bwOAE security of BM with respect to pred_{trivial} and func_{out}. Then we build adversaries A_{bwOAE-of-SP} and A_{CR} such that

\[
\text{Adv}_{\text{bwOAE}}^{\text{BM}, \text{pred}_{\text{trivial}}, \text{func}_{\text{out}}} (A_{\text{bwOAE-of-BM}}) \leq \text{Adv}_{\text{bwOAE}}^{\text{ENC}[M, NE]} \left( A_{\text{bwOAE-of-SP}} \right)
+ \text{Adv}_{\text{CR}}^{\text{H}} (A_{\text{CR}}).
\]

The formal proof of Theorem 4 is in Appendix C.4.

Note that we prove the OAE security of BM with respect to pred_{trivial} and func_{out}. As discussed in Section 3, pred_{trivial} essentially requires BM to have ciphertext integrity. Our result relies on the security of SP with respect to pred_{trivial-exception-user} and func_{out}. Recall that pred_{trivial-exception-user} basically requires SP to have ciphertext integrity, except it allows for an honest ciphertext to be successfully decrypted even with respect to a wrong user identifier; the latter is not considered a “valid” forgery. This does not translate to an attack against BM because it only uses SP to encrypt header messages m\_header = (ad, u, g, h\_body) that contain u, and the BM.VerDec algorithm verifies that the group identifier it received as input matches the one that was parsed from m\_header.

### 6.3 Out-Group AE Security of SealPacket

Out-group AE security of SP = SEAL-PACKET-SS[H, DS, NE] reduces to the security NE and DS (see Fig. 16). In particular, wOAE[ENC[M, NE]] security holds (for any M ⊆ \{0, 1\}^*) if NE provides authenticated encryption for key-dependent messages and DS produces M-sparse signatures. We introduce these security notions below.

**Intuition.** Recall that in the wOAE[ENC[M, NE]] security game, the adversary is provided with signcryption and unsigncryption oracles SigEnc and VerDec for SP, and an encryption oracle Enc for NE. Each of these returns output based on a challenge bit that is shared between them. The adversary can use three approaches to learn the challenge bit. It can (a) attempt SP forgeries by calling its SP decryption oracle VerDec; (b) make left-or-right queries to its NE encryption oracle Enc; (c) make left-or-right queries to its SP encryption oracle SigEnc.

The adversary is allowed to expose users’ signing keys so it could attempt to forge an SP ciphertext using an exposed DS signing key and its ENC oracle. The adversary would then query the resulting
ciphertext to its VerDec oracle in an attempt to trivially win the game. We show that the adversary is unable to accomplish this. The ENC oracle is defined to only produce encryptions of the messages from the set $\mathcal{M}$. In the implementation of Keybase, the messages from $\mathcal{M}$ have a specific encoding; we will rely on this property in our proof. In contrast, any ciphertext successfully decrypted by VerDec must encrypt a message of the form $m_e = s \parallel m$ where $s$ is a valid DS signature. So the adversary needs to find a signature $s$ that is consistent with the message encoding that is permitted by ENC. The $\mathcal{M}$-sparseness of DS signatures, which we formalize below, prevents this. It follows that the VerDec oracle does not help the adversary to win the game by querying ciphertexts that were previously returned by ENC.\(^3\)

Now we can reimage the ENC and SIGEnc oracles as producing NE encryptions of key-dependent messages. The SIGEnc oracle requires messages to be derived as a specific function of the symmetric group key $K$. The ENC oracle can be thought of as allowing messages that are derived from “constant” functions, meaning the chosen messages do not depend on $K$. We can also view the VerDec oracle as an NE decryption oracle that prevents the adversary from trivially winning the game by merely querying the ciphertexts it previously obtained from either ENC or SIGEnc. We define the AE security of NE for key-dependent messages and show that the adversary can only win the wOAE[ENC[$\mathcal{M}$, NE]] game against SP if it can win the KDMAE game against NE.

**The Strength of the Targeted Security Notion.** During the analysis of the out-group AE security of BM in Section 6.2, we discussed that the left-or-right security of SP is not required for us to prove the security of BM. Meaning that the requirement (c) listed above arose as a result of us choosing to define the wOAE[ENC[$\mathcal{M}$, NE]] game in a way that it captures a stronger security notion than strictly necessary. We note that in our wOAE[ENC[$\mathcal{M}$, NE]] security analysis of SP, case (c) follows from case (b), and does not need additional assumptions.

**Reliance on the Message Encoding in Keybase.** We mentioned in the intuition that we rely on the encoding of messages in $\mathcal{M}$ in our proof. We emphasize that avoiding this dependency is non-trivial. The cyclic key dependency within SP and the key reuse between BM and SP pose significant challenges when considering the possibility of an alternate proof.

**$\mathcal{M}$-Sparse Signatures.** Consider game $\varphi^{\text{SPARSE}}$ of Fig. 17, defined for a digital signature scheme DS, a set $\mathcal{M} \subseteq \{0, 1\}^*$, and an adversary $A^{\text{SPARSE}}$. The advantage of $A^{\text{SPARSE}}$ in breaking the $\mathcal{M}$-SPARSE security of DS is defined as $\text{Adv}^{\text{SPARSE}}_{\text{DS}, \mathcal{M}}(A^{\text{SPARSE}}) = \text{Pr}[\varphi^{\text{SPARSE}}_{\text{DS}, \mathcal{M}}(A^{\text{SPARSE}})]$. Intuitively, this game captures the inability of an adversary to produce a signature that conforms to the message space $\mathcal{M}$ even though the adversary chooses the public key used to verify the signature. More formally, the adversary wins if it can return $(vk, m, s, \gamma)$ such that $s$ verifies as a signature over the message $m$ under the verification key $vk$ and $s \parallel \gamma \in \mathcal{M}$. We stress that the adversary is allowed to choose an arbitrary – possibly malformed – verification key. The adversary is not required to know the corresponding signing key, and such a key may in fact not exist.

We verify our intuition about the $\mathcal{M}$-sparsity of the Ed25519 signature scheme underlying SP in Appendix C.1. Ed25519 is a deterministic signature scheme introduced by Bernstein, Duif, Lange, Schwabe, and Yang in [BDL+11]. It is obtained by applying the commitment-variant of the Fiat-Shamir transform to an identification scheme. Therefore a signature produced by Ed25519 consists of the commitment and response of the identification scheme. The adversary can only win the SPARSE game of Ed25519 if it is able to produce an accepting conversation transcript for the identification scheme such that the corresponding commitment conforms to $\mathcal{M}$. Commitments in the identification scheme underlying Ed25519 are elements of a prime-order group. We prove that finding such a commitment is only possible if the adversary can find a group element and its discrete logarithm such that the group element is in $\mathcal{M}$ which we show is hard in the generic group model.

**The Message Space $\mathcal{M}$.** Keybase uses the MessagePack serialization format [Fur] to encode plaintext messages. Plaintext messages are represented using a custom data structure in Keybase. So the serialized MessagePack encoding of a plaintext is a byte sequence that not only stores the plaintext itself but also some metadata about the data structure that represents it. For messages encrypted by BM, the metadata

\(^3\) The wOAE[ENC[$\mathcal{M}$, NE]] game itself also prevents the adversary from trivially winning by querying VerDec on a ciphertext that was previously returned by SigEnc.
about the data structure happens to be located in the first 17 bytes of the encoding. This means that
the encoding of every plaintext encrypted by BM contains a fixed 17-byte prefix. Let this 17-byte prefix
be \text{pre}. Then we define the message space of BM by \(BM.MS = \{\text{pre} || \nu \mid \nu \in \{0,1\}^*\}\).

**Message-Deriving Functions.** Let \(\phi\) be any function that
takes a symmetric key \(K\) as input and uses it to derive and return
some message \(m\). We call \(\phi\) a *message-deriving* function and will
consider some classes (i.e. sets) \(\Phi\) of message-deriving functions.
For simplicity, we require that the length of an output returned
by \(\phi\) must not depend on its input; we denote the output length
of \(\phi\) by \(||\phi||\).

**AE Security of NE for Key-Dependent Messages.** Consider
the game \(g^{\Phi}_{\text{KDMAE}}\) of Fig. 18, defined for a nonce-based encryption
scheme \(NE\), a class of message-deriving functions \(\Phi\), and an adversary
\(A_{\text{KDMAE}}\). The advantage of \(A_{\text{KDMAE}}\) in breaking the \(\Phi\)-KDMAE
security of \(NE\) is defined as
\[
\text{Adv}_{NE,\Phi}^{\text{KDMAE}}(A) = 2 \cdot \Pr[g^{\Phi}_{\text{KEAD}}(A)] - 1.
\]
This game can be thought of as a modification of the AEAD
security game for NE (Fig. 4) which does not require nonce-misuse
resistance. The core difference is that the \(\text{Enc}\) oracle now takes message-deriving functions \(\phi_0, \phi_1 \in \Phi\)
as input (rather than challenge messages \(m_0, m_1\)). The challenge message is derived as
\(m_0 \leftarrow \phi_0(K[g])\) for \(b \in \{0,1\}\), where \(K[g]\) is the symmetric group key associated to \(g\). Trivial attacks are prevented by
requiring that \(\phi_0, \phi_1\) have the same output length and that \(\phi_0 = \phi_1\) whenever \(\text{Enc}\) is called for a corrupt
group.

Our definition is based on prior work [BRS03,BPS07,BK11,BMT14].
There are strong impossibility results [BK11] regarding the existence of schemes that are secure with respect to very large classes
of message-deriving functions \(\Phi\). We sidestep these results by considering a very narrow and simple class
\(\Phi_{\text{SP}}\) that we define below.

**The Class of Message-Deriving Functions \(\Phi_{\text{SP}}\).** At the beginning of this section we discussed that
in the wOAE[\(\text{ENC}[M,NE]\)] security game for \(\text{SP}\), the \(\text{SigEnc}\) and \(\text{Enc}\) oracles can be thought of as
returning an \(NE\) ciphertext that encrypts an output of some message-deriving function. For example, the
\(\text{Enc}\) oracle can be expressed roughly as
\(\text{Enc}(g,n,m_0,m_1) := \text{NE.Enc}(K[g],n,\phi_{m_0}(K[g]))\) with respect to
any class \(\Phi\) that for every \(m \in \{0,1\}^*\) contains the constant function \(\phi_{m}(K) := m\) (i.e. \(\phi_{m}\) ignores its

---

**Fig. 17:** Game defining \(M\)-sparsity of a digital signature scheme DS for a set \(M\).

**Fig. 18:** Game defining authenticated-encryption security of \(NE\) for \(\Phi\)-key-dependent messages, where \(\Phi\) is
a class of message-deriving functions and \(G = \{\text{NewHonGroup}, \text{ExposeGroup}, \text{NewCorrGroup}\}\).
input $K$). We now define the class $\Phi_{SP}$ containing all message-deriving functions that are used by either SIGENC or ENC.

**Construction 5.** Let $NE$ be a nonce-based encryption scheme. Let $H$ be a hash function. Let $DS$ be a digital signature scheme. Let SIGENC-DER and ENC-DER be the parameterized message-deriving functions that are defined as follows, each taking an NE key $K \in \{0,1\}^{NE.nl}$ as input.

\[
\begin{align*}
\text{SIGENC-DER[NE,H,DS,m,n,sk]}(K) & \\
\hline
\begin{array}{l}
h \leftarrow H(m) ; \\
m_s \leftarrow \text{“Keybase-Chat-2” || (K,n,h)} \\
s \leftarrow DS.Sig(sk,m_s) ; \\
m_e \leftarrow s || m \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{ENC-DER[m]}(K) & \\
\hline
\begin{array}{l}
\text{Return } m
\end{array}
\end{align*}
\]

Then $\Phi_{SP} = \text{MSG-DER-FUNC[NE,H,DS]}$ is the class of message-deriving functions that is defined as follows.

\[
\Phi_{SP} = \{ \text{SIGENC-DER[NE,H,DS,m,n,sk]} \mid m \in \{0,1\}^* , n \in \{0,1\}^{NE.nl}, sk \in \{0,1\}^{DS.skip} \}
\]

\[
\cup \{ \text{ENC-DER[m]} \mid m \in \{0,1\}^* \}.
\]

Note that SIGENC-DER only uses $K$ as a part of the message $m_s$ signed by $DS.Sig$. Keybase instantiates $DS$ with Ed25519 which computes two SHA-512 hashes of $m_s$ (mixed with other inputs). The resulting signature does not depend on $m_s$ in any other way. Using this observation and an indifferentiability result of Bellare, Davis, and Di [BDD23] (for SHA-512 with output reduced modulo a prime) we capture SIGENC-DER as a special class of message-deriving functions for which we can prove security in the random oracle model.

**KDMAE Security for Messages Derived from a Hashed Key.** Let $H$ be a hash function. Let $\Phi$ be a class of message-deriving functions such that each $\phi \in \Phi$ on input $K$ is only allowed to derive messages from the hash value $H(K)$, and never directly from $K$. We will roughly show that every AEAD-secure nonce-based encryption scheme $NE$ is also $\Phi$-KDMAE-secure, provided that $H$ is modeled as a random oracle. We formalize this class of functions as follows.

**Definition 4.** We say $\Phi$ derives messages from a hashed key if there exists a set $\Gamma$ and a function $H$ (modeled as a random oracle) such that $\Phi = \{ \phi_\gamma \mid \phi_\gamma(\cdot) = \gamma(H(\cdot)), \gamma \in \Gamma \}$. In Appendix C.2 we show how to capture $\Phi_{SP}$ as satisfying this definition. Thereby, the following result will give us $\Phi_{SP}$-KDMAE security.

**Proposition 1.** Let $NE$ be a nonce-based encryption scheme. Let $\Phi$ be a class of message-deriving functions that derives messages from a hash key. Let $A_{KDMAE}$ be an adversary against the $\Phi$-KDMAE security of $NE$ making $q_{\text{NewHonGroup}}$ queries to its $\text{NewHonGroup}$ oracle. Then we can build adversaries $A_{KR}$ and $A_{AEAD}$ such that (in the random oracle model)

\[
\text{Adv}_{NE,\Phi}^{\text{KDMAE}}(A_{KDMAE}) \leq 2 \cdot \text{Adv}_{NE}^{\text{KR}}(A_{KR}) + \text{Adv}_{NE}^{\text{AEAD}}(A_{AEAD}) + \frac{q_{\text{NewHonGroup}}}{2^{NE.nl}}.
\]

The constructed adversaries will not repeat $(g,n)$ across ENC queries, so non-nonce-misuse resistant $NE$ suffices. To prove this, we first assert that a $\Phi$-KDMAE adversary $A_{KDMAE}$ can never directly query the random oracle on any of the (non-exposed) honest keys; otherwise, we could use $A_{KDMAE}$ to break the key-recovery security of AEAD. But then $A_{KDMAE}$ cannot distinguish between messages derived from $H(K[g])$ or from some $H^*(g)$. Here $H$ is the actual random oracle and $H^*$ is a simulated random oracle whose output depends on a group’s identifier $g$ instead of this group’s key $K[g]$. We switch from using $H(K[g])$ to $H^*(g)$, thus breaking the dependency of each challenge message on the corresponding $NE$ key. The AEAD security of $NE$ then guarantees that $A_{KDMAE}$ cannot guess the challenge bit. The formal proof of Proposition 1 is in Appendix C.3.

**Out-group AE Security of SealPacket.** We prove wOAE security of $SP$.

**Theorem 5.** Let $M \subseteq \{0,1\}^*$. Let $SP = \text{SEAL-OAE-SS}[H,DS,NE]$ be the symmetric signcryption scheme built from some $H,DS,NE$ as specified in Construction 4. Let $\text{pred}_{\text{trivial}}$ be the ciphertext-triviality predicate as defined in Fig. 8. Let $\text{func}_{\text{out}}$ be the output-guarding function as defined in Fig. 9. Let
wOAE[\text{ENC}, M, NE]] be the security notion as defined in Definition 3. Let \( \Phi_{\text{SP}} = \text{MSG-DER-FUNC}[\text{NE}, H, DS] \) be the class of message-deriving functions defined in Construction 5. Let \( A_{\text{wOAE-of-SP}} \) be an adversary against the wOAE[\text{ENC}, M, NE]] security of \( \text{SP} \) with respect to \( \text{pred}_{\text{trivial}}^{\text{wOAE-of-SP}} \) and \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \). Then we can build adversaries \( A_{\text{KDMAE}} \) and \( A_{\text{SPARSE}} \) such that
\[
\text{Adv}_{\text{SP, pred}_{\text{trivial}}^{\text{wOAE-of-SP}}, \text{func}_{\text{out}}^{\text{wOAE-of-SP}}} (A_{\text{wOAE-of-SP}}) \leq \text{Adv}_{\text{KDMAE}, \Phi_{\text{SP}}} (A_{\text{NE}}) + 2 \cdot \text{Adv}_{\text{SPARSE}, \text{M}} (A_{\text{SPARSE}}).
\]

The formal proof of Theorem 5 is in Appendix C.5.

We prove wOAE[\text{ENC}, M, NE]] security of \( \text{SP} \) with the ciphertext-triviality predicate \( \text{pred}_{\text{trivial}}^{\text{wOAE-of-SP}} \) and the output-guarding function \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \). The predicate \( \text{pred}_{\text{trivial}}^{\text{wOAE-of-SP}} \) requires ciphertext integrity, except ciphertexts do not bind to the user’s identifier. As pointed out in Section 3 and Section 6.2, the choice of this predicate follows from the level of ciphertext integrity achieved by \( \text{SP} \). The use of \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \) makes the VerDec oracle less informative. We chose \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \) despite that fact because proving security with respect to a stronger parameter like \( \text{func}_{\text{trivial}}^{\text{wOAE-of-SP}} \) comes at the cost of adding complexity to our reductions.

### 6.4 Stronger Out-Group AE Security Results

The OAE security results that we showed in Sections 6.2 and 6.3 used the output-guarding function \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \). In this section, we show that for the symmetric signcryption schemes that we consider in this work, the OAE security of a scheme with respect to \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \) implies its OAE security with respect to the stronger output guarding function \( \text{func}_{\text{out}}^{\text{wOAE-of-SP}} \). This requires us to first introduce a new security notion for digital signature schemes and symmetric signcryption schemes that we call unique verifiability.

**Unique Verifiability.** Consider game \( g^{\text{UV}} \) of Fig. 19, defined for a digital signature scheme \( \text{DS} \) and an adversary \( A_{\text{UV}} \) on the left, and for a symmetric signcryption scheme \( \text{SS} \) and an adversary \( A_{\text{UV}} \) on the right. The advantage of \( A_{\text{UV}} \) in breaking the UV security of \( \text{DS} \) (resp. \( \text{SS} \)) is defined as \( \text{Adv}_{\text{DS}}^{\text{UV}} (A_{\text{UV}}) = \text{Pr}[g^{\text{DS}}_{\text{UV}} (A_{\text{UV}})] \) (resp. \( \text{Adv}_{\text{SS}}^{\text{UV}} (A_{\text{UV}}) = \text{Pr}[g^{\text{SS}}_{\text{UV}} (A_{\text{UV}})] \)).

Intuitively, for \( \text{DS} \), this game captures the inability of an attacker to produce a signature by executing the underlying signing algorithm with some signing key \( sk \) such that the resulting signature verifies successfully under an attacker-chosen verification key \( vk' \) that is different from the “honest” verification key corresponding to \( sk \). Analogously, for \( \text{SS} \), the game captures the inability of an attacker to produce a ciphertext by executing the underlying encryption algorithm with some signing key \( sk \) such that the resulting ciphertext decrypts successfully under an attacker-chosen verification key \( vk' \) that is different from the “honest” verification key corresponding to \( sk \).

\[
\begin{array}{c}
\text{Game}\ g^{\text{DS}}_{\text{UV}} (A_{\text{UV}}) \\
(s, vk', m) \leftarrow A_{\text{UV}} \\
s \leftarrow \text{DS.Sig}(sk, m) \\
\text{win}_0 \leftarrow \text{DS.Ver}(vk', m, s) \\
\text{win}_1 \leftarrow ((s, vk') \notin [\text{DS.Kg}]) \\
\text{Return } \text{win}_0 \text{ and } \text{win}_1
\end{array}
\]

\[
\begin{array}{c}
\text{Game}\ g^{\text{SS}}_{\text{UV}} (A_{\text{UV}}) \\
(g, K_p, u, u', sk_u, vk_{u'}, n, m, ad) \leftarrow A_{\text{UV}} \\
c \leftarrow \text{SS.SigEnc}(g, K_p, u, sk_u, n, m, ad) \\
\text{win}_0 \leftarrow (\text{SS.VerDec}(g, K_p, u', vk_{u'}, n, c, ad) \neq \bot) \\
\text{win}_1 \leftarrow ((sk_u, vk_u) \notin [\text{SS.UserKg}]) \\
\text{Return } \text{win}_0 \text{ and } \text{win}_1
\end{array}
\]

Fig. 19: **Left pane:** Game defining the unique verifiability of a digital signature scheme \( \text{DS} \). **Right pane:** Game defining the unique verifiability of a symmetric signcryption scheme \( \text{DS} \).

As an intermediate result, we first show that the unique verifiability of the symmetric signcryption schemes \( \text{StE} = \text{SIGN-THEN-ENCRYPT-SS}[\text{DS, NE}], \text{EtS} = \text{ENCRYPT-THEN-SIGN-SS}[\text{NE, DS}], \) and \( \text{SP} = \text{SEAL-PACKET-SS}[H, \text{DS, NE}] \) defined in Sections 4 and 5 is implied by the unique verifiability of the underlying digital signature scheme \( \text{DS} \). In Appendix D.2 we prove that in the random oracle model, the Ed25519 digital signature scheme underlying \( \text{SP} \) achieves unique verifiability. We omit
\[ BM = BOX\text{-MESSAGE-SS}[\mathcal{M}, NE, H, SP] \] from Proposition 2 for simplicity, but it is easy to see that the unique verifiability of BM follows from the unique verifiability of SP.

**Proposition 2.** Let \( StE = \text{SIGN-THEN-ENCRYPT-SS}[DS, NE] \), \( EtS = \text{ENCRYPT-THEN-SIGN-SS}[NE, DS] \), and \( SP = \text{SEAL-PACKET-SS}[H, DS, NE] \) be symmetric signcryption schemes built from some DS, NE, H as specified in Construction 1, Construction 2, and Construction 4 respectively. Let \( SS \in \{StE, EtS, SP\} \). Let \( \mathcal{A}_{UV}^{SS} \) be any adversary against the UV security of SS. Then we can build adversary \( \mathcal{A}_{DS}^{UV} \) such that

\[ \text{Adv}_{SS}^{UV}(\mathcal{A}_{UV}) \leq \text{Adv}_{DS}^{UV}(\mathcal{A}_{UV}). \]

The proof of Proposition 2 for SS = StE is fairly straightforward and is given in Appendix D.1.

