Cryptography and Security — Final Exam
Solution

Serge Vaudenay

29.1.2018

- duration: 3h
- no documents allowed, except one 2-sided sheet of handwritten notes
- a pocket calculator is allowed
- communication devices are not allowed
- the exam invigilators will **<u>not</u>** answer any technical question during the exam
- readability and style of writing will be part of the grade

The exam grade follows a linear scale in which each question has the same weight.

1 Collision on Sponge-Based Hash Functions

The sponge is a construction for cryptographic hashing. It uses a cryptographic permutation $f : \{0,1\}^b \to \{0,1\}^b$. To instantiate the sponge with f, we have to pick the parameters r (called the "rate") and c (called the "capacity"), such that $r + c = b$. The sponge also allows to set an arbitrary length h of the output. We fix f, c and h and set $H = \mathsf{Sponge}[f, c, h](M)$ defined as follows. We first define $\mathsf{pad}(a) = 10^{r-(a+2 \bmod r)}1$ for an integer a (i.e. the bitstring consisting of one bit 1 followed with $r - (a + 2 \bmod r)$ bits 0, then one bit 1: this is not a power of 10). Here, $|m|$ denotes the length of the bitstring m. We use the notation $\|$ for the concatenation of bitstrings. To compute the hash $H(m)$ of a message m (specified as a bitstring), we use the following pseudocode:

```
Input: $m$
 1: $M \leftarrow m\|\mathsf{pad}(|m|)$ {$\mathsf{pad}(|m|) = 10^{r-(|m|+2 \bmod r)}1$}
 2: $\ell \leftarrow |M|/r$
 3: split $M = M_1\|M_2\|\cdots\|M_\ell$ {split $M$ into $\ell$ blocks of $r$ bits $M_1, \ldots, M_\ell$}
 4: $S \leftarrow 0^b$
 5: for $i = 1$ to $\ell$ do
 6:     $S \leftarrow S \oplus (M_i\|0^c)$
 7:     $S \leftarrow f(S)$
 8: end for
 9: $Z \leftarrow$ empty string
10: while $|Z| < h$ do
11:     $Z \leftarrow Z\|\mathsf{left}_r(S)$
12:     $S \leftarrow f(S)$
13: end while
14: return $\mathsf{left}_h(Z)$
```

where $\mathsf{left}_r(Z)$ returns the r leftmost bits of a binary string Z. (We similarly define $\mathsf{right}_c(Z)$ so that $Z = \mathsf{left}_r(Z)\|\mathsf{right}_c(Z)$ whenever Z is of b bits.) We see that the rate impacts performance, the bigger it is, the less times we need to call f to hash a message. Now we investigate how the capacity influences the security of a sponge-based hash function.

Q.1 Say why it is called a sponge and in which well-known algorithm is this construction used.

The loop on step 5–8 is called the absorbing *phase, as it absorbs the M_i blocks. The loop on step 10–13 is called the* squeezing *phase, as we extract output from the state. This is used in the SHA-3 hash standard (the Keccak hash function).*

Q.2 Briefly describe a generic collision-search attack on the hash function H. What is its time and memory complexity? (It is convenient to measure the time complexity in computations which are equivalent to one call of f.)

To find a collision for a hash function $H : \{0,1\}^* \to \{0,1\}^h$, we select a constant θ and we pick $N = \theta \cdot 2^{h/2}$ messages $m_1, \ldots, m_N$. For each $i = 1, \ldots, N$ we compute $T_i = H(m_i)$ and store (T_i, m_i) in a hash table until there are two records (T, m) and (T, m') with colliding hashes. We output m, m'. The probability of success of this attack is

$$p \approx 1 - \left(1 - 2^{-h}\right)^{\frac{N^2}{2}} \approx 1 - e^{-\frac{\theta^2}{2}}$$

The time and memory complexity are both $\mathcal{O}(2^{h/2})$.

Q.3 Prove that given two randomly chosen strings $x \neq x'$ such that $|x| = |x'| = b$, the probability that $\mathsf{right}_c(f(x)) = \mathsf{right}_c(f(x'))$ is about 2^{-c}. (To make this estimation, you can assume that f is a "random permutation".)

