Technical Report TI-5/02 (9th August 2002)
TU Darmstadt, Fachbereich Informatik

Provably Secure Public-Key Encryption for Length-Preserving Chaumian Mixes

Bodo Möller

Technische Universität Darmstadt, Fachbereich Informatik
moeller@cdc.informatik.tu-darmstadt.de

Abstract. Mix chains as proposed by Chaum allow sending untraceable electronic e-mail without requiring trust in a single authority: messages are recursively public-key encrypted to multiple intermediates (mixes), each of which forwards the message after removing one layer of encryption. To conceal as much information as possible when using variable (source routed) chains, all messages passed to mixes should be of the same length; thus, message length should not decrease when a mix transforms an input message into the corresponding output message directed at the next mix in the chain. Chaum described an implementation for such length-preserving mixes, but it is not secure against active attacks. We show how to build practical cryptographically secure length-preserving mixes. The conventional definition of security against chosen ciphertext attacks is not applicable to length-preserving mixes; we give an appropriate definition and show that our construction achieves provable security.

Keywords: Cryptographic remailers, chosen ciphertext attack security, public-key cryptosystems with length-expanding decryption

1 Introduction

Chaum's *mix* concept [5] is intended to allow users to send untraceable electronic e-mail without having to trust a single authority. The idea is to use a number of intermediates such that it suffices for just one of these to be trustworthy in order to achieve untraceability. (The sender does not have to decide which particular intermediate he is willing to trust, he just must be convinced that at least one in a given list will behave as expected.) These intermediates, the *mixes,* are remailers accepting public-key encrypted input. Messages must be of a fixed size (shorter messages can be padded, longer messages can be split into multiple parts). To send a message, it is routed through a chain of mixes $\mathsf{M}_1, \ldots, \mathsf{M}_n$: the sender obtains the public key of each mix; then he recursively encrypts the message (including the address of the final recipient) yielding $E_{\mathsf{M}_1}\big(E_{\mathsf{M}_2}(\ldots E_{\mathsf{M}_n}(\mathit{payload})\ldots)\big)$ where E_{M_i} denotes encryption with M_i's public key, and sends the resulting ciphertext to mix M_1. (The public-key cryptosystem will typically be hybrid, i.e. involve use of symmetric-key cryptography for bulk data encryption.) Each mix removes the corresponding layer of encryption and

forwards the decrypted message to the next mix; thus mix M_n will finally recover *payload*.

Each mix is expected to collect a large batch of messages before forwarding the decryption results. The messages in the batch must be reordered (mixed) to prevent message tracing. It is important to prevent replay attacks – a mix must not process the same message twice, or active adversaries would be able to trace messages by duplicating them at submission to cause multiple delivery. (Timestamps can be used to limit the timespan for which mixes have to remember which messages they have already processed; see [5] and [6].)

Usually it is desireable to allow *source routing,* i.e. let senders choose mix chains on a per-message basis. This increases the flexibility of the whole scheme: senders can make use of new mixes that go into operation, and they can avoid mixes that appear not to work properly; in particular, they can avoid mixes that suppress messages (be it intentionally or because of technical problems). For source routing, in the recursively encrypted message, each layer must contain the address of the next mix in the chain so that each mix knows where to forward the message. A problem with the straightforward implementation of source routing is that messages will shrink as they proceed through the chain, not only because of the forwarding address for each layer that must be included, but also because public-key encryption increases message sizes. For optimal untraceability, we need *length-preserving mixes:* the messages that mixes receive should essentially look like the resulting messages that they forward to other mixes.

A construction of length-preserving mixes is given in [5] (a variant of this is used in fielded systems [12]): mix messages consist of a fixed number of slots of a fixed size. The first slot is public-key encrypted so that it can be read the first mix in the chain. Besides control information directed at the respective mix (such as the address of the next mix in the chain or, in case of the last mix, the address of the final recipient), decryption yields a symmetric key that the mix uses to decrypt all the other slots. Then slots are shifted by one position: the decryption of the second slot becomes the new first slot, and so on. A new final slot is added consisting of random (garbage) data to obtain a mix message of the desired fixed length, which can the be forwarded to the next mix. On the way through the chain, each mix message consists of a number of slots with valid data followed by enough slots with garbage to fill all available slots; mixes are oblivious of how many slots contain valid data and how many are random. Each mix in the chain, when decrypting and shifting the slots, will "decrypt" the garbage slots already present and add a new final slot, thus increasing the number of random slots by one. We note that while the transformation of an incoming mix message to the corresponding outgoing mix message is *length-preserving,* the decryption step itself is actually *length-expanding* because some of the data obtained by decryption is control data directed at the current mix.

The problem with this method for obtaining a hybrid public-key cryptosystem with length-expanding decryption is that it is not secure against active attacks: assume that an adversary controls at least one mix, and that all senders submit well-formed messages. Now when the victim submits a message, the ad-

versary can mark it by modifying one of the slots. This mark will persist as the message is forwarded through the mix chain: due to the decryption and slot shifting performed by each mix, the corresponding slot will essentially contain garbage. If the final mix is controlled by the adversary, the adversary may be able to notice the modification and thus break untraceability.

To rule out such attacks, the hybrid public-key cryptosystem should provide security against adaptive chosen ciphertext attacks (CCA security): that is, it should be secure against an adversary who can request the decryption of arbitrary ciphertexts.

