15 Authenticated Encryption & AEAD

to-do *Disclaimer: You're reading a rough first draft of this chapter.*

It can be helpful to think of encryption as providing a secure *logical* channel between two users who only have access to an insecure *physical* channel. Below are a few things that an attacker might do to the insecure physical channel:

- An attacker may **passively eavesdrop**; *i.e.*, simply observe the channel. A CPA-secure encryption scheme provides **confidentiality** and prevents the attacker from learning anything by eavesdropping.
- An attacker may **drop** messages sent along the channel, resulting in a denial of service. If the attacker can do this on the underlying physical channel, then it cannot be overcome through cryptography.
- An attacker may try to **modify** messages that are sent along the channel, by tampering with their ciphertexts. This sounds like what CCA-secure encryption protects against, right?
- An attacker may try to **inject** new messages into the channel. If successful, Bob might receive a message and mistake it for something that Alice meant to send. Does CCA security protect against this? If it is indeed possible to inject new messages into the channel, then an attacker can delete Alice's ciphertexts and replace them with their own. This would seem to fall under the category of "modifying" messages on the channel, so message-injection and message-modification are somewhat connected.
- An attacker may try to **replay** messages that were sent. For example, if Bob was convinced that a ciphertext c came from Alice, then an attacker can re-send the same c many times, and Bob may interpret this as Alice wanting to re-send the same plaintext many times. Does CCA security protect against this?

Although it might seem that CCA-secure encryption guarantees protection against many of these kinds of attacks, it does not!

To see why, consider the SPRP-based encryption scheme of Construction 9.3. We proved that this scheme has CCA security. However, it never raises any errors during decryption. *Every* ciphertext is interpreted as a valid encryption of *some* plaintext. An attacker can choose an arbitrary ciphertext, and when Bob decrypts it he might think Alice was trying to send some (presumably garbled) message. The only thing that CCA security guarantees is that **if** an attacker is able to make a ciphertext that decrypts without error, then it must decrypt to something that is unrelated to the contents of other ciphertexts.

In order to achieve protection against message-modification and message-injection on the secure channel, we need a stronger/better security definition. **Authenticated encryption (AE)** formalizes the extra property that *only* someone with the secret key can find ciphertexts that decrypt without error. For example, encrypt-then-MAC (Construction 10.9) already has this property.

In this chapter we will discuss authenticated encryption and a closely-related concept of encryption with **associated data (AD)**, which is designed to help prevent message-replay attacks. These two concepts are the "gold standard" for encryption.

15.1 Definitions

Authenticated Encryption

As with CPA and CCA flavors of security, we can define AE security in both a "left-vs-right" style or a "pseudorandom ciphertexts" style. Both are reasonable choices. To make life simpler we will only define the pseudorandom-ciphertexts-style of AE security in this chapter.

In CCA$ security, the attacker has access to the decryption algorithm (except for ciphertexts generated by the library itself). This captures the idea that the result of decrypting adversarially-generated ciphertexts cannot help distinguish the contents of other ciphertexts. For AE security, we want a stronger condition that $\text{Dec}(k, c) = \texttt{err}$ for every adversarially-generated ciphertext c. Using the same ideas used to define security for MACs, we express this requirement by saying that the attacker shouldn't be able to distinguish access to the "real" Dec algorithm, or one that always outputs `err`:

Definition 15.1 (AE) *Let Σ be an encryption scheme. We say that Σ has **authenticated encryption (AE) security** if $\mathcal{L}^{\Sigma}_{\text{ae\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{ae\$-rand}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{ae\$-real}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

CTXT($m \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\text{Enc}(k, m)$
  $\mathcal{S} := \mathcal{S} \cup \{c\}$
  return c

DECRYPT($c \in \Sigma.\mathcal{M}$):
  if $c \in \mathcal{S}$: return `err`
  return $\Sigma.\text{Dec}(k, c)$

$\mathcal{L}^{\Sigma}_{\text{ae\$-fake}}$

CTXT($m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.\mathcal{C}(|m|)$
  return c

DECRYPT($c \in \Sigma.\mathcal{M}$):
  return `err`

Discussion

The two libraries are different from each other in two major ways: whether the calling program sees real ciphertexts or random strings (that have nothing to do with the given plaintext), and whether the calling program sees the true result of decryption or an error

message. With these two differences, we are demanding that two conditions be true: the calling program can't tell whether it is seeing real or fake ciphertexts, it also cannot generate a ciphertext (other than the ones it has seen) that would cause Dec to output anything except `err`.