If we assume that f behaves "randomly", using the law of total probability, we have that

$$\begin{aligned}
&\Pr\left[\mathsf{right}_c(f(x)) = \mathsf{right}_c(f(x'))\right] \\
&= \sum_{z \in \{0,1\}^c} \Pr\left[\mathsf{right}_c(f(x)) = z \mid \mathsf{right}_c(f(x')) = z\right] \cdot \Pr\left[\mathsf{right}_c(f(x')) = z\right] \\
&= \sum_{z \in \{0,1\}^c} \frac{2^r - 1}{2^b - 1} \cdot \frac{2^r}{2^b} \\
&= \frac{2^r - 1}{2^b - 1} \approx \frac{1}{2^c}.
\end{aligned}$$

The probabilities that an image under f has the rightmost c bits equal to z are derived by counting how many possible values of $f(x)$ (or $f(x')$) end with z versus how many possible values we have in total.

Q.4 Describe an algorithm that will find two $x \neq x'$ such that $|x| = |x'| = b$ and $\mathsf{right}_c(f(x)) = \mathsf{right}_c(f(x'))$. What is its time and memory complexity?

We can define $F(x) = \mathsf{right}_c(f(x))$ and mount the generic collision search on F from Q.2. As the output size of F is c bits, we run the attack with $\theta \cdot 2^{c/2}$ messages, so the time and memory complexities will both be $\mathcal{O}(2^{c/2})$.

Q.5 Assume that $h > c$. Describe a collision-search attack on the hash function $H = \mathsf{Sponge}[f, c, h](M)$ that is more efficient than the generic collision search from Q.2. What is its time and memory complexity?
HINT: You can assume that after processing $\lceil c/2r \rceil$ randomly chosen message blocks, the state S behaves as $f(x)$ with a random input x.

The idea of the attack is to find a collision in the rightmost c bits of the state S and then force a collision on the rest of the state using the message block.
Using the hint and the previous question, we pick $\theta \cdot 2^{c/2}$ random strings m_i of at least $r \cdot \lceil c/2r \rceil$ bits. For each $i = 1, \ldots, \theta \cdot 2^{c/2}$, we parse m_i into blocks (without padding!) and partially evaluate the sponge to obtain the final state S_i, compute and store $(\mathsf{left}_r(S_i), \mathsf{right}_c(S_i), m_i)$ in a hash table until there are two entries (L, R, m) and (L', R, m') with colliding R.
We output $\bar{m} = m \| L$ and $\bar{m}' = m' \| L'$ so that the final state becomes $f(0^r \| R)$ for both messages. As both messages have the same length, they will be subject to the same padding so will produce the same states. We will have $H(\bar{m}) = H(\bar{m}')$, because the internal state of the sponge after processing all-but-the-last blocks will share the rightmost c bits for both $\bar{m}$ and $\bar{m}'$. The complexity will be $O(2^{c/2})$, which is less than the complexity of the generic collision search because $h > c$.

2 LPN with Extreme Noise Parameters

We consider a set of parameters consisting of an integer k and a probability τ. We assume that a secret vector $s \in \{0,1\}^k$ is initially selected with a uniform distribution. We define a source $\mathcal{S}$ which generates some random pairs (v, b) as follows: first, the source picks a random vector $v \in \{0,1\}^k$ with a uniform distribution and (independently) a random bit e such that $\Pr[e = 1] = \tau$. Then, the source produces $b = \langle v, s\rangle \oplus e$ where we use the notation $\langle ., .\rangle$ for the dot product in $\mathbf{Z}_2^k$ defined by

$$\langle v, s\rangle = (v_1 s_1 + \cdots + v_k s_k) \bmod 2$$

and we use the notation $\oplus$ for the addition in $\mathbf{Z}_2$. The components of v and s are the v_i and s_i for $i = 1, \ldots, k$, respectively. The output (v, b) of the source $\mathcal{S}$ is called a *sample*. Upon a query to $\mathcal{S}$, the source produces a sample. By making a query, we get only the freshly generated sample (v, b) but we see neither the secret s nor the noise bit e.

The *Learning Parity with Noise problem* (LPN) consists of designing an algorithm $\mathcal{A}^{\mathcal{S}}(k, \tau)$ which recovers the secret vector s by only querying the source $\mathcal{S}$ and knowing the parameters k and τ, with a "large enough" probability of success (e.g. at least 50%):

$$\Pr[\mathcal{A}^{\mathcal{S}}(k, \tau) \to s] \geq \frac{1}{2}$$

We assume that τ is a function of k and we measure the *complexity* of $\mathcal{A}$ with a *time* complexity and a *query* complexity. Both are asymptotic in terms of k. The time complexity is the asymptotic expected complexity in number of binary operations. The query complexity is the number of queries to the source. The LPN problem is often used in cryptography with τ between $\frac{\mathsf{cte}}{\sqrt{k}}$ and $\frac{1}{2} - \frac{\mathsf{cte}}{k}$, for a constant cte, as it is believed to be hard (even with the help of quantum computers!). In this exercise, we investigate values of τ which are not in this range.