In this paper, we present a secure and practical hybrid construction for length-preserving mixes, which is also more flexible than the slot-based approach. The standard notion of CCA security for public-key cryptosystems is not applicable to our hybrid public-key cryptosystem with length-expanding decryption because the encryption operation must be defined differently for our application: encryption cannot be treated as an atomic black-box operation that takes a plaintext and returns a ciphertext – in this model, we cannot have length-extending decryption. Rather, our encryption process first is input part of the desired *ciphertext* and determines a corresponding part of what will be the decryption result (it is this step that provides length expansion); then it is input the payload plaintext and finally outputs the complete ciphertext, which includes the portion requested in the first input.

We describe the hybrid construction in section 2. Section 3 discusses appropriate security notions and gives a provable security result for the construction.

This paper shows for the first time how to implement length-preserving mixes cryptographically secure against active attacks. We use many ideas and techniques described by Cramer and Shoup in [7], but adapt them to fit the new notions of security needed in the context of length-preserving mixes.

1.1 Notation

Strings are binary, i.e. elements of $\{0,1\}^*$. The concatenation of strings s and t is denoted $s \parallel t$. The length of string s is $|s|$. For $|s| \geq w$, $\text{prefix}_w(s)$ denotes the string consisting of the leftmost w bits of s; thus, $\text{prefix}_{|s|}(s \parallel t) = s$. Messages (plaintexts, ciphertexts) are strings.

Algorithms are usually probabilistic (randomized).

2 A Construction for Length-Preserving Mixes

Our construction allows much flexibility in the choice of cryptographic schemes, even in a single chain. The single parameter that must be fixed for all mixes is the desired mix message length ℓ. Also it may be desireable to define a maximum length for the actual message payload, i.e. the part of the plaintext as recovered by the final mix that can be chosen by the sender: as a message proceeds through the mix chain, more and more of the data will be pseudo-random gibberish; the length of the useful part of the final plaintext reveals that chains leaving less

than this amount cannot have been used. The length n of the mix chain need not be fixed.

For purposes of exposition, we number the mixes $\mathsf{M}_1, \ldots, \mathsf{M}_n$ according to their position in the chain chosen by the sender (note that the same mix might appear multiple times in one chain). For each mix M_i, the following must be defined and, with the exception of the secret key $\mathsf{SK}_{\mathsf{M}_i}$, known to senders who want to use the mix:

– A *key encapsulation mechanism*

$$\mathsf{KEM}_{\mathsf{M}_i}$$

and a key pair

$$(\mathsf{PK}_{\mathsf{M}_i}, \mathsf{SK}_{\mathsf{M}_i})$$

consisting of a *public key* $\mathsf{PK}_{\mathsf{M}_i}$ and a *secret key* $\mathsf{SK}_{\mathsf{M}_i}$ for this key encapsulation mechanism. A key encapsulation mechanism, similarly to a public-key encryption mechanism, provides an algorithm

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}$$

using the public key and an algorithm

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Decrypt}$$

using the secret key. In contrast with public-key encryption, the first algorithm takes no input apart from the public key: $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}(\mathsf{PK}_{\mathsf{M}_i})$ generates a ciphertext $\mathfrak{K}$ of a fixed length

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen}$$

corresponding to a (pseudo-)random string K of a fixed length

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{OutLen}$$

and outputs the pair $(K, \mathfrak{K})$. Evaluation of $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Decrypt}(\mathsf{SK}_{\mathsf{M}_i}, \mathfrak{K})$ will return said string K. That is, ciphertext $\mathfrak{K}$ encapsulates the random message K (which can be used as a key for symmetric-key cryptography). On arbitrary inputs $\mathfrak{K}$, the computation $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Decrypt}(\mathsf{SK}_{\mathsf{M}_i}, \mathfrak{K})$ may either return some string K or fail (return the special value $\mathsf{invalid}$).
Associated with key encapsulation mechanism $\mathsf{KEM}_{\mathsf{M}_i}$ is a *key generation algorithm*

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{KeyGen}$$

that returns appropriate key pairs $(\mathsf{PK}, \mathsf{SK})$. If M_i is a correctly operating mix, its key pair $(\mathsf{PK}_{\mathsf{M}_i}, \mathsf{SK}_{\mathsf{M}_i})$ has been generated by this algorithm. (Otherwise $\mathsf{PK}_{\mathsf{M}_i}$ may have been constructed differently; it might even be invalid in the sense that there is no corresponding secret key $\mathsf{SK}_{\mathsf{M}_i}$ that could be used for decryption.) Note that we use no explicit security parameter, so desired key sizes are implicit to $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{KeyGen}$.

One example of a key encapsulation mechanism is the Diffie-Hellman key exchange [8] with static keys for one party and ephemeral keys for the other party where the concatenation of the ephemeral public key with the common secret group element is hashed to obtain a symmetric key (cf. [1], [14]); $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen}$ can be kept particularly small for elliptic curve Diffie-Hellman using just x coordinates of points (see [11]). Specifications for various key encapsulation mechanisms can be found in [14].