Whenever the calling program calls DECRYPT(c) for a ciphertext c that was produced by the library (in CTXT), both libraries will return `err` by construction. Importantly, the difference in the libraries is the behavior of DECRYPT on ciphertexts that were *not* generated by the library (*i.e.*, generated by the attacker).

Associated Data

AE provides a secure channel between Alice and Bob that is safe from message-modification and message-injection by the attacker (in addition to providing confidentiality). However, AE still does not protect from **replay** of messages. If Alice sends a ciphertext c to Bob, we know that Bob will decrypt c without error. The guarantee of AE security is that Bob can be sure that the message originated from Alice in this case. If an attacker re-sends the same c at a later time, Bob will likely interpret that as a sign that Alice wanted to say the same thing again, even though this was not Alice's intent. It is still true that Alice was the originator of the message, but just not at this time.

You may wonder how it is possible to prevent this sort of attack. If a ciphertext c is a valid ciphertext when Alice sends it, then it will *always* be a valid ciphertext, right? A clever way around this problem is for Alice to not only authenticate the ciphertext as coming from her, but to authenticate it also to a *specific context.* For example, suppose that Alice & Bob are exchanging encrypted messages, and the 5th ciphertext is c, sent by Alice. The main idea is to let Alice authenticate the fact that "I meant to send c as the 5th ciphertext in the conversation." If an attacker re-sends c later (*e.g.*, as the 11th ciphertext, a different context), Bob will attempt to authenticate the fact that "Alice meant to send c as the 11th ciphertext," and this authentication will fail.

What I have called "context" is called **associated data** in an encryption scheme. In order to support associated data, we modify the syntax of the encryption and decryption algorithms to take an additional argument d. The ciphertext $c = \mathsf{Enc}(k, d, m)$ is an encryption of m with associated data d. In an application, d could be a sequence number of a conversation, a hash of the entire conversation up to this point, an IP address + port number, etc. — basically, as much information as you can think of regarding this ciphertext's intended context. Decrypting c with the "correct" associated data d via $\mathsf{Dec}(k, d, c)$ should result in the correct plaintext m. Decrypting c with any other associated data should result in an error, since that reflects a mismatch between the sender's and receiver's contexts.

The intuitive security requirement for **authenticated encryption with associated data (AEAD)** is that an attacker who sees many encryptions c_i of chosen plaintexts, each authenticated to a particular associated data d_i, cannot generate a different (c^*, d^*) that decrypts successfully. The security definition rules out attempts to modify some c_i under the same d_i, or modify some d_i for the same c_i, or produce a completely new (c^*, d^*).

Definition 15.2 (AEAD) *Let Σ be an encryption scheme. We write $\Sigma.\mathcal{D}$ to denote the space of supported associated data signifiers ("contexts"). We say that Σ has **authenticated encryption with associated data (AEAD) security** if $\mathcal{L}^{\Sigma}_{\text{aead\$-real}} \approx \mathcal{L}^{\Sigma}_{\text{aead\$-rand}}$, where:*

$\mathcal{L}^{\Sigma}_{\text{aead\$-real}}$

$k \leftarrow \Sigma.\text{KeyGen}$
$\mathcal{S} := \emptyset$

CTXT($d \in \Sigma.\mathcal{D}, m \in \Sigma.\mathcal{M}$):
  $c := \Sigma.\text{Enc}(k, d, m)$
  $\mathcal{S} := \mathcal{S} \cup \{(d, c)\}$
  return c

DECRYPT($d \in \Sigma.\mathcal{D}, c \in \Sigma.\mathcal{M}$):
  if $(d, c) \in \mathcal{S}$: return err
  return $\Sigma.\text{Dec}(k, d, c)$