**Proposition 3.** Let \( DS = \text{Ed25519-DS}[k, p, G, G_p, B, H_1, H_2, H_3] \) be the digital signature scheme built from some \( k, p, G, G_p, B, H_1, H_2, H_3 \) as specified in Construction 6. Let \( \mathcal{A}_{UV} \) be any adversary against the UV security of DS. Let \( H_3 \) be modeled as a random oracle and \( q_{it} \) be the number of random oracle queries that \( \mathcal{A}_{UV} \) makes. Then

\[ \text{Adv}_{Ed25519}^{UV}(\mathcal{A}_{UV}) \leq (2 \cdot q_{it})/p + q_{it}^2/p. \]

We now show that OAE security of a symmetric signcryption scheme \( SS \in \{StE, EtS, SP, BM\} \) with respect to \( \text{func}_{\text{out}} \) follows from its OAE security with respect to \( \text{func}_{\text{out}} \) and its UV security.

<table>
<thead>
<tr>
<th>Scheme SS</th>
<th>Security notion OAE*</th>
<th>Triviality predicate pred_{trivial}</th>
</tr>
</thead>
<tbody>
<tr>
<td>StE = SIGN-THEN-ENCRYPT-SS[DS, NE]</td>
<td>OAE</td>
<td>pred_{trivial}</td>
</tr>
<tr>
<td>EtS = ENCRYPT-THEN-SIGN-SS[NE, DS]</td>
<td>OAE</td>
<td>pred_{trivial}</td>
</tr>
<tr>
<td>BM = BOX-MESSAGE-SS[\mathcal{M}, NE, H, SP]</td>
<td>bwOAE</td>
<td>pred_{trivial}</td>
</tr>
<tr>
<td>SP = SEAL-PACKET-SS[H, DS, NE]</td>
<td>wOAE[\text{ENC}[\mathcal{M}, NE]]</td>
<td>pred_{except-user}</td>
</tr>
</tbody>
</table>

Table 1: Table specifying the OAE* security notion and the predicate pred_{trivial} for different values of SS. Here, \( StE = \text{SIGN-THEN-ENCRYPT-SS}[DS, NE] \) and \( EtS = \text{ENCRYPT-THEN-SIGN-SS}[NE, DS] \) are the symmetric signcryption schemes built from some DS, NE as specified in Construction 1 and Construction 2 respectively. The scheme \( BM = BOX\text{-MESSAGE-SS}[\mathcal{M}, NE, H, SP] \) is the symmetric signcryption scheme built from \( \mathcal{M} = \{0, 1\}^* \) and some NE, H, SP as specified in Construction 3. Finally, the scheme \( SP = SEAL\text{-PACKET-SS}[H, DS, NE] \) is the symmetric signcryption scheme built from some H, DS, NE as specified in Construction 4.

**Proposition 4.** Let \( StE, EtS, BM, \) and \( SP \) be the symmetric signcryption schemes defined in Constructions 1, 2, 3 and 4 respectively. Let \( \text{pred}_{\text{trivial}} \) and \( \text{pred}_{\text{except-user}} \) be the ciphertext-triviality predicates as defined in Fig. 8. Let \( \text{func}_{\text{out}} \) and \( \text{func}_{\text{sec}} := \text{func}_{\text{silence-with-mt}}[\text{pred}_{\text{trivial}}] \) be the output-guarding functions as defined in Fig. 9. Let \( SS \in \{StE, EtS, SP, BM\} \), and OAE* and \( \text{pred}_{\text{trivial}} \) as defined in Table 1. Let \( \mathcal{A}_{OAE^*} \) be any adversary against the OAE* security of SS with respect to \( \text{pred}_{\text{trivial}} \) and \( \text{func}_{\text{out}} \). Then we can build adversaries \( \mathcal{A}_{UV} \) and \( \mathcal{A}_{OAE^*} \) such that

\[ \text{Adv}_{SS,\text{pred}_{\text{trivial}},\text{func}_{\text{out}}}(\mathcal{A}_{OAE^*}) \leq \text{Adv}_{SS}^{UV}(\mathcal{A}_{UV}) + \text{Adv}_{SS,\text{pred}_{\text{trivial}},\text{func}_{\text{out}}}(\mathcal{A}_{OAE^*}). \]

The proof of Proposition 4 is in Appendix D.3.

### 6.5 Instantiation and Interpretation

In this section, we summarize the analysis of BoxMessage that we provided in Sections 6.1 to 6.3. For each of the two core security notions in our paper, we state a separate corollary about what is achieved.
by `BoxMessage`. We choose to present our corollaries in a precise and formal way, meaning they do not simplify or approximate any of the advantage terms from our analysis in the prior sections. However, instead of using the abstract schemes, we phrase our corollaries with respect to the real-world primitives that are used to instantiate `BoxMessage` in Keybase, as described in Section 5.

In the following, we show that our analysis of the in-group unforgeability of `BoxMessage` in Section 6.1 reduces the IUF security of `BoxMessage` with respect to the predicate $\text{pred}_{\text{trivial}}$ to the collision resistance of the SHA-256 and SHA-512 hash functions and the SUFCMA security of the Ed25519 signature scheme. Recall that in the IUF security game, we use ciphertext triviality predicates to detect ciphertexts that were trivially forwarded from the `SigEnc` oracle to the `VerDec` oracle. In particular, the predicate $\text{pred}_{\text{trivial}}$ captures the strong unforgeability of ciphertexts by checking whether $z \in C$. We note here that the following result holds for any choice of the message space $M$. Moreover, the SUFCMA security of Keybase’s implementation of Ed25519 is shown in [BCJZ21].

**Corollary 1.** Let SHA-256, SHA-512, Ed25519, XSalsa20-Poly1305 be the standard real-world schemes used in Keybase. Let $SP = \text{SEAL-PACKET-SS}[\text{SHA}-512, \text{Ed25519}, \text{XSalsa20-Poly1305}]$ be the symmetric signcryption scheme as defined in Construction 4. Let $BM = \text{BOX-MESSAGE-SS}[M, \text{XSalsa20-Poly1305}, \text{SHA}-256, SP]$ be the symmetric signcryption scheme as defined in Construction 3, instantiated for any message space $M$. Let $\text{pred}_{\text{trivial}}$ be the ciphertext-triviality predicate as defined in Fig. 8. Let $\mathcal{A}_{\text{IUF-of-BM}}$ be any adversary against the IUF security of $BM$ with respect to $\text{pred}_{\text{trivial}}$. Then we can build adversaries $\mathcal{A}_{\text{SUFCA}}, \mathcal{A}_{\text{CR-of-SHA}512}$, and $\mathcal{A}_{\text{CR-of-SHA}256}$ such that

$$
\text{Adv}_{BM, \text{pred}_{\text{trivial}}}(\mathcal{A}_{\text{IUF-of-BM}}) \leq \text{Adv}_{\text{E25519}}^{\text{SUFCA}}(\mathcal{A}_{\text{SUFCA}}) + \text{Adv}_{\text{SHA}-256}^{\text{CR}}(\mathcal{A}_{\text{CR-of-SHA}512}) + \text{Adv}_{\text{SHA}-256}^{\text{CR}}(\mathcal{A}_{\text{CR-of-SHA}256}).
$$

Next, we show that our analysis of the out-group AE security of `BoxMessage` in Sections 6.2 and 6.3 reduces the bwOAE security of `BoxMessage` with respect to the predicate $\text{pred}_{\text{trivial}}$ and the output-guarding function $\text{func}_{\text{out}}$ to the key-recovery and AEAD security of XSalsa20-Poly1305, the sparsity of Ed25519 with respect to the message space $M$ underlying `BoxMessage`, and the collision resistance of SHA-256. Recall that bwOAE is a variant of the OAE security notion that modifies the group-nonce uniqueness condition of the latter. In the bwOAE game, we use ciphertext triviality predicates to detect ciphertexts that were trivially forwarded from the `SigEnc` oracle to the `VerDec` oracle and the function $\text{func}_{\text{out}}$ to appropriately respond with $\perp$ when the `VerDec` oracle is queried on such forwarded ciphertexts.

**Corollary 2.** Let SHA-256, SHA-512, Ed25519, XSalsa20-Poly1305 be the standard real-world schemes used in Keybase. Let $SP = \text{SEAL-PACKET-SS}[\text{SHA}-512, \text{Ed25519}, \text{XSalsa20-Poly1305}]$ be the symmetric signcryption scheme as defined in Construction 4. Let $BM = \text{BOX-MESSAGE-SS}[M, \text{XSalsa20-Poly1305}, \text{SHA}-256, SP]$ be the symmetric signcryption scheme as defined in Construction 3, instantiated for the message space $M$ that contains application-layer messages encoded in the MessagePack format. Let $\text{pred}_{\text{trivial}}$ be the ciphertext-triviality predicate as defined in Fig. 8. Let $\text{func}_{\text{out}}$ be the output-guarding function as defined in Fig. 9. Let $\mathcal{A}_{\text{bwOAE-of-BM}}$ be any adversary against the bwOAE security of BM with respect to $\text{pred}_{\text{trivial}}$ and $\text{func}_{\text{out}}$, making at most $q_{\text{NewHonGroup}}$ queries to its `NewHonGroup` oracle. Then we can build adversaries $\mathcal{A}_{\text{KR}}, \mathcal{A}_{\text{AEAD}}, \mathcal{A}_{\text{SPARSE}}$, and $\mathcal{A}_{\text{CR-of-SHA}256}$ such that

$$
\text{Adv}_{BM, \text{pred}_{\text{trivial}}}(\mathcal{A}_{\text{bwOAE-of-BM}}) \\
\leq 2 \cdot \text{Adv}_{\text{XSalsa20-Poly1305}}^{\text{KR}}(\mathcal{A}_{\text{KR}}) + \text{Adv}_{\text{XSalsa20-Poly1305}}^{\text{AEAD}}(\mathcal{A}_{\text{AEAD}}) + \frac{q_{\text{NewHonGroup}}^2}{2^{257}} + \text{Adv}_{\text{E25519,M}}^{\text{SUFCA}}(\mathcal{A}_{\text{SPARSE}}) + \text{Adv}_{\text{SHA}-256}^{\text{CR}}(\mathcal{A}_{\text{CR-of-SHA}256}).
$$

We emphasize that for this result, we model the (modularly reduced) instances of SHA-512 within Ed25519 as a random oracle. We also note from our analysis of the sparsity of Ed25519 with respect to the message space $M$ in Appendix C.1 that we expect the $\text{Adv}_{\text{E25519,M}}^{\text{SUFCA}}(\mathcal{A}_{\text{SPARSE}})$ term in the bound above to be very small for the parameters that `BoxMessage` is instantiated within Keybase.
7 Conclusions

Combining Theorem 2 with Theorem 3 and Theorem 4 with Theorem 5 establishes the in-group unforgeability and out-group authenticated encryption security of Keybase’s BoxMessage algorithm. These results rely on some standard security assumptions (unforgeability of Ed25519 and collision resistance of SHA-256) as well as some non-standard assumptions (key-dependent message security of XSalsa20-Poly1305 and sparsity of Ed25519). These non-standard assumptions arose, respectively, from the key cycle in SealPacket and the key reuse without explicit context separation in BoxMessage. While we were able to justify these assumptions, we consider them brittle as they are not well studied, their justifications required ideal models, and (in the case of sparsity) they required properties of the specific messaging encoding format used by Keybase.

The comparative simplicity of our Sign-then-Encrypt construction speaks to the value of formalizing the syntax and security of symmetric signcryption. Explicit goals allow designing schemes in parallel with writing proofs to identify precisely what is needed.

References


### A Proofs for Sign-Then-Encrypt Based Symmetric Signcryption

#### A.1 In-Group Unforgeability of StE (Proof of Theorem 1)

<table>
<thead>
<tr>
<th>Games $G_0$–$G_2$</th>
<th>\text{VER}\text{Dec}(g, u, n, c, ad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_0$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>require $K[g] \neq \perp$ and $vk[u] \neq \perp$ and $u \in \text{members}[g]$</td>
</tr>
<tr>
<td>$G_1$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>$ad_{\text{NE}} \leftarrow (u, ad)$ ; $m_e \leftarrow \text{NE}\text{.Dec}(K[g], n, c, ad_{\text{NE}})$</td>
</tr>
<tr>
<td>$G_2$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>$m_e \leftarrow (g, n, m, ad)$</td>
</tr>
<tr>
<td>$G_3$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>If $m_e = \perp$ then return $\perp$</td>
</tr>
<tr>
<td>$G_4$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>$s \parallel m \leftarrow m_e$ // s.t. $</td>
</tr>
<tr>
<td>$G_5$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>$m_e \leftarrow (g, n, m, ad)$</td>
</tr>
<tr>
<td>$G_6$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>If $\neg\text{DS.Ver}(vk[u], m_e, s)$ then return $\perp$</td>
</tr>
<tr>
<td>$G_7$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>$z \leftarrow ((g, u, n, m, ad), c)$</td>
</tr>
<tr>
<td>$G_8$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorr}User</td>
<td>If $z \in C$ then return $m$</td>
</tr>
<tr>
<td>$G_9$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorrUser}</td>
<td>If $\neg\text{user_is_corrupt}[u]$ then</td>
</tr>
<tr>
<td>$G_{10}$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorrUser}</td>
<td>$\text{bad}<em>t \leftarrow \text{true}$ // $A</em>{\text{SUFCMA}}$ breaks SUFCMA of DS.</td>
</tr>
<tr>
<td>$G_{11}$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorrUser}</td>
<td>$\text{win} \leftarrow \text{true}$ // $G_{0(1)}$</td>
</tr>
<tr>
<td>$G_{12}$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorrUser}</td>
<td>$\text{bad}<em>t \leftarrow \text{true}$ // $G</em>{0(2)}$</td>
</tr>
<tr>
<td>$G_{13}$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorrUser}</td>
<td>Return $m$</td>
</tr>
</tbody>
</table>

---

**Fig. 20: Top pane:** Games $G_0$–$G_2$ for the proof of Theorem 1. The code highlighted in gray was added by expanding the symmetric signcryption algorithms StE.SimEnc, StE.VerDec and the ciphertext-triviality predicate pred_{trivial} in game $G_{0}$\text{U}SimEnc,SimVerDec,Sim\text{Ex}User,Sim\text{New}User,Sim\text{NewCorrUser} ($A_{\text{URF}}$). The code added for the transitions between games is highlighted in green. **Bottom pane:** Adversary $A_{\text{SUFCMA}}$ for the proof of Theorem 1. The highlighted instructions mark the changes in the code of the simulated game $G_0$.

**OVERVIEW.** This proof uses games $G_0$ through $G_2$ in Fig. 20. We establish the following claims.
Consider what conditions need to be met for a proof using games. In Fig. 20, we build an adversary that corrupts in game 0. Each of these transitions removes an instruction that sets the flag in each of its conditional branches. It follows that

\[ \text{Pr}[\text{bad}_0] + \text{Pr}[\text{bad}_1] \leq \text{Adv}^{\text{SUFCA}}(A^{\text{SUFCA}}). \]

We justify our claims in the following paragraphs.

**GAME 0.** Game 0 is functionally equivalent to game 0 (\(A^{\text{SUFCA}}\)). The former expands multiple instructions in the latter; the expanded code is marked in gray. Game 0 also contains newly added code that is highlighted in green and does not affect its functionality; this code will be used for transitions between games. In particular, the if-then-else statement that was added in oracle VERDEC sets the win flag in each of its conditional branches. It follows that

\[ \text{Adv}^{\text{SUFCA}}(A^{\text{SUFCA}}) = \text{Pr}[\text{bad}_0]. \]

**TRANSITIONS FROM 0 TO 1.** Let \(i \in \{0, 1\}\). Games 0 and 0 are identical until bad is set, therefore

\[ \text{Pr}[\text{bad}_1] = \text{Pr}[\text{bad}_1] \leq \text{Pr}[\text{bad}_1]. \]

Each of these transitions removes an instruction that sets the win flag. The win flag can no longer be set in game 1, making it impossible for 0 to win. We have

\[ \text{Pr}[\text{bad}_1] = 0. \]

**BOUNDING Pr[bad_0].** In Fig. 20, we build an adversary 0 against the SUFCMA security of DS. It simulates game 0 for adversary 0 as follows. When adversary 0 calls the user oracles in the SUFCMA security game, adversary 0 responds by using its own user oracles from the SUFCMA security game. In response to 0’s calls to the group oracles in the SUFCMA security game, we have 0 respond by honestly sampling and managing its own NE keys. When 0 calls the SigEnc oracle, adversary 0 can now simulate it by using its Sign oracle. Finally, 0’s calls to the VERDEC oracle can be trivially simulated by 0 because this oracle does not use DS signing keys. Observe that bad can be in game 0 only after VERDEC confirms the following three statements to be true: DS.Ver(vk, m, s), \(\neg\text{user_is_corrupt}(u)\), and \((u, m, s) \notin S\). These are exactly the three winning conditions used in the SUFCMA security game, so 0 halts with abort whenever it detects bad being set in the simulated SUFCMA game. We have

\[ \text{Pr}[\text{bad}_0] \leq \text{Pr}[\text{SUFCA}] = \text{Adv}^{\text{SUFCA}}(A^{\text{SUFCA}}). \]

**BOUNDING Pr[bad_1].** Now consider what conditions need to be met for bad to be set in game 1. This only happens when \(z \notin C\) and \((u, m, s) \in S\). We can use the definitions of the encoding \(m\) and the tuple \(z\) in order to expand \(u, m, s \in S\) into \((u, (g, n, m, ad), s) \in S, z \notin C\) into \((g, u, n, m, ad), c) \notin C\). The former means that one of the prior calls to the SigEnc oracle produced the ciphertext \(c' = \text{NE.Enc}[K[g], m, m, ad]\) for \(m = s \parallel m\) and \(ad = (u, ad)\). But the latter implies that in the current call to the VERDEC oracle a distinct ciphertext \(c \neq c'\) was used to recover \(m = \text{NE.Dec}[K[g], n, c, ad]\). Together this means that \(\text{NE.Enc}[K[g], n, \text{NE.Dec}[K[g], n, c, ad], ad] \neq c\), violating the assumed tidiness property of NE. We get a contradiction, and hence

\[ \text{Pr}[\text{bad}_1] = 0. \]

**A.2 Out-group Authenticated Encryption of StE (Proof of Theorem 1)**

**OVERVIEW.** This proof uses games 0 through 0 in Fig. 21. We establish the following claims.
Games $G_0$–$G_2$

$b' \leftarrow \{0, 1\}; \quad b' \leftarrow A_{\text{OAE}}^G, \text{StEnc, VerDec}; \quad \text{Return } b' = b'$

$\text{SigEnc}(g, u, n, m_0, m_1, ad)$

require $K[g] \neq \bot$ and $|m_0| = |m_1|

require $\text{sk}[u] \neq \bot$ and $u \in \text{members}[g]$

require $\forall d \in \{0, 1\}, (g, u, n, m_d, ad) \notin N_d$

If $m_0 \neq m_1$ then

If $\text{group_is_corrupt}[g]$ then return $\bot$ else $\text{chal}[g] \leftarrow \text{true}$

$m_s \leftarrow (g, n, m_0, ad); \quad s \leftarrow \text{DS.Sig}(\text{sk}[u], m_s)$

$m_e \leftarrow s \parallel |m_b}; \quad \text{adNE} \leftarrow \langle u, ad \rangle$

$c \leftarrow \text{Enc}(g, n, m_e, adNE)$

$N_0 \leftarrow N_0 \cup \{(g, u, n, m_0, ad)\}$

$N_1 \leftarrow N_1 \cup \{(g, u, n, m_1, ad)\}$

$C \leftarrow C \cup \{(g, u, n, m_0, ad), c\}$

$Q \leftarrow Q \cup \{(g, u, n, m_0, m_1, ad), c\}$

$W[g, u, n, c, ad] \leftarrow m_1; \quad \text{Return } c$

Adversary $A_{\text{AEAD}}^G, \text{Enc, Dec}$

$b' \leftarrow A_{\text{OAE}}^G, \text{StEnc, StEnc, StEnc, VerDec}; \quad \text{Return } b'$

$\text{SimSigEnc}(g, u, n, m_0, m_1, ad)$

require $K[g] \neq \bot$ and $|m_0| = |m_1|

require $\text{sk}[u] \neq \bot$ and $u \in \text{members}[g]$

require $\forall d \in \{0, 1\}, (g, u, n, m_d, ad) \notin N_d$

If $m_0 \neq m_1$ then

If $\text{group_is_corrupt}[g]$ then return $\bot$ else $\text{chal}[g] \leftarrow \text{true}$

$m_s \leftarrow (g, n, m_0, ad); \quad s \leftarrow \text{DS.Sig}(\text{sk}[u], m_s)$

$m_e \leftarrow s \parallel |m_b}; \quad \text{adNE} \leftarrow \langle u, ad \rangle$

$c \leftarrow \text{Enc}(g, n, m_e, adNE)$

Else \quad // $m_0 \neq m_1$

$m_{s, 0} \leftarrow (g, n, m_0, ad); \quad s_0 \leftarrow \text{DS.Sig}(\text{sk}[u], m_{s, 0})$

$m_{s, 1} \leftarrow (g, n, m_1, ad); \quad s_1 \leftarrow \text{DS.Sig}(\text{sk}[u], m_{s, 1})$

$m_{e, 0} \leftarrow s_0 \parallel |m_0}; \quad m_{e, 1} \leftarrow s_1 \parallel |m_1}; \quad \text{adNE} \leftarrow \langle u, ad \rangle$

$c \leftarrow \text{Enc}(g, n, m_{e, 0}, m_{e, 1}, adNE)$

$N_0 \leftarrow N_0 \cup \{(g, u, n, m_0, ad)\}$

$N_1 \leftarrow N_1 \cup \{(g, u, n, m_1, ad)\}$

$W[g, u, n, c, ad] \leftarrow m_1; \quad \text{Return } c$

SimExposer$(g)$

SimEncExposer$(g)$

SimNewHonGroup$(g)$

SimNewCorrGroup$(g, K)$

SimNewUser$(u)$

SimNewGroup$(g, sk, vk)$

SimNewGroup$(g, K, users)$

SimNewUser$(u)$

SimNewGroup$(g, sk, vk)$

SimNewCorrUser$(g, u, sk, vk)$

SimNewHonUser$(u)$

SimNewCorrUser$(g, u, sk, vk)$

// These oracles are identical to the corresponding oracles in the OAE game for StE as per Fig. 7.

Fig. 21: Top pane: Games $G_0$–$G_2$ for the proof of Theorem 1. The code highlighted in gray was added by expanding the symmetric signcryption algorithms StE.SigEnc, StE.VerDec and the ciphertext-triviality predicate $\text{pred}_{\text{triv}}$ in game $G_0^{\text{OAE}}$. The code added for the transitions between games is highlighted in green. Bottom pane: Adversary $A_{\text{AEAD}}$ for the proof of Theorem 1. The highlighted instructions mark the changes in the code of the simulated game $G_2$. 

35
By combining the above, we have
\[
\text{Adv}^\text{OAE}_{\text{StE.predict/out}}(A_{\text{OAE}}) = 2 \cdot \text{Pr}[g_0] - 1
\]

We justify our claims in the following paragraphs.