– A *one-time message authentication code*

$$\mathsf{MAC}_{\mathsf{M}_i}$$

with key length

$$\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen}$$

and output length

$$\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}.$$

A one-time message authentication code specifies an efficient deterministic algorithm that takes as input a key K of length $\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen}$ and a bit string s and returns a string $\mathsf{MAC}_{\mathsf{M}_i}(K, s)$ of length $\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}$. In our construction, $\mathsf{MAC}_{\mathsf{M}_i}$ will only be used for strings s of fixed length $\ell - \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} - \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}$.
Candidate one-time message authentication codes are UHASH [9][1] and HMAC [2].

– A *pseudo-random bit string generator*

$$\mathsf{STREAM}_{\mathsf{M}_i}$$

taking as input a key K of length

$$\mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{KeyLen}$$

and deterministically generating an output string $\mathsf{STREAM}_{\mathsf{M}_i}(K)$ of length

$$\mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{OutLen}.$$

This will be used for an XOR-based stream cipher. A convenient example implementation is the so-called *counter mode* of a block cipher (the output sequence is the prefix of appropriate length of $E_K(0) \parallel E_K(1) \parallel E_K(2) \parallel \ldots$ where E_K denotes block cipher encryption using key K; see [3] and [10]).

– An integer

$$\mathsf{PlainLen}_{\mathsf{M}_i}$$

specifying the length of the prefix of each decrypted message that is considered control data directed to mix $\mathsf{M}_{\mathsf{M}_i}$ and will not be forwarded. (This is the amount of message expansion: the decrypted message minus the prefix must be of size ℓ because that is what will be sent to the next mix.)

[1] UHASH is the combination of a key derivation function with an almost strongly universal hash function [15] and is the core of UMAC as specified in [9]. Note that the security arguments provided for the earlier version of UMAC in [4] are based on a different approach.

The parameters must fulfil the following conditions:

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} + \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen} + \mathsf{PlainLen}_{\mathsf{M}_i} < \ell \quad (1)$$

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{OutLen} = \mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{KeyLen} + \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen} \quad (2)$$

$$\mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{OutLen} = \mathsf{PlainLen}_{\mathsf{M}_i} + \ell \quad (3)$$

2.1 Encryption

We now describe encryption for sending a message through a chain $\mathsf{M}_1, \ldots, \mathsf{M}_n$. Let *payload* be the message of length

$$|payload| = \ell - \sum_{1 \le i \le n} (\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} + \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen} + \mathsf{PlainLen}_{\mathsf{M}_i}) \quad (4)$$

(messages shorter than this maximum should be randomly padded on the right). For each i, let $plain_i$ be the control message of length $\mathsf{PlainLen}_{\mathsf{M}_i}$ directed to the respective mix. The encryption algorithm for these arguments is denoted

$$chain\_encrypt_{\mathsf{M}_1,\ldots,\mathsf{M}_n}(plain_1, \ldots, plain_n; payload)$$

or

$$chain\_encrypt_{\mathsf{M}_1,\ldots,\mathsf{M}_n}(plain_1, \ldots, plain_n; payload; \lambda)$$

where λ is the empty string. The algorithm is defined recursively. Let $1 \le i \le n$, and let C_i be a string of length

$$|C_i| = \sum_{1 \le k < i} (\mathsf{KEM}_{\mathsf{M}_k}.\mathsf{CipherLen} + \mathsf{MAC}_{\mathsf{M}_k}.\mathsf{OutLen} + \mathsf{PlainLen}_{\mathsf{M}_k}) \quad (5)$$

(thus specifically $|C_1| = 0$). Then algorithm

$$chain\_encrypt_{\mathsf{M}_i,\ldots,\mathsf{M}_n}(plain_i, \ldots, plain_n; payload; C_i)$$

works as follows:

1. Use $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}(\mathsf{PK}_{\mathsf{M}_i})$ to generate a pair $(K_i, \mathfrak{K}_i)$.
2. Split K_i in the form
$$K_i = K_{i,\mathsf{MAC}} \parallel K_{i,\mathsf{STREAM}}$$
such that $|K_{i,\mathsf{MAC}}| = \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen}$ (so $|K_{i,\mathsf{STREAM}}| = \mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{KeyLen}$ by (2)).
3. Compute $\mathsf{STREAM}_{\mathsf{M}_i}(K_{i,\mathsf{STREAM}})$ and split this string in the form
$$stream_{i,\mathsf{L}} \parallel stream_{i,\mathsf{R}}$$
such that the left part $stream_{i,\mathsf{L}}$ is of length
$$\ell - |C_i| - \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} - \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}$$
and the right part $stream_{i,\mathsf{R}}$ accordingly, by (3), is of length
$$|C_i| + \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} + \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen} + \mathsf{PlainLen}_{\mathsf{M}_i}.$$

4. If $i = n$ (last mix), then by (4) and (5) it follows that $|stream_{n,\mathsf{L}}| = \mathsf{PlainLen}_n + |payload|$. In this case, set

$$\mathfrak{C}_n = \big(stream_{n,\mathsf{L}} \oplus (plain_n \parallel payload)\big) \parallel C_n.$$

   Otherwise, let

$$C_{i+1} = stream_{i,\mathsf{R}} \oplus (C_i \parallel 0^{\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen}+\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}+\mathsf{PlainLen}_{\mathsf{M}_i}}),$$

   by recursion compute

$$x_i = chain\_encrypt_{\mathsf{M}_{i+1},\ldots,\mathsf{M}_n}(plain_{i+1}, \ldots, plain_n; payload; C_{i+1}),$$

   and let

$$\mathfrak{C}_i = \big(stream_{i,\mathsf{L}} \oplus (plain_i \parallel \mathrm{prefix}_{|stream_{i,\mathsf{L}}|-\mathsf{PlainLen}_{\mathsf{M}_i}}(x_i))\big) \parallel C_i.$$

5. Compute $\mathfrak{M}_i = \mathsf{MAC}_i(K_{i,\mathsf{MAC}}, \mathfrak{C}_i)$.
6. Return the ciphertext $\mathfrak{K}_i \parallel \mathfrak{M}_i \parallel \mathfrak{C}_i$, which is of length ℓ.