$\mathcal{L}^{\Sigma}_{\text{aead\$-fake}}$

CTXT($c \in \Sigma.\mathcal{D}, m \in \Sigma.\mathcal{M}$):
  $c \leftarrow \Sigma.C(|m|)$
  return c

DECRYPT($d \in \Sigma.\mathcal{D}, c \in \Sigma.\mathcal{M}$):
  return err

Discussion

One way to "authenticate a message to some context d" is to encrypt $m \| d$ instead of just m (in an AE scheme). This would indeed work! Including d as part of the plaintext would indeed authenticate it, but it would also *hide* it. The point of differentiating between plaintext and associated data is that we assume the associated data is *shared context* between both participants. In other words, we assume that the sender and receiver both already know the context d. Therefore, *hiding* d is overkill — only authentication is needed. By making a distinction between plaintext and associated data separately in AEAD, the **ciphertext length can depend only on the length of the plaintext**, and not depend on the length of the associated data.

The fact that associated data d is public is reflected in the fact that the calling program chooses it in the security definition.

"Standard" AE corresponds to the case where d is always empty: all ciphertexts are authenticated to the same context.

15.2 Achieving AE/AEAD

The Encrypt-then-MAC construction (Construction 10.9) has the property that the attacker cannot generate ciphertexts that decrypt correctly. Even though we introduced encrypt-then-MAC to achieve CCA security, it also achieves the stronger requirement of AE.

Claim 15.3 *If E has CPA security and M is a secure MAC, then EtM (Construction 10.9) has AE security.*

to-do *There is a slight mismatch here, since I defined AE/AEAD security as a "pseudorandom ciphertexts" style definition. So you actually need CPA\$+PRF instead of CPA+MAC. But CPA+MAC is enough for the left-vs-right style of AE/AEAD security.*

The security proof is essentially the same as the proof of CCA security (Claim 15.5). In that proof, there is a hybrid in which the DECRYPT subroutine always returns an error. Stopping the proof at that point would result in a proof of AE security.

Encrypt-then-MAC with Associated Data

Recall that the encrypt-then-MAC construction computes a MAC of the ciphertext. To incorporate associated data, we simply need to compute a MAC of the ciphertext along with the associated data.

Recall that most MACs in practice support variable-length inputs, but the length of the MAC tag does not depend on the length of the message. Hence, this new variant of encrypt-then-MAC has ciphertexts whose length does not depend on the length of the associated data.

Construction 15.4 (Enc+MAC+AD)

$\underline{\mathsf{Enc}((k_e, k_m), d, m)}$:
$c \leftarrow E.\mathsf{Enc}(k_e, m)$
$t := M.\mathsf{MAC}(k_m, d \| c)$
return (c, t)

$\underline{\mathsf{Dec}((k_e, k_m), d, (c, t))}$:
if $t \neq M.\mathsf{MAC}(k_m, d \| c)$:
  return err
return $E.\mathsf{Dec}(k_e, c)$

Claim 15.5 *If E has CPA security and M is a secure MAC, then Construction 15.4 has AEAD security, when the associated data has fixed length (*i.e.*, $\mathcal{D} = \{0,1\}^n$ for some fixed n).*

to-do *This construction is insecure for variable-length associated data. It is not terribly hard to fix this; see exercises.*

15.3 Carter-Wegman MACs

Suppose we construct an AE[AD] scheme using the encrypt-then-MAC paradigm. A good choice for the CPA-secure encryption scheme would be CBC mode; a good choice for the MAC scheme would be ECBC-MAC. Combining these two building blocks would result in an AE[AD] scheme that invokes the block cipher *twice* for each plaintext block — once for the CBC encryption (applied to the plaintext) and once more for the ECBC-MAC (applied to that ciphertext block).

Is it possible to achieve AE[AD] with less cost? In this section we will explore a more efficient technique for variable-length MACs, which requires only one multiplication operation per message block along with a single invocation of a block cipher.

Universal Hash Functions

The main building block in Carter-Wegman-style MACs is a kind of hash function called a **universal hash function** (UHF). While the name "universal hash function" sounds like it must be an incredibly strong primitive, a UHF actually gives a much weaker security guarantee than a collision-resistant or second-preimage-resistant hash function.

