Problem 1: Fun with Digital Signatures

a. The vulnerability in this scheme is that if you have a signature for a message, you can obtain a signature for any message obtained by permuting the blocks of the original message.

Let \( M = m_1 \circ m_2 \circ m_3 \circ \cdots \circ m_n \) be the message that you want to forge on. Ask the signer for a signature on \( M' = m_2 \circ m_1 \circ m_3 \circ \cdots \circ m_n \). It will return a signature
\[
\Sigma(M') = \sigma(m_2) \circ \sigma(m_1) \circ \sigma(m_3) \circ \cdots \circ \sigma(m_n).
\]
Output the signature \( \Sigma(M) = \sigma(m_1) \circ \sigma(m_2) \circ \sigma(m_3) \circ \cdots \circ \sigma(m_n) \).

b. The vulnerability in this scheme is that if you have a signature for a message, you can obtain a signature for any message obtained by removing blocks from the end of the original message (as long as your indices are represented using the same number of bits as in the original message).

Let \( M \) be the message that you want to forge on. Choose the smallest \( n \) such that
\[
\lceil \log_2(n+1) + \frac{|M|}{n} \rceil \leq k.
\]
If \( \lceil \log_2(n+1) \rceil \neq \lceil \log_2(n+2) \rceil \), then fail. Otherwise, pad \( M \) with 0’s to create an \( M' \) such that \( |M'| \) is a multiple of \( k - \lceil \log_2(n+1) \rceil \). Ask the signer for a signature on \( M'' = M' \circ m_{n+1} \). It will return a signature
\[
\Sigma(M'') = \sigma(m_1) \circ \cdots \circ \sigma(m_n) \circ \sigma(m_{n+1}).
\]
Output the signature \( \Sigma(M) = \sigma(m_1) \circ \cdots \circ \sigma(m_n) \).

This attack will fail on messages of length \( |M'| \), where for the smallest \( n \) such that
\[
\lceil \log_2(n+1) + \frac{|M'|}{n} \rceil \leq k,
\]
it is the case that \( \lceil \log_2(n+1) \rceil \neq \lceil \log_2(n+2) \rceil \). The reason we had to fail in this case is that if we increase the length of the message, then the signer will use a different number of bits to represent the indices, and therefore will break the message into smaller blocks. That is, if the original message \( M \) was broken into \( m_1 \circ m_2 \circ \cdots \circ m_n \) by the signature scheme, and our message \( M' \) was broken into \( m'_1 \circ m'_2 \circ \cdots \circ m'_l \), it will not be the case that \( m_i = m'_i \). In fact, it may not even be the case that \( l = n + 1 \).

c. The vulnerability in this scheme is that if you have signatures for two messages, then you can obtain a signature for any message built from blocks in the original messages, as long as the blocks remain in the same position as they did in the original messages.

Let \( M \) be the message that you want to forge on. Choose \( n \) and break \( M = m_1 \circ \cdots \circ m_n \) as in Scheme 3. Choose \( m'_i \leftarrow \{0, 1\}^{k - \lceil 2\log_2(n+1) \rceil} \) for \( i \) from 1 to \( n \). Let
\[
M_1 = m'_1 \circ m_2 \circ \cdots \circ m_n, \quad \text{and} \quad M_2 = m_1 \circ m'_2 \circ \cdots \circ m'_n.
\]
Get the signatures for each of these messages, giving you

\[ \Sigma(M_1) = \sigma(m_1') \circ \sigma(m_2) \circ \cdots \circ \sigma(m_n), \quad \text{and} \]
\[ \Sigma(M_2) = \sigma(m_1) \circ \sigma(m_2') \circ \cdots \circ \sigma(m_n'). \]

Output the signature \( \Sigma(M) = \sigma(m_1) \circ \sigma(m_2) \circ \cdots \circ \sigma(m_n). \)

d. Choose a session id \( s \leftarrow \{0, 1\}^{k/4} \). Choose the smallest \( n \) such that

\[ \left\lceil 2 \log_2(n + 1) + \frac{|M|}{n} \right\rceil \leq \frac{3k}{4}. \]