Appendix A.1 illustrates a ciphertext generated by $chain\_encrypt_{\mathsf{M}_1,\ldots,\mathsf{M}_n}$.

2.2 Decryption

The decryption algorithm for mix M_i ($1 \le i \le n$) works as follows, given a length-ℓ ciphertext $\mathfrak{K} \parallel \mathfrak{M} \parallel \mathfrak{C}$ (split into its three components according to parameters $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} = |\mathfrak{K}|$ and $\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen} = |\mathfrak{M}|$). We denote it

$$mix\_decrypt_{\mathsf{M}_i}(\mathfrak{K} \parallel \mathfrak{M} \parallel \mathfrak{C}).$$

Remember that M_i in general is not aware of the mix chain used by the sender or even of its own position i in the chain.

1. Compute $K = \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Decrypt}(\mathsf{SK}_{\mathsf{M}_i}, \mathfrak{K})$. If this computation fails, abort with an error (return invalid).
2. Split K in the form
$$K = K_{\mathsf{MAC}} \parallel K_{\mathsf{STREAM}}$$
   such that $|K_{\mathsf{MAC}}| = \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen}$.
3. Compute
$$\widetilde{\mathfrak{M}} = \mathsf{MAC}_{\mathsf{M}_i}(K_{\mathsf{MAC}}, \mathfrak{C})$$
   and test whether
$$\widetilde{\mathfrak{M}} = \mathfrak{M}.$$
   If this is not the case, abort with an error (return invalid).
4. Compute the string
$$stream = \mathsf{STREAM}_{\mathsf{M}_i}(K_{\mathsf{STREAM}})$$
   of length $\mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{OutLen}$.

5. Compute

$$stream \oplus (\mathfrak{C} \parallel 0^{\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen}+\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}+\mathsf{PlainLen}_{\mathsf{M}_i}})$$

and split the resulting string in the form

$$plain_i \parallel P$$

where $plain_i$ is of length $\mathsf{PlainLen}_{\mathsf{M}_i}$ (and thus, by (3), P is of length ℓ).
6. Return the pair $(plain_i, P)$.

If this algorithm finishes without an error, $plain_i$ is a control message directed to the current mix, and P is the message to be forwarded to another mix or to the final recipient (as requested by the control message).

It is straightforward to verify that for ciphertexts computed as

$$chain\_encrypt_{\mathsf{M}_1,\ldots,\mathsf{M}_n}(plain_1, \ldots, plain_n; payload),$$

iterative decryption by mixes $\mathsf{M}_1, \ldots, \mathsf{M}_n$ will indeed work without an error and recover the respective strings $plain_i$ and finally also the message *payload* concatenated with some (useless) extra data. Appendix A.2 illustrates decryption.

3 Provable Security

The mix concept is intended to provide security when at least one mix can be trusted and behaves correctly. (However note that denial-of-service attacks by incorrectly operating mixes cannot be ruled out, the only option is to avoid mixes that appear to malfunction.) Thus we assume that some mix M_i works as expected while all other mixes are controlled by the adversary and may not follow the protocol (also the cryptographic schemes $\mathsf{KEM}_{\mathsf{M}_j}$, $\mathsf{MAC}_{\mathsf{M}_j}$, and $\mathsf{STREAM}_{\mathsf{M}_j}$ associated with mixes M_j, $j \neq i$, might be not secure).

This leaves only M_i to be attacked: outer layers of encryption for mixes that appear before M_i in a mix chain are easily removed by the adversary and thus are not relevant for security; and inner layers of encryption for mixes that appear after M_i in a mix chain are involved in the encryption process

$$chain\_encrypt_{\mathsf{M}_i,\ldots,\mathsf{M}_n}(plain_i, \ldots, plain_n; payload; C_i)$$

described in section 2.1, but cannot provide protection against the adversary.

Section 3.1 shows how the security of the mix construction can be captured formally. We then provide security definitions for the underlying key encapsulation method (section 3.2), one-time message authentication code (section 3.3), and pseudo-random bit string generator (section 3.4). Finally, section 3.5 presents a security result that relates the security of the mix encryption scheme to the security of these cryptographic schemes: we will see that if the mix encryption scheme is not secure, then this is because one of the underlying cryptographic schemes is not secure.