**GAME $g_0$**. Game $g_0$ is functionally equivalent to game $g_{\text{OAE}}^\text{StE.predict/out}$ (A_{\text{OAE}}). The former expands multiple instructions in the latter; the expanded code is marked in gray. Game $g_0$ also contains newly added code that is highlighted in green and does not affect its functionality; this code will be used for transitions between games. It follows that
\[
\text{Adv}^\text{OAE}_{\text{StE.predict/out}}(A_{\text{OAE}}) = 2 \cdot \text{Pr}[g_0] - 1.
\]

**TRANSITION FROM $g_0$ TO $g_1$**. We now explain how game $g_1$ differs from game $g_0$, and will we argue that these two games are functionally equivalent. Game $g_1$ maintains a record of all ciphertexts that were previously returned by its SigEnc oracle. When such a ciphertext gets subsequently passed as an input to the VerDec oracle (with the matching values of $g, u, n, ad$), game $g_1$ immediately returns $\bot$. To implement this functionality, game $g_1$ uses a table $W$ that maps $(g, u, n, c, ad)$ to $m_1$. We claim that
\[
\text{Pr}[g_0] = \text{Pr}[g_1].
\]

In order to justify this transition, let us look at what happens in game $g_0$ when an honestly produced ciphertext is forwarded from the SigEnc oracle to the VerDec oracle. Consider an arbitrary SigEnc query that took some $(g^*, u^*, n^*, m_0^*, m_1^*, ad^*)$ as input to return a ciphertext $c^*$. At the end of this SigEnc call, the sets $N$, $C$, and $Q$ are updated to insert new elements that are created from the SigEnc’s input $(g^*, u^*, n^*, m_0^*, m_1^*, ad^*)$ and output $c^*$; our analysis below will crucially rely on these sets containing the newly added elements. Now consider what happens if the VerDec oracle is queried on $(g^*, u^*, n^*, c^*, ad^*)$ at any point later on. If the condition checking $\text{group.is.corrupt}[g]$ passes, then the decryption correctness of NE and the correctness of DS jointly guarantee that VerDec builds $z = ((g^*, u^*, n^*, m_0^*, ad^*), c^*)$. But exactly the same tuple was inserted into $C$ during the earlier SigEnc call. It follows that $z \in C$ is true and the VerDec oracle returns $\bot$. Therefore, the outputs of VerDec are consistent across $g_0$ and $g_1$.

**TRANSITION FROM $g_1$ TO $g_2$**. Note that the condition checking $z \in C$ in oracle VerDec of game $g_1$ can never be reached. In particular, if the set $C$ contains an element $z = ((g, u, n, m, ad), c)$ for any value of $m$, then $W[g, u, n, c, ad] \neq \bot$ must be true. The latter causes VerDec to return $\bot$ right away. So in game $g_2$ we remove the two lines of code that build $z$ and check whether it belongs to $C$. Game $g_2$ is functionally equivalent to game $g_1$, so
\[
\text{Pr}[g_1] = \text{Pr}[g_2].
\]

**BOUNDING $Pr[g_2]$**. In Fig. 21 we build an adversary $A_{\text{AEAD}}$ against the AEAD security of NE that simulates game $g_2$ for adversary $A_{\text{OAE}}$. Adversary $A_{\text{AEAD}}$ simulates the group oracles for $A_{\text{OAE}}$ by calling its own group oracles, and it simulates the user oracles for $A_{\text{OAE}}$ by sampling its own keys for DS.

Adversary $A_{\text{AEAD}}$ simulates the SigEnc oracle for $A_{\text{OAE}}$ by evaluating $\text{NE.Enc}$ using its encryption oracle ENC. When SigEnc is queried with $m_0 = m_1$ as input, adversary $A_{\text{AEAD}}$ ensures that ENC is likewise called with both of its challenge-message arguments being equal. This maintains consistency of the challenge maps cha[·] between the simulated game $g_2$ and the AEAD security game of $A_{\text{AEAD}}$. When SigEnc is queried with $m_0 \neq m_1$ as input, note that $|m_{e,0}| = |m_{e,1}|$ is guaranteed to hold whenever $|m_0| = |m_1|$ does, because in Section 2.1 we require DS to produce signatures of a fixed length DS.nl. Finally, the nonce-misuse condition in oracle SigEnc that checks $(g, n, m_1, ad) \notin N$ is equivalent to the corresponding condition in oracle ENC that checks $(g, u, n, m_1, ad) \notin N$, because $A_{\text{AEAD}}$ queries ENC on an ad field that uniquely encodes the $u, ad$ fields of SigEnc.

Adversary $A_{\text{AEAD}}$ simulates the VerDec oracle for $A_{\text{OAE}}$ by evaluating $\text{NE.Dec}$ using its decryption oracle DEC. If DEC returns $m_0 \neq \bot$ then the challenge bit in game $g_{\text{AEAD}}^\text{NE}(A_{\text{AEAD}})$ is $b = 1$ and hence
$\mathcal{A}_{AEAD}$ immediately calls $\text{abort}(1)$ in order to halt and return $1$ as output. Otherwise (i.e. $m_0 = \perp$) adversary $\mathcal{A}_{AEAD}$ returns $\perp$ as the output of the simulated $\mathcal{V}_{Dec}$ oracle. Note that the $\mathcal{AEAD}$ security game defines $\mathcal{Dec}$ to return $\perp$ in response to the NE ciphertexts that were previously produced by its ENC oracle. This is equivalent to the case where a ciphertext is forwarded from $\mathcal{SigEnc}$ to $\mathcal{V}_{Dec}$; the use of the $W$ map at the start of the simulated $\mathcal{V}_{Dec}$ oracle ensures that $\mathcal{Dec}$ is never called on a replayed NE ciphertext.

Let $d$ denote the challenge bit in game $\mathcal{G}_d^{AEAD}(\mathcal{A}_{AEAD})$, and let $d'$ denote the corresponding output of adversary $\mathcal{A}_{AEAD}$. If $d = 0$ then $\mathcal{A}_{AEAD}$ perfectly simulates game $\mathcal{G}_2$ with challenge bit $b = 0$ for adversary $\mathcal{A}_{OAE}$, and returns $d' = 1$ iff $\mathcal{A}_{OAE}$ returns $b' = 1$. In particular, adversary $\mathcal{A}_{AEAD}$ perfectly simulates the $\mathcal{V}_{Dec}$ oracle because in game $\mathcal{G}_2$ this oracle always returns $\perp$ if $W[g, u, n, c, ad] = \perp$ and $b = 0$. We write $\Pr[d' = 1 | d = 0] = \Pr[b' = 1 | b = 0]$. If $d = 1$ then $\mathcal{A}_{AEAD}$ perfectly simulates game $\mathcal{G}_2$ with challenge bit $b = 1$ for adversary $\mathcal{A}_{OAE}$ until the first time $\mathcal{N}_E \mathcal{D}_{Dec}$ produces non-$\perp$ in oracle $\mathcal{V}_{Dec}$ of game $\mathcal{G}_2$. When the latter happens, $\mathcal{A}_{AEAD}$ halts with $d' = 1$ regardless of whether $\mathcal{A}_{OAE}$ would have returned $b' = 1$. We write $\Pr[d' = 1 | d = 1] \geq \Pr[b' = 1 | b = 1]$. It follows that

$$\text{Adv}_{\mathcal{NE}}^{\mathcal{AEAD}}(\mathcal{A}_{AEAD}) = \Pr[d' = 1 | d = 1] - \Pr[b' = 1 | b = 0]$$

The latter equality relies on the standard fact that $\mathcal{A}_{OAE}$’s advantage in game $\mathcal{G}_2$ can be expressed in two alternative ways that are equivalent.

## B Proofs for the In-Group Unforgeability of Keybase

### B.1 In-Group Unforgeability of BoxMessage (Proof of Theorem 2)

**Overview.** This proof uses games $\mathcal{G}_0$ through $\mathcal{G}_2$ in Fig. 22. We establish the following claims.

- $\text{Adv}_{BM, \text{predinit}}^{\mathcal{IUF}}(\mathcal{A}_{\mathcal{IUF} - \mathcal{BM}}) = \Pr[\mathcal{G}_0]$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\mathcal{bad}_0]$  
- $\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\mathcal{bad}_1]$

By combining the above, we have

$$\text{Adv}_{BM, \text{predinit}}^{\mathcal{IUF}}(\mathcal{A}_{\mathcal{IUF} - \mathcal{BM}}) = \Pr[\mathcal{G}_0] = 1 \sum_{i=0}^1 (\Pr[\mathcal{G}_i] - \Pr[\mathcal{G}_{i+1}]) + \Pr[\mathcal{G}_2]$$

$$\leq \Pr[\mathcal{bad}_0] + \Pr[\mathcal{bad}_1]$$

$$\leq \text{Adv}_{SP, \text{predinit}}^{\mathcal{IUF}-\mathcal{SP}}(\mathcal{A}_{\mathcal{IUF} - \mathcal{SP}}) + \text{Adv}_{CR}^{\mathcal{IUF}}(\mathcal{A}_{\mathcal{CR}}).$$

We justify our claims in the following paragraphs.

**Game $\mathcal{G}_0$.** Game $\mathcal{G}_0$ is functionally equivalent to game $\mathcal{G}_{BM, \text{predinit}}^{\mathcal{IUF}}$. The former expands multiple instructions in the latter; the expanded code is marked in gray. Game $\mathcal{G}_0$ also contains newly added code that is highlighted in green and does not affect its functionality; this code will be used for transitions between games. In particular, the if-then-else statement that was added in oracle $\mathcal{V}_{Dec}$ sets the win flag in each of its conditional branches. It follows that

$$\text{Adv}_{BM, \text{predinit}}^{\mathcal{IUF}}(\mathcal{A}_{\mathcal{IUF} - \mathcal{BM}}) = \Pr[\mathcal{G}_0].$$

**Transitions from $\mathcal{G}_0$ to $\mathcal{G}_2$.** Let $i \in \{0, 1\}$. Games $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$ are identical until $\mathcal{bad}_i$ is set, therefore

$$\Pr[\mathcal{G}_i] - \Pr[\mathcal{G}_{i+1}] \leq \Pr[\mathcal{bad}_i].$$

Each of these transitions removes an instruction that sets the win flag. In game $\mathcal{G}_2$ the win flag is only set when the conditions $3\gamma' : (g', u, n_{\text{header}} \oplus m_{\text{header}} \oplus c_{\text{header}} \oplus e), c_{\text{header}} \in C_{\text{SP}}$ and $n_{\text{body}} \| c_{\text{body}} = T[u, n_{\text{header}}, c_{\text{header}}, ad, g, h_{\text{body}}]$ are satisfied. Since $m_{\text{header}} = \langle ad, u, g, h_{\text{body}} \rangle$, the former condition is necessarily true for
Fig. 22: **Top pane:** Games $G_0 \rightarrow G_2$ for the proof of Theorem 2. The code highlighted in gray was added by expanding the symmetric signcryption algorithms BM.SigEnc, BM.VerDec and the ciphertext-triviality predicate $\text{pred}_{\text{trivial}}$ in game $G_2^{\text{BM.pretrivial}(A_{\text{UF-of-BM}})}$. The code added for the transitions between games is highlighted in green. **Bottom pane:** Adversary $A_{\text{UF-of-SP}}$ for the proof of Theorem 2. The highlighted instructions mark the changes in the code of the simulated game $G_1$. 
In Fig. 22 we build an adversary $A_{\text{IUF-of-SP}}$ against the IUF security of SP with respect to the ciphertext-triviality predicate $\text{pred}_{\text{trivial}}$. It perfectly simulates game $G_0$ for adversary $A_{\text{IUF-of-BM}}$. In particular, it simulates the responses to the user and group oracles using the corresponding oracles from its own IUF security game. Note that the IUF security game allows to expose all group keys, so every time an honest group key is created in the simulated game $G_0$, adversary $A_{\text{IUF-of-SP}}$ immediately exposes it. While simulating the SigEnc oracle for $A_{\text{IUF-of-BM}}$, it replaces calls to $\text{SP.SigEnc}$ with its SigEnc oracle. Analogously, while simulating the VerDec oracle, it substitutes calls to $\text{SP.VerDec}$ with its VerDec oracle.

Finally, when the $\text{bad}_0$ flag is set in the simulated game $G_0$, adversary $A_{\text{IUF-of-SP}}$ wins by setting the win flag in the IUF security game it plays in. Note that the $\text{bad}_0$ flag is only set in $G_0$ when $\neg \text{user is corrupt}[u]$, and $c_{\text{header}}$ successfully decrypts under $\text{SP.VerDec}$ and $\exists g': ((g', u, n_{\text{header}}, m_{\text{header}}, \varepsilon), c_{\text{header}}) \in C_{\text{SP}}$. These are the same as the winning conditions in the IUF security game of SP and so

$$\Pr[\text{bad}_0] \leq \text{Adv}_{\text{SP.\text{pred}_{\text{except-group}}} \left( A_{\text{IUF-of-SP}} \right)}.$$ 

Bounding $\Pr[\text{bad}^{0}]$. In order to upper-bound $\Pr[\text{bad}^{0}]$, consider an adversary $A_{\text{CR}}$ against the collision resistance of $H$ that is defined as follows. Adversary $A_{\text{CR}}$ perfectly simulates game $G_1$ for $A_{\text{IUF-of-BM}}$ until the $\text{bad}_1$ flag is set. To simulate the user oracles $U$ and the group oracles $G$, adversary $A_{\text{CR}}$ itself samples and maintains all user keys and group keys. Whenever the $\text{bad}_1$ flag is set in $\text{VerDec}$, adversary $A_{\text{CR}}$ returns a pair of strings $(m_{\text{body}} \parallel c_{\text{body}}, T[u, n_{\text{header}}, c_{\text{header}}, ad, g, h_{\text{body}}])$. Note that the table entry $T[u, n_{\text{header}}, c_{\text{header}}, ad, g, h_{\text{body}}]$ is not empty because $\text{bad}_1$ could only be set once the following two conditions were satisfied in the current $\text{VerDec}$ call: $\exists g': ((g', u, n_{\text{header}}, m_{\text{header}}, \varepsilon), c_{\text{header}}) \in C_{\text{SP}}$ and $m_{\text{header}} = \langle ad, u, g, h_{\text{body}} \rangle$. This means there was a prior SigEnc call where all of the values indexing the table $T$ (as per the current $\text{VerDec}$ call) were simultaneously in use. The two strings returned by $A_{\text{CR}}$ are distinct because a conditional statement in game $G_1$ verifies this immediately before setting the $\text{bad}_1$ flag. We now show that these strings also produce the same hash value. First, observe that the table entry $T[u, n_{\text{header}}, c_{\text{header}}, ad, g, h_{\text{body}}]$ was filled during a prior call to oracle SigEnc, with a value for which $h_{\text{body}} = \text{H}(T[u, n_{\text{header}}, c_{\text{header}}, ad, g, h_{\text{body}}])$ is true. But the same hash was obtained in the current $\text{VerDec}$ call upon evaluating $h_{\text{body}} \leftarrow \text{H}(n_{\text{body}} \parallel c_{\text{body}})$. So the strings returned by $A_{\text{CR}}$ produce a collision. It follows that

$$\Pr[\text{bad}^{1}] \leq \text{Adv}_{H}^{CR}(A_{\text{CR}}).$$

B.2 In-Group Unforgeability of SealPacket (Proof of Theorem 3)

Overview. This proof uses games $G_0$ through $G_3$ in Fig. 23. We establish the following claims.

- $\text{Adv}_{\text{SP.\text{pred}_{\text{except-group}}} \left( A_{\text{IUF-of-SP}} \right)} \left( A_{\text{IUF-of-SP}} \right) = \Pr[\text{bad}_0]$
- $\Pr[\text{bad}_0] = 0$
- $\Pr[\text{bad}_0] \leq \text{Adv}_{\text{DS}}^{\text{SUFCA}}(A_{\text{SUFCA}})$
- $\Pr[\text{bad}_0] \leq \text{Adv}_{H}(A_{\text{CR}})$
Games $g_0 \rightarrow g_3$

```
▷ G, SigEnc, VinDec
SigEnc(g, u, n, m, ad) // $ad = \varepsilon$

require $K[g] \neq \perp$ and $vk[u] \neq \perp$ and $u \in \text{members}[g]$
$h \leftarrow H(m) ; m_s \leftarrow \text{"Keybase-Chat-2"}$ $\parallel (K[g], n, h)$
$s \leftarrow DS.Sig(kv[u], m_s) ; T[u, m_s, s] \leftarrow m$
$m_e \leftarrow s \parallel m ; c \leftarrow NE.Enc(K[g], n, m_e)$
$C \leftarrow C \cup \{(g, u, n, m, ad), c\} ; \text{Return } c$
```

NewHonUser(u)
NewHonGroup(g, users)
ExposeUser(u)
ExposeGroup(g)
NewCorrUser(u, sk, vk)
NewCorrGroup(g, K, users)

// These oracles are identical to the corresponding
// oracles in the IUF game for SP as per Fig. 7.

Adversary $A^{\text{IUF-SUF_CMA}}_{\text{SimEnc, SigEnc, VinDec}}$

```
SimSigEnc(g, u, n, m, ad) // $ad = \varepsilon$

require $K[g] \neq \perp$ and $sk[u] \neq \perp$ and $u \in \text{members}[g]$
$h \leftarrow H(m) ; m_s \leftarrow \text{"Keybase-Chat-2"}$ $\parallel (K[g], n, h)$
$s \leftarrow \text{SIGN}(u, m_s) \parallel T[u, m_s, s] \leftarrow m$
$m_e \leftarrow s \parallel m ; c \leftarrow NE.Enc(K[g], n, m_e)$
$C \leftarrow C \cup \{(g, u, n, m, ad), c\} ; \text{Return } c$
```

SimNewHonUser(u)
SimExposeUser(u)
SimNewCorrUser(u, sk, vk)
SimNewHonGroup(g, K, users)
SimExposeGroup(g)
SimNewCorrGroup(g, K, users)

// These oracles are identical to the corresponding
// oracles in the IUF game for SP as per Fig. 7.

```
SimNewHonUser(u)

require $sk[u] = vk[u] = \perp$ oracles in the

sk[u] \leftarrow "dummy"

vk[u] \leftarrow NewHonUser(u)

Return $vk[u]$
```

```
SimExposeUser(u)

require $sk[u] \neq \perp$

user_is_corrupt[u] \leftarrow true

sk[u] \leftarrow EXPOSEUser(u)

Return $sk[u]$
```

```
SimNewCorrUser(u, sk, vk)

require $sk[u] = vk[u] = \perp$

user_is_corrupt[u] \leftarrow true

NEWCorrUser(u, sk, vk)

sk[u] \leftarrow sk ; vk[u] \leftarrow vk
```

Fig. 23: **Top pane:** Games $g_0 \rightarrow g_3$ for the proof of Theorem 3. The code highlighted in gray was added by expanding the symmetric signcryption algorithms $SP.SigEnc, SP.VerDec$ and the ciphertext-triviality predicate $\text{pred}_{\text{SUF-CMA}}$ in game $g_{SP, pred_{\text{SUF-CMA}},\text{group}}^{\text{IUF}}(A^{\text{IUF}}_{\text{of-SF}})$. The code added for the transitions between games is highlighted in **green**. Bottom pane: Adversary $A^{\text{SUF-CMA}}$ for the proof of Theorem 3. The **highlighted** instructions mark the changes in the code of the simulated game $g_0$.

By combining the above, we have

$$\text{Adv}_{SP, pred_{\text{SUF-CMA}},\text{group}}^{\text{IUF}}(A^{\text{IUF}}_{\text{of-SF}}) = \text{Pr}[g_0] = \sum_{i=0}^{2} \left( \text{Pr}[g_i] - \text{Pr}[g_{i+1}] \right) + \text{Pr}[g_3]$$

$$\leq \sum_{i=0}^{2} \text{Pr}[\text{bad}_{i}] + \text{Pr}[g_3]$$

$$\leq \text{Adv}_{\text{SUF-CMA}}^{\text{SUF-CMA}} + \text{Adv}_{\text{H}}^{\text{H}}(A_{\text{CR}}).$$
We justify our claims in the following paragraphs.