(Assume that \( |M| \) is small enough and \( k \) is large enough that this is possible.) Then break \( M \) up into \( M = m_1 \circ \cdots \circ m_n \), where each \( m_i \) is such that \( |m_i| = \frac{3k}{4} - \left\lceil 2 \log_2(n + 1) \right\rceil \), and \( m_n \) is padded with 0s as necessary. Then let our signature be

\[ \Sigma(M) = (\sigma(s \circ n \circ 1 \circ m_1), \sigma(s \circ n \circ 2 \circ m_2), \ldots, \sigma(s \circ n \circ n \circ m_n)), \]

where each index is represented using \( \lceil \log_2(n + 1) \rceil \) bits.

This scheme ensures that you can’t mix and match pieces from different signatures to construct a new Frankenstein signature because, with high probability, the session ids won’t match.

**Problem 2: The Schnorr Signature**

a. First we notice that \( g \) has order \( q \), and thus exponents of \( g \) \( \pmod{p} \) can be implicitly treated as working \( \pmod{q} \). In other words, \( a \equiv b \pmod{q} \) implies that \( g^a \equiv g^b \pmod{p} \). With this in mind,

\[ g^a \equiv Ah^c \pmod{p} \]
\[ g^{r + xc} \equiv g^x(g^c)^x \pmod{p} \]
\[ g^{r + xc} \equiv g^{x + xc} \pmod{p} \]

b. **Proof.** Suppose we have an efficient adaptive adversary \( A \) satisfying the random oracle model, which—when given a public key \( PK \) for the Schnorr construction—is capable of producing with non-negligible probability a pair \((m^*, \sigma^*)\) such that \( \sigma^* \) is a valid signature for \( m^* \) under \( PK \), but \((m^*, \sigma^*)\) is not one of the polynomial number of signatures which it received in its queries to the signing oracle. Then we can construct \( B \) which takes as input:

- A \( k \)-bit prime number \( p^* = aq^* + 1 \), for prime \( q \) of length \( \Theta(k) \).
- An element \( g^* \) of \( \mathbb{Z}_p^* \) with order \( q^* \).
- An element \( h^* \) of \( \mathbb{Z}_p^* \) equal to \((g^*)^{x^*} \), where \( x^* \) is an unknown, randomly-distributed element of \( \mathbb{Z}_q \).

and which outputs \( x' = x^* \) with non-negligible probability.

This algorithm \( B \) constructs a Schnorr public key \( PK^* = (H^*, p^*, q^*, g^*, h^*) \), where \( H^* \) is a callback to \( B \) implementing an ideal hash function, and invokes \( A \) on \( PK^* \) and begins handling its signing and hashing queries. Since \( A \) can make only a polynomial number of such queries, \( B \) can keep a table of all its hash responses in order to make sure it always returns the same value on repeat requests. The table starts out empty, and every time \( A \) makes a query \( B \) picks a random number \( z \), records it in the table, and outputs \((g^*)^z \pmod{q} \). This distribution is random, and duplicates are handled properly, so \( A \) doesn’t notice anything is amiss.

HW 10-2
To handle signature queries, $B$ can use its control over $H$ to manipulate the values of $c$ so that its signatures verify, even though it doesn’t know $x^*$. It does this by picking a random value for $s$ and $c = (g^s)^2$, then computing $A \equiv g^s h^{-c} \pmod{p}$ so that the signature will verify. Before returning $(A, s)$ to $A$, it makes sure that this does not conflict with a previous entry in its hash function table, then fixing $c$ in the table so subsequent calls to $H$ will be consistent with the signature. If there is a conflict, it can just retry, either repeating a previous successful signature or guessing at another one, with proportional probabilities determined by the number of queries already made for the particular message. However, such conflicts will occur with negligible probability, since there can only be a polynomial number of queries already in the table. Note that $s$ should indeed be distributed uniformly at random, since it is the linear combination of the independently-random $r$ and $c$.