3.1 Security of the Mix Encryption Scheme

To show that our construction for length-preserving mixes is secure, we want to model an active adversary who is able to launch an adaptive chosen ciphertext attack (CCA). Due to the recursive nature of our encryption algorithm for mix chains, we cannot directly apply the usual CCA definitions for ordinary public-key encryption; we adapt the attack game described in [7, section 3.2] (which goes back to [13]) as follows to take into account the special properties of our construction:

1. The adversary queries a *key generation oracle*, which uses $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{KeyGen}$ to compute a key pair
$$(\mathsf{PK}, \mathsf{SK})$$
and responds with PK (and secretly stores SK).
2. The adversary makes a sequence of queries to a *decryption oracle.* Each query is an arbitrary string s of length ℓ, and the oracle responds with
$$mix\_decrypt_{\mathsf{M}_i}(s),$$
using the secret key SK from step 1. That is, each oracle response is either a pair $(plain, P)$ where $|plain| = \mathsf{PlainLen}_{\mathsf{M}_i}$ and $|P| = \ell$, or the special value $\mathsf{invalid}$.
3. The adversary uses an *interactive encryption oracle* as follows (compare with the encryption algorithm in section 2.1):
   - First the adversary submits some string C_i subject only to the condition that
   $$0 \le |C_i| < \ell - \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} - \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}.$$
   - The interactive encryption oracle uses $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}(\mathsf{PK})$ to generate a pair $(K_{\mathsf{oracle}}, \mathfrak{K}_{\mathsf{oracle}})$ and splits K_{oracle} in the form
   $$K_{\mathsf{oracle}} = K_{\mathsf{MAC}} \parallel K_{\mathsf{STREAM}}$$
   such that $|K_{\mathsf{MAC}}| = \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen}$; it computes $\mathsf{STREAM}_{\mathsf{M}_i}(K_{\mathsf{STREAM}})$ and splits the resulting string *stream* in the form
   $$stream = stream_{\mathsf{L}} \parallel stream_{\mathsf{R}}$$
   such that $|stream_{\mathsf{L}}| = \ell - |C_i| - \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} - \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}$; and it computes
   $$C_{i+1} = stream_{\mathsf{R}} \oplus (C_i \parallel 0^{\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen}+\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}+\mathsf{PlainLen}_{\mathsf{M}_i}})$$
   and sends C_{i+1} to the adversary.
   - The adversary submits a pair of strings (m_0, m_1) satisfying
   $$|m_0| = |m_1| = \ell - |C_i| - \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} - \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}.$$

– The interactive encryption oracle chooses a uniformly random bit $b \in \{0,1\}$, determines

$$\mathfrak{C} = (\mathit{stream}_{\mathsf{L}} \oplus m_b) \parallel C_i$$

and

$$\mathfrak{M} = \mathsf{MAC}_{\mathsf{M}_i}(K_{\mathsf{MAC}}, \mathfrak{C}_i),$$

and responds with

$$\mathfrak{K}_{\mathsf{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}.$$

4. The adversary again makes a sequence of queries to a *decryption oracle* as in step 2, except that this time the decryption oracle refuses being asked for the challenge ciphertext $\mathfrak{K}_{\mathsf{oracle}} \parallel \mathfrak{M} \parallel \mathfrak{C}$ from step 3 (it returns $\mathsf{invalid}$ for this case).
5. The adversary outputs a bit $\widetilde{b} \in \{0,1\}$.

The bit $\widetilde{b}$ output by the adversary is its guess for the value of b.

The essential difference to the attack game for ordinary public-key encryption is that we have made the encryption oracle interactive to reflect the recursiveness of the encryption algorithm for mix chains, where encryption to mixes later in the chain than M_i can have potentially arbitrary effects on a large part of the plaintext.

Let A be any adversary (interactive probabilistic algorithm with bounded runtime) in the above attack game. Its *CCA advantage* against M_i's instantiation of the mix encryption scheme is

$$\mathsf{AdvCCA}_{\mathsf{M}_i,A} = \left|\Pr\left[\widetilde{b} = 1 \mid b = 1\right] - \Pr\left[\widetilde{b} = 1 \mid b = 0\right]\right|.$$

3.2 Security of the Key Encapsulation Mechanism

CCA security for the key encryption mechanism $\mathsf{KEM}_{\mathsf{M}_i}$ is defined through the following attack game (cf. [7, section 7.1.2]):

1. The adversary queries a *key generation oracle*, which uses $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{KeyGen}$ to compute a key pair $(\mathsf{PK}, \mathsf{SK})$ and responds with PK.
2. The adversary makes a sequence of queries to a *decryption oracle.* Each query is an arbitrary string s of length $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen}$; the oracle responds with

$$\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Decrypt}(\mathsf{SK}, s).$$

   Thus each oracle response is either a string of length $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{OutLen}$ or the special value $\mathsf{invalid}$.
3. The adversary queries an *encryption oracle,* which works as follows: it uses $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}(\mathsf{PK})$ to obtain a pair $(K_0, \mathfrak{K}_{\mathsf{oracle}})$, it generates a uniformly randomly string K_1 such that $|K_0| = |K_1|$, chooses a uniformly random bit $b_{\mathsf{KEM}} \in \{0,1\}$, and responds with $(K_{b_{\mathsf{KEM}}}, \mathfrak{K}_{\mathsf{oracle}})$.
4. The adversary again makes a sequence of queries to a *decryption oracle* as in step 2, but this time the oracle refuses the specific query $\mathfrak{K}_{\mathsf{oracle}}$ and responds $\mathsf{invalid}$ in this case.