Q.1 For $\tau = 0$, prove that the LPN problem is easy to solve: there exists a polynomially bounded (in terms of k) algorithm to solve it. Briefly describe this algorithm and its complexity (time and query).

If $\tau = 0$, each sample gives a random $\mathbf{Z}_2$-linear equation which is satisfied by s. Using a bit more than k samples, it is likely that we find k linearly independent equations, so obtain a linear system which uniquely characterizes s. This is a system of k independent linear equations in k unknowns. We solve it by Gaussian elimination. Straightforward implementations have complexity $\mathcal{O}(k^3)$. The number of samples is $\mathcal{O}(k)$.

Q.2 For $\tau = \frac{1}{2}$, prove that the LPN problem is impossible to solve.
HINT: first show that the source is uniform whatever s.

For $\tau = \frac{1}{2}$, we know from the construction that (v, e) is uniformly distributed in $\mathbf{Z}_2^{k+1}$. For any bit β, we have $\Pr[b = \beta|v] = \Pr[e = \langle v, s\rangle \oplus \beta|v] = \Pr[e = \langle v, s\rangle \oplus \beta] = \frac{1}{2}$. So, b is independent from v and uniformly distributed. Hence, $\mathcal{S}$ is a uniform source. Because it is uniform, it is independent from s.
An arbitrary algorithm $\mathcal{A}$ fed with samples from the source will produce an output with a given distribution. If the secret s is changed into some s', the distribution of the source is exactly the same, so $\mathcal{A}$ will produce an output with same distribution. Hence, the output of $\mathcal{A}$ is independent from s. Since s is uniformly distributed, the probability that the output of $\mathcal{A}$ equals s is 2^{-k}. Therefore, it cannot succeed except with the tiny probability of 2^{-k}.

Q.3 For $\tau = \mathcal{O}(\frac{1}{k})$, we show in the next questions that the LPN problem is still easy to solve.

Q.3a Prove that we can easily get k samples (v_i, b_i), $i = 1, \ldots, k$ such that all v_i are linearly independent. Briefly describe this algorithm with a pseudocode and its complexity (time and query).

Assuming that we have $n < k$ linearly independent v vectors $v_1, \ldots, v_n$, we can get fresh samples (v, b) until we have one v which is linearly independent from $v_1, \ldots, v_n$. We can denote it v_{n+1}. The previous ones $v_1, \ldots, v_n$ span a vector space of dimension n, so of 2^n elements. A new one v is dependent with probability $2^n/2^k \leq \frac{1}{2}$. So, with a constant number of trials $\mathcal{O}(1)$, we can find an independent one v_{n+1}. The algorithm starts with $n = 0$ and iterates the above principle. Clearly, it needs $\mathcal{O}(k)$ samples. Instead of testing linear independence at every step, we can directly generate these samples then extract k linearly independent vectors from the pool with complexity $\mathcal{O}(k^3)$, by a Gaussian algorithm.

Q.3b We denote $b_i = \langle v_i, s\rangle \oplus e_i$. Prove that the number X of $i \in \{1, \ldots, k\}$ such that $e_i = 1$ is a random variable which has an expected value equal to $k\tau$ and a variance of $k\tau(1 - \tau)$.

This number X of i is simply $X = e_1 + \cdots + e_k$ (with addition over $\mathbf{Z}$). Using linearity, we have $E(X) = k \cdot E(e_i) = k\tau$. Since all e_i are linearly independent, we can compute the variance $V(X) = k \cdot V(e_i)$. We have

$$V(e_i) = E(e_i^2) - E(e_i)^2 = E(e_i) - E(e_i)^2 = \tau - \tau^2 = \tau(1 - \tau)$$

because $e_i^2 = e_i$. The standard deviation is $\sqrt{V(X)} = \sqrt{k\tau(1 - \tau)}$. This is a binomial distribution.

Q.3c By using the previous questions, prove that for $\tau = \mathcal{O}(\frac{1}{k})$ we can recover s from the obtained samples. Briefly describe an algorithm with a pseudocode and its complexity (time and query).

HINT: there may be an exhaustive search on a tiny space.

HINT^2: $\Pr[X > k\tau] \leq 37\%$.