Finally, we can derive our result from $A$ by running it once until it gets a result (which will be correct with non-negligible probability), then rewinding its execution to right before it queried for the hash of the message and $A$-value that it returned a forgery for. We can assume without loss of generality that $A$ always tries to verify its candidate solution and immediately returns it if it is correct (otherwise some other $A'$ must exist which behaves this way\(^1\)), and furthermore we note that $c$ is independent of everything $A$ does, meaning that $A$ cannot hope to systematically manipulate $r$ to get a particular $c$. So $A$’s non-negligible probability of success must be dependent solely on its ability to choose $r$ before knowing $c$ and to then perform some computation given its chosen $r$ and a completely random $c$.

This finally leads us to the fact that $A$’s non-negligible success implies that, with non-negligible probability, it will have some non-negligible probability of succeeding again in the situation we rewind to, because we can assume (1) that it has chosen $m$ and $r$ to maximize its chances of success given its prior queries, (2) that situations in which it has gathered enough information to have a non-negligible success rate contribute non-negligibly to its overall success rate, and (3) that if it does indeed succeed again with the same $m$ and $r$ but with a different $c$, then it will then return with a new forgery with the same $A$ but a different $s$. So given the non-negligible probability of $A$ succeeding on the first run, and the non-negligible probability of it having a non-negligible probability of succeeding on the second run, there is yet a smaller (but still non-negligible) probability that it will succeed in both runs. And if it does, then we can use the distinct $s_0$, $s_1$, $c_0$, and $c_1$ to efficiently compute $x$ as stated. We can then return $x$, which as shown will be a correct discrete log of $h$ with non-negligible probability. \(\square\)

**Problem 3: Damaging a CCA-Secure Cryptosystem**

a. Proof. We prove this using a reduction. In this case, we assume there exists an algorithm $A$ such that

$$\Pr[(pk, sk) \leftarrow G(1^k); (m_0, m_1, state) \leftarrow A^{D(sk,:)}(pk); b \leftarrow \{0, 1\};$$

$$c \leftarrow E(pk, m_b) : A^{D'(sk,:)}(state, c) = b] = \frac{1}{2} + \epsilon(k)$$

for some non-negligible $\epsilon(\cdot)$. We will use these algorithms to construct an adversary $B$ that breaks the CCA-1 security of $(G_1, E_1, D_1)$. This algorithms two phases run as follows:

- On input $pk$, $B$ will invoke $A$ on input $pk$. Any time $A$ asks for the decryption of a ciphertext $(c_i, b)$, $B$ will query $D_1$ on input $c_i$ to obtain a decryption, which it then gives

---

\(^1\)This is a subtle point. In many cases, we cannot make restrictions on the adversary because we may accidentally not include some way the adversary can attack our scheme. However, this restriction is not on what the adversary does in *its attack*. It is clear that, for every adversary that does not check its answer, there exists an adversary that checks its answer but is otherwise identical, and they will be correct with exactly the same probability.
Problem 4: The Cramer-Shoup Cryptosystem

a. We can construct this adversary as follows: on input \( PK \), \( \mathcal{A} \) is allowed to spend its first interactive phase as many queries as it wants, although none are necessary. In the end, \( \mathcal{A} \) will output two messages \( m_0, m_1 \) chosen at random from the message space (as well as the necessary state information \( s \)). Upon receiving \( m_0, m_1, s \) and a ciphertext

\[
C = (g^r, h^r, A^r \cdot m_b, B^r)
\]

for \( b \in \{0, 1\} \), \( \mathcal{A} \) will pick a random value \( r' \leftarrow \mathbb{Z}_q \) and compute

\[
C' = (g^r \cdot g^{r'}, h^r \cdot h^{r'}, A^r \cdot m_b \cdot A'^r, B^r \cdot B'^r).
\]

Since this is re-randomized, it will be a different ciphertext and so the decryption oracle will decrypt it properly and output the underlying plaintext \( m_b \). \( \mathcal{A} \) can then simply check if \( m_0 = m_b \) and output 0 if and only if this equality holds.