5. The adversary outputs a bit $\widetilde{b}_{\mathsf{KEM}} \in \{0,1\}$.

The *CCA advantage* of an adversary A_1 (an interactive probabilistic algorithm with bounded runtime) against $\mathsf{KEM}_{\mathsf{M}_i}$ in this attack game is

$$\mathsf{AdvCCA}_{\mathsf{KEM}_{\mathsf{M}_i},A_1} = \Big|\Pr\left[\widetilde{b}_{\mathsf{KEM}} = 1 \mid b_{\mathsf{KEM}} = 1\right] - \Pr\left[\widetilde{b}_{\mathsf{KEM}} = 1 \mid b_{\mathsf{KEM}} = 0\right]\Big|.$$

3.3 Security of the One-Time Message Authentication Code

To define the security of $\mathsf{MAC}_{\mathsf{M}_i}$, we use the following attack game (cf. [7, section 7.2.2]:

1. The adversary submits a string s (which for our purposes may be assumed to be of fixed length $\ell - \mathsf{KEM}_{\mathsf{M}_i}.\mathsf{CipherLen} - \mathsf{MAC}_{\mathsf{M}_i}.\mathsf{OutLen}$) to an oracle. The oracle generates a uniformly random string K of length $\mathsf{MAC}_{\mathsf{M}_i}.\mathsf{KeyLen}$ and responds with $\mathsf{MAC}_{\mathsf{M}_i}(K, s)$.
2. The adversary outputs a list $(s_1, t_1), (s_2, t_2), \ldots, (s_m, t_m)$ of pairs of string.

An adversary A_2 again is an interactive probabilistic algorithm with bounded runtime; the runtime bound implies a bound on the list length m. We say that adversary A_2 has *produced a forgery* if $s_k \neq s$ and $\mathsf{MAC}_{\mathsf{M}_i}(K, s_k) = t_k$ for some k ($1 \leq k \leq m$). Its advantage against $\mathsf{MAC}_{\mathsf{M}_i}$, denoted $\mathsf{AdvForge}_{\mathsf{MAC}_{\mathsf{M}_i},A_2}$, is the probability that it produces a forgery in the above game.

3.4 Security of the Pseudo-Random Bit String Generator

To define the security of $\mathsf{STREAM}_{\mathsf{M}_i}$, we use the following attack game:

1. The adversary queries an oracle. The oracle generates a uniformly random string K of length $\mathsf{STREAM}_{\mathsf{M}_i}.\mathsf{KeyLen}$, computes $\mathit{stream}_0 = \mathsf{STREAM}_{\mathsf{M}_i}(K)$, generates a uniformly random string stream_1 with $|\mathit{stream}_0| = |\mathit{stream}_1|$, chooses a uniformly random bit $b_{\mathsf{STREAM}} \in \{0,1\}$, and responds with stream_b.
2. The adversary outputs a bit $\widetilde{b}_{\mathsf{STREAM}} \in \{0,1\}$.

The advantage of an adversary A_3 (again an interactive probabilistic algorithm with bounded runtime) against $\mathsf{STREAM}_{\mathsf{M}_i}$ is

$$\mathsf{Adv}_{\mathsf{STREAM}_{\mathsf{M}_i},A_3} = \Big|\Pr\left[\widetilde{b}_{\mathsf{STREAM}} = b_{\mathsf{STREAM}}\right] - \frac{1}{2}\Big|.$$

3.5 Security Result

Let A be an adversary A against the mix encryption scheme attacking a mix M_i as described in section 3.1. It can be shown that there are adversaries A_1, A_2, A_3 against $\mathsf{KEM}_{\mathsf{M}_i}$, $\mathsf{MAC}_{\mathsf{M}_i}$, $\mathsf{STREAM}_{\mathsf{M}_i}$ all having essentially the same runtime as A such that

$$\mathsf{AdvCCA}_{\mathsf{M}_i,A} \leq 2 \cdot \big(\mathsf{AdvCCA}_{\mathsf{KEM}_{\mathsf{M}_i},A_1} + \mathsf{AdvForge}_{\mathsf{MAC}_{\mathsf{M}_i},A_2} + \mathsf{Adv}_{\mathsf{STREAM}_{\mathsf{M}_i},A_3}\big).$$

The proof uses standard techniques; details can be found in appendix B.

References

1. Abdalla, M., Bellare, M., and Rogaway, P. DHAES: An encryption scheme based on the Diffie-Hellman problem. Submission to IEEE P1363a. `http://grouper.ieee.org/groups/1363/P1363a/Encryption.html`, 1998.
2. Bellare, M., Canetti, R., and Krawczyk, H. Keying hash functions for message authentication. In *Advances in Cryptology – CRYPTO '96* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, pp. 1–15.
3. Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science (FOCS '97)* (1997), IEEE Computer Society, pp. 394–403.
4. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology – CRYPTO '99* (1999), M. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, pp. 216–233.
5. Chaum, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24* (1981), 84–88.
6. Cottrell, L. Mixmaster & remailer attacks. `http://www.obscura.com/%7Eloki/remailer/remailer-essay.html`, 1997.
7. Cramer, R., and Shoup, V. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. Manuscript, `http://shoup.net/papers/`, 2001.
8. Diffie, W., and Hellman, M. E. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654.
9. Krovetz, T., Black, J., Halevi, S., Hevia, A., Krawczyk, H., and Rogaway, P. UMAC: Message authentication code using universal hashing. Internet-Draft `draft-krovetz-umac-01.txt`, `http://www.cs.ucdavis.edu/~rogaway/umac/`, 2000.
10. Lipmaa, H., Rogaway, P., and Wagner, D. Comments to NIST concerning AES modes of operation: CTR-mode encryption. `http://csrc.nist.gov/encryption/modes/workshop1/papers/lipmaa-ctr.pdf`, 2000.
11. Miller, V. S. Use of elliptic curves in cryptography. In *Advances in Cryptology – CRYPTO '85* (1986), H. C. Williams, Ed., vol. 218 of *Lecture Notes in Computer Science*, pp. 417–428.
12. Mixmaster anonymous remailer software. `http://sourceforge.net/projects/mixmaster/`.
13. Rackoff, C. W., and Simon, D. R. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology – CRYPTO '91* (1992), J. Feigenbaum, Ed., vol. 576 of *Lecture Notes in Computer Science*, pp. 433–444.
14. Shoup, V. A proposal for an ISO standard for public key encryption. Version 2.1, December 20, 2001. `http://shoup.net/papers/`.
15. Wegman, M. N., and Carter, J. L. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences 22* (1981), 265–279.

