CSCI-GA.3210-001 MATH-GA.2170-001	**Introduction to Cryptography**	April 4, 2012

Lecture 11

Lecturer: Yevgeniy Dodis *Spring 2012*

Last time we defined almost universal hash functions, and showed how they are useful for message authentication. Recall, such family $\mathcal{H} = \{h_t : \{0,1\}^L \to \{0,1\}^\ell\}$ has the property that for all $m \neq m'$, $\Pr_t(h_t(m) = h_t(m')) \leq \varepsilon$, where ε is negligible in the secuirty parameters. We now give a variety of almost universal families $\mathcal{H}$. As will will see, this primitive is quite easy to construct, both information-theoretically and computationally. Then we proceed to study the resulting MACs, as well as several other ways to design MACs. Then we switch our attention to collision-resistant hash funstions.

1 Information-Theoretic Examples

Inner Product Construction. Let F be a finite field of size roughly 2^ℓ. In particular, $F = GF[2^\ell]$ is most convenint, but $F = \mathbb{Z}_p$ is also OK for $p \approx 2^\ell$. View the message $m \in \{0,1\}^L$ as n elements $m_1 \dots m_n$ of F, where $n \approx L/\ell$. For example, if $F = GF[2^\ell]$, we simply split the message into ℓ-bit chucks $m_1 \dots m_n$ and view each block m_i as an element of $GF[2^\ell]$.

The secret key t of h consists of n elements $a_1 \dots a_n$ of F. Thus, the length of t is (roughly) L, equal to the length of the message m. Now, define

$$h_{a_1 \dots a_n}(m_1, \dots, m_n) = \sum_{i=1}^{n} a_i m_i \qquad \text{(the operations are in } F\text{)}$$

Let us now examine the probability of a collision for any $m \neq u$. Let $z_i = m_i - u_i$. As $x \neq y$, at least one of the z_i is a non-zero element of F. By symmetry and for the ease of notation, let us assume that this is $z_1 \neq 0$. Now, in order for $h_{a_1 \dots a_n}(m_1, \dots, m_n) = h_{a_1 \dots a_n}(u_1, \dots, u_n)$, we must have

$$\sum_{i=1}^{n} a_i m_i = \sum_{i=1}^{n} a_i u_i \quad \Leftrightarrow \quad a_1 z_1 = -\sum_{i=2}^{n} a_i z_i \quad \Leftrightarrow \quad a_1 = -\left(\sum_{i=2}^{n} a_i z_i\right)/z_1$$

Now, what is the probability that a random field element a_1 is equal to the last expression (whatever that experssion is, notice that the choice of a_1 is *independent* of it)? Clearly, it is $1/|F| \approx 2^{-\ell}$. In particular, it is the optimal value $2^{-\ell}$ when $F = GF[2^\ell]$. Thus, this construction achieves optimal ε, but the key length of t is equal to L, which is too large. Instead, we would like the key to be $O(\ell)$, independent of the size L of the message!

Polynomial Construction. As before, let F be a finite field of size roughly 2^ℓ (either $\mathbb{Z}_p$, or, more conveniently, $GF[2^\ell]$ since it takes exactly ℓ bits to represent an element in this field). As before, view $m = m_1, \dots, m_n$ (i.e., $|m| = L \approx n\ell$), where each $m_i \in F$. Now,

however, we view $m_1 \dots m_n$ as n coefficients of a degree $(n-1)$ polynomial over F (see below). We will also select a random point $x \in F$ as the key to a function h_x in the hash family, defined as

$$h_x(m_1, \dots, m_n) = q_m(x) = \sum_{i=1}^{n} m_i \cdot x^{i-1}$$

where all the operations are done in F. Let's examine the probability of a collision between two distinct "polynomials" m and u. A collision here means

$$h_x(m) = h_x(u) \Longleftrightarrow q_m(x) = q_u(x) \Longleftrightarrow q_{m-u}(x) = 0 \Longleftrightarrow \sum_{i=1}^{n} (m_i - u_i) \cdot x^{i-1} = 0$$

where at least one $m_i - u_i \neq 0$, i.e. $q_{m-u}(\cdot)$ is a *non-zero* polynomial of degree at most $(n-1)$. It is a well known fact that any *non-zero* polynomial of degree d can have at most d roots in F. Since the point (our key) $x \in F$ was chosen at random, the probability that x is one of these at most $(n-1)$ roots of $q_{m-u}(\cdot)$ is at most $\frac{n-1}{|F|} \approx \frac{L}{2^\ell \ell}$, which is negligible.