Because the ciphertext \( C' \) is a valid ciphertext and distinct from \( C \), the decryption oracle will always decrypt it correctly and return the message \( m_b \). This means \( \mathcal{A} \) succeeds with probability 1 and thus breaks the CCA-2 security of the cryptosystem.

b. A well-formed ciphertext will be of the form

\[
C = (g^r, h^r, A^r \cdot m_s, (BC^3)^r)
\]

HW 10-4
where
\[ \beta = H(g^r, h^r, A^r \cdot m). \]

If we write this as a tuple of the form \((R, S, P, T)\), we see that
\[ T = (BC^\beta)^r = ((g^a h^b)(g^\beta w h^\beta))^{r^k} = (g^{a+\beta w} h^{b+\beta z})^r = (g^r)^{a+\beta w} (h^r)^{b+\beta z} = R^{a+\beta w} S^{b+\beta z}, \]

so that the cryptosystem really is correct.

**Problem 5: Adaptive Security**

a. Define \((G^*, E^*, D^*)\) by:
\[
G^*(1^k) = G(1^k) \\
E^*(pk, m) = \begin{cases} 0^k & \text{if the first } k \text{ bits of } pk \text{ are equal to } m \\ E(pk, m) & \text{otherwise} \end{cases} \\
D^*(sk, c) = \begin{cases} \text{first } k \text{ bits of } pk & \text{if } c = 0^k \\ D(sk, c) & \text{otherwise} \end{cases}
\]

This cryptosystem satisfies Definition 4, but not Definition 3. First, we note that any secure cryptosystem must satisfy that the range of \(G(1^k)\) has larger-than-polynomial size in \(k\), as otherwise an adversary could enumerate them all and decrypt any message in polynomial time. This means that a non-adaptive adversary Eve would have only a negligible chance of seeing \(E^*(pk, pk)\), and thus our contrived cryptosystem is still non-adaptively semantically secure.

On the other hand, an adaptive adversary can output the first \(k\) bits of \(pk\) as their chosen message, ensuring that they will always see \(0^k\) when looking at a real ciphertext. Thus, our contrived cryptosystem is not adaptively semantically secure.

b. Definition 3 does imply Definition 4.

*Proof by contradiction.* Assume for the sake of contradiction that there exists a cryptosystem \((G, E, D)\) that satisfies Definition 3, but does not satisfy Definition 4. Thus, for every \(Sim\), there must exist some poly-length message sequence \(M\) and some adversary \(A\) which can succeed in the experiment in Definition 3 with non-negligible advantage. Let us build an adversary \(B\) which breaks the adaptive semantic security of the cryptosystem.

Let us fix a particular simulator \(Sim\). Assume that \(B\) has \(m_k\) hardcoded into its description when run on \(1^k\). \(B\), in the first round, return \(m_k\) as its chosen message. Once it gets the encryption \(c\), \(B\) runs \(A\) with input \(c_b\) and output whatever it outputs. Since we are attempting to distinguish from the same simulator as \(A\), and the ciphertext we give to \(A\) is the same

\[2\text{In the case that } E(pk, m) = 0^k \text{ for some other } m, \text{ we can have } E^*(pk, m) \text{ output } E(pk, pk) \text{ instead. We assume for simplicity that no such } m \text{ exists.} \]

\[3\text{WLOG we assume that } |pk| \geq k; \text{ otherwise, we could simply pad } pk \text{ with 0's.} \]
distribution as that given in the Real distribution of Definition 3, we will be correct with the same probability as $A$. This breaks Definition 4, and thus we must have that Definition 3 implies Definition 4.