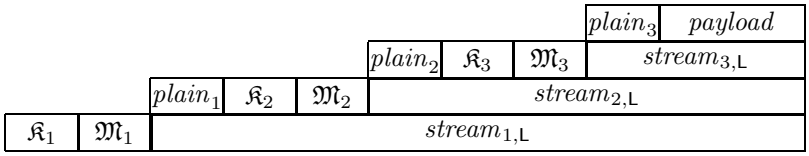
A Illustrations

A.1 Encryption

We depict a ciphertext generated by the encryption algorithm

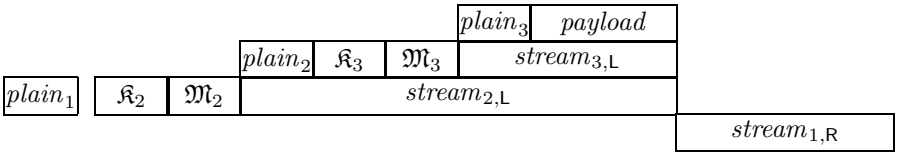
$$chain\_encrypt_{\mathsf{M}_1,\mathsf{M}_2,\mathsf{M}_3}(plain_1, plain_2, plain_3; payload)$$

as described in section 2.1. In the illustration, we have concatenation horizontally and XOR vertically (i.e. boxes in the same row represent bit strings that are concatenated, and the ciphertext is the XOR of the multiple rows shown).

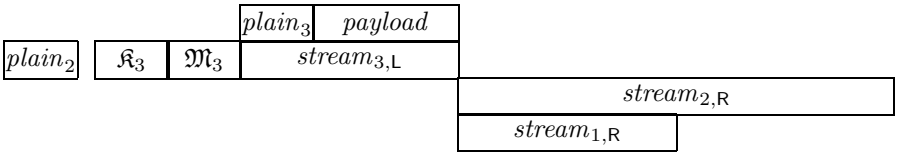


A.2 Decryption

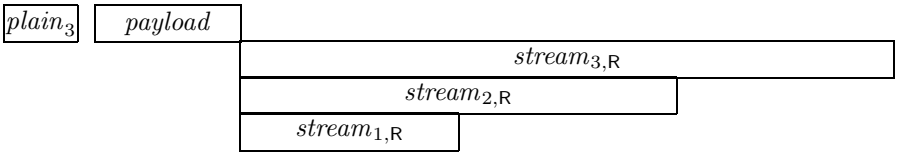
The result obtained by mix M_1 when applying the decryption algorithm from section 2.2 to the ciphertext from appendix A.1 is composed as follows:



The string $plain_1$ is directed to M_1. The remainder of the decryption result is a ciphertext that should be forwarded to the next mix in the chain, M_2, which will then obtain the following result:



Similarly, mix M_3 will obtain the following final decryption result:



B Security Proof

We prove the security result for the mix encryption scheme given in section 3.5. Let $\mathbf{G}_0$ denote the attack game from section 3.1. We will modify it in multiple steps, essentially disabling the underlying cryptographic schemes (key encryption method, one-time message authentication code, and pseudo-random bit string generator) one after another.

$\mathbf{G}_1$ is like $\mathbf{G}_0$ except that a uniformly random string is used for K_{oracle}, whereas $\mathfrak{K}_{\mathsf{oracle}}$ is still generated by $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}(\mathsf{PK})$. In the decryption oracle, K is substituted whenever $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Decrypt}(\mathsf{SK}, \mathfrak{K}_{\mathsf{oracle}})$ would have to be computed.

$\mathbf{G}_2$ is like $\mathbf{G}_1$ except that the decryption oracle always responds with $\mathsf{invalid}$ when faced with any query prefixed with string $\mathfrak{K}_{\mathsf{oracle}}$. (This applies to both step 2 and step 4 in section 3.1. Thus, the invocation of $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{Encrypt}(\mathsf{PK})$ to generate $(K_{\mathsf{oracle}}, \mathfrak{K}_{\mathsf{oracle}})$ must be advanced from step 3 to an earlier step; this is only a descriptive change, it does not affect the behaviour observed by the adversary.)

$\mathbf{G}_3$ is like $\mathbf{G}_2$ except that the interactive encryption oracle uses a uniformly random string *stream* instead of computing $\mathsf{STREAM}_{\mathsf{M}_i}(K_{\mathsf{STREAM}})$.