**Game \( g_0 \).** Game \( g_0 \) is functionally equivalent to game \( \text{IUF}_{\text{SUFCMA}} \). The former expands multiple instructions in the latter; the expanded code is marked in gray. Game \( g_0 \) also contains newly added code that is highlighted in green and does not affect its functionality; this code will be used for transitions between games. In particular, the if-then-else statement that was added in oracle \( \text{VERDEC} \) sets the win flag in each of its conditional branches. It follows that

\[
\text{Adv}_{\text{SUFCMA}}^{\text{IUF}_{\text{SUFCMA}}} (A_{\text{SUFCMA}}) = \Pr[g_0].
\]

**Transitions from \( g_0 \) to \( g_3 \).** Let \( i \in \{0, 1, 2\} \). Games \( g_i \) and \( g_{i+1} \) are identical until \( \text{bad} \) is set, therefore

\[
\Pr[g_i] - \Pr[g_{i+1}] \leq \Pr[\text{bad}_i^0].
\]

Each of these transitions removes an instruction that sets the win flag. The win flag can no longer be set in game \( g_3 \), making it impossible for \( A_{\text{IUF}_{\text{SUFCMA}}} \) to win. We have

\[
\Pr[g_3] = 0.
\]

**Bounding \( \Pr[\text{bad}_0^0] \).** In Fig. 23 we build an adversary \( A_{\text{SUFCMA}} \) against the SUFCMA security of DS. It perfectly simulates game \( g_0 \) for adversary \( A_{\text{IUF}_{\text{SUFCMA}}} \) until the \( \text{bad}_0 \) flag is set. Adversary \( A_{\text{SUFCMA}} \) responds to \( A_{\text{IUF}_{\text{SUFCMA}}} \)'s user-oracle queries using its own user oracles, and it simulates group oracles by sampling and maintaining group keys locally. While simulating the SigEnc oracle, it substitutes calls to the signing algorithm DS.Sig with its signing oracle Sign. Whenever \( \text{bad}_0 \) is set, adversary \( A_{\text{SUFCMA}} \) halts the simulation and returns \( (u, m_s, s) \) as the forgery in its own game. Note that in any VERDEC query that sets \( \text{bad}_0 \), the following conditions must be satisfied: \( DS.\text{Ver}(vk[u], m_s, s) \), \( \neg \text{user_is_corrupt}[u] \), and \( T[u, m_s, s] = \bot \). These are equivalent to the three conditions that are necessary to win in the unforgeability game against DS. We have

\[
\Pr[\text{bad}_0^0] \leq \text{Adv}_{\text{DS}}^{\text{SUFCMA}} (A_{\text{SUFCMA}}).
\]

**Bounding \( \Pr[\text{bad}_i^0] \).** In order to upper-bound \( \Pr[\text{bad}_i^0] \), consider an adversary \( A_{\text{CR}} \) against the collision resistance of \( H \) that is defined as follows. Adversary \( A_{\text{CR}} \) perfectly simulates game \( g_i \) for adversary \( A_{\text{IUF}_{\text{SUFCMA}}} \) until the \( \text{bad}_i \) flag is set. To simulate the user oracles \( U \) and the group oracles \( G \), adversary \( A_{\text{CR}} \) itself samples and maintains all user keys and group keys. Whenever the \( \text{bad}_i \) flag is set in \( \text{VERDEC} \), adversary \( A_{\text{CR}} \) returns a pair of strings \( (T[u, m_s, s], m) \). Note that immediately before setting \( \text{bad}_i \), game \( g_i \) verifies that the table entry \( T[u, m_s, s] \) is not empty and that the two strings are distinct. We now show that these strings also produce the same hash value. First, observe that the table \( T \) is filled by oracle SigEnc. For any string assigned to the table entry \( T[u, m_s, s] \), we know that \( h = H(T[u, m_s, s]) \) is true for the hash value \( h \) that is encoded in \( m_s \). While reconstructing the value of \( m_s \) in the current \( \text{VERDEC} \) call, the same hash must have been obtained upon evaluating \( h \leftarrow H(m) \). So the strings returned by \( A_{\text{CR}} \) produce a collision. It follows that

\[
\Pr[\text{bad}_i^0] \leq \text{Adv}_{\text{HS}}^{\text{CR}} (A_{\text{CR}}).
\]

**Bounding \( \Pr[\text{bad}_1^2] \).** Consider a VERDEC query \( (g, u, n, c, \varepsilon) \) that sets \( \text{bad}_2 \) in game \( g_2 \). Immediately prior to setting \( \text{bad}_2 \), game \( g_2 \) verifies that \( T[u, m_s, s] = m \) for \( m_s \) = “Keybase-Chat-2” \( || \langle K[g], n, h \rangle \) that was obtained during the current \( \text{VERDEC} \) call. Since the table \( T \) is only populated inside SigEnc, it follows that the SigEnc oracle was previously queried with \( \langle g', u, n, m, \varepsilon \rangle \) as input, for some \( g' \) satisfying \( K[g'] = K[g] \). In that SigEnc query, let the output of \( \text{NE.Enc} \) be \( c' \). Then \( (\langle g', u, n, m, \varepsilon \rangle, c' \in C \). However, the \( \text{VERDEC} \) oracle in game \( g_2 \) also requires that \( \langle g, u, n, m, \varepsilon \rangle \in C \) before the \( \text{bad}_2 \) flag could be set. This is only possible if adversary \( A_{\text{IUF}_{\text{SUFCMA}}} \) was able to find a distinct \( \text{NE} \) ciphertext \( c \neq c' \) that decrypts to the same value as \( c' \). By combining the information we deduced about the SigEnc and \( \text{VERDEC} \) queries above, we have \( \text{NE.Enc}(K[g], n, \text{NE.Dec}(K[g], n, c, \varepsilon, \varepsilon) \neq c' \). This contradicts the assumed tidiness property of \( \text{NE} \), therefore

\[
\Pr[\text{bad}_1^2] = 0.
\]
C Analysis and Proofs for the Out-Group AE Security of Keybase

C.1 \( M \)-sparsity of Ed25519

In Section 6.3 we defined the \( M \)-SPARSE security of a digital signature scheme for an arbitrary set \( M \subseteq \{0,1\}^* \). In this section, we reduce the \( M \)-SPARSE security of Ed25519 for any sufficiently small \( M \). To do so, we first introduce and analyze two definitions. The first is Discrete Logarithm with Prefix Matching (DL-PM), which is a variant of the Discrete Logarithm (DL) assumption. We verify the DL-PM assumption in the generic group model. The second is Impersonation with Prefix Matching (IMP-PM), which is a new security definition for identification schemes. Intuitively, IMP-PM captures an adversary’s inability to generate a conversation transcript in which the commitment contains a predefined prefix such that an honest verifier would accept the transcript even if the adversary is allowed to choose the verification key used by the verifier. We prove that the IMP-PM security of the identification scheme underlying Ed25519 is implied by the hardness of the DL-PM assumption in the underlying prime-order group. We then use this result to prove that Ed25519 achieves SPARSE security. We introduce these definitions with a particular focus on their relevance to our analysis of Keybase. Our primary intention is not to present these definitions as stand-alone concepts of independent interest but rather to use them as tools essential for our examination of Keybase’s SEAL-Packet-SS scheme.

**Notation and Conventions.** For any positive integer \( n \in \mathbb{N} \setminus \{0\} \), we let \([n] = \{1,2,\ldots,n\} \). We use \( G \) to denote a group. We use the additive notation to write group operations and use \( 0_\ell \) to denote the identity element of \( G \). We assume that the membership test is efficiently computable in \( G \) for all groups we consider. Wherever specified, we let the elements of \( G \) be represented as \( k \)-bit strings where \( k \) is a positive integer. For a cyclic group \( G_p \) of order \( p \) with a generator \( B \), the discrete logarithm of an element \( X \in G_p \) is the value \( x \in \mathbb{Z}_p \) such that \( xB = X \).

![Fig. 24: Left pane: Game defining the discrete logarithm with prefix matching security of a group \( G_p \) with respect to a message space \( M \). Right pane: Game defining the impersonation with prefix matching security of an identification scheme \( ID \) with respect to a message space \( M \).](image)

**The DL-PM Assumption for \( G_p \).** Let \( G_p \) be a cyclic group of order \( p \), and let \( B \in G_p \) be its generator. Consider game \( g^{DL-PM}_{G_p,M}(A_{DL-PM}) \) defined for \( G_p \), a set \( M \subseteq \{0,1\}^* \), and an adversary \( A_{DL-PM} \). The advantage of \( A_{DL-PM} \) against the DL-PM security of \( G_p \) with respect to \( M \) is defined as \( \text{Adv}_{G_{p,M}}^{DL-PM}(A_{DL-PM}) = \text{Pr}[g^{DL-PM}_{G_p,M}(A_{DL-PM})] \). The adversary wins if it finds a tuple \((r,R,\gamma)\) such that \( r \) is the discrete logarithm of \( R \) and \( R \parallel \gamma \in M \). We prove the hardness of DL-PM in the generic group model for sufficiently small \( M \).

**Identification Schemes.** An identification scheme \( ID \) is an interactive protocol involving two parties, the prover and the verifier. The prover is instantiated with a secret key \( sk \) that it uses to send the first message called a commitment. The verifier, instantiated with a public key \( vk \), then samples and sends a random challenge. Lastly, the prover sends a response to the verifier’s challenge. The verifier uses \( vk \) and the conversation transcript to accept or reject the conversation. Formally, \( ID \) specifies three algorithms \( ID.Kg, ID.P = (P_1, P_2) \), and \( ID.V = (V_1, V_2) \). Associated to every instance of \( ID \) is a challenge set \( ChalSet \). The key generation algorithm \( ID.Kg \) returns a secret key \( sk \) and a public verification key \( vk \).
The prover ID.P is a pair of algorithms ID.P₁ and ID.P₂. The commitment algorithm ID.P₁ takes sk as input and returns a commitment com and some state st. The deterministic response algorithm ID.P₂ takes a challenge ch ∈ ChalSet sent by the verifier ID.V₁ and a state st, and returns a response rsp to send to the verifier ID.V₂. The verifier ID.V is a pair of algorithms ID.V₁ and ID.V₂. The algorithm ID.V₁ samples and returns a random challenge ch ∈ ChalSet. The deterministic verification algorithm ID.V₂ takes the tuple (vk, com, ch, rsp) as input and returns true if it accepts the conversation, and false otherwise.

**The IMP-PM Security of ID.** The IMP-PM game GIMP,PMID,M for an identification scheme ID, a message space M, and an adversary AIMP,PMID,M is run in Fig. 24. The advantage of AIMP,PMID,M against the IMP-PM security of ID is given by AdvIMP,PMID,M(AIMP,PM) = Pr[GIMP,PMID,M(AIMP,PM)]. The adversary AIMP,PM is run twice over the course of the game. Its first execution is at the beginning of the game after which it returns the tuple (vk, com, st). The game then proceeds to sample the challenge ch using ID.V₁. Subsequently, the adversary is run once again on input (ch, st). Its final output is (rsp, γ). The adversary wins if ID.V₂(vk, com, ch, rsp) = true and com || γ ∈ M.

**Sparse Security of the Ed25519 Signature Scheme.**

Construction 6. Let k = 256 and prime p ≈ 2^{252}. Let G be a group whose elements are encoded as k-bit strings. Let G_p be the cyclic subgroup of G whose order is p and generator is B. Let H₁ = H₂ = H₃ = SHA-512. Then Ed25519-DS[k, p, G, G_p, B, H₁, H₂, H₃] is the signature scheme as defined in Fig. 25 with Ed25519-DS.sk = 256 and Ed25519-DS.sl = 512.

<table>
<thead>
<tr>
<th>Ed25519.Kg</th>
<th>Ed25519.Sig(sk, m)</th>
<th>Ed25519.Ver(vk, m, s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sk ← s {0, 1}^k // k = 256</td>
<td>c₁</td>
<td></td>
</tr>
<tr>
<td>c₁</td>
<td></td>
<td>c₂ ← H₁(sk) // H₁ = SHA-512</td>
</tr>
<tr>
<td>t ← 2^{k-2} + \sum_{i=1}^{k-3} 2^i \cdot c₁[i]</td>
<td>a ← t mod p</td>
<td>If z ≥ p then return false</td>
</tr>
<tr>
<td>a ← t mod p // p ≈ 2^{252}</td>
<td>r ← H₂(c₂, m) // H₂ = SHA-512</td>
<td>R ← rB; A ← aB</td>
</tr>
<tr>
<td>vk ← aB</td>
<td>ch ← H₃(R, A, m) // H₃ = SHA-512</td>
<td>If R \not∈ G_p or A \not∈ G then return false</td>
</tr>
<tr>
<td>Return (sk, vk)</td>
<td>z ← (r + ch \cdot a) mod p</td>
<td>Return z \cdot B = R + ch \cdot A</td>
</tr>
<tr>
<td></td>
<td>s ← R</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return s</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 25: The Ed25519 signature scheme.

**Proposition 5** (IMP-PM EdID ⇒ SparseEd25519). Let Ed25519 = Ed25519-DS[k, p, G, G_p, B, H₁, H₂, H₃] be the digital signature scheme built from k, p, G, G_p, B, H₁, H₂, H₃ as specified in Construction 6. Let M be a message space that does not depend on H₃. Let A Sparse be any adversary against the M-Sparse security of Ed25519. Let H₃ be modeled as a random oracle and q₁ be the number of random oracle queries that A Sparse makes. Then we build an adversary AIMP,PM such that

\[ \text{Adv}_{\text{Sparse}, \text{Ed25519}, \text{M}}(A_{\text{Sparse}}) \leq q₁ + 1 \cdot \text{Adv}_{\text{EdID}, \text{M}}(A_{\text{IMP,PM}}). \]

**Proof.** We begin with the game G Sparse9Ed25519, M in Fig. 26. The hash function H₃ in Fig. 27 is implicitly used as H₃(·) mod p. So we use the indifferentiability results of Bellare, Davis, and Di [BDD23] to model H₃ as a random oracle. We have

\[ \text{Pr}_{\text{Sparse}, \text{Ed25519}, \text{M}}(A_{\text{Sparse}}) = \text{Adv}_{\text{Sparse}, \text{Ed25519}, \text{M}}(A_{\text{Sparse}}). \]

In Fig. 26 we build an adversary AIMP,PM against the IMP-PM security of EdID that perfectly simulates the sparse game G Sparse9Ed25519, M for A Sparse. The adversary AIMP,PM guesses the random oracle query i for which A Sparse will ultimately attempt a forgery. Note that A Sparse may attempt a forgery on values that it never queried to the random oracle. So AIMP,PM makes a call to the random oracle after A Sparse has attempted to forge a signature. Therefore i ∈ [q₁ + 1]. For every query j ∈ [q₁ + 1] such that j ≠ i,
Proof. We start with the game $\text{Ed25519}$ run on the inputs $\text{oracle query}$ $A$ (on the inputs against the $\text{Proposition 6}$) in $\text{ch}$ used in the proof of $\text{Proposition 5}$. The function $\text{Left pane}$: Adversary $A$ is the identification scheme defined in Fig. 27 with $\text{EdID}$, whose order is $\text{prime}$ $p$. Let $G_p$ be the cyclic subgroup of $G$ whose order is $p$ and generator is $B$. Let $H = \text{SHA-512}$. Then $\text{Ed25519-ID}[k, p, G, G_p, B, H]$ is the identification scheme defined in Fig. 27 with $\text{Ed25519-ID}$. ChalSet $= Z_p$.

$A_{\text{IMP-PM}}$ lazily simulates the output of $H$. For the $j = i$ case, we introduce a new instruction $\text{output}$ which is to be interpreted as follows. When $A_{\text{IMP-PM}}$ encounters this instruction during the $i^{\text{th}}$ random oracle query $(R^*, A^*, m^*)$, it returns the values $A^*, R^*$, and its current state $\text{st}$ to its own game. It is then run on the inputs $(\text{ch}, \text{st})$. It uses the state $\text{st}$ to resume executing in the context of $\text{SimH}$, assigning $\text{ch}$ to the variable $\text{ch}^*$. Finally, $A_{\text{IMP-PM}}$ returns $\text{ch}$ as the output of the $i^{\text{th}}$ $H$ query. Recall that the challenge $\text{ch}$ in $A_{\text{IMP-PM}}$'s game is sampled uniformly at random from $Z_p$. Also, note that $\text{output}$ is guaranteed to execute exactly once. So, this change is undetectable for $A_{\text{ASPARSE}}$.

At some point, $A_{\text{ASPARSE}}$ returns the forgery $(vk, m, s, \gamma)$ where $s$ is of the form $R || z$. Adversary $A_{\text{IMP-PM}}$ returns $(R, z || \gamma)$ and wins whenever $(vk, R) = (A^*, R^*)$ and $A_{\text{ASPARSE}}$ wins. We know that

$$\Pr[(vk, R) = (A^*, R^*)] \geq 1/(q_h + 1)$$

Consider the winning conditions for $A_{\text{ASPARSE}}$. We have $A^* \in G$, $R^* \in G_p$, $z \cdot B = R^* + \text{ch}^* \cdot A^*$ and $s || \gamma \in \mathcal{M}$ all hold. The last condition simplifies to $R || (z || \gamma) \in \mathcal{M}$. These are exactly the same as the winning conditions for $A_{\text{IMP-PM}}$. Hence,

$$\text{Adv}_{\text{EdID,\mathcal{M}}}^{\text{IMP-PM}}(A_{\text{IMP-PM}}) \geq 1/(q_h + 1) \cdot \text{Adv}_{\text{Ed25519,\mathcal{M}}}^{\text{ASPARSE}}(A_{\text{ASPARSE}})$$

This concludes the proof.

**IMP-PM SECURITY OF THE ID SCHEME UNDERLYING Ed25519.**

**Construction 7.** Let $k = 256$ and prime $p = 2^{252}$. Let $G$ be a group whose elements are encoded as $k$-bit strings. Let $G_p$ be the cyclic subgroup of $G$ whose order is $p$ and generator is $B$. Let $H = \text{SHA-512}$. Then $\text{Ed25519-ID}[k, p, G, G_p, B, H]$ is the identification scheme defined in Fig. 27 with $\text{Ed25519-ID}$. ChalSet $= Z_p$.

**Proposition 6** ($\text{DL-PM}_{G_p} \Rightarrow \text{IMP-PM}_{\text{EdID}}$). Let $\text{EdID} = \text{Ed25519-ID}[k, p, G, G_p, B, H]$ be the identification protocol built from $k$, $p$, $G$, $G_p$, $B$, $H$ as specified in **Construction 7**. Let $A_{\text{IMP-PM}}$ be any adversary against the IMP-PM security of EdID with respect to the message space $\mathcal{M}$. Then, we build an adversary $A_{\text{DL-PM}}$ such that

$$\text{Adv}_{\text{EdID,\mathcal{M}}}^{\text{IMP-PM}}(A_{\text{IMP-PM}}) \leq \sqrt{\text{Adv}_{G_p,\mathcal{M}}^{\text{DL-PM}}(A_{\text{DL-PM}})} + 1/p.$$  

Proof. We start with the game $g_{\text{EdID,\mathcal{M}}}^{\text{IMP-PM}}$ in Fig. 28. We have

$$\Pr[g(A_{\text{IMP-PM}})] = \text{Adv}_{\text{EdID,\mathcal{M}}}^{\text{IMP-PM}}(A_{\text{IMP-PM}}).$$

Adversary $A_{\text{DL-PM}}$ runs $A_{\text{IMP-PM}}$ and simulates the game $g_{\text{IMP-PM}}^{\text{EdID,\mathcal{M}}}$. At some point, $A_{\text{IMP-PM}}$ returns the tuple $(vk, \text{com}, \text{st})$. Then $A_{\text{DL-PM}}$ samples a challenge $\text{ch}_0$ uniformly at random from $Z_p$ and runs $A_{\text{IMP-PM}}$ on the inputs $(\text{ch}_0, \text{st})$. Finally, $A_{\text{IMP-PM}}$ terminates with output $(z_0, \gamma_0)$. Subsequently, $A_{\text{DL-PM}}$ resets
Fig. 27: The identification protocol EdID for the proof of Proposition 6.

\[
\begin{array}{|c|c|c|}
\hline
\text{EdID.Kg} & \text{EdID.P}_1(sk) & \text{EdID.P}_2(ch, st) \\
\hline
sk \leftarrow \{0, 1\}^k & e_1 \parallel e_2 \leftarrow H(sk) & (r, a) \leftarrow st \\
& H = \text{SHA-512} & z \leftarrow (r + ch - a) \mod p \\
& |e_1| = |e_2| = 256 & \text{rsp} \leftarrow z; \text{Return rsp} \\
t \leftarrow 2^{k-2} + \sum_{i=3}^{k-2} 2^i \cdot e_1[i] & a \leftarrow t \mod p; r \leftarrow s \mod Z_p & \\
a \leftarrow t \mod p & \text{st} \leftarrow (r, a) & \\
vk \leftarrow aB & R \leftarrow rB; \text{com} \leftarrow R & \\
\text{Return} (sk, vk) & \text{Return} (\text{com, st}) & \\
\end{array}
\]

Fig. 28: Left pane: The security game used in the proof of Proposition 6. We use gray highlighting to show code from EdID. Right pane: Adversary \( \mathcal{A}_{\text{DL-PM}} \) used in the proof of Proposition 6. The interpretation of the line highlighted green is explained in the text below.

\( \mathcal{A}_{\text{IMP-PM}} \)'s state back to the point just after when it had returned \((vk, \text{com, st})\), chooses a new random challenge \( ch_1 \) and sends it to \( \mathcal{A}_{\text{IMP-PM}} \). The adversary \( \mathcal{A}_{\text{IMP-PM}} \) returns another response \((z_1, \gamma_1)\).

To explain the step \((Q + T) \leftarrow vk\), we present some details about the group \( G \) underlying EdID. The group \( G \) satisfies \( n = 8 \cdot p = 2^5 \cdot p \) where \( n \) is the order of \( G \). Using Kronecker’s decomposition theorem we have that in addition to the prime order subgroup \( G_p \), the group \( G \) contains another subgroup \( G_t \) of order \( 8 \) such that the group \( G \) is the direct product of \( G_p \) and \( G_t \), i.e., \( G = G_p \times G_t \). This means that every element \( x \in G \) can be uniquely written as \( Q + T \) for \( Q \in G_p \) and \( T \in G_t \). We call \( G_t \) the torsion subgroup of \( G \), and \( T \) the torsion component of \( X \). Consider the check \( z \cdot B = R + ch \cdot A \). This is equivalent to \( z \cdot B - R = ch \cdot A \) which in turn equivalent to \( z \cdot B - R = ch + T \). By the time this check is performed by \( \mathcal{A}_{\text{DL-PM}} \), we know that \( z \cdot B - R \in G_p \). For equality to hold, \( ch + T \in G_p \) must be true. This can only happen if \( T = 0 \). Therefore, \( z \cdot B - R = ch \cdot A \Rightarrow z \cdot B - R = ch \cdot Q \).

Let \( E \) be the event that all three conditional statements in Fig. 28 evaluate to true i.e. \( ch_0 \neq ch_1, z_0 \cdot B = R + ch_0 \cdot Q, z_1 \cdot B = R + ch_1 \cdot Q \), and \( R \parallel \gamma_i \in M \) for \( i \in \{0, 1\} \) all hold simultaneously. Whenever the event \( E \) occurs, \( \mathcal{A}_{\text{DL-PM}} \) computes the discrete log \( r \) of \( R \) by solving the system of equations \( z_0 = r + ch_0 \cdot q \) and \( z_1 = r + ch_1 \cdot q \), returns \((r, R, \gamma_0)\) and wins its game. It follows that

\[
\text{Adv}_{\mathcal{G}_p, M}^{\mathcal{A}_{\text{DL-PM}}} \geq \text{Pr}[E].
\]

We use the Reset Lemma [BP02] to bound \( \text{Pr}[E] \). The Reset Lemma upper-bounds the advantage of a dishonest prover \( \mathcal{A}_{\text{IMP-PM}} \) as a function of the probability that the prover produces two accepting
responses in an experiment that resets it as

$$\text{Adv}_{\text{EID}, M}^{\text{IMP-PM}}(A_{\text{IMP-PM}}) \leq 1/|\text{ChalSet}| + \Pr[E]$$

where ChalSet is the set from which the verifier samples challenges. In our case ChalSet = $Z_p$. Combining the probabilities above we get

$$\text{Adv}_{\text{EID}, M}^{\text{IMP-PM}}(A_{\text{IMP-PM}}) \leq \sqrt{\text{Adv}_{G_{p}, M}^{\text{DL-PM}}(A_{\text{DL-PM}})} + 1/p.$$ 

This concludes the proof.

**Shoup’s Generic Group Model.** We analyze the DL-PM security of $G_p$ in Shoup’s generic group model which uses a random injective map $\sigma : Z_p \rightarrow S$ for some set of strings $S$ and thinks of the string $\sigma(x)$ as the representation of the group element $x \Bbb{B}$. It gives the adversary access to two oracles, the generator exponentiation oracle Exp and the group operation oracle Op. On input $x$, the Exp oracle returns $\sigma(x)$ which is the representation of $x \Bbb{B}$. On input $(a, b, X, Y)$, the Op oracle returns the representation of $aX + bY$ if $X$ and $Y$ are valid string representations of group elements and returns $\perp$ otherwise.