Recall that (x, x') is a **collision** under salt s if $x \neq x'$ and $H(s, x) = H(s, x')$. A universal hash function has the property that it is hard to find such a collision . . .

. . . when x and x' are chosen without knowledge of the salt,

. . . and when the attacker has *only one attempt at finding a collision* for a particular salt value.

These constraints are equivalent to choosing the salt *after* x and x' are chosen, and a collision should be negligibly likely under such circumstances.

The definition can be stated more formally:

Definition 15.6 (UHF) *A hash function H with set of salts $\mathcal{S}$ is called a **universal hash function (UHF)** if $\mathcal{L}^H_{\text{uhf-real}} \approx \mathcal{L}^H_{\text{uhf-fake}}$, where:*

$\mathcal{L}^H_{\text{uhf-real}}$

$\underline{\text{TEST}(x, x' \in \{0,1\}^*):}$
$s \leftarrow \mathcal{S}$
$b := \left[H(s,x) \stackrel{?}{=} H(s,x')\right]$
return (s, b)

$\mathcal{L}^H_{\text{uhf-fake}}$

$\underline{\text{TEST}(x, x' \in \{0,1\}^*):}$
$s \leftarrow \mathcal{S}$
return $(s, \texttt{false})$

This definition is similar in spirit to the formal definition of collision resistance (Definition 11.1). Just like that definition, this one is cumbersome to use in a security proof. When using a hash function, one typically does not explicitly check for collisions, but instead just proceeds as if there was no collision.

In the case of UHFs, there is a different and helpful way of thinking about security. Consider a "**blind collision-resistance**" game, where you try to find a collision under H without access to the salt, and even *without seeing the outputs of H!* It turns out that if H is a UHF, then it is hard to find collisions in such a game:

Claim 15.7 *If H is a UHF, then the following libraries are indistinguishable:*

$\mathcal{L}^H_{\text{bcr-real}}$

$s \leftarrow \mathcal{S}$
$H_{\text{inv}} :=$ empty assoc. array

$\underline{\text{TEST}(x \in \{0,1\}^*):}$
$y := H(s,x)$
if $H_{\text{inv}}[y]$ defined and $H_{\text{inv}}[y] \neq x$:
return $H_{\text{inv}}[y]$
$H_{\text{inv}}[y] := x$
return $\texttt{false}$

$\approx$

$\mathcal{L}^H_{\text{bcr-fake}}$

$\underline{\text{TEST}(x \in \{0,1\}^*):}$
return $\texttt{false}$

In these libraries, the calling program chooses inputs x to the UHF. The $\mathcal{L}_{\text{bcr-real}}$ library maintains a private record of all of the x values and their hashes, in the form of a reverse lookup table. $H_{\text{inv}}[y]$ will hold the value x that was hashed to result in y.

If the calling program calls $\text{TEST}(x)$ on a value that collides with a previous x', then $\mathcal{L}_{\text{bcr-real}}$ will respond with this x' value (the purpose of this is just to be helpful to security proofs that use these libraries); otherwise it will respond with $\texttt{false}$, giving no information about s or $H(s,x)$. The other library always responds with $\texttt{false}$. Hence, the two are indistinguishable only if finding collisions is hard.

to-do *Proof to come. It's not hard but tedious.*

Constructing UHFs using Polynomials

UHFs have much weaker security than other kinds of hashing, and they can in fact be constructed unconditionally. One of the mathematically simplest constructions has to do with polynomials.

Claim 15.8 *Let p be a prime and g be a nonzero polynomial with coefficients in $\mathbb{Z}_p$ and degree at most d. Then g has at most d zeroes from $\mathbb{Z}_p$.*

This observation leads to a simple UHF construction, whose idea is to interpret the string x as the coefficients of a polynomial, and evaluate that polynomial at point s (the salt of the UHF). In more detail, let p be a prime with $p > 2^\lambda$, and let the salt s be a uniformly chosen element of $\mathbb{Z}_p$. To compute the hash of x, first split x into λ-bit blocks, which will be convenient to index as $x_{d-1}\|x_{d-2}\|\ldots\|x_0$. Interpret each x_i as a number mod p. Then, the value of the hash $H(s,x)$ is:

$$s^d + x_{d-1}s^{d-1} + x_{d-2}s^{d-2} + \cdots + x_0 \pmod p$$

This is the result of evaluating a polynomial with coefficients $(1, x_{d-1}, x_{d-2}, \ldots, x_0)$ at the point s. A convenient way to evaluate this polynomial is by using **Horner's rule:**

$$\cdots s \cdot (s \cdot (s + x_{d-1}) + x_{d-2}) + x_{d-3} \cdots$$

The construction is described formally below.