Also, the key size is only ℓ bits, independent of the message length $L = n\ell$ (instead, the error depends on L). It turns out that one can achieve the best of both world — small key length and error probability close to $2^{-\ell}$. Concretely, one can acheive $|t| = O(\ell + \log N)$ and $\varepsilon = 2^{1-\ell}$. But we will not give this construction here.

2 Computational Examples (XOR-MAC, CBC-MAC, HMAC)

The next several examples use a PRF family $\mathcal{F} = \{f_t : \{0,1\}^\ell \to \{0,1\}^\ell\}$. Notice, we are slightly cheating here for 2 reasons. First, we are using "short-input" PRF f_t to build "long-input" *computationally* almost universal $\mathcal{H} = \{h_t\}$. This means that for any PPT attacker who outputs two messages $x' \neq x'$, the probability that $h_t(x) = h_t(x')$ is negligible:

$$\Pr(h_t(x) = h_t(x') \mid t \leftarrow \$, (x, x') \leftarrow A(1^k)) = \mathsf{negl}(k)$$

This is luckily enough for our purposes (i.e., the composition of "short" PRF with computationally almost universal $\mathcal{H}$ still yields "long" PRF). But the reason this comes up is that in the analysis of ε-universaility we will immediately replace f_t by a truly random function R. But this change means that the actual family we construct using $\mathcal{F}$ is only computationally almost universal.

Second, to build our "long-input" PRF, we will have to combine our h_t constructed using f_t with another *independently selected* PRF f_s, via $f_s(h_t(\cdot))$. As we will see, however, a simple general trick allows us to avoid making s and t independent. Namely, sacrifize 1 bit in ℓ, and always apply $f_s(1, \cdot)$ when constructing the hash function $h_t(\cdot)$, and use $f_s(0, h_t(\cdot))$ on the outer layer. Using the "random function paradigm", $f_s(0, \cdot)$ and $f_s(1, \cdot)$ indeed look like two independent random function. In fact, in specific cases will not even have to do that (see below), even though it is a very inexpensive "loss" anyway. Below, we describe the hash function without the domain separation "trick" above.

To summarize, the advantage of using a PRF in bulding $\mathcal{H}$ is saving on the key size + making the construction possibly very efficient (since "practical" PRF's are very cheap). As a downside, the error probabilities will be worse, and will depend on the "computational

closeness" of our PRF to a truly random function. Namely, to prove the universality of the hash function, we first assume that f_t is a truly random function (by the "random function paradigm"), and then prove the information-theoretic security as before.[1]

In all the examples below, we assume that: $m = m_1 \dots m_n$, where all $|m_i| = \ell'$, $L = \ell' n$, $\ell' \approx \ell$ (see below for details), *the number of blocks n is fixed*,[2] and t is a random key for our "base" PRF.

Using XOR Mode. Define

$$h_t(m_1 \dots m_n) = f_t(m_1, 1) \oplus f_t(m_2, 2) \oplus \cdots \oplus f_t(m_n, n)$$