**DL-PM in Shoup’s Generic Group Model.**

**Proposition 7.** Let $S = \{0, 1\}^k$. Let $M = \{\text{pre} \parallel \nu \mid \nu \in \{0, 1\}^*\}$ where pre is the 17-byte prefix defined in Section 6.3. Let $A_{\text{DL-PM}}$ be any adversary against the DL-PM security of $G_p$ with respect to $M$, making at most $q_{\text{Exp}}$ and $q_{\text{Op}}$ queries to its Exp and Op oracles respectively. Then we show that

$$\text{Adv}_{G_{p}, M}^{\text{DL-PM}}(A_{\text{DL-PM}}) \leq ((q_{\text{Exp}} + q_{\text{Op}} + 2) \cdot (2^{k-|\text{pre}|} + q_{\text{Exp}} + 2 \cdot q_{\text{Op}} + 2))/2^k + (q_{\text{Exp}} + q_{\text{Op}} + 2)^2/p.$$ 

**Overview.** This proof uses games $g_0$ through $g_4$ in Fig. 29 and Fig. 30. We establish the following claims.

- $\text{Adv}_{G_{p}, \text{pre}}^{\text{DL-PM}}(A_{\text{DL-PM}}) = \Pr[g_0]$  
- $\Pr[g_0] = \Pr[g_1] = (\Pr[g_1] - \Pr[g_2]) + \Pr[g_2]$  
- $\Pr[g_1] - \Pr[g_2] \leq \Pr[\text{bad}_0]$  
- $\Pr[\text{bad}_1] \leq (q_{\text{Exp}} + q_{\text{Op}} + 2) \cdot (2^{k-|\text{pre}|} + q_{\text{Exp}} + 2 \cdot q_{\text{Op}} + 2))/2^k$  
- $\Pr[\text{bad}_2] \leq (q_{\text{Exp}} + q_{\text{Op}} + 2)^2/p$  
- $\Pr[\text{bad}_3] \leq (q_{\text{Exp}} + q_{\text{Op}} + 2)^2/p$  
- $\Pr[\text{bad}_4] = 0$

By combining the above, we have

$$\text{Adv}_{G_{p}, \text{pre}}^{\text{DL-PM}}(A_{\text{DL-PM}}) = \Pr[g_0] = \Pr[g_1] = (\Pr[g_1] - \Pr[g_2]) + \Pr[g_2] \leq \Pr[\text{bad}_0] + \Pr[g_3] \leq \Pr[\text{bad}_0] + (\Pr[g_3] - \Pr[g_4]) + \Pr[g_4] \leq \Pr[\text{bad}_1] + \Pr[\text{bad}_4] \leq (q_{\text{Exp}} + q_{\text{Op}}) \cdot (2^{k-|\text{pre}|} + q_{\text{Exp}} + 2 \cdot q_{\text{Op}})/2^k + (q_{\text{Exp}} + q_{\text{Op}} + 2)^2/p.$$ 

We justify our claims in the following paragraphs.

**Game $g_0$.** We start with the game $g_0$ in Fig. 29. The game $g_0$ is a modified version of the game $g_{G_{p}, M}^{\text{DL-PM}}$ shown in Fig. 24. We use the notation $\Phi(Z_p, S)$ to denote the set of all injections with domain $Z_p$ and range $S$. The game $g_0$ replaces the check $R \parallel \gamma \in M$ of $g_{G_{p}, M}^{\text{DL-PM}}$ with the check $R \in M$. Recall that the length of the prefix pre used to define $M$ is 136 bits long and that group elements are represented as 256-bit strings. Thus $R \parallel \gamma \in M$ is only possible is $R \in M$. It follows that

$$\text{Adv}_{G_{p}, M}^{\text{DL-PM}}(A_{\text{DL-PM}}) = \Pr[g_0].$$

**Transition from $g_0$ to $g_1$.** The game $g_1$ performs a lazy simulation of the random injection $\sigma$. To do this the game uses the table $T$, initially empty, that partially represents $\sigma$. Every time the adversary makes an Exp query on a value $x$, the oracle lazily samples the output and returns it. Now the adversary may query Op on inputs $(a, b, X, Y)$ such that either $X$ or $Y$ is not in $\text{Rng}(T)$. For such queries, the oracle Op needs to probabilistically decide if they are in $\text{Rng}(\sigma)$. Whenever the oracle decides that a point $X \in S$ does not have a preimage, it remembers that decision by maintaining a set $S_{\text{reject}}$ of rejected points. The oracle assigns a preimage to $X$ with probability $(p - \text{Rng}(T))/(|S| - |\text{Rng}(T)| - |S_{\text{reject}}|)$ by sampling $j \leftarrow [|S| - \text{Rng}(T) - |S_{\text{reject}}|]$ and checking if $j \leq p - \text{Rng}(T)$. We compute this probability as follows. There are $p$ points in $\text{Rng}(\sigma)$. This means that when a query $(a, b, X, Y)$ is made to Op such that $X \notin \text{Rng}(T)$ there are at most $p - |\text{Rng}(T)|$ points that the oracle can decide to assign a preimage for.
Games $g_0$. We use the notation $\Phi(\mathbb{Z}_p, S)$ to denote the set of all injections with domain $\mathbb{Z}_p$ and range $S$.

The codomain of $\sigma$ contains $|S|$ points of which $|\text{Rng}(T)|$ already have a preimage and $|S_{\text{reject}}|$ have been rejected. This leaves a total of $|S| - |\text{Rng}(T)| - |S_{\text{reject}}|$ points in the codomain. Therefore the probability that a point $X \not\in \text{Rng}(T)$ belongs in $\text{Rng}(\sigma)$ is $(p - \text{Rng}(T))/(|S| - |\text{Rng}(T)| - |S_{\text{reject}}|)$. The table $T$ perfectly simulates the injection $\sigma$. Therefore,

$$\Pr[g_0] = \Pr[\hat{g}_1].$$

Games $g_1-4$. We use $S_{\text{pre}}$ to denote the set of strings in $S$ that contain the prefix $\text{pre}$.

**Transition from $g_1$ to $g_2$.** Games $g_1$ and $g_2$ are identical until bad is set, therefore

$$\Pr[g_1] - \Pr[g_2] \leq \Pr[\text{bad}_0].$$

**Bounding $\Pr[\text{bad}_0].** The bad flag is set in $g_1$ if the EXP oracle samples a string containing the prefix pre from $S \setminus (\text{Rng}(T) \cup S_{\text{reject}})$. Game $g_2$ adds an instruction that reassigns $T[x]$ to a value that does not contain the prefix pre. We use a counting argument to bound $\Pr[\text{bad}_0]$. Strings in $S$ are $k$ bits long. We use $S_{\text{pre}}$ to denote the set of strings in $S$ that contain the prefix pre. We have $|S_{\text{pre}}| = 2^{k-\text{pre}}$. Of these, $|(\text{Rng}(T) \cup S_{\text{reject}}) \cap S_{\text{pre}}|$ have either already been added to $\text{Rng}(T)$ or to $S_{\text{reject}}$ during prior OP queries. The probability that EXP picked one of the remaining $2^{k-\text{pre}} - |(\text{Rng}(T) \cup S_{\text{reject}}) \cap S_{\text{pre}}|$ values is $(2^{k-\text{pre}} - |(\text{Rng}(T) \cup S_{\text{reject}}) \cap S_{\text{pre}}])/(|S| - |\text{Rng}(T)| - |S_{\text{reject}}|)$. Hence, the probability that a query
to Exp set $\text{bad}_0$ is bounded by
\[
\frac{2^k - |\text{pre}| - (|\text{Rng}(T) \cup \text{S}_{\text{reject}}| \cap \text{S}_{\text{pre}})}{|S| - |\text{Rng}(T)| - |\text{S}_{\text{reject}}|} \leq \frac{2^k - |\text{pre}|}{2^k}.
\]
Applying a union bound over all Exp queries, we have
\[
\Pr[\text{bad}_0] \leq \frac{(q_{\text{Exp}} + q_{\text{OP}} + 2) \cdot (2^k - |\text{pre}| + q_{\text{Exp}} + 2 \cdot q_{\text{OP}} + 2)}{2^k}.
\]

**Transition $\mathcal{G}_2$ to $\mathcal{G}_3$.** In game $\mathcal{G}_2$ we update the table $T$ such that it performs a “symbolic” simulation of the injection $\sigma$. Specifically, we introduce indeterminates $Z_i$ so the table $T$ is of the form $T : \mathbb{Z}_p[Z_i] \rightarrow S$, i.e., the domain of $T$ now consists of (linear) polynomials of the form $P(Z_i) = \alpha Z_i + \beta$ for $\alpha, \beta \in \mathbb{Z}_p$.

Note that the adversary can only make constant polynomial queries directly to Exp. However, it may indirectly query Exp on a non-constant polynomial via an OP query. Whenever a query $\text{Exp}(x)$ is made in $\mathcal{G}_2$, the oracle checks if $\exists P \in \text{Dom}(T) : P(z) = x(z)$ where $z$ is the vector of all $z_i \in \mathbb{Z}_p$ sampled so far. If the conditional evaluates to true, the oracle sets the $\text{bad}_1$ flag and sets $T[x]$ to $T[P]$. This ensures that games $\mathcal{G}_2$ and $\mathcal{G}_3$ are identical. It follows that
\[
\Pr[\mathcal{G}_2] = \Pr[\mathcal{G}_3].
\]

**Transition $\mathcal{G}_3$ to $\mathcal{G}_4$.** Games $\mathcal{G}_3$ and $\mathcal{G}_4$ are identical until $\text{bad}_1$ is set, therefore
\[
\Pr[\mathcal{G}_3] - \Pr[\mathcal{G}_4] \leq \Pr[\text{bad}_1^4].
\]

Game $\mathcal{G}_3$ removes the instruction that assigns $T[P]$ to $T[x]$.

**Bounding $\Pr[\text{bad}_1^4]$.** We use the Schwartz-Zippel lemma to bound $\Pr[\text{bad}_1^4]$. Note that $\text{bad}_1$ is only set if $T[x] = \bot$ and $\exists P \in \text{Dom}(T) : P(z) = x(z)$. This implies that $x \neq P$ and $x(z) = P(z)$ or $(P - x)(z) = 0$. As $P - x$ is a linear polynomial and $z$ is sampled uniformly at random from $\mathbb{Z}_p$, we use the Schwartz-Zippel lemma to bound the probability that a particular Exp query sets $\text{bad}_1$ as $\Pr[\text{bad}_1] \leq \Pr[\text{bad}_1^4]$.

Applying a union bound over all Exp queries, we have
\[
\Pr[\text{bad}_1^4] \leq \frac{(q_{\text{Exp}} + q_{\text{OP}} + 2)^2}{p}.
\]

**Bounding $\Pr[\mathcal{G}_4]$.** The adversary cannot win $\mathcal{G}_4$. To win, $A_{\text{DL-PM}}$ must return a constant polynomial $r$ and a bitstring $r$ such that $R$ begins with $\text{pre}$ and $T[r] = R$ holds at the end of the execution of the game. This is impossible because values written into $T$ by Exp never begin with $\text{pre}$ and OP only writes to $T$ for non-constant polynomials. Hence,
\[
\Pr[\mathcal{G}_4] = 0.
\]

### C.2 $\Phi_{\text{SP}}$ Derives Messages from Hashed NE Keys

In this section, we justifity our claim that the key cycle in SealPacket can be viewed as an instance of deriving the message from a hashed key. Let $\Phi_{\text{SP}} = \text{SIGENC-DER}[\text{NE}, H, DS, m, n, sk] \cup \text{ENC-DER}[m]$ be the message deriving function defined in Construction 5. Then we build functions $\gamma$ and $H$ for $\Phi_{\text{SP}}$ that satisfy Definition 4. The case of ENC-DER[m] is trivial: the corresponding $\gamma$ is the constant function that always returns $m$ (regardless of its input $H$). So we focus on the other case now. Let $\phi = \text{SIGENC-DER}[\text{NE}, H, DS, m, n, sk]$ for any $m \in \{0,1\}^*$, $n \in \{0,1\}^{\text{NE.nl}}$, and $sk \in \{0,1\}^{\text{DS.skd}}$. In Keybase, $\text{NE} = \text{XSalsa20-Poly1305}, H = \text{SHA-512}$, and $\text{DS} = \text{Ed25519}$. We provide the expanded definition of $\phi$ in Fig. 31 wherein $\mathbb{G}_p$ be the cyclic group underlying the Ed25519 scheme as specified in Construction 6. In particular, the order of $\mathbb{G}_p$ is $p$, the generator of $\mathbb{G}_p$ is $B$, and the elements of $\mathbb{G}_p$ are encoded as 256-bit strings.

Note that the only lines within the message deriving function $\phi$ that use the input key $K$ are the ones that are indicated with a \textbf{box}. Additionally, we define $r$ and $ch$ to be the outputs of the SHA-512 calls modulo $p$. These values were likewise used only modulo $p$ in the function $\phi$; we merely made this explicit in Fig. 31.
We define the function $H$ corresponding to $\phi$ in Fig. 31. It encapsulates the two SHA-512 mod $p$ instances within $\phi$. In recent work [BDD23] formalized and showed the indifferentiability of SHA-512 reduced modulo a sufficiently large prime $p$ as a part of the broader analysis of the Ed25519 scheme. If we show that the two SHA-512 mod $p$ instances in $\phi$ (namely $H_0$ and $H_1$) can be modeled as two independent random oracles, then we can directly apply the result of [BDD23] to prove that $H$ can be reasonably treated as a random oracle. For this, it would suffice to show that (a) each SHA-512 call takes inputs of a fixed length, and (b) the input lengths of these SHA-512 calls are distinct. These properties follow from $K, n, h, e_2, R$, and $A$ being fixed-length. With this definition of $H$, we can straightforwardly build $\gamma$ such that $\phi = \gamma(H(K))$, where $H$ can be modeled as a random oracle.

**Fig. 31: The function $\phi = \text{SIGENC-DER}[\text{NE}, H, \text{Ed25519}, m, n, sk]$**

- $H(K, n, h, X)$
  - $\text{require } |n| = \text{XSalsa20-Poly1305.nl}$
  - $\text{require } |h| = \text{SHA-512.cl}$
  - $\text{require } |K| = \text{XSalsa20-Poly1305.kl}$
  - $\text{require } |X| \in \{256, 512\}$
  - $m_s \leftarrow \text{"Keybase-Chat-2" } \lang K, n, h \rang$

If $|X| = 256$ then
- Return $H_0(X, m_s)$ // $H_0(\cdot) = \text{SHA-512(\cdot)} \mod p$
- Return $H_1(X, m_s)$ // $H_1(\cdot) = \text{SHA-512(\cdot)} \mod p$

**CHALLENGES IN MODELLING $H$ AS A RANDOM ORACLE.** Here we enumerate the subtle issues with claiming that the function $H$ defined above satisfies Definition 4.

First of all, according to Definition 4, the domain of the function $H$ should be $\{0, 1\}^{\text{XSalsa20-Poly1305.kl}}$ where XSalsa20-Poly1305.kl = 256. However, the functions $H_0$ and $H_1$ in Fig. 31 take inputs of the form $(e_2, m_s, R, A, m_s)$ where $|e_2| = |R| = |A| = 256$ and $m_s = 1072$. This means that the domain $D$ of $H$ is $\{0, 1\}^{256} \times \{0, 1\}^{1072} \cup \{0, 1\}^{256+1072}$ and its range $R$ is $\mathbb{Z}_p$. We fix this in the next subsection by defining a function $F : \{0, 1\}^{\text{XSalsa20-Poly1305}} \rightarrow \{f : f' \rightarrow R\}$ where $D' = \{0, 1\}^{704} \times \{0, 1\}^{256} \cup \{0, 1\}^{512}$ such that $F(K)(x) = H(K, x)$. Then we define $\gamma_0$ corresponding to $F$ such that $\phi = \gamma_0(F(K))$. Since $F$ is the composition of an injection and a function that can be modeled as a random oracle, $F$ can also be modeled as a random oracle.

Secondly, we have thus far omitted discussing the two other SHA-512 (not mod $p$) calls within $\phi$. If we tried to formally reduce security to the indifferentiability result of [BDD23] these extra uses of SHA-512 require small tweaks to the approach to handle. We provide an overview of these tweaks in the last subsection of this appendix.

**PER-KEY RANDOM ORACLARES.** Let $D' = \{0, 1\}^{704} \times \{0, 1\}^{256} \cup \{0, 1\}^{512}$ and $R = \mathbb{Z}_p$. Then we define the functions $F$ and $\gamma_0$ as discussed above in Fig. 32. The function $F$ takes an XSalsa20-Poly1305 key $K \in \{0, 1\}^{\text{XSalsa20-Poly1305.kl}}$ as input, and it returns the description of another function $f[K] : D' \rightarrow R$. We emphasize that the output of $F(K)$ specifies the function $f[K]$ that can itself be evaluated on arbitrary input tuples $(u, h, X)$. Note that there is domain separation. For $K \neq K'$, $f[K]$ and $f[K']$ will never call SHA-512 mod $p$ on the same inputs. Consequently, (modeling SHA-512 mod $p$ as a random oracle) each $f[K]$ is independent and random, so $F$ can be viewed as a random oracle. We also define $\gamma_0[m, n, sk]$ corresponding to $\phi$ in Fig. 32. It is straightforward to see that $\phi = \text{SIGENC-DER}[\text{NE}, H, \text{DS}, m, n, sk]$ can be expressed as $\gamma_0(F(K))$. Except for the nuance we discuss in the next subsection, we have built the functions $H = F$ and $\gamma = \gamma_0$ corresponding to the message deriving function $\phi$ such that they satisfy Definition 4.
In game

In this game we embed the code for the simulator from the previous step inside the

In the following game transitions, we start with the game $G_0$, $A_{KDMAE}$ internally calls the standard model function $SHA-512$ and picks $\phi = SIGENC-DER|XSalsa20-Poly1305, SHA-512, Ed25519, m, n, sk$ which runs the standard model calls to both plain $SHA-512$ (of the form $h \leftarrow SHA-512(m)$ and $e_1||e_2 \leftarrow SHA-512(sk)$) and $SHA-512$ mod $p$.

2. **Game $G_1$.** In $G_1$, we replace the function $\phi$ with its modified version $\tilde{\phi}$. Unlike the former, the latter does not invoke plain $SHA-512$ but instead takes the values $h$ and $e_1||e_2$ as part of its input. We then have $A_{KDMAE}$ compute the values $h$ and $e_1||e_2$ locally by running $SHA-512$ before its calls to $\phi$. As this transition only involves reorganization of logic, $G_1$ is equivalent to $G_0$.

3. **Game $G_2$.** In game $G_2$, we note that $SHA-512$ is an MD-based hash function, computed by repeatedly invoking the compression function $sha$. We replace all calls to $SHA-512$ mod $p$ with the MD construction mod $p$ that uses the compression function $sha$ inside of adversary and the function $\phi$. We replace all calls to plain $SHA-512$ with the MD construction using the compression function $sha$ inside of adversary. So this step does not change the view of the adversary and hence $G_2$ is equivalent to $G_1$.

4. **Game $G_3$.** Now we switch from the standard model to the ideal model and replace all instances of the compression function $sha$ with the ideal function $h$. Here we assume that the compression function $sha$ can be modeled as an idealized compression function. This heuristic step sets us for the indifferentiability analysis.

5. **Game $G_4$.** In game $G_4$, we use the indifferentiability results of [BDD23] to replace the MD mod $p$ construction (with access to $h$) used by $\tilde{\phi}$ with the random oracle $H$. This changes all other queries to $h$ (particularly, those internal to $A_{KDMAE}$ that arose from the calls to plain $SHA-512$) to instead use a stateful simulator with access to $H$ to simulate $h$.

6. **Game $G_5$.** In this game we embed the code for the simulator from the previous step inside $A_{KDMAE}$.

So instead of having direct access to the simulator, the adversary runs it as part of its own code. This step only constitutes syntactic changes and hence $G_5$ is equivalent to $G_4$.

Fig. 32: **Left pane:** The function $F$ corresponding to $\phi$. **Right pane:** The function $\gamma_0[m, n, sk]$ corresponding to $\phi$.

### Proof Flow from Indifferentiability
As we mentioned earlier, the presence of the two plain (i.e. not reduced modulo $p$) $SHA-512$ calls in $\phi$ introduces subtleties that require us to tweak our steps so far. Recall that we want to show that $\phi$ derives messages from a hashed key and use this property to prove the KDMAE security of $XSalsa20-Poly1305$ with respect to $\phi$. We now give an overview of the required tweaks and the overall proof flow that would be required to leverage the indifferentiability results of [BDD23].

In the following game transitions, we start with the game $G_0$ that captures the KDMAE security of $NE = XSalsa20-Poly1305$ with respect to the message deriving function $\phi$. We sketch intermediate game transitions to ultimately reach $G_6$ that captures the KDMAE security of $XSalsa20-Poly1305$ with respect to the function $\gamma_1(F(.))$ where we define $\gamma_1$ along the way and $F$ is a random oracle.

1. **Game $G_0$.** In $G_0$, $A_{KDMAE}$ internally calls the standard model function $SHA-512$ and picks $\phi = SIGENC-DER|XSalsa20-Poly1305, SHA-512, Ed25519, m, n, sk$ which runs the standard model calls to both plain $SHA-512$ (of the form $h \leftarrow SHA-512(m)$ and $e_1||e_2 \leftarrow SHA-512(sk)$) and $SHA-512$ mod $p$.

2. **Game $G_1$.** In $G_1$, we replace the function $\phi$ with its modified version $\tilde{\phi}$. Unlike the former, the latter does not invoke plain $SHA-512$ but instead takes the values $h$ and $e_1||e_2$ as part of its input. We then have $A_{KDMAE}$ compute the values $h$ and $e_1||e_2$ locally by running $SHA-512$ before its calls to $\phi$. As this transition only involves reorganization of logic, $G_1$ is equivalent to $G_0$.

3. **Game $G_2$.** In game $G_2$, we note that $SHA-512$ is an MD-based hash function, computed by repeatedly invoking the compression function $sha$. We replace all calls to $SHA-512$ mod $p$ with the MD construction mod $p$ that uses the compression function $sha$ inside of adversary and the function $\phi$. We replace all calls to plain $SHA-512$ with the MD construction using the compression function $sha$ inside of adversary. So this step does not change the view of the adversary and hence $G_2$ is equivalent to $G_1$.

4. **Game $G_3$.** Now we switch from the standard model to the ideal model and replace all instances of the compression function $sha$ with the ideal function $h$. Here we assume that the compression function $sha$ can be modeled as an idealized compression function. This heuristic step sets us for the indifferentiability analysis.

5. **Game $G_4$.** In game $G_4$, we use the indifferentiability results of [BDD23] to replace the MD mod $p$ construction (with access to $h$) used by $\tilde{\phi}$ with the random oracle $H$. This changes all other queries to $h$ (particularly, those internal to $A_{KDMAE}$ that arose from the calls to plain $SHA-512$) to instead use a stateful simulator with access to $H$ to simulate $h$.

6. **Game $G_5$.** In this game we embed the code for the simulator from the previous step inside $A_{KDMAE}$.

So instead of having direct access to the simulator, the adversary runs it as part of its own code. This step only constitutes syntactic changes and hence $G_5$ is equivalent to $G_4$. $G_4$ Incorporating the plain $SHA-512$ calls into the adversary $A_{KDMAE}$'s code in $G_1$ was done as a set up for this step, where it is important that the simulator is only ever run internally to $A_{KDMAE}$ and not in the message derivation function. Otherwise we would have issues as the simulator is stateful.
7. Game $g_6$. In our final transition, we replace $H$ and $\hat{\phi}$ with $F$ and $\gamma_1(F(\cdot))$ respectively. Here $F$ is as defined in Fig. 32 except SHA-512 mod $p$ is replaced with $H$ and $\gamma_1$ is the same as $\gamma_0$ except it does not make calls to plain SHA-512 and is parameterized by $[h, n, e_1, |e_2|]$. Thus we have shown that the message deriving function $\phi$ derives messages from a hashed key.