Now we consider an adversary A as in section 3.1, exposed to these different attack games $\mathbf{G}_x$, $0 \leq x \leq 3$, and look at the respective success probabilities $\Pr_{\mathbf{G}_x}\left[\widetilde{b} = b\right]$. Based on A, adversaries A_1, A_2, A_3 against $\mathsf{KEM}_{\mathsf{M}_i}$, $\mathsf{MAC}_{\mathsf{M}_i}$, $\mathsf{STREAM}_{\mathsf{M}_i}$ will be built all having essentially the same runtime as A.

A_1 attacks $\mathsf{KEM}_{\mathsf{M}_i}$ as follows (cf. section 3.2). At first, it generates $b \in \{0,1\}$ uniformly at random. Then, it runs the adversary A; when A queries its decryption oracle, A_1 uses its own decryption oracle to perform the decryption algorithm from section 2.2, and when A queries its encryption oracle, A_1 queries its own encryption oracle to obtain a pair $(K_{\mathsf{oracle}}, \mathfrak{K}_{\mathsf{oracle}})$ and performs step 3 from section 3.1 using this pair and the pregenerated bit b. Finally, when A outputs its bit $\widetilde{b}$, A_1 outputs 1 if $\widetilde{b} = b$ and 0 otherwise. Observe that

$$\left|\Pr_{\mathbf{G}_1}\left[\widetilde{b} = b\right] - \Pr_{\mathbf{G}_0}\left[\widetilde{b} = b\right]\right| = \mathsf{AdvCCA}_{\mathsf{KEM}_{\mathsf{M}_i}, A_1}$$

($\mathbf{G}_0$ corresponds to $b_{\mathsf{KEM}} = 0$, $\mathbf{G}_1$ corresponds to $b_{\mathsf{KEM}} = 1$ in section 3.2).

A_2 attacks $\mathsf{MAC}_{\mathsf{M}_i}$ as follows (cf. section 3.3). At first, it generates $b \in \{0,1\}$ and a string K_{oracle} of length $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{OutLen}$ uniformly at random. Then, it runs the adversary A, playing the roles of the key generation oracle, decryption oracle, and interactive encryption oracle, substituting the pregenerated string K_{oracle} and the pregenerated bit b in step 3 of section 3.1. Whenever A submits a query $\mathfrak{K} \parallel \mathfrak{M} \parallel \mathfrak{C}$ to the decryption oracle, A_2 adds the pair $(\mathfrak{C}, \mathfrak{M})$ to its own output (A's final output bit $\widetilde{b}$ is ignored). Observe that

$$\left|\Pr_{\mathbf{G}_2}\left[\widetilde{b} = b\right] - \Pr_{\mathbf{G}_1}\left[\widetilde{b} = b\right]\right| \leq \mathsf{AdvForge}_{\mathsf{MAC}_{\mathsf{M}_i}, A_2}$$

($\mathbf{G}_2$ behaves differently from $\mathbf{G}_1$ only if a forgery has been produced).

A_3 attacks $\mathsf{STREAM}_{\mathsf{M}_i}$ as follows (cf. section 3.4). First, it generates $b \in \{0,1\}$ and a string K_{oracle} of length $\mathsf{KEM}_{\mathsf{M}_i}.\mathsf{OutLen}$ uniformly at random. Then, it runs the adversary A, playing the roles of the key generation oracle, decryption oracle, and interactive encryption oracle, substituting the pregenerated string K_{oracle} and the pregenerated bit b in step 3 of section 3.1. When A outputs its bit $\widetilde{b}$, A_3 outputs 1 if $\widetilde{b} = b$ and 0 otherwise. Observe that

$$\left|\Pr_{\mathbf{G}_3}\left[\widetilde{b} = b\right] - \Pr_{\mathbf{G}_2}\left[\widetilde{b} = b\right]\right| = \mathsf{Adv}_{\mathsf{STREAM}_{\mathsf{M}_i},A_3}$$

($\mathbf{G}_2$ corresponds to $b_{\mathsf{STREAM}} = 0$, $\mathbf{G}_3$ corresponds to $b_{\mathsf{STREAM}} = 1$ in section 3.4).

Finally observe that $\Pr_{\mathbf{G}_3}\left[\widetilde{b} = b\right] = \frac{1}{2}$.

From this we obtain the inequality

$$\begin{aligned}
\mathsf{AdvCCA}_{\mathsf{M}_i,A} &= 2 \cdot \left|\frac{1}{2} - \Pr_{\mathbf{G}_0}\left[\widetilde{b} = b\right]\right| \\
&= 2 \cdot \left|\sum_{1 \le x \le 3} \left(\Pr_{\mathbf{G}_x}\left[\widetilde{b} = b\right] - \Pr_{\mathbf{G}_{x-1}}\left[\widetilde{b} = b\right]\right)\right| \\
&\le 2 \cdot \left(\mathsf{AdvCCA}_{\mathsf{KEM}_{\mathsf{M}_i},A_1} + \mathsf{AdvForge}_{\mathsf{MAC}_{\mathsf{M}_i},A_2} + \mathsf{Adv}_{\mathsf{STREAM}_{\mathsf{M}_i},A_3}\right),
\end{aligned}$$

which concludes our security proof.