(so that the input to the PRF is slightly longer: $\ell = \ell' + \log n$ bits long). Assuming f is a truly random function from ℓ to ℓ bits, and if $(u_1, \dots, u_n) \neq (m_1, \dots, m_n)$, say $m_i \neq u_i$, we get that

$$\begin{aligned} \Pr_f[f(m_1, 1) \oplus \cdots \oplus f(m_n, n) = f(u_1, 1) \oplus \cdots \oplus f(u_n, n)] &= \Pr_f[f(m_i, i) \oplus f(u_i, i) = \alpha] \\ &= \frac{1}{2^\ell} \end{aligned}$$

where α is some string independent[3] of $f(m_i, i) \oplus f(u_i, i)$, which in turn is random since $u_i \neq m_i$. As we indicated, to build a PRF out of it, we actually use

$$f_s(0,\ f_s(1, m_1, 1) \oplus \cdots \oplus f_s(1, m_n, n))$$

Using CBC Mode (CBC-MAC). We can view this construction as simply applying the CBC mode of operation with IV being 0^ℓ (a string of ℓ zeros), and outputting the *last block only* (remember, we do not need to "decrypt", only to "tag"):

$$h_t(m_1 \dots m_n) = f_t(m_n \oplus f_t(m_{n-1} \oplus \dots \oplus f_t(m_2 \oplus f_t(m_1)) \dots)) \tag{1}$$

The proof of (computational) universality of this $\mathcal{H}$ is a bit tricky, so we omit it. The main ideas are similar to what we have done earlier with CBC-encryption: intuitively, if $m \neq u$, say $m_i \neq u_i$, and f is a truly random function, the values $h_t(m)$ and $h_t(u)$ "diverge once and for all" w.h.p., starting at the i-th application of the f.

Lemma 1 *The function h_t defined in Equation (1) is computationally AU.*

In order to get a PRF out of this variant of CBC, it seems like we need to apply an indepependent PRF f_s to the h_t above. Indeed, this variant is called *encrypted CBC-MAC*, and we will again come back to it in Section 4:

$$\text{Encrypted-CBC}(m) = f_s(f_t(m_n \oplus f_t(m_{n-1} \oplus f_t(\dots f_t(m_2 \oplus f_t(m_1)) \dots))))$$

[1] Of course, the construction will be inefficient with a truly random function, but this does not concern us: the efficient PRF construction is what we are using, only the *proof* uses a random function.

[2] See Section 4 for more on this restrictive assumption.

[3] That is why we used the block number inside f.

However, by revisiting the analysis of Lemma 1 more carefully, we see that it actually shows more. Namely, even without applying an outside f_s to the above construction, we already get a PRF! More specifically, consider the following function known as the *CBC-MAC*, which is the same as the the function in Equation (1), except we renamed t to s:

$$\text{CBC-MAC}(m) = f_s(m_n \oplus f_s(m_{n-1} \oplus f_s(\ldots f_s(m_2 \oplus f_s(m_1))\ldots)))$$

Theorem 1 *CBC-MAC is a* PRF *on L bit inputs, if f_s is a* PRF *on ℓ-bit inputs.*

The proof of this result is slightly tedious, but follows the same structure as the proof of almost universality we mentioned. Essentially, on any two distinct messages $m \neq u$, at the first message block i where $m_i \neq u_i$, the current computation values of CBC-MAC(m) and CBC-MAC(u) will diverge *to random* once and for all. So all the output values are random and unrelated, meaning that we get a PRF.

CBC-MAC scheme is extremely popular, and is extensively used in practice. We also remark that we do not actually need $\mathcal{F}$ be a PRP family here (unlike for the encryption where we need to recover the message), any length-preserving PRF family is enough!

Using Cascade Mode (and HMAC). This next example builds a hash function $h_t : \{0,1\}^L \to \{0,1\}^\ell$ using a different PRF family $\{f_t\}$. Specifically, we do not care as much about the input size of f_t (but the larger the better), let use call it b, but care that the output size is ℓ and the key size k is at most ℓ. In practice, for example, one uses input of size $b = 512$, and output and key size both either $\ell = 128$ or $\ell = 160$. However, we will see that the construction works even for $b = 1$!