Let $c = k\tau$. We know that $c = \mathcal{O}(1)$. Let $e = (e_1, \ldots, e_k)$. This vector has a weight around $k\tau = c$, with standard deviation less than $\sqrt{c}$. We take an arbitrary constant $u > 0$. The vector e has weight bounded by $w = c + u\sqrt{c}$ with probability greater than $\frac{1}{2}$. (More precisely, with $u = 0$, we have $w = c$ and the probability that the weight exceed w is bounded by 37%, due to the Chernoff-Hoeffding bound.) The set of all e with weight bounded by w has a cardinality bounded by k^w. This is polynomially bounded. We can do an exhaustive search on this set, so essentially guess the vector e. If we guess e, we solve s from a system of linear equations. Once s is obtained, we can check consistency with more samples. So, we can recover s with a polynomially bounded algorithm. Our algorithm works as follows:

```
1:  collect $\mathcal{O}(k)$ samples
2:  extract $(v_1, b_1), \ldots, (v_k, b_k)$ such that $v_1, \ldots, v_k$ are linearly independent
3:  collect $k$ new samples $(v'_i, b'_1), \ldots, (v'_k, b'_k)$
4:  for all $e \in \{0,1\}^k$ with weight bounded by $w$ do
5:      solve the system $b_i \oplus e_i = \langle v_i, \bar{s} \rangle$ in $\bar{s}$
6:      count $w'$: how many $i$ are such that $b'_i \oplus \langle v'_i, \bar{s} \rangle = 1$
7:      if $w' \leq w$ then
8:          stop and yield $\bar{s}$
9:      end if
10: end for
```

Here, consistency is checked by measuring that with k independent samples, the weight w' of the error vector is also bounded by w.

3 Cross-Authentication based on Short Authenticated Strings

The following exercise is inspired from SAS-Based Authenticated Key Agreement *by Pasini and Vaudenay, published in the proceedings of PKC 2006.*

Let r and k be two integers.

We consider a commitment scheme commit with which we commit to (m_A, K) (where m_A is a message and K is a hash key) by picking some coins ρ and producing $c = \mathsf{commit}(m_A, K; \rho)$ and we open the commitment by releasing (m_A, K, ρ).

We also consider a strongly universal hash function family h_K from the set of all finite bitstrings to $\{0,1\}^r$, defined by a key $K \in \{0,1\}^k$. This is such that for any fixed m, m' such that $m \neq m'$, the pair $(h_K(m), h_K(m'))$ is uniformly distributed in $(\{0,1\}^r)^2$ when $K \in \{0,1\}^k$ is uniformly distributed.

We define the following SAS-based message cross-authentication protocol: if Alice wants to send the message m_A and Bob wants to send the message m_B, they run the following protocol. Communications with the $\mathsf{authenticate}$ tags are authenticated (i.e. sent through a special authenticated channel) in the sense that an adversary cannot create or modify the message. Other communications are done through an insecure channel which may be corrupted by an adversary. For this reason, we put a hat on notations to designate received values. For instance, if Alice sends K to Bob through an insecure channel, Bob receives $\hat{K}$ which is equal to K only if the adversary lets the message unchanged.

$$\begin{array}{rcl}
\textbf{Alice} & & \textbf{Bob}\\
\textbf{input}: m_A & & \textbf{input}: m_B\\
\text{pick } K \in_U \{0,1\}^k & & \text{pick } R \in_U \{0,1\}^r\\
\text{pick } \rho & & \\
c \leftarrow \mathsf{commit}(m_A, K; \rho) & \xrightarrow{m_A, c} & \\
& \xleftarrow{m_B, R} & \\
& \xrightarrow{K, \rho} & \text{check } \hat{c} = \mathsf{commit}(\hat{m}_A, \hat{K}; \hat{\rho})\\
\text{compute } \mathsf{SAS} \leftarrow \hat{R} \oplus h_K(\hat{m}_B) & \xrightarrow{\mathsf{authenticate}(\mathsf{SAS})} & \text{verify } \mathsf{SAS} = R \oplus h_{\hat{K}}(m_B)\\
\text{verify } \mathsf{SAS} = \hat{R} \oplus h_K(\hat{m}_B) & \xleftarrow{\mathsf{authenticate}(\mathsf{SAS})} & \text{compute } \mathsf{SAS} \leftarrow R \oplus h_{\hat{K}}(m_B)\\
& & \\
\textbf{output}: \hat{m}_B & & \textbf{output}: \hat{m}_A
\end{array}$$