Construction 15.9 (Poly-UHF)

$p = a\ prime\ > 2^\lambda$
$\mathcal{S} = \mathbb{Z}_p$

$\underline{H(s,x)}$:
write $x = x_{d-1}\|x_{d-2}\|\cdots\|x_0$,
where each $|x_i| = \lambda$
$y := 1$
for $i = d-1$ downto 0:
$y := s \cdot y + x_i \ \%\ p$
return y

Claim 15.10 *The Poly-UHF construction is a secure UHF.*

Proof It suffices to show that, for any $x \neq x'$, the probability that $H(s,x) = H(s,x')$ (taken over random choice of s) is negligible. Note that $H(s,x) = g(s)$, where g is a polynomial whose coefficients are $(1, x_{d-1}, \ldots, x_0)$, and $H(s,x') = g'(s)$, where g' is a similar polynomial derived from x'. Note that x and x' may be split into a different number of blocks, leading to different degrees (d and d') for the two polynomials.

In order to have a collision $H(s,x) = H(s,x')$, we must have

$$\begin{aligned} & g(s) \equiv_p g'(s) \\ \iff\ & g(s) - g'(s) \equiv_p 0 \end{aligned}$$

Note that the left-hand side in this equation is a polynomial of degree at most $d^* = \max\{d, d'\}$. Furthermore, that polynomial $g - g'$ is not the zero polynomial because g

and g' are different polynomials. Even if the original strings x and x' differ only in blocks of 0s, the resulting g and g' will be different polynomials because they include an extra leading coefficient of 1.

A collision happens if and only if s is chosen to be one of the roots of $g - g'$. From Claim 15.8, the polynomial has at most d^* roots, so the probability of choosing one of them is at most:

$$d^*/p \leqslant d^*/2^\lambda.$$

This probability is negligible since d^* is polynomial in λ (it is the number of blocks in a string that was written down by the attacker). ■

to-do *Fine print: this works but modular multiplication is not fast. If you want this to be fast, you would use a binary finite field. It is not so bad to describe what finite fields are, but doing so involves more polynomials. Then when you make polynomials whose coefficients are finite field elements, it runs the risk of feeling like polynomials over polynomials (because at some level it is). Not sure how I will eventually deal with this.*

Carter-Wegman UHF-based MAC

A UHF by itself is not a good MAC, even when its salt s is kept secret. This is because the security of a MAC must hold even when the attacker sees the function's outputs, but a UHF provides security (blind collision-resistance) only when the attacker does not see the UHF outputs.

The Carter-Wegman MAC technique augments a UHF by sending its output through a PRF, so the MAC of m is $F(k, H(s, m))$ where H is a UHF and F is a PRF.

Construction 15.11 (Carter-Wegman) *Let H be a UHF with n bits of output, and let F be a secure PRF with $in = n$. The Carter-Wegman construction combines them as follows:*

KeyGen:	$\text{MAC}\big((k, s), x\big)$:
$k \leftarrow \{0,1\}^\lambda$	$y := H(s, x)$
$s \leftarrow \mathcal{S}$	return $F(k, y)$
return (k, s)	

We will show that the Carter-Wegman construction is a secure PRF. Recall that this implies that the construction is also a secure MAC (Claim 10.4). Note that the Carter-Wegman construction also *uses* a PRF as a building block. However, it uses a PRF for short messages, to construct a PRF for arbitrary-length messages. Furthermore, it only calls the underlying PRF once, and all other computations involving the UHF are comparitively "cheap."