Now, split the message m into $m_1 \ldots m_n$, except now each chuck is of size b, so that $L = bn$. The initial key t to h_t is chosen at random from $\{0,1\}^\ell$, and then we inductively define values $x_0 \ldots x_n \in \{0,1\}^\ell$ as follows:

$$\begin{aligned} x_0 &= t \\ x_i &= f_{x_{i-1}}(m_i) \end{aligned}$$

Finally, the output $h_t(m_1 \ldots m_n) = x_n$. To describe it differently, $f_t(m_1)$ determines the PRF key x_1 to be used in the next round with input m_2, which in turn defines the PRF key x_2 to be used with the next input block m_3, and so on. This construction is called *cascade* or *Merkle-Damgard*. Notice, it really works for any input size $b \geq 1$, at the price of using L/b evaluations of the underlying PRF f (so larger b yields more efificency).

The intuition behind this construction is quite similar to the case of CBC-MAC, and is the following. First, since all x_i's are PRF outputs, they are computationally indistinguishable from random. Second, the very first block i separating two L-bit messages m and u would result in two computationally independent PRF keys x_i derived after the i-th call to f, and from this point on evaluating h on m and u looks totally independent. Of course, with small probability the "chains" might "converge" again, but by simple birthday argument this convergence is quite unlikely (we omit formal bounds here). In particular, we can argue

Lemma 2 *The cascade construction defines a computationally AU family of hash functions $\{h_t : \{0,1\}^L \to \{0,1\}^\ell\}$.*

In fact, the analysis shows more. Not only is $\mathcal{H}$ computational AU (meaning it can be composed with a *freshly keyed* PRF), but it is a PRF *by itself*! Intuitively, the above argument really said that the moment message diverge, everything stays random, and since any non-equal messages must diverge evetually, we get a PRF!

Theorem 2 *The cascade construction defines a* PRF *from L bits to ℓ bits.*

Why do we then care about Lemma 2 if we have Theorem 2? The reason will be clear in Section 4: it will have to do with our assumption that the message length L is fixed, which is a bit too restrictive in practice. But now let us try to see what the cascade construction gives us:

(1) It actually gives us a PRF by itself. In fact, it turns any PRF with large enough output size (and not too large key size) into an arbitrary (but FIXED) length PRF, no matter how small the original input size b is. In fact, when $b = 1$ the base PRF $f_t : \{0,1\} \to \{0,1\}^\ell$, where $|t| = \ell$, simply becomes a length doubling PRG $G : \{0,1\}^\ell \to \{0,1\}^{2\ell}$ via $G(t) = f_t(0) \circ f_t(1)$! Moreover, applying the cascade to this PRG G reduces the cascade construction to the GGM construction of PRFs from PRGs! (check it yourself!) In essense, using larger $b > 1$ lets us use a 2^b-ary tree instead of the binary tree, which brings the depth from $L = L/1$ to L/b (meaning that one need L/b evaluations of f to compute the cascade).

(2) It also gives us a computational AU family of functions. Thus, if we combine it with another PRF $g_s : \{0,1\}^\ell \to \{0,1\}^c$, we get a composed PRF as well. The only problem here is that we would like implement g_s using f_s, but the domains do not exactly match. g_s should take ℓ-bit inputs, and f_s takes b-bit inputs (and outputs ℓ-bit output). If $b \geq \ell$, which is the case in practice, this is not a problem: simply view the ℓ-bit input $h_t(m)$ to f_s as a b-bit input (i.e., pad it with $b - \ell$ zeros or something). Even otherwise, we can use Lemma 2 and build an ℓ-bit input PRF g_s out of f_s. But since this is never used, we'll assume $b \geq \ell$, and write f_s to mean an ℓ-bit input PRF (even though it can take potentially longer inputs). The resulting construction is called *NMAC*. More specifically, *NMAC* uses PRF f from b bits to ℓ bits (and key size ℓ), where in practice $b \geq \ell$, has two independent keys s and t, and essentially does $f_s(h_t(m))$, where h_t is the cascades mode applied to m. In practice, we do not like to have two keys though, so a variant of *NMAC* which uses only one key is called *HMAC*. A sound implementation of *HMAC* should have sacrificed one input bit and prepended 0 for s and 1 for t like we described before, but instead it does something more heuristic. More or less, it sets $s = t +$ constant, where the constant is heuristically chosen and fixed. Thus, in the future we will only concentrate on the theoretically-cound *NMAC* mode.