C.3 KDMAE for Messages Derived from a Hashed Key (Proof of Proposition 1)

**OVERVIEW.** This proof uses games $g_0$ through $g_4$ in Fig. 33. We establish the following claims.

- $\text{Adv}^{\text{KDMAE}}_{\text{NE}, \phi}(A_{\text{KDMAE}}) = 2 \cdot \text{Pr}[g_0] - 1$
- $\text{Pr}[g_0] - \text{Pr}[g_1] \leq \text{Pr}[\text{bad}_0^6]$
- $\text{Pr}[\text{bad}_0^6] \leq 3^{\gamma_0} \text{NewHonGroup}/2^{\text{NE}.k_i+1}$
- $\text{Pr}[g_1] = \text{Pr}[g_2]$

By combining the above, we have

$$\text{Adv}^{\text{KDMAE}}_{\text{NE}, \phi}(A_{\text{KDMAE}}) = 2 \cdot \text{Pr}[g_0] - 1 = 2 \cdot \left( \sum_{i=0}^{3} (\text{Pr}[g_i] - \text{Pr}[g_{i+1}]) + \text{Pr}[g_4] \right) - 1$$

$$\leq 2 \cdot (\text{Pr}[\text{bad}_0^6] + \text{Pr}[\text{bad}_1^6] + \text{Pr}[g_4]) - 1$$

$$\leq 2\gamma_0 \text{NewHonGroup}/2^{\text{NE}.k_i} + 2 \cdot \text{Adv}^{KR}_{\text{NE}}(A_{KR}) + \text{Adv}^{\text{AEAD}}_{\text{NE}}(A_{\text{AEAD}}).$$

We justify our claims in the following paragraphs.

**GAME $g_0$.** As per Proposition 1, consider game $g_{\text{KDMAE}}^{\text{NE}, \phi}$ where $\phi$ derives messages from a hashed key so every message-deriving function is of the form $\phi(x)(K[g]) = \gamma(H(K[g]))$, where $H$ is modeled as a random oracle $R$. We let $R$ denote the random oracle’s range. We rewrite this game in a functionally equivalent way, obtaining game $g_0$ as our starting point in this proof. The highlighted code in $g_0$ marks the changes that we introduced while rewriting $g_{\text{KDMAE}}^{\text{NE}, \phi}$. The code highlighted in **green** is only used for transitions between games $g_0$ through $g_4$; we will explain this code later. The code highlighted in gray introduces the following changes: (1) it writes $\phi(x)(K[g])$ as $\gamma(H(K[g]))$, (2) it populates and updates the sets $S_{\text{honest}}, S_{\text{reused}}$ that contain “honest” and “reused” NE keys, and (3) it adds a simulated random oracle $H_{\text{secret}}$ that is meant to be called only on inputs that are keys of honest groups.

We will now explain the meaning of these sets, and justify that they do not alter the functionality of the initial game. It will follow that

$$\text{Adv}^{\text{KDMAE}}_{\text{NE}, \phi}(A_{\text{KDMAE}}) = 2 \cdot \text{Pr}[g_0] - 1.$$

First, we introduce the set $S_{\text{honest}}$ that contains all honest symmetric keys at any point in time. Whenever the adversary creates an honest group, the newly sampled key is added to $S_{\text{honest}}$ and whenever it exposes a group key, the exposed key is removed from $S_{\text{honest}}$. If the NewHonGroup oracle ever samples the same symmetric key for two different groups, we set the bad_0 flag. Next, we rewrite the random oracle $H$ such that if it is queried on a (currently) honest key $K$, it delegates the lazy sampling of the response to the simulated oracle $H_{\text{key}}^{\text{secret}}$. Otherwise, it responds by lazily sampling the output itself. Note that $H$ and $H_{\text{secret}}$ use different tables $T_0$ and $T_1$ for responding to queries. Whenever an honest key is exposed on top of removing it from the set $S_{\text{honest}}$, the game updates the table $T_0$ to be consistent with the table $T_1$ by setting $T_0[K[g]]$ to $H_{\text{secret}}(K[g])$. Note that a key $K$ is exposed iff $T_0[K] \neq \bot$. If $H$ is called on an exposed key $K$, it returns $T_0[K]$. If $A_{\text{KDMAE}}$ queries its NewCorrGroup oracle on inputs $(g, K)$ such that $K[g] = \bot$ and $K \in S_{\text{honest}}$, we call the key $K$ a "reused" key. We maintain a set $S_{\text{reused}}$ of reused keys and set the bad_1 flag whenever a reused key is queried to NewCorrGroup. We require that $H$ answer a query on a reused key on its own rather than delegating it to the simulated random oracle $H_{\text{key}}^{\text{secret}}$. We assume, without loss of generality, that the adversary makes all of its NewHonGroup queries before making any other queries. Lastly, we add conditional statements in the Enc oracle that result in three branches. All three branches execute the same line of code that assigns to z the output of $H(K[g])$. None of these changes affect the functionality of $g_{\text{KDMAE}}^{\text{NE}, \phi}$.  

51
Games $G_0 - G_4$

$b' \leftarrow \{0, 1\}; \ b \leftarrow A_{K^{\text{KDMAE}}}$

Return $b = b'$

$H(K)$

If $K \in S_{\text{honest}}$ and $K \notin S_{\text{reused}}$ then

$\text{bad}_1 \leftarrow \text{true} // A_{KR}$ breaks KR of NE.

Return $H^{\text{sym}}(g) // g_0(0, 2)$

$H^{\text{secret}}(\text{group}[K]) // g_2(3, 4)$

If $T_0[K] = \perp$ then $T_0[K] \leftarrow \mathcal{R}$

Return $T_0[K]$

$H^{\text{key}}(g)$

If $T_1[K] = \perp$ then $T_1[K] \leftarrow \mathcal{R}$

Return $T_1[K]$

$H^{\text{secret}}(g) // g_2(2, \infty)$

If $T_0[g] = \perp$ then $T_1[g] \leftarrow \mathcal{R}$

Return $T_1[g] // g_2(2, \infty)$

NewHONGroup($g$)

require $K[g] = \perp$

$K[g] \leftarrow \{0, 1\} \text{ NE.}$

$S_{\text{honest}} \subseteq S_{\text{honest}} \cup \{K[g]\}$

If $\text{group}[K[g]] \neq \perp$ then

// Birthday bound.

$\text{bad}_0 \leftarrow \text{true}$

$\text{stop(width)} // g_3(1, 3)$

Return $\text{K}[g]$

ExposeGroup($g$)

require $K[g] \neq \perp$ and $\text{chal}[g]$

$\text{group_is_corrupt}[g] \leftarrow \text{true}$

NewCORRGroup($g, K$)

require $K[g] = \perp$

$\text{group_is_corrupt}[g] \leftarrow \text{true}$

If $K \in S_{\text{honest}}$ and $K \notin S_{\text{reused}}$ then

$\text{bad}_1 \leftarrow \text{true} // A_{KR}$ breaks KR of NE.

$S_{\text{honest}} \subseteq S_{\text{honest}} \cup \{K[g]\}$

$T_0[K] \leftarrow H^{\text{sym}}(K[g]) // g_0(0, 2)$

$T_1[K] \leftarrow H^{\text{key}}(g) // g_2(2, \infty)$

$T_0[K] \leftarrow H^{\text{secret}}(K) // g_2(2, 4)$

$T_1[K] \leftarrow H^{\text{secret}}(\text{group}[K]) // g_2(2, 4)$

$K[g] \leftarrow K$

Adversary $A_{KR}^{\text{Enc, Dec, Guess}}$

$b \leftarrow \{0, 1\}$

$b' \leftarrow A_{K^{\text{KDMAE}}}$

SimH(K)

$\text{Guess}(K)$

If $T_0[K] = \perp$ then $T_0[K] \leftarrow \mathcal{R}$

Return $T_0[K]$

$H^{\text{group}}(g)$

If $T_1[K] = \perp$ then $T_1[K] \leftarrow \mathcal{R}$

Return $T_1[K]$

SimNEWHONGroup($g$)

require $K[g] = \perp$

$K[g] \leftarrow \text{"dummy"}$

NewHONGroup($g$)

SimEXPOSEGroup($g$)

require $K[g] \neq \perp$ and $\text{chal}[g]$

$K[g] \leftarrow \text{EXPOSEGroup}(g)$

SimNEWCORRGroup($g, K$)

require $K[g] = \perp$

$\text{group_is_corrupt}[g] \leftarrow \text{true}$

Guess($K$)

$T_0[K] \leftarrow H^{\text{secret}}(g)$

Return $T_0[K]$

SimDEC($g, n, c$)

require $K[g] \neq \perp$ and $\text{group_is_corrupt}[g] 

$m \leftarrow \text{DEC}(g, n, c)$

If $b = 0$ then return $\perp$ else return $m$

Fig. 33: Top pane: Games $G_0 - G_4$ for the proof of Proposition 1. The code highlighted in gray rewrites parts of game $G_{\text{NE.}}(A_{K^{\text{KDMAE}}})$ in an equivalent way. The code added for the transitions between games is highlighted in green. Bottom pane: Adversary $A_{KR}$ for the proof of Proposition 1. The highlighted instructions mark the changes in the code of the simulated game $G_4$. 

52
Transition from $g_0$ to $g_1$. In game $g_1$, the NewHonGroup oracle executes the \texttt{stop}(false) instruction whenever $\text{bad}_0$ is set. This instruction immediately halts the game and returns false as its final output. As the games $g_0$ and $g_1$ are identical-until-bad, we have
\[
\Pr[g_0] - \Pr[g_1] \leq \Pr[\text{bad}_0^B].
\]

Bounding $\Pr[\text{bad}_0^B]$. The probability that $\text{bad}_0$ is set in $g_0$ is bounded by the probability that the NewHonGroup oracle assigns the same key $K$ to two different groups. As keys are sampled uniformly at random from $\{0,1\}^{\text{NE} \cdot \text{id}}$, we can use the birthday bound to get
\[
\Pr[\text{bad}_0^B] \leq q^2_{\text{NewHonGroup}} / (2 \cdot 2^{\text{NE} \cdot \text{id}}).
\]

Transition from $g_1$ to $g_2$. We note that the change introduced in the previous game transition ensures that $\forall K \in \text{honest} \setminus \text{reused}$, there is a unique mapping between $K$ and the corresponding group identifier $g$. Moreover, for every query $H(K)$ for which $K \in S_{\text{honest}} \setminus \text{reused}$, the query is relayed to $H^\text{secret}$. Now in $g_2$ we use a new simulated oracle $H^\text{group}$ to respond to all $H$ queries satisfying $K \in S_{\text{honest}} \setminus \text{reused}$. In essence we switch from using $H^\text{secret}$ to using $H^\text{group}$. The unique mapping between group identifiers and keys enables us to make this switch without changing the functionality of $g_1$. Hence,
\[
\Pr[g_1] = \Pr[g_2].
\]

Transition from $g_2$ to $g_3$. In game $g_3$ the game no longer aborts when $\text{bad}_0$ is set. Note that this only increases the chances that the adversary wins the game. It follows that
\[
\Pr[g_2] \leq \Pr[g_3].
\]

Transition from $g_4$ to $g_1$. In game $g_4$, the oracles $H$ and NewCorrGroup do not execute any other instructions in the conditional branches that set the $\text{bad}_1$ flag. As games $g_3$ and $g_4$ are identical until $\text{bad}_1$ is set in $g_1$, we have
\[
\Pr[g_3] - \Pr[g_4] \leq \Pr[\text{bad}_1^B].
\]

Bounding $\Pr[\text{bad}_1^B]$ To bound $\Pr[\text{bad}_1^B]$, we build an adversary $A_{KR}$ in Fig. 33 that simulates $g_4$ for $A_{KDMAE}$. The adversary $A_{KR}$ uses its group oracle $G$ to respond to group oracle queries made by $A_{KDMAE}$. The set $\text{reused}$ is always empty in $g_4$. So $A_{KR}$ can ignore the conditional that checks if $K[g] \in \text{reused}$ while simulating the ENC oracle. To simulate the other two conditional branches, $A_{KR}$ uses the $\text{group}_\text{is}\_\text{corrupt}[g]$. Note that $K[g] \in S_{\text{honest}}$ iff $\neg \text{group}_\text{is}\_\text{corrupt}[g]$. It replaces calls to $\text{NE} . \text{Enc}$ with its ENC oracle and calls to $\text{NE} . \text{Dec}$ with its DEC oracle. Finally, $A_{KR}$ guesses every key queried to the random oracle $H$ or the NewCorrGroup oracle. The $\text{bad}_1$ flag is only set in $g_4$ if the adversary queried an honest key to either the random oracle $H$ or to the NewCorrGroup oracle. By guessing such a key, $A_{KR}$ wins its game. Thus,
\[
\Pr[\text{bad}_1^B] \leq \text{Adv}_{\text{NE}}^{A_{KR}}(A_{KR}).
\]

Bounding $\Pr[g_4]$. We build an adversary $A_{AEAD}$ in Fig. 34 that perfectly simulates game $g_4$ for $A_{KDMAE}$ and wins whenever $A_{KDMAE}$ does. It uses its group oracles $G$ to simulate responses to group oracle queries made by $A_{KDMAE}$ and its oracles ENC and DEC to simulate the corresponding oracles in $g_4$. Therefore,
\[
2 \cdot \Pr[g_4] - 1 \leq \text{Adv}_{\text{AEAD}}^{A_{AEAD}}(A_{AEAD}).
\]

C.4 Out-Group AE Security of BoxMessage (Proof of Theorem 4)

Overview. This proof uses games $g_0$ through $g_3$ in Fig. 35. We establish the following claims.

- $\text{Adv}_{\text{BM}, \text{pred}_{\text{inital}, \text{func}_{\text{out}}}^{\text{sp}}}^{\text{OAE}}(A_{\text{BM} \cdot \text{OAE} - \text{of} \cdot \text{BM}}) = 2 \cdot \Pr[g_0] - 1$
- $\text{Pr}[g_0] = \Pr[g_1]$
- $\Pr[g_1] - \Pr[g_2] \leq \Pr[\text{bad}_0^B]$
- $\Pr[\text{bad}_0^B] \leq \text{Adv}_{\text{CR}}^{A_{\text{CR}}}(A_{\text{CR}})$
- $\Pr[\text{bad}_0^B] \leq \text{Adv}_{\text{BM}, \text{pred}_{\text{inital}, \text{func}_{\text{out}}}^{\text{sp}}}^{\text{OAE}}(A_{\text{BM} \cdot \text{OAE} - \text{of} \cdot \text{BM}})$
- $\Pr[g_2] - \Pr[g_3] \leq \Pr[\text{bad}_0^B]$
- $\text{Pr}[\text{bad}_0^B] = 0$
- $2 \cdot \Pr[g_3] - 1 \leq \text{Adv}_{\text{SP}, \text{pred}_{\text{inital}, \text{func}_{\text{out}}}^{\text{sp}}}^{\text{OAE}}(A_{\text{BM} \cdot \text{OAE} - \text{of} \cdot \text{SP}})$
Fig. 34: Adversary $A_{AEAD}$ for the proof of Proposition 1. The highlighted instructions mark the changes in the code of the simulated game $G_4$. 

<table>
<thead>
<tr>
<th>Adversary $A_{AEAD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b' \leftarrow A_{SimG,SimEnc,SimDec,SimH}$</td>
</tr>
<tr>
<td>Return $b'$</td>
</tr>
<tr>
<td>( SimH(K) )</td>
</tr>
<tr>
<td>If $T_0[K] = \perp$ then $T_0[K] \leftarrow R$</td>
</tr>
<tr>
<td>Return $T_0[K]$</td>
</tr>
<tr>
<td>$H_{group}^g(g)$</td>
</tr>
<tr>
<td>If $T_1[g] = \perp$ then $T_1[g] \leftarrow R$</td>
</tr>
<tr>
<td>Return $T_1[g]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$SimEnc(g, n, \phi_0, \phi_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{s.t. } \phi_b(K[g]) = \gamma_b(H(K[g])) )</td>
</tr>
<tr>
<td>require $K[g] \neq \perp$ and $(g, n) \not\in N$</td>
</tr>
<tr>
<td>require $\phi_0, \phi_1 \in \Phi$ and $|\phi_0| = |\phi_1|$</td>
</tr>
<tr>
<td>If $\phi_0 \neq \phi_1$ then</td>
</tr>
<tr>
<td>If $\text{group_is_corrupt}[g]$ then return $\perp$</td>
</tr>
<tr>
<td>$\text{chal}[g] \leftarrow \text{true}$</td>
</tr>
<tr>
<td>If $\neg \text{group_is_corrupt}[g]$ then $z \leftarrow H_{group}^g(g)$</td>
</tr>
<tr>
<td>else $z \leftarrow SimH(K[g])$</td>
</tr>
<tr>
<td>$m_0 \leftarrow \gamma_0(z); m_1 \leftarrow \gamma_1(z); c \leftarrow ENC(g, n, m_0, m_1)$</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>$m \leftarrow \gamma_0(z); c \leftarrow ENC(g, n, m)$</td>
</tr>
<tr>
<td>$N \leftarrow N \cup {(g, n)}; C \leftarrow C \cup {(g, n, c)}; \text{Return } c$</td>
</tr>
<tr>
<td>$SimDec(g, n, c)$</td>
</tr>
<tr>
<td>require $K[g] \neq \perp$ and $\neg \text{group_is_corrupt}[g]$ and $(g, n, c) \not\in C$</td>
</tr>
<tr>
<td>$m \leftarrow \text{DEC}(g, n, c)$</td>
</tr>
<tr>
<td>If $b = 0$ then return $\perp$</td>
</tr>
<tr>
<td>else return $m$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$SimNewHonGroup(g)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>require $K[g] = \perp$</td>
</tr>
<tr>
<td>$K[g] \leftarrow \text{&quot;dummy&quot;}$</td>
</tr>
<tr>
<td>$NewHonGroup(g)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$SimExposeGroup(g)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>require $K[g] \neq \perp$ and $\neg \text{chal}[g]$</td>
</tr>
<tr>
<td>$\text{group_is_corrupt}[g] \leftarrow \text{true}$</td>
</tr>
<tr>
<td>$K[g] \leftarrow \text{EXPOSEGROUP}(g)$</td>
</tr>
<tr>
<td>$T_0[K[g]] \leftarrow H_{group}^g(g)$</td>
</tr>
<tr>
<td>Return $K[g]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$SimNewCorrGroup(g, K)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>require $K[g] = \perp$</td>
</tr>
<tr>
<td>$\text{group_is_corrupt}[g] \leftarrow \text{true}$</td>
</tr>
<tr>
<td>$K[g] \leftarrow \text{NewCorrGroup}(g, K)$</td>
</tr>
<tr>
<td>$T_0[K[g]] \leftarrow H_{group}^g(g)$</td>
</tr>
<tr>
<td>$K[g] \leftarrow K$</td>
</tr>
</tbody>
</table>
Fig. 35: **Top pane:** Games $G_0 - G_3$ for the proof of Theorem 4. The code highlighted in gray was added by expanding the symmetric signcryption algorithms $\text{BM.SigEnc}, \text{BM.VerDec},$ and the ciphertext-triviality predicate $\text{pred}_\text{trivial}$ in game $G_0 \otimes \text{BM.pred}_\text{trivial}^{\text{funcal}}(A^\text{BM.of-BM}).$ The code added for the transitions between games is highlighted in green. **Bottom pane:** Adversary $A^\text{wOAE-of-SP}$ for the proof of Theorem 4. The highlighted instructions mark the changes in the code of the simulated game $G_3.$
By combining the above, we have

\[ \text{Adv}^{\text{OAE}}_{\text{BM, pred\text{-}func}}(A_{\text{bwOAE\text{-}of\text{-}BM}}) = 2 \cdot \Pr[\hat{g}_0] - 1 = 2 \cdot \Pr[\hat{g}_1] - 1 \]

\[ = 2 \cdot \left( \sum_{i=0}^{1} (\Pr[\hat{g}_{i+1}] - \Pr[\hat{g}_{i+2}]) + \Pr[\hat{g}_3] \right) - 1 \]

\[ = 2 \cdot \left( \sum_{i=0}^{1} (\Pr[\text{bad}^{\hat{g}_{i+1}}]) + \Pr[\hat{g}_3] \right) - 1 \]

\[ \leq 2 \cdot \text{Adv}^{\text{CR}}_H(A_H) + \text{Adv}^{\text{OAE}}_{\text{SP, pred\text{-}func\text{-}func}}(A_{\text{bwOAE\text{-}of\text{-}SP}}). \]

We justify our claims in the following paragraphs.