To understand the security of Carter-Wegman, we work backwards. The output $F(k, H(s, x))$ comes directly from a PRF. These outputs will look random as long as the inputs to the PRF are *distinct*. In this construction, the only way for PRF inputs to repeat is for there to be a collision in the UHF H. However, we have to be careful. We can only reason about the collision-resistance of H when its salt is secret and its outputs are hidden from the attacker. The salt is indeed hidden in this case (kept as part of the Carter-Wegman

key), but its outputs are being used as PRF inputs. Fortunately, the guarantee of a PRF is that its outputs appear *unrelated* to its inputs. In other words, the PRF outputs leak no information about the PRF inputs (H-outputs). Indeed, this appears to be a situation where the UHF outputs are hidden from the attacker, so we can argue that collisions in H are negligibly likely.

Claim 15.12 *If H is a secure UHF and F is a secure PRF, then the Carter-Wegman construction (Construction 15.11) is a secure PRF, and hence a secure MAC as well.*

Proof We will show that $\mathcal{L}^{\text{CW}}_{\text{prf-real}} \approx \mathcal{L}^{\text{CW}}_{\text{prf-rand}}$ using a standard hybrid technique.

$\mathcal{L}^{\text{CW}}_{\text{prf-real}}$

$k \leftarrow \{0,1\}^\lambda$
$s \leftarrow \mathcal{S}$

LOOKUP(x):
  $y := H(s,x)$
  return $F(k,y)$

The starting point is $\mathcal{L}^{\text{CW}}_{\text{prf-real}}$.

$T :=$ empty assoc. array
$s \leftarrow \mathcal{S}$

LOOKUP(x):
  $y := H(s,x)$
  if $T[y]$ undefined:
    $T[y] \leftarrow \{0,1\}^{out}$
  return $T[y]$

We have applied the security of F, by factoring out in terms of $\mathcal{L}^{F}_{\text{prf-real}}$, replacing it with $\mathcal{L}^{F}_{\text{prf-rand}}$, and inlining the result.

cache := empty assoc. array
$T :=$ empty assoc. array
$s \leftarrow \mathcal{S}$

LOOKUP(x):
  if *cache*[x] undefined:
    $y := H(s,x)$
    if $T[y]$ undefined:
      $T[y] \leftarrow \{0,1\}^{out}$
    cache[x] := $T[y]$
  return *cache*[x]

The LOOKUP subroutine has the property that if it is called on the same x twice, it will return the same result. It therefore does no harm to cache the answer every time. The second time LOOKUP is called on the same value x, the previous value is loaded from cache rather than recomputed. This change has no effect on the calling program.

```
cache := empty assoc. array
H_inv := empty assoc. array
T := empty assoc. array
s ← S

LOOKUP(x):
  if cache[x] undefined:
    y := H(s, x)
    if H_inv[y] defined:
      x' := H_inv[y]
      return cache[x']
    if T[y] undefined:
      T[y] ← {0,1}^out
    H_inv[y] := x
    cache[x] := T[y]
  return cache[x]
```

Note that if LOOKUP is first called on x' and then later on x, where $H(s,x) = H(s,x')$, LOOKUP will return the same result. We therefore modify the library to keep track of H-outputs and inputs. Whenever the library computes $y = H(s,x)$, it stores $H_{\text{inv}}[y] = x$. However, if $H_{\text{inv}}[y]$ already exists, it means that this x and $x' = H_{\text{inv}}[y]$ are a collision under H. In that case, the library directly returns whatever it previously returned on input x'. This change has no effect on the calling program.

```
cache := empty assoc. array
H_inv := empty assoc. array
T := empty assoc. array
s ← S

LOOKUP(x):
  if cache[x] undefined:
    y := H(s, x)
    if H_inv[y] defined:
      x' := H_inv[y]
      return cache[x']
    if H_inv[y] undefined:
      T[y] ← {0,1}^out
    H_inv[y] := x
    cache[x] := T[y]
  return cache[x]
```