3 A Different XOR-MAC

We also mentioning another popular MAC paradigm which uses a PRF's and the XOR mode of operation. Namely, let $\mathcal{F}$ be the PRF family and $\mathcal{H}$ be a hash family from L to ℓ bits, whose properties will be given in a second. Rather than making the MAC output $f_s(h_t(m))$,

we now let it output $(nonce, f_s(nonce) \oplus h_t(m))$. The verification of $(nonce, v)$ checks that $v = f_s(nonce) \oplus h_t(m)$. Here *nonce* is the value that w.h.p. never repeats again, like a random string, or a counter (notice the similarity with encryption). In particular, this method is typically either randomized (*nonce* is random), or stateful (*nonce* is a counter), unlike our previous fully deterministic methods. Also, one has to either know or transmit the *nonce*. Finally, it is used only to make a MAC, and not a (more general) "long-input" PRF.

Still, what are the properties of $\mathcal{H}$ that make this method go through? As a simple attack, given a valid tag $(nonce, v)$ of m and a value a, the adversary can try to output a "forgery" $(nonce, v \oplus a)$ for some $m' \neq m$. It is easy to see that this will be successful if and only if $h_t(m) \oplus h_t(m') = a$. Since a, m, m' are arbitrary, at the very least we must have that for any $m \neq m'$, and any $a \in \{0,1\}^\ell$, we have

$$\Pr_t(h_t(m) \oplus h_t(m') = a) \leq \varepsilon$$

(where ε is negligible). Such families are called *ε-xor-universal* (or *almost XOR-universal*, or simply, *AXU*). Notice, regular ε-universality corresponds to $a = 0$ since $h_t(m) = h_t(m')$ iff $h_t(m) \oplus h_t(m') = 0$. Thus, a further disadvantage of this method is that it uses more restrictive classes of hash functions! However, the latter criticism is typically not a big deal, since most natural universal families are actually xor-universal. It turns out that xor-universality is sufficient:

Theorem 3 *$f_s(nonce) \oplus h_t(\cdot)$ defines a secure* MAC *whenever all the nonces are unique w.h.p., $\mathcal{F}$ is a* PRF *family and $\mathcal{H}$ is AXU.*

The most used xor-universal family comes from the XOR mode of the previous section (and uses PRF to build h_t):

$$h_t(m_1 \dots m_n) = f_t(m_1, 1) \oplus f_t(m_2, 2) \oplus \cdots \oplus f_t(m_n, n)$$

It is easy to see that our proof from the previous section in fact showed that $\mathcal{H}$ is AXU (check it). As in the previous section, we have to use the trick with prepending 0 and 1 to make the final MAC construction and use the same key:

$$\mathsf{Tag}_s(m) = (nonce,\ f_s(0, nonce) \oplus f_s(1, m_1, 1) \oplus \cdots \oplus f_s(1, m_n, n))$$

This is called the XOR-MAC. Naturally, it has a randomized or counter flavor depending on whether the *nonce* is random, or is a counter (in the later case the *nonce* need not be explicitly sent over).

But why use this method given its two disadvantages (only a MAC + slightly stronger assumption of h)? The point is that the security of $\mathcal{H}$ depends on the *output size* of the PRF, rather than the *input size* like we had in the $f_s(h_t(m))$ composition. Namely, if previously $h : \{0,1\}^L \to \{0,1\}^{\underline{\text{input length of}}\ f}$, now we have $h : \{0,1\}^L \to \{0,1\}^{\underline{\text{output length of}}\ f}$. And since it is much easier to extend the output of a PRF than it's input, we get that this XOR-MODE might yeild considerably better exact security in practice, especially if used with counters (so that one does not have to pay a birthday bound on *nonce* which depends on the input length of f). Overall, which MAC is better depends on a variety of parameters, with most constructions being incomporable (i.e., for different circumstances either one could be better).