**GAME \( g_0 \).** The game \( g_0 \) is functionally equivalent to \( \text{Adv}^{\text{OAE}}_{\text{BM, pred\text{-}func}}(A_{\text{bwOAE\text{-}of\text{-}BM}}) \). Additionally, the former includes code lines highlighted in green that we use for transitions between games. Note that the if-then-else statement that was added in oracle VERDEC returns \( m_{\text{body}} \) under all conditional branches. These changes do not affect the functionality of \( g_0 \) and we have

\[ \text{Adv}^{\text{OAE}}_{\text{BM, pred\text{-}func}}(A_{\text{bwOAE\text{-}of\text{-}BM}}) = 2 \cdot \Pr[\hat{g}_0] - 1. \]

**TRANSITION FROM \( g_0 \) TO \( g_1 \).** In the game \( g_0 \), if the adversary forwards an honestly generated ciphertext from SIGENC to VERDEC, then the condition \( z \in C \) holds and VERDEC will always return \( \bot \). In game \( g_1 \), we remove the conditional that checks if \( z \in C \) from \( g_1 \) and use the table \( W \) instead to achieve the same behavior without actually having to decrypt forward ciphertexts. For every SIGENC query, the table \( W \) indexed by the tuple \((g, u, n, c, ad)\) stores the challenge message \( m_{\text{body},1} \) where \( c \) is the ciphertext returned for that query. This table \( W \) is then used to respond to VERDEC queries on trivially forward ciphertexts. Note that the decryption correctness of NE and SP gives \( W[g, u, n, c, ad] \neq \bot \iff ((g, u, n, m_{\text{body}}, ad), c) \in C \) i.e. \( W[g, u, n, c, ad] \neq \bot \iff z \in C \) holds in \( g_0 \). However, if the condition \( W[g, u, n, c, ad] \neq \bot \) evaluates to true in \( g_0 \), the VERDEC oracle immediately returns \( \bot \). This means that the conditional checking whether \( z \in C \) is unreachable in \( g_0 \), and removing it in \( g_1 \) does not change the output of VERDEC. It follows that

\[ \Pr[\hat{g}_0] = \Pr[\hat{g}_1]. \]

**TRANSITIONS FROM GAME \( g_1 \) TO \( g_3 \).** Let \( i \in \{0, 1\} \). Games \( g_{i+1} \) and \( g_{i+2} \) are identical until \( \text{bad} \) is set. Each of these transitions removes an instruction returning \( m_{\text{body}} \) and returns \( \bot \) instead. Therefore

\[ \Pr[\hat{g}_{i+1}] - \Pr[\hat{g}_{i+2}] \leq \Pr[\text{bad}^{\hat{g}_{i+1}}]. \]

**BOUNDING \( \Pr[\text{bad}^{\hat{g}_0}] \).** We describe adversary \( A_{\text{CR}} \) against the collision resistance of \( H \) that simulates game \( g_1 \) for \( A_{\text{bwOAE\text{-}of\text{-}BM}} \). Adversary \( A_{\text{CR}} \) runs the code of \( g_1 \) by sampling and maintaining all user and group keys locally. Consider a VERDEC query that sets the \( \text{bad}_0 \) flag. During this query VERDEC computed \( h_{\text{body}} = H(n_{\text{body}} \parallel c_{\text{body}}) \). Now, the \( \text{bad}_0 \) flag is only set when \( \exists u' : ((g, u', n_{\text{header}}, m_{\text{header}}, \varepsilon), c_{\text{header}}) \in C_{\text{SP}} \) and \( (n_{\text{body}}, c_{\text{body}}) \neq T[g, n_{\text{header}}] \). The former implies that there was a prior call to SIGENC during which SP.SigEnc was invoked on inputs \((g, u', n_{\text{header}}, m_{\text{header}}, \varepsilon)\) and produced output \( c_{\text{header}} \). Let the body ciphertext for this query be \( c'_{\text{body}} \). We know that for the current VERDEC query \( z \notin C \). Therefore \( c_{\text{body}} \neq c'_{\text{body}} \). Note that the message \( m_{\text{header}} \) uniquely encodes \( h_{\text{body}} = H(n_{\text{body}} \parallel c_{\text{body}}) \). By the uniqueness of \((g, n_{\text{header}})\) we know that \( T[g, n_{\text{header}}] \) uniquely maps to the pair \((n_{\text{body}} \parallel c'_{\text{body}})\). Then we get \( h_{\text{body}} = H(n_{\text{body}} \parallel c'_{\text{body}}) = H(n_{\text{body}} \parallel c'_{\text{body}}) \) and \( (n_{\text{body}}, c'_{\text{body}}) \neq (n_{\text{body}}, c_{\text{body}}) \). Therefore, by returning \((n_{\text{body}} \parallel c_{\text{body}}, n_{\text{body}} \parallel c'_{\text{body}})\), adversary \( A_{\text{CR}} \) wins the collision-resistance game of \( H \) whenever \( \text{bad}_0 \) is set. This gives

\[ \Pr[\text{bad}^{\hat{g}_0}] \leq \text{Adv}^{\text{CR}}_{H}(A_{\text{CR}}). \]

**BOUNDING \( \Pr[\text{bad}^{\hat{g}_2}] \).** The conditions that set \( \text{bad}_1 \) in \( g_2 \) are \( \exists u' : ((g, u', n_{\text{header}}, m_{\text{header}}, \varepsilon), c_{\text{header}}) \in C_{\text{SP}} \) and \((n_{\text{body}}, c_{\text{body}}) = T[g, n_{\text{header}}] \). Since \( m_{\text{header}} \) encodes \( u \), we get \( u' = u \). Then we can infer that \((g, u, n_{\text{header}}, m_{\text{header}}, \varepsilon), c_{\text{header}}) \in C_{\text{SP}} \). This means that there was a prior call to SIGENC during which
SP.SigEnc was invoked on inputs \((g, u, n\text{header}, m\text{header}, \varepsilon)\) and produced output \(c\text{header}\). By the construction of \(m\), we know that the values \(ad\) and \(h\text{body}\) were the same across a prior SigEnc query and the current VerDec query. The condition \((n\text{body}, c\text{body}) = T[g, n\text{header}]\) implies that \((n\text{body}, c\text{body})\) were also repeated across the two queries. Combining the above, we get that the values \(g, u, n, c\), and \(ad\) were repeated, and hence \(W[g, u, n, c, ad] \neq \perp\). Therefore it is impossible to set \(bad_1\) in \(G_2\) and
\[
\Pr[bad_1] = 0
\]

**Bounding \(\Pr[g_1]\).** In Fig. 35 we show adversary \(A_{\text{wOAE-of-SP}}\) against the OAE security of SP that simulates the game \(G_2\) for adversary \(A_{\text{OAE}}\) until \(bad_2\) is set and wins whenever \(A_{\text{OAE}}\) does. It simulates the user oracles using the corresponding oracles from its OAE security game and simulates responses to the group oracles using its group oracles. While simulating the SigEnc oracle for \(A_{\text{OAE}}\), it replaces calls to SP.SigEnc with its SigEnc oracle and analogously, while simulating VerDec, it substitutes calls to SP.VerDec with its VerDec oracle. To simulate calls to NE.Enc, adversary \(A_{\text{wOAE-of-SP}}\) uses its Enc oracle. Finally, if the \(bad_2\) flag is never set and \(A_{\text{OAE}}\) returns \(b'\), adversary \(A_{\text{wOAE-of-SP}}\) also returns \(b'\). Now consider the case that \(bad_2\) is set in the simulated game. It must be the case that \(\exists u'\) such that \((g, u', n\text{header}, m\text{header}, \varepsilon), c\text{header} \in \mathcal{C}_{\text{SP}}\) and \(m\text{header} \neq \perp\). This is only possible when \(A_{\text{wOAE-of-SP}}\) has successfully forged a ciphertext and the bit \(b\) in its game is 1. Therefore \(A_{\text{wOAE-of-SP}}\) executes \(\text{abort}(1)\) and wins its game whenever \(bad_2\) is set, giving
\[
2 \cdot \Pr[g_1] - 1 \leq Adv_{\text{SP, pred}_{\text{trivial, except-user}}, func_{\text{out}}} (A_{\text{wOAE-of-SP}}).
\]

### C.5 Out-group AE Security of SealPacket (Proof of Theorem 5)

**OVERVIEW.** This proof uses games \(g_0\) through \(g_3\) in Fig. 36. We establish the following claims.

1. \(Adv_{\text{SP, pred}_{\text{trivial, except-user}}, func_{\text{out}}} (A_{\text{wOAE-of-SP}}) = 2 \cdot \Pr[g_0] - 1\)
2. \(\Pr[g_0] = \Pr[g_1]\)
3. \(\Pr[g_1] - \Pr[g_2] \leq \Pr[bad_0]\)

By combining the above, we have
\[
Adv_{\text{SP, pred}_{\text{trivial, except-user}}, func_{\text{out}}} (A_{\text{wOAE-of-SP}}) = 2 \cdot (\Pr[g_1] - \Pr[g_2]) - 1
\leq 2 \cdot (\Pr[bad_0] + \Pr[g_3]) - 1
\leq 2 \cdot Adv_{\text{DS, M}} (A_{\text{SPARSE}}) + Adv_{\text{ND, M}} (A_{\text{AEAD}}).
\]

We justify our claims in the following paragraphs.

**GAME \(g_0\).** We claim that the view of adversary \(A_{\text{wOAE-of-SP}}\) in game \(g_0\) is identical to its view in game \(g_{\text{SP, pred}_{\text{trivial, except-user}}, func_{\text{in}}} (A_{\text{wOAE-of-SP}})\). We use green highlighting to show code that was added for transitions between games. In particular, we add the set \(C_{\text{NE}}\) in SigEnc and the set \(L\) in Enc. The set \(C_{\text{NE}}\) stores the tuple \((g, n, c)\) for every SigEnc query where \(c\) is the output of NE.Enc for that query. The set \(L\) stores the tuple \((g, n\text{body}, c\text{body})\) for every valid ENC query where \(c\text{body}\) is the output of ENC for that query. The latter set is then used in VerDec to check whether the adversary forwarded a ciphertext returned by ENC. If that is the case then the \(bad_0\) flag is set. This does not affect the functionality of the game. Therefore,
\[
Adv_{\text{SP, pred}_{\text{trivial, except-user}}, func_{\text{out}}} (A_{\text{wOAE-of-SP}}) = 2 \cdot \Pr[g_0] - 1.
\]

**Transition from \(g_0\) to \(g_1\).** In game \(g_1\), we use the set \(C_{\text{NE}}\) in VerDec to check if \((g, n, c) \in C_{\text{NE}}\) for a query \((g, u, n, c, ad)\). If the condition is true, the oracle returns \(\perp\). Now, \((g, n, c) \in C_{\text{NE}} \iff \exists u' : ((g, u', n, m, ad), c) \in C\) where \(m\) is the unisignryption of \(c\). This implies that \((g, n, c) \in C_{\text{NE}} \iff pred_{\text{trivial, except-user}} (z, C) = \text{true}\). By definition of the output-guarding function \(func_{\text{out}}\), the game \(g_0\) returns \(\perp\) whenever \(pred_{\text{trivial, except-user}} (z, C) = \text{true}\). The game \(g_1\) returns \(\perp\) whenever \((g, n, c) \in C_{\text{NE}}\). Therefore the outputs of games \(g_0\) and \(g_1\) are consistent and we have
\[
\Pr[g_0] = \Pr[g_1].
\]
Games \(g_0, g_3\) for the proof of Theorem 5. The code highlighted in gray was added by expanding the symmetric signcryption algorithms \(\text{SP.SigEnc}, \text{SP.VerDec}\), the ciphertext-triviality predicate \(\text{pred}_\text{trivial}^{\text{pf-except-user}}\), and the output-guarding function \(\text{func}_{\text{out}}\) in game \(\text{g}_{\text{SP.SigEnc}}\). The code added for the transitions between games is highlighted in green. Bottom pane: Adversary \(A_{\text{KDMAE}}\) for the proof of Theorem 5. The highlighted instructions mark the changes in the code of the simulated game \(g_3\).
TRANSITION FROM $G_1$ TO $G_2$. In game $G_2$ we modify VERDEC to return $\bot$ whenever $bad_0$ is set. As the games $G_1$ and $G_2$ are identical-until-bad, $Pr[G_1] \leq Pr[G_2] + Pr[bad_0^0]$.

BOUNDING $Pr[bad_0^0]$. We bound the probability that $bad_0$ is set in $G_1$ by describing adversary $\mathcal{A}_{\text{SPARSE}}$ that perfectly simulates $G_1$ for $A_{\text{OAEP-SP}}$. Adversary $A_{\text{SPARSE}}$ plays in the $\text{SPARSE}$ security game against DS with respect to the message set $M$. It simply runs the code for the game $G_1$ by sampling and maintaining the user and group keys. Whenever the flag $bad_0$ is set on a VERDEC query, adversary $A_{\text{SPARSE}}$ returns the corresponding $(vk[u], m_s, s)$ and wins its game. Note that the $bad_0$ flag is set when the output $c$ of a prior ENC query is successfully unscrypted by VERDEC. The correctness of NE guarantees that $m_e = \text{NE.Dec}((g, n, c)$ is the same as the message previously encrypted by ENC. As ENC requires that $m_e \in M$, we have $\text{DS.Ver}(vk[u], m_s, s) = \text{true}$ and $s \parallel m_e \in M$. These are the same as the winning conditions in the $\text{SPARSE}$ game of DS. Thus

$$Pr[bad_0^0] \leq \text{Adv}_{DS, M}^{\text{SPARSE}}(A_{\text{SPARSE}}).$$

TRANSITION FROM $G_2$ TO $G_3$. In game $G_3$, we modify the VERDEC oracle. We move the instruction that checks if $(g, n, c) \in L$ to the top of the oracle. This does not change the output of the oracle and we have $Pr[G_2] = Pr[G_3]$.

BOUNDING $Pr[G_3]$. In Fig. 36, we define the adversary $A_{\text{KDMAE}}$ that simulates $G_3$ for $A_{\text{OAEP-SP}}$. The reduction simulates responses to group oracle queries by using its own group oracles and simulates the user oracles by sampling and maintaining keys locally. While simulating the StE oracle, it replaces calls to NE.Enc with its oracle ENC using the function SIGENC-DEC. The function SIGENC-DEC executes exactly the same lines of code that computed $m_e$ from these parameters in $G_3$. This means that

$$A_{\text{KDMAE}}^\text{SIGENC-DEC}(\text{NE}, H, DS, m_0, n, sk[u](K)) \leq \text{SIGENC-DEC}(\text{NE}, H, DS, m_1, n, sk[u](K)).$$

To simulate the encryption oracle ENC, the adversary $A_{\text{KDMAE}}$ uses its oracle ENC with the function ENC-DEC. The function ENC-DEC is parameterized by a message $m$ and returns $m$. Note that the output of $\text{ENC}(g, n_0, m_0, m_0, 0)(K), \text{ENC-DEC}(m_0, 1)(K))$ is the same as $\text{NE.Enc}(K, n_0, m_0, 0, m_0, 1)(K))$. While simulating the VERDEC oracle, the reduction replaces calls to NE.Dec with its DEC oracle. Adversary $A_{\text{AEAD}}$ wins whenever $A_{\text{OAEP-SP}}$ does. Therefore,

$$Pr[G_3] \leq \text{Adv}_{GE, M}^{\text{KDMAE}}(A_{\text{AEAD}}).$$

D  Proofs for the UV Stronger Out-Group AE Security Results

D.1 Proof for the UV Security of StE (Proof of Proposition 2)

Proof. This proof uses the game $G$ in Fig. 37. We establish that $\text{Adv}_{\text{StE}}^{\text{DS}}(A_{\text{StE}}^{\text{UV}}) \leq \text{Adv}_{\text{DS}}^{\text{DS}}(A_{\text{StE}}^{\text{UV}})$. Game $G_0$ is equivalent to the game $G_0^{\text{StE}}$. The former expands multiple instructions in the latter; the expanded code is marked in gray. Therefore, $\text{Adv}_{\text{StE}}^{\text{UV}}(A_{\text{StE}}^{\text{UV}}) = Pr[G_0]$. In Fig. 37 we show adversary $A_{\text{StE}}^{\text{UV}}$ against the UV security of DS that simulates the game $G_0$ for $A_{\text{StE}}^{\text{UV}}$ and wins whenever $A_{\text{StE}}^{\text{RS}}$ does. Once $A_{\text{StE}}^{\text{RS}}$ returns $(g, K, K', u, sk_{u'}, vk_{u'}, n, m, ad)$, the reduction constructs $m_s = (g, n, m, ad)$, and returns $(sk_{u'}, vk_{u'}, m_s)$. Note that whenever $A_{\text{StE}}^{\text{RS}}$ wins, it must be the case that $\text{DS.Ver}(g, K, K', u, sk_{u'}, vk_{u'}, n, c) \neq \bot$ and $(sk_{u'}, vk_{u'}) \notin [\text{SS.UserKg}]$. By the correctness of NE and the construction of StE, this implies that $\text{StE.Ver}(vk_{u'}, m_s, s)$ and $(sk_{u'}, vk_{u'}) \notin [\text{SS.UserKg}]$ where $m_s = (g, n, m, ad)$. These two are exactly the winning conditions of the UV security game of DS. Therefore, $A_{\text{StE}}^{\text{UV}}$ wins whenever $A_{\text{StE}}^{\text{RS}}$ does and it follows that $\text{Adv}_{\text{StE}}^{\text{UV}}(A_{\text{StE}}^{\text{UV}}) \leq Pr[G_0]$. For simplicity, we only give the proof of Proposition 2 for $\text{SS} = \text{StE}$. However, the adversary $A_{\text{StE}}^{\text{UV}}$ of Fig. 37 remains the same for all $\text{SS} \in \{\text{StE, EtS, SP}\}$ except for the line in the box that constructs the value $m_s$. For other choices of SS, this line would be replaced as per the construction of the scheme.
D.2 Proof for the UV Security of Ed25519 (Proof of Proposition 3)

Proof. **Overview.** This proof uses games $g_0$ through $g_2$ in Fig. 38. We establish the following claims.

<table>
<thead>
<tr>
<th>$g_0$</th>
<th>$g_1$</th>
<th>$g_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(sk, vk', m) \xrightarrow{} A_{UV}^{H}$</td>
<td>$s \xrightarrow{} Ed2519.Sig(sk, m)$</td>
<td>$R \parallel s \parallel A' = vk'$</td>
</tr>
<tr>
<td>$H(x)$</td>
<td>If $H[x] = \bot$ then</td>
<td>$X \leftarrow Z_p$</td>
</tr>
<tr>
<td></td>
<td>$X \leftarrow X$</td>
<td>$H[x] = X$</td>
</tr>
<tr>
<td></td>
<td>$(R, A, m) \leftarrow x; ch \leftarrow H[x]$</td>
<td>$(R, A, m) \leftarrow x; ch \leftarrow H[x]$</td>
</tr>
<tr>
<td></td>
<td>If $ch = 0$ then</td>
<td>If $ch = 0$ then</td>
</tr>
<tr>
<td></td>
<td>$bad_0 \leftarrow true$</td>
<td>$bad_0 \leftarrow true$</td>
</tr>
<tr>
<td></td>
<td>$ch \leftarrow Z_p \setminus {0}$</td>
<td>$ch \leftarrow Z_p \setminus {0}$</td>
</tr>
<tr>
<td></td>
<td>// $g_{1,\infty}$</td>
<td>// $g_{1,\infty}$</td>
</tr>
<tr>
<td></td>
<td>$H[x] = ch$</td>
<td>$H[x] = ch$</td>
</tr>
<tr>
<td></td>
<td>$// g_{1,\infty}$</td>
<td>$// g_{1,\infty}$</td>
</tr>
<tr>
<td></td>
<td>$T(R, A, m) = ch \cdot A$</td>
<td>$T(R, A, m) = ch \cdot A$</td>
</tr>
<tr>
<td></td>
<td>Return $H[x]$</td>
<td>Return $H[x]$</td>
</tr>
</tbody>
</table>

Fig. 38: Games $g_0$–$g_2$ used in the proof of Proposition 3.

By combining the above, we have

$$Adv_{Ed25519}^{UV}(A_{UV}) = Pr[g_0] = Pr[g_0] = (Pr[g_0] - Pr[g_1]) + Pr[g_1]$$
$$\leq Pr[bad_0^0] + Pr[g_1] \leq Pr[bad_0^0] + (Pr[g_1] - Pr[g_2]) + Pr[g_2]$$
$$\leq Pr[bad_0^0] + Pr[bad_1^2] \leq (2 \cdot q_h)/p + q_h^2/p.$$
We justify our claims in the following paragraphs.

GAME $\mathcal{G}_0$. The game $\mathcal{G}_0$ is functionally equivalent to $g_{UV}^{Ed25519}$. We use gray highlighting to show code from $Ed25519$. Additionally, $\mathcal{G}_0$ includes code lines highlighted in green that we use for transitions between games. In particular, the random oracle $H$ sets the $\text{bad}_0$ flag if the challenge $\text{ch}$ sampled for a query $x = (R, A, m)$ satisfies $\text{ch} \cdot A = 0$. The random oracle $H$ also maintains a dictionary $T$ that stores the value $\text{ch} \cdot A$ corresponding to every query $(R, A, m)$ where $\text{ch}$ is the output for that query. It sets the $\text{bad}_1$ flag if the booleans $\exists A' : T[(R, A', m)] = \text{ch} \cdot A$ and $A \neq A'$ evaluate to true for a query $(R, A, m)$. These changes do not affect the functionality of $\mathcal{G}_0$. It follows that

$$\Pr[\mathcal{G}_0] = \text{Adv}_{Ed25519}^{UV}(A_{UV}).$$

TRANSITIONS FROM $\mathcal{G}_0$ TO $\mathcal{G}_2$. Let $i \in \{0, 1\}$. Games $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$ are identical until $\text{bad}_i$ is set, therefore

$$\Pr[\mathcal{G}_i] - \Pr[\mathcal{G}_{i+1}] \leq \Pr[\text{bad}_i^0].$$

BOUNDING $\Pr[\text{bad}_0^0]$. We now compute $\Pr[\text{bad}_0^0]$. Consider the condition $\text{ch} \cdot A = 0$ that sets the $\text{bad}_0$ flag for an input $(R, A, m)$ in game $\mathcal{G}_0$. Recall that $\text{ch}$ is sampled uniformly at random from $\mathbb{Z}_p$. Now for a given $A = a \cdot B$ there are exactly two values for $\text{ch} = 0_B$ and $a^{-1} \mod p$ and that satisfy $\text{ch} \cdot A = 0$. Therefore the probability that a particular random oracle query sets the $\text{bad}_0$ flag in $\mathcal{G}_0$ is $2/p$. Applying a union bound over all $H$ queries we have

$$\Pr[\text{bad}_0^0] = q_H \cdot 2/p.$$

BOUNDING $\Pr[\text{bad}_1^0]$. Consider the condition that sets $\text{bad}_1$ in $\mathcal{G}_1$. We have $T(R, A', m) = \text{ch} \cdot A$ and $A' \neq A$. Let $\text{ch}'$ be the response returned for the query $(R, A', m)$. Then we have $\text{ch}' = A'$. At the time $\text{bad}_1$ is set, $\text{ch}'$, $A'$, and $A$ are fixed. The probability that for a particular random oracle query $\text{ch} = (\text{ch}' \cdot A') \cdot A^{-1}$ is bounded by $q_H/p$. Applying a union bound over all random oracle queries we have

$$\Pr[\text{bad}_1^0] = q_H^2/(p - 1).$$

GAME $\mathcal{G}_2$. Consider the winning conditions in $\mathcal{G}_2$. The adversary $A_{UV}$ wins the game $\mathcal{G}_2$ if it returns $(sk, vk', m)$ such that $(z \cdot B = R + \text{ch}' \cdot vk')$ and $(z, vk) \notin [DS, Kg]$ where $(R \| z) = Ed25519.Sig(sk, m)$ and $\text{ch}'$. Let $vk$ be the verification key that satisfies $(sk, vk) \in [DS, Kg]$. Then we have from the correctness of $DS$ that $z \cdot B = R + \text{ch}' \cdot vk$. We also know that $vk \neq vk'$. So we have $z \cdot B = R + \text{ch}' \cdot vk'$ and $z \cdot B = R + \text{ch}' \cdot vk$. These two conditions can simultaneously hold only if one of $\text{ch} \cdot A = 0$ or $\text{ch}' \cdot A' = \text{ch} \cdot A$ is true. In our transitions from $\mathcal{G}_0$ to $\mathcal{G}_2$, we precluded either of these conditions from being true. Therefore, it is impossible for $A_{UV}$ to win $\mathcal{G}_2$. It follows that

$$\Pr[\mathcal{G}_0] = 0.$$

D.3 Stronger Out-Group AE Security Results (Proof of Proposition 4)

OVERVIEW. This proof uses games $\mathcal{G}_0$ through $\mathcal{G}_2$ in Fig. 39. We establish the following claims.

- $\text{Adv}_{SS, pred_{minal}, func_{out}}^{\text{OA}_E}(A_{OA}_E) = 2 \cdot \Pr[\mathcal{G}_0] - 1$
- $\Pr[\mathcal{G}_1] = \Pr[\mathcal{G}_2]$
- $2 \cdot \Pr[\mathcal{G}_2] - 1 \leq \text{Adv}_{SS, pred_{minal}, func_{out}}^{\text{OA}_E}(A_{OA}_E)$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^0]$
- $\Pr[\text{bad}_0^0] \leq \text{Adv}_{SS}^{UV}(A_{UV})$
- $2 \cdot \Pr[\mathcal{G}_2] - 1 \leq \text{Adv}_{SS, pred_{minal}, func_{out}}^{\text{OA}_E}(A_{OA}_E)$

By combining the above, we have

$$\text{Adv}_{SS, pred_{minal}, func_{out}}^{\text{OA}_E}(A_{OA}_E) = 2 \cdot \Pr[\mathcal{G}_0] - 1 = 2 \cdot (\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] + \Pr[\mathcal{G}_1]) - 1$$

$$\leq 2 \cdot (\Pr[\text{bad}_0^0] + \Pr[\mathcal{G}_1]) - 1$$

$$\leq \text{Adv}_{SS}^{UV}(A_{UV}) + \text{Adv}_{SS, pred_{minal}, func_{out}}^{\text{OA}_E}(A_{OA}_E).$$

We justify our claims in the following paragraphs.
These changes do not affect the functionality of the corresponding oracles in the OAE game for Ste as per Fig. 7.