In the previous hybrid, $T[y]$ is set at the same time $H_{\text{inv}}[y]$ is set — on the first call LOOKUP(x) such that $H(s,x) = y$. Therefore, it has no effect on the calling program to check whether $T[y]$ is defined or check whether $H_{\text{inv}}[y]$ is defined.

```
cache := empty assoc. array
H_inv := empty assoc. array
s ← S

LOOKUP(x):
  if cache[x] undefined:
    y := H(s, x)
    if H_inv[y] defined:
      x' := H_inv[y]
      return cache[x']
    if H_inv[y] undefined:
      cache[x] ← {0,1}^out
    H_inv[y] := x
  return cache[x]
```

Note that if $H_{\text{inv}}[y]$ is defined, then LOOKUP returns within that if-statement. The line $cache[x] := T[y]$ is therefore only executed in the case that $H_{\text{inv}}[y]$ was not initially defined. Instead of choosing $T[y]$ only to immediately assign it to $cache[x]$, we just assign directly to $cache[x]$. This change has no effect on the calling program, and it does away with the T associative array entirely.

The if-statements involving H_{inv} in this hybrid are checking whether x has collided with any previous x' under H. All of this logic, including the evaluation of H, can be factored out in terms of $\mathcal{L}^{H}_{\text{bcr-real}}$. At this point in the sequence of hybrids, the output of H is not needed, except to check whether a collision has been encountered (and if so, what the offending inputs are). Again, this change has no effect on the calling program. The result is:

```
cache := empty assoc. array

LOOKUP(x):
  if cache[x] undefined:
    if TEST(x) = x' ≠ false :
      return cache[x']
    else:
      cache[x] ← {0,1}^out
  return cache[x]
```

$\diamond$

$\mathcal{L}^{H}_{\text{bcr-real}}$

```
s ← S
H_inv := empty assoc. array

TEST(x):
  y := H(s, x)
  if H_inv[y] defined:
    return H_inv[y]
  H_inv[y] := x
  return false
```

The security of H is that we can swap $\mathcal{L}^{H}_{\text{bcr-real}}$ for $\mathcal{L}^{H}_{\text{bcr-fake}}$, with negligible effect on the calling program. Note that TEST algorithm in $\mathcal{L}_{\text{bcr-fake}}$ always returns `false`. This leads us to simply remove the "if MYTEST$(x) \neq$ `false`" clause, resulting in the following:

$\mathcal{L}^{\text{CW}}_{\text{prf-rand}}$

```
cache := empty assoc. array
LOOKUP(x):
  if cache[x] undefined:
    cache[x] ← {0,1}^out
  return cache[x]
```

Since this is exactly $\mathcal{L}^{\text{CW}}_{\text{prf-rand}}$, we are done. We have shown that $\mathcal{L}^{\text{CW}}_{\text{prf-rand}} \approx \mathcal{L}^{\text{CW}}_{\text{prf-rand}}$. ■

15.4 Galois Counter Mode for AEAD

The most common block cipher mode for AEAD is called **Galois Counter Mode (GCM)**. GCM is essentially an instance of encrypt-then-MAC, combining CTR mode for encryption and the polynomial-based Carter-Wegman MAC for authentication. GCM is relatively inexpensive since it requires only one call to the block cipher per plaintext block, plus one multiplication for each block of ciphertext + associated data.

Rather than using polynomials over $\mathbb{Z}_p$, GCM mode uses polynomials defined over a finite field with 2^λ elements. Such fields are often called "Galois fields," which leads to the name Galois counter mode.

to-do *More information about GCM might be nice. Again, would be nice to have a crash course in finite fields.*

Exercises

to-do *. . . more on the way . . .*

15.1. Suppose Enc-then-MAC+AD is instantiated with CBC mode and any secure MAC, as described in Construction 15.4. The scheme is secure for fixed-length associated data. Show that if variable-length associated data is allowed, then the scheme does **not** provide AEAD security.

Note: you are not attacking the MAC! Take advantage of the fact that $d\|c$ is ambiguous when the length of d is not fixed and publicly known.

15.2. Suggest a way to make Construction 15.4 secure for variable-length associated data. Prove that your construction is secure.

15.3. Show that if you know the salt s of the Poly-UHF construction (Construction 15.9), you can efficiently find a collision.

15.4. Show that if you are allowed to see only the output of Poly-UHF (*i.e.*, the salt remains hidden), on chosen inputs then you can compute the salt.