4 Variable Length-Inputs

So far we made a convenient simplifying assumption that all the inputs to a MAC are of the same length L. In practice, this assumption is extremely inconvenient, and we would like to build *variable-length* MACs: namely, MACs which work for any input size in $\{0,1\}^*$.

First, let us revisit our constructions so far (whose key length is independent of the message length), and see which ones are right away secure variable-length MACs. As we will see, the answer is essentially "all except cascade and CBC-MAC".

- **Polynomial Construction.** Although the construction is insecure the way we stated it, — since the polynomial corresponding to $m_1 \dots m_n$ is the same as the one corresponding $0^\ell m_1 \dots m_n$, — it is very easy to fix. Simply prepend a fixed non-zero block a to each message. Thus, polynomial corresponding to m is now $q_m(x) = ax^n + m_n x^{n-1} + \ldots + m_1$. Now if m has n blocks, u has b blocks, and $m \neq u$, then the difference between $q_m(x)$ and $q_u(x)$ is a *non-zero* polynomial of degree at most $\max(n, b)$. Indeed, if $n = b$, then ax^n cancels, but the remaining polynomials won't cancel since $m \neq u$; else, say $n > b$, the term ax^n will not cancel (and, similarly, when $n < b$).

- **AU-based or AXU-based XOR Modes.** It is easy to see from the analyses of either mode that it can directly handle variable-length messages, since the block number is always included when evaluating f_t, so both the computational AU and the AXU properties still hold.

- **CBC-MAC and cascade.** It is not hard to see that either one of these modes is *not secure* when dealing with variable length messages. We give the reason for the cascade, leaving the (slightly more complicated) attack on the CBC-MAC as an exercise. The problem is the so called *extension attack*. Given a cascade of the message $m = m_1 \dots m_n$, we can easily forge a tag for any extended messages $m' = m_1 \dots m_n m_{n+1} \dots m_b$, where $b > n$. The reason is that the output of $x = cascade(m)$ is the PRF key we need to plug in to continue evaluating $x' = cascade(m')$ starting from the $(n+1)$-st block. Specifically, x' is simply the cascade of $m_{n+1} \dots m_b$ with the key x. Thus, if we learn x, we can compute the tag of any extended message by ourselves! Thus, Theorem 2 and Theorem 1 are not true for variable-length messages!

  On the positive side, it is easy to see that the above attack if the *only* attack on the cascade and the CBC-MAC. In particular, if we encode messages we tag in a *prefix-free* form, — namely, no encoded message is a prefix of another encoded message, — the cascade and the CBC-MAC are still secure.

- **Encrypted CBC-MAC and HMAC.** We claim that the encrypted versions of CBC-MAC and cascade (i.e., the NMAC) are still secure, even for variable-legth messages. To prove this, we only need to show that CBC-MAC and cascade remain *computational almost universal* even for variable length messages. In other words, we claim that Lemma 1 and Lemma 2 are still true!

  The argument is an extension of the one used to prove Lemma 1 and Lemma 2. There, we used the fact that ones the messages $m \neq u$ "diverge", they never meet again. As

we said, this analysis also works for prefix-free messages $m \neq u$. In the general case, say, when m is a prefix of u, we also have to argue that it is unlikely to have short "cycles". The argument is not very hard, and uses the same kind of birthday bounds we gave so far. So we will omit it from here.

To summarize, for variable-length messages, it is always safe to use HMAC and encrypted CBC-MAC. If one additionally knows (or can enforce) the messages to be prefix-free, then basic cascade and CBC-MAC are also secure.

5 CCA-secure and Authenticated Encryption

We briefly touch upon more advanced topics. First, recall from the homework the notion of CCA-security.