Fig. 39: Games $g_0,g_2$ used in the proof of Proposition 4 for $\text{pred}_{\text{trivial}} = \text{pred}_{\text{trivial}}^{\text{same}}$ and $\text{pred}_{\text{trivial}} = \text{pred}_{\text{same}}^{\text{except-user}}$. The $\text{VerDec}$ oracle corresponding to $\text{pred}_{\text{trivial}}^{\text{same}}$ is shown on the right and the $\text{VerDec}$ oracle corresponding to $\text{pred}_{\text{trivial}}^{\text{same}}$ is shown on the right.

The game $g_0$ is functionally equivalent to $\rho^{\text{OAE}}_{\text{SS,}$pred$_{\text{trivial}}$,func$_{\text{out}}$. Additionally, the former includes code lines highlighted in green that we use for transitions between games. The $\text{VerDec}$ oracle corresponding to $\text{pred}_0^{\text{same}}$ is shown at the bottom-left of Fig. 39 and that corresponding to $\text{pred}_0^{\text{same}}$ is shown at the bottom-right of Fig. 39. The Derive,$vk$ function defined in ?? derives the verification key $vk$ corresponding to the input signing key $sk$. The conditional statement that was added in $\text{VerDec}$ oracle sets the bad flag whenever the boolean $(vk[u],sk[u]) \notin \text{SS.UserKg}$ evaluates to true for $\text{pred}_{\text{trivial}}^{\text{same}}$ and whenever the boolean $(vk[u],sk[u]) \notin \text{SS.UserKg}$ evaluates to true for $\text{pred}_{\text{trivial}}^{\text{same}}$. These changes do not affect the functionality of $g_0$ and we have $Adv'^{\text{OAE}}_{\text{SS,}$pred$_{\text{trivial}}$,func$_{\text{out}}}(\mathcal{A}_{\text{OAE}}) = 2 \cdot Pr[\text{bad}] - 1$. 

<table>
<thead>
<tr>
<th>Game $g_0$</th>
<th>$g_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b \leftarrow {0,1}$; $b' \leftarrow \mathcal{A}_{\text{OAE}}(g,\text{StDec},\text{VerDec})$ Return $b = b'$</td>
<td></td>
</tr>
<tr>
<td>$\text{NEWHONUser}(u)$ $\text{NEWGROUP}(g,users)$ $\text{EXPOSEUser}(u)$ $\text{EXPOSEGroup}(g)$ $\text{NEWCORRUser}(u,sk,vk)$ $\text{NEWCORRGroup}(g,\mathcal{K},users)$</td>
<td></td>
</tr>
<tr>
<td>$\forall$ oracles in the game for $\text{EtS}$ these oracles are identical to the corresponding $\text{SS.UserKg}$ $\text{SS.UserKg}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$</td>
<td></td>
</tr>
<tr>
<td>$\text{NEWHONUser}(u)$ $\text{NEWGROUP}(g,users)$ $\text{EXPOSEUser}(u)$ $\text{EXPOSEGroup}(g)$ $\text{NEWCORRUser}(u,sk,vk)$ $\text{NEWCORRGroup}(g,\mathcal{K},users)$</td>
<td></td>
</tr>
<tr>
<td>$\forall$ oracles in the game for $\text{EtS}$ these oracles are identical to the corresponding $\text{SS.UserKg}$ $\text{SS.UserKg}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$</td>
<td></td>
</tr>
<tr>
<td>$\text{NEWHONUser}(u)$ $\text{NEWGROUP}(g,users)$ $\text{EXPOSEUser}(u)$ $\text{EXPOSEGroup}(g)$ $\text{NEWCORRUser}(u,sk,vk)$ $\text{NEWCORRGroup}(g,\mathcal{K},users)$</td>
<td></td>
</tr>
<tr>
<td>$\forall$ oracles in the game for $\text{EtS}$ these oracles are identical to the corresponding $\text{SS.UserKg}$ $\text{SS.UserKg}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$</td>
<td></td>
</tr>
<tr>
<td>$\text{NEWHONUser}(u)$ $\text{NEWGROUP}(g,users)$ $\text{EXPOSEUser}(u)$ $\text{EXPOSEGroup}(g)$ $\text{NEWCORRUser}(u,sk,vk)$ $\text{NEWCORRGroup}(g,\mathcal{K},users)$</td>
<td></td>
</tr>
<tr>
<td>$\forall$ oracles in the game for $\text{EtS}$ these oracles are identical to the corresponding $\text{SS.UserKg}$ $\text{SS.UserKg}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$ $\text{SS.Group}$</td>
<td></td>
</tr>
</tbody>
</table>
In game $g_1$ we modify the VERDEC oracle such that it returns $\perp$ whenever bad$^0$ is set. As the games $g_0$ and $g_1$ are identical-until-bad, we have $$\Pr[g_0] \leq \Pr[g_1] + \Pr[\text{bad}^0_0].$$

Bounding $\Pr[\text{bad}^0_0]$. In order to bound $\Pr[\text{bad}^0_0]$, consider an adversary $A_{UV}$ against the UV security of SS that is defined as follows. It perfectly simulates the game $g_0$ for adversary $A_{OAE}$ until the bad$^0_0$ flag is set. To simulate the user oracles $U$ and the group oracles $G$, adversary $A_{UV}$ itself samples and maintains all user keys and group keys.

Let us first define $A_{UV}$ for the case $\text{pred}_{\text{trivial}} = \text{pred}_{\text{trivial}}^\text{int}$. Whenever the bad$^0_0$ flag is set in the simulated game during a VERDEC query, adversary $A_{UV}$ returns the corresponding $(g, K[g], u, u, sk[u], vk[u], n, m, ad)$. Consider a VERDEC query $(g, u, n, c, ad)$ that sets the bad$^0_0$ flag in $g_0$. During this query, the verification key $vk[u]$ successfully verifies and decrypts the ciphertext $c$, i.e., $SS.\text{VerDec}(g, K[g], u, vk[u], n, c, ad) \neq \perp$. Moreover, we have $z = (g, u, n, m, ad) \in C$. Therefore, $c = SS.\text{SigEnc}(g, K[g], u, sk[u], n, c, ad)$. Note that immediately before the bad$^0_0$ flag is set, game $g_0$ verifies that $(vk[u], sk[u]) \notin [SS.\text{UserKg}]$. This implies that $(sk[u], vk[u]) \notin [DS.Kg]$. So the bad$^0_0$ flag is set when the following conditions are true –

$$SS.\text{VerDec}(g, K[g], u, vk[u], n, c, ad) \neq \perp$$

where $c = SS.\text{SigEnc}(g, K[g], u', sk[u'], n, c, ad)$ and $(sk[u'], vk[u]) \notin [DS.Kg]$.

We now define $A_{UV}$ for the case $\text{pred}_{\text{trivial}} = \text{pred}_{\text{trivial}}^\text{trivial}$. Whenever the bad$^0_0$ flag is set in the simulated game during a VERDEC query, adversary $A_{UV}$ returns the corresponding $(g, K[g], u', usk[u'], vk[u], n, m, ad)$. Consider a VERDEC query $(g, u, n, c, ad)$ that sets the bad$^0_0$ flag in $g_0$. During this query, the verification key $vk[u]$ successfully verifies and decrypts the ciphertext $c$, i.e., $SS.\text{VerDec}(g, K[g], u, vk[u], n, c, ad) \neq \perp$. Moreover, we have $\exists u' : ((g, u', n, c, ad), c) \in C$. Therefore, $c = SS.\text{SigEnc}(g, K[g], u', sk[u'], n, c, ad)$. Note that immediately before the bad$^0_0$ flag is set, game $g_0$ verifies that $(vk[u], sk[u']) \notin [SS.\text{UserKg}]$. This implies that $(sk[u'], vk[u]) \notin [DS.Kg]$. So the bad$^0_0$ flag is set when the following conditions are true –

$$SS.\text{VerDec}(g, K[g], u, vk[u], n, c, ad) \neq \perp$$

where $c = SS.\text{SigEnc}(g, K[g], u', sk[u'], n, c, ad)$ and $(sk[u'], vk[u]) \notin [DS.Kg]$. In both cases, the conditions under which bad$^0_0$ is set in $g_0$ are exactly the winning conditions in the UV game of SS. Therefore, by returning the corresponding values, $A_{UV}$ wins its game and we have $$\Pr[\text{bad}^0_0] \leq \text{Adv}_{SS}(A_{UV}).$$

Transition from $g_1$ to $g_2$. First, we address this transition for the case $\text{pred}_{\text{trivial}} = \text{pred}_{\text{trivial}}^\text{trivial}$. In the game $g_1$, if the adversary forwards an honestly generated ciphertext from SigEnc to VERDEC, then $z \in C$ evaluates to true in VERDEC and VERDEC returns the corresponding challenge message $m_1$ if $(vk[u], sk[u]) \in [SS.\text{UserKg}]$ and otherwise. In game $g_2$ we remove the conditional that checks if $\text{pred}_{\text{trivial}}(z, C)$ is true from VERDEC and instead use the table $W$ to achieve the same behavior without actually having to decrypt forwarded ciphertexts. For every SigEnc query, the table $W$ indexed by $g, u, n, c$, and $ad$ stores the challenge message $m_1$ where $c$ is the ciphertext returned for that query. This table $W$ is then used to respond to VERDEC queries on trivially forwarded ciphertexts. Note that the decryption correctness of SS gives $W[g, u, n, c, ad] \neq \perp$ evaluates to true in $g_2$ iff $\text{pred}_{\text{trivial}}(z, C)$ for $z = ((g, u, n, m, ad), c)$. Therefore the outputs of VERDEC are consistent between the two games.

Now we analyze the case $\text{pred}_{\text{trivial}} = \text{pred}_{\text{trivial}}^\text{except-user}$ analogously. In the game $g_1$, if the adversary forwards an honestly generated ciphertext from SigEnc to VERDEC, then $\exists u' : ((g, u', n, m, ad), c) \in C$ evaluates to true in VERDEC and VERDEC returns the corresponding challenge message $m_1$ if $(vk[u], sk[u']) \in [SS.\text{UserKg}]$ and otherwise. Again we use the table $W$ to replace the conditional that checks the triviality predicate. The decryption correctness of SS gives $\exists u' : W[g, u', n, c, ad] \neq \perp$ evaluates to true in $g_2$ iff $\exists u' : ((g, u', n, m, ad), c) \in C$. So the outputs of VERDEC are consistent between the two games.

In both cases, the outputs of VERDEC remain unchanged between $g_1$ and $g_2$. It follows that $$\Pr[g_1] = \Pr[g_2].$$

Bounding $\Pr[g_2]$. In Fig. 40, we define the adversary $A_{OAE.L}$ against the OAE security of SS with respect to the predicate $\text{pred}_{\text{trivial}}$ and the output guarding function $F^{\perp}$.

It perfectly simulates the game $g_2$ for adversary $A_{OAE}$. In particular, it simulates the responses to the user and group oracles using the
Fig. 40: Adversary $A_{\text{AWE}}$ used in the proof of Proposition 4. The SIMVERDEC oracle for pred$_{\text{trivial}}$ is shown on the left and the SIMVERDEC oracle for pred$_{\text{trivial-except-user}}$ is shown on the right.
corresponding oracles from its own OAE security game. Note that the OAE security game allows to expose all user keys, so every time an honest user key-pair is created in the simulated game \(g_2\), adversary \(A_{OAE\perp}\) immediately exposes it. It simulates the SigEnc oracle using the corresponding oracle from its security game and simulates the VerDec oracle using the corresponding oracle from its security game and the table \(W\). The simulated VerDec oracle corresponding to \(\text{pred}_\text{trivial} = \text{pred}_\text{suf-except-user}\) and \(\text{pred}_\text{trivial} = \text{pred}_\text{suf}\) are shown at the bottom-left and bottom-right respectively of Fig. 40. Finally, if the \(m \neq \bot\) condition is never true in the simulated VerDec and \(A_{OAE}\) returns \(b'\), adversary \(A_{OAE\perp}\) also returns \(b'\). Now consider the case that \(m \neq \bot\) evaluates to true in the simulated game. It must be the case that \(\text{pred}_\text{trivial}(z, C) = \text{false}\) and \(m \neq \bot\). This is only possible when \(A_{OAE}\) has successfully forged a ciphertext and the bit \(b\) is 1. Therefore \(A_{OAE\perp}\) executes \(\text{abort}(1)\) and wins its game. We have

\[
2 \cdot \Pr[g_i] - 1 \leq \text{Adv}_{\text{SS, pred}_\text{trivial}, \text{func}_\text{suf}}^{\text{OAE\perp}}.
\]

### E Other Uses of Symmetric Signcryption in Keybase

In this appendix, we briefly touch upon other uses of the SealPacket and BoxMessage algorithms in Keybase. We present three such instances – SealPacket as a standalone scheme, the algorithm SealWhole that is used to encrypt long strings, and the algorithm PostFileAttachment that is used to encrypt and upload file attachments to a third-party storage.

#### E.1 Security of SealPacket as a Standalone Encryption Scheme

In Keybase, the SealPacket subroutine is not only used within the BoxMessage construction to signencrypt the header but also as a standalone scheme. For example, it is used in isolation to signencrypt small chat attachments. Here we prove the wOAE security of SealPacket as a standalone scheme with respect to the predicate \(\text{pred}_{\text{suf-except-user}}\) and the output guarding function \(\text{func}_{\text{suf}}\).

Recall that we analyzed the security of SealPacket when used within BoxMessage in Theorem 5. In particular, we had reduced the wOAE[Enc[\(M, NE\)] security of SealPacket with respect to the predicate \(\text{pred}_{\text{suf-except-user}}\) and the output guarding function \(\text{func}_{\text{suf}}\) to the \(M\)-sparsity of the underlying digital signature scheme DS and the KDMAE security of the underlying nonce based encryption scheme NE.

One could capture the wOAE security of the standalone SealPacket scheme with respect to the predicate \(\text{pred}_{\text{suf-except-user}}\) and the function \(\text{func}_{\text{suf}}\) as a special case of this result by restricting the adversary from making any calls to its Enc oracle. By doing so, the reduction to the SPARSE security of the digital signature scheme is no longer possible. Moreover, the class of message-deriving functions considered during the reduction to the KDMAE security of the nonce-based encryption scheme would be smaller. This is because in the proof of Theorem 5 the function ENC-DER was only required to simulate the Enc oracle, which is never used when treating SealPacket as a standalone scheme, allowing its omission from the class of message-deriving functions considered.

In the following, we show that the wOAE security of the standalone SealPacket scheme with respect to the predicate \(\text{pred}_{\text{suf-except-user}}\) and the output guarding function \(\text{func}_{\text{suf}}\) implies its wOAE security with respect to the stronger output-guarding function \(\text{func}_{\text{sec}}\).

**Proposition 8.** Let \(DS = \text{Ed25519-DS}[k, p, G, G_p, B, H_1, H_2, H_3]\) be the digital signature scheme built from some \(k, p, G, G_p, B, H_1, H_2, H_3\) as specified in Construction 6. Let \(SP = \text{SEAL-PACKET-SS}[H, DS, NE]\) be the symmetric signcryption scheme built from some \(H, NE,\) and \(DS\). Let \(\text{pred}_{\text{suf-except-user}}\) be the ciphertext-triviality predicate defined in Fig. 8. Let \(\text{func}_{\text{suf}}\) and \(\text{func}_{\text{sec}} := \text{func}_{\text{suf}}\) be the output-guarding functions as defined in Fig. 9. Let \(wOAE\) be the security notion as defined in Definition 1. Let \(A_{wOAE\perp, SP}\) be any adversary against the wOAE security of \(SP\) with respect to \(\text{pred}_{\text{suf-except-user}}\) and \(\text{func}_{\text{suf}}\). Then we can build adversaries \(A_{\text{adv}}\) and \(B_{wOAE\perp, SP}\) such that

\[
\text{Adv}_{wOAE, \text{pred}_{\text{suf-except-user}, \text{func}_{\text{suf}}}^{\text{wOAE\perp, SP}}} = \text{Adv}_{DS, \text{adv}}(\mathcal{A}_{\text{adv}}) + \text{Adv}_{wOAE, \text{pred}_{\text{suf-except-user}, \text{func}_{\text{suf}}}^{\text{wOAE\perp, SP}}}.
\]

**Proof.** This proof is analogous to the proof of Proposition 4.
E.2 Multi-purpose Scheme SealWhole for Encrypting Long Strings

**Basic Notation.** For any \( x \in \{0, 1\}^* \) and \( \ell \in \mathbb{N} \) such that \( |x| \leq \ell \), we write \( \langle x \rangle_\ell \) to denote the bit-string of length \( \ell \) that is built by padding \( x \) with leading zeros.

**Symmetric Signcryption Syntax.** In this section, we augment the definition of SS from Section 3 with a ciphertext-length function \( SS.c.l: \mathbb{N} \to \mathbb{N} \) that is associated with every instance of SS. We then require that the signcryption algorithm \( SS.SigEnc \) encrypts any message \( m \in SS.MS \) into a ciphertext \( c \) of the corresponding (fixed) length \( |c| = SS.c.l(|m|) \).

The SealWhole algorithm [Key] is a wrapper around the standalone SealPacket algorithm and is used in Keybase to encrypt large strings and attachments. We present the pseudocode for the SealWhole algorithm in Fig. 41 and give a high-level description of the algorithm. The SealWhole algorithm accepts the same inputs as the SealPacket algorithm. To encrypt a message \( m \) with a nonce \( n \), the SealWhole.SigEnc algorithm first parses the message \( m \) into chunks \( m_i \) of length 1 MB. It then encrypts each \( m_i \) by calling the SealPacket algorithm with nonce \( n_i = n \parallel (i \mod 2^{64})_{64} \) to obtain ciphertext \( c_i \). The SealWhole algorithm returns the concatenation of all \( c_i \) obtained above.

```
SealWhole.SigEnc(g, K_g, u, sk_u, n, m, ad) // where n \in \{0, 1\}^{128}, ad = \varepsilon
m_0 \parallel m_1 \parallel ... \parallel m_l \leftarrow m // \text{[}|m_i| = 1 \text{MB for all } i < l \text{ and } 0 \leq |m_i| < 1 \text{MB.]
For i = 0 to l do
    n_i \leftarrow n \parallel (i \mod 2^{64})_{64}
    c_i \leftarrow SealPacket.SigEnc(g, K_g, u, sk_u, n_i, m_i, ad)
Return c_0 \parallel c_1 \parallel ... \parallel c_l

SealPacket.SigEnc(g, K_g, u, sk_u, n, m, ad) // n \in \{0, 1\}^{192}, ad = \varepsilon
h_{chunk} \leftarrow SHA-512(m_{chunk})
\sigma_{chunk} \leftarrow Ed25519.Sig(sk_u, "Keybase-Chat-2" || \langle K_g, n_{chunk}, h_{chunk} \rangle)
c_{chunk} \leftarrow XSalsa20-Poly1305.Enc(K_g, n_{chunk}, \sigma_{chunk} \parallel m_{chunk})
Return c_{chunk}
```

Fig. 41: The SealWhole algorithm used in Keybase for encrypting long strings. We reproduce the SealPacket algorithm from Section 5 for convenience.

E.3 Encryption of Attachments with PostFileAttachment

The PostFileAttachment [Key] algorithm is used to encrypt and upload files in private Keybase folders. We present the pseudocode for the PostFileAttachment algorithm in Fig. 42 and give a high-level description of the algorithm. The PostFileAttachment algorithm accepts the same inputs as the BoxMessage algorithm, except the message \( m_{\text{attach}} \) is a file that a sender wants to upload to a third-party storage, and the associated data \( ad \) is a pair \( (ad_{\text{attach}}, ad_{\text{wrap}}) \). To encrypt the file attachment \( m_{\text{attach}} \), the PostFileAttachment algorithm first samples a 256-bit attachment encryption key \( k_{\text{attach}} \) and generates an ephemeral signing key pair \( (sk_{\text{attach}}, vk_{\text{attach}}) \) by calling SealWhole.UserKg. It then computes the nonce \( n_{\text{attach}} \) as the first 128 bits of SHA-256(\( ad_{\text{attach}} \)). Next it calls the SealWhole.SigEnc algorithm to encrypt \( m_{\text{attach}} \) using the key \( k_{\text{attach}} \), nonce \( n_{\text{attach}} \), and signing key \( sk_u \) to obtain the ciphertext \( c_{\text{attach}} \). Subsequently it calls \( \text{SHA-256} \) on \( n_{\text{attach}} \) and \( c_{\text{attach}} \) to produce the hashes \( h_m \) and \( h_c \). It then builds the wrapper message \( m_{\text{wrap}} = (k_{\text{attach}}, vk_{\text{attach}}, n_{\text{attach}}, h_m, h_c) \). Finally PostFileAttachment.SigEnc invokes BoxMessage on \( m_{\text{wrap}}, K_g, sk_u, n_{\text{wrap}}, \) and \( ad_{\text{wrap}} \) to obtain the ciphertext \( c_{\text{wrap}} \). The algorithm returns \( (c_{\text{wrap}}, c_{\text{attach}}, n_{\text{attach}}) \).

We do not further analyze the SealWhole and PostFileAttachment algorithms in this work.
PostFileAttachment.SigEnc\( (g, K_g, u, sk_u, n_{\text{wrap}}, m_{\text{attach}}, ad) \)  // \( K_g \in \{0,1\}^{256} \)

\[
(ad_{\text{attach}}, ad_{\text{wrap}}) \leftarrow ad \\
(k_{\text{attach}}) \leftarrow \{0,1\}^{256} \\
(vk_{\text{attach}}, vk_{\text{attach}}) \leftarrow \text{SealWhole.UserKg}  // \text{SealWhole} = \text{SEAL-WHOLE-SS} \\
(n_{\text{attach}}) \leftarrow \text{H}(ad_{\text{attach}})[1..128]  // \text{H} = \text{SHA-256} \; n_{\text{attach}} \text{ assigned the first 128 bits of the H output} \\
c_{\text{attach}} \leftarrow \text{SealWhole.SigEnc}(g, k_{\text{attach}}, u, sk_{\text{attach}}, n_{\text{attach}}, m_{\text{attach}}, \varepsilon) \\
h_m \leftarrow \text{H}(m_{\text{attach}}) \\
h_c \leftarrow \text{H}(c_{\text{attach}}) \\
m_{\text{wrap}} \leftarrow (k_{\text{attach}}, vk_{\text{attach}}, n_{\text{attach}}, h_m, h_c) \\
c_{\text{wrap}} \leftarrow \text{BM.SigEnc}(g, K_g, u, sk_u, n_{\text{wrap}}, m_{\text{wrap}}, ad_{\text{wrap}}) \\
c \leftarrow (c_{\text{wrap}}, c_{\text{attach}}, n_{\text{attach}}) \\
\text{Return } c
\]

Fig. 42: The PostFileAttachment algorithm used in Keybase to encrypt and upload file attachments to a third-party storage.