DEFINITION 1 SKE (Gen, E, D) is IND-secure against CCA attack iff $\forall \mathsf{PPT} B = (B_1, B_2)$,

$$\Pr[\, b = \tilde{b} \;\mid\; \begin{array}{l} s \leftarrow G(1^k); \\ (m_0, m_1, \beta) \leftarrow B_1^{E_s, D_s}(1^k); \\ b \leftarrow \{0,1\}; \\ \tilde{c} \leftarrow E_s(m_b); \\ \tilde{b} \leftarrow B_2^{E_s, D_s}(\tilde{c}, \beta); \,] \leq \frac{1}{2} + \mathsf{negl}(k) \end{array}$$

where B_2 cannot call D_s on input c. ♢

The definition is quite natural, and allowes the attacker to have oracle access to both the encryption and decryption functionality. Recall also from the homework that the following encryption scheme, based of a pseudorandom permutation g_s is CCA-secure: $E_s(m; r) = g_s(m \circ r)$, where $m \circ r$ is concatentation of m and randomness r. The decryption simply recovers $m \circ r$ by computing $g_s^{-1}(c)$ and "drops" r.

We will now give another way of constructing CCA-secure schemes, which is more general and has stronger security properties. In fact, recall that encryption schemes and message authentication *schemes* have roughly the same syntax, but different goals. Both take message m, and convert it into some other "enveloped" message c, by using the secret key s. And the recipient should recover m (or output invalid) from c, again using s. For encryption, we cared about privacy: no information about m should be contained in c, while for authentication we cared about authenticity: the recipeint should be sure that m came from the sender, irrespective of whether or not m is hidden. What if we combine these two goal? We get an extremely useful primitive called authenticated encryption.

In brief, authenticated encryption is again a triple of algorithms (G, E, D). G is the key generation algorithm, i.e. $G(1^k)$ produces the shared secret key (usually, a truly random sting of some length). As usual, $c \leftarrow E_s(m)$ produces the ciphertext, while $D_s(c) \rightarrow \tilde{m} \in M \cup \{\perp\}$, where M is the message space (say, $M = \{0,1\}^k$), and $\perp$ denotes "invalid". For privacy, we want (G, E, D) to be an IND-secure encryption scheme against CPA (chosen plaintext attack). However, now we also want (G, E, D) to be a secure message authentication scheme (strongly) existentially unforgeable against chosen message attack. Notice, here a successful forgery constitutes producing c s.t. $D_s(c) \neq \perp$, and c was

never returned by the "tagging" (i.e., "encryption")[4] oracles. Also notice the attacker does not necessarily need to "know" the message he is forging.

It is a good exercise (see homework) to see what is "wrong" with the CCA-secure encryption scheme $g_s(m \circ r)$ mentioned above, in the context of authenticated encryption. In fact, in the homework you will show that $(r, g_s(m \circ r))$ is in fact a secure authenticated encryption. Here, however, we want to mention several useful properties of authenticated encryption:

Theorem 4 *A secure symmetric-key authenticated encryption is always a* CCA*-secure symmetric-key encryption. Namely,* CPA*-security coupled with (strong) unforgeability implies* CCA*-security.*

The theorem is simple, so we only hint on the proof. Intuitively, if the scheme is unforgeable, then the decryption oracel is "useless": either the attacker already knows the answer (i.e., he got it from the encryption oracle before), or he gets $\perp$ (which is "useless") or he forged a valid ciphertext never returned by the encryption oracle.

Thus, instead of directly shoing CCA-security, it suffices to show CPA-security plus unforgeability. In fact, doing so will give a strictly stronger primitive of authenticated encryption!

Constructions of AE. This is an active area of research, and there are many interesting constructions. We already mentioned one above. Here we just metnion another: encrypt-then-mac. The secret key consists of two keys s for CPA-secure encryption and u for strongly unforgeable MAC. Then, $AE_{s,u}(m) = (c \leftarrow Enc_s(m), t \leftarrow Tag_u(c))$, while authenticated decryption first checks if t is a valid tag of c, and only then outputs $m = Dec_s(c)$. We leave it a simple exercise to argue the security of this scheme.

[4]Notice, in this scenario the "tagging" and "encryption" oracles are *the same*.