In the considered attacks in this exercise, we have a (wo)man-in-the-middle (let's call her Eve) who controls the (unauthenticated) channels and substitutes all variables by some variants with a hat that she computes. She wants that for any $m_A, m_B, \hat{m}_A, \hat{m}_B$, there exists an *efficient* way to make Alice and Bob succeed (i.e., both verifications succeed), with input/output $m_A/\hat{m}_B$ and $m_B/\hat{m}_A$, respectively. If not specified otherwise, the attack is supposed to succeed with probability 100%. We assume that h and commit are *easily* computable. We assume that r

is small so that a loop with 2^r iterations is considered as *efficient*. Eve can use $m_A, m_B, \hat{m}_A, \hat{m}_B$ as input before the attack starts.

Q.1 Explain what this protocol could be used for and a typical use case.

The protocol could be used to mutually authenticate ephemeral Diffie-Hellman public keys m_A and m_B. This way, we would have an authenticated key agreement protocol. It could be used to initialize a secure association. Similar protocols are used in Bluetooth pairing, like the Simple Secure Pairing *based on* Numeric Comparison. *The* SAS *is typically a 6-digit number (so $r \approx 30$), displayed by device A and device B, and a human operator check that they are equal to authenticate them.*

Q.2 (Case of a non-hiding commitment.) If there exists an efficiently implementable function reveal such that for all m, K, ρ, we have

$$\mathsf{reveal}(m, \mathsf{commit}(m, K; \rho)) = K$$

prove that there is an attack.

After Alice releases m_A, c, Eve computes $K = \mathsf{reveal}(m, c)$ then forwards the messages to Bob but substitutes $\hat{m}_A$ to m_A and $\hat{c} = \mathsf{commit}(\hat{m}_A, K; \hat{\rho})$ to c. So, $\hat{K} = K$. After Bob releases m_B, R, Eve sends $\hat{m}_B$ and

$$\hat{R} = R \oplus h_K(m_B) \oplus h_K(\hat{m}_B)$$

to Alice. Alice then sends m_A, K, ρ which are replaced with $\hat{m}_A, K, \hat{\rho}$ by Eve. Alice computes

$$\mathsf{SAS} = \hat{R} \oplus h_K(\hat{m}_B) = R \oplus h_K(m_B)$$

and Bob computes the same. As the commitment verifies well on Bob's side, the protocol always completes.

Q.3 (Case of a modified protocol.) If we modify the protocol by removing the commitment (as shown below), prove that there is an attack using a loop of 2^r iterations.

$$\begin{array}{rcl}
\text{pick } K \in_U \{0,1\}^k & \xrightarrow{\quad m_A \quad} & \text{pick } R \in_U \{0,1\}^r \\
 & \xleftarrow{\quad m_B, R \quad} & \\
 & \xrightarrow{\quad K \quad} & \\
\mathsf{SAS} = \hat{R} \oplus h_K(\hat{m}_B) & \xleftrightarrow{\mathsf{authenticate}(\mathsf{SAS})} & \mathsf{SAS} = R \oplus h_{\hat{K}}(m_B)
\end{array}$$

Eve picks some random $\hat{R}$. She substitutes all messages in the modified protocol following the name of the variables. Then, she does a preimage attack on the function f defined by $f(\hat{K}) = h_{\hat{K}}(m_B)$ to find $\hat{K}$ such that $f(\hat{K}) = R \oplus \hat{R} \oplus h_K(\hat{m}_B)$. This phase needs roughly 2^r trials, which is simple if r is small. This allows her to substitute $\hat{K}$ in the last message and to finalize the attack. The two SAS *will match.*

Q.4 (Case of a non-binding commitment.) If there exists an efficiently implementable function $\mathsf{equivocate}$ returning random coins ρ such that for all $c, \hat{m}, \hat{K}$, we have

$$\mathsf{commit}(\hat{m}, \hat{K}; \mathsf{equivocate}(c, \hat{m}, \hat{K})) = c$$

prove that there is an attack using a loop of 2^r iterations.

Eve picks some random $\hat{c}$ and $\hat{R}$. She substitute all messages following the name of the variables. Then, she does a preimage attack on the function f defined by $f(\hat{K}) = h_{\hat{K}}(m_B)$ to find $\hat{K}$ such that $f(\hat{K}) = R \oplus \hat{R} \oplus h_K(\hat{m}_B)$. This phase needs roughly 2^r trials, which is simple if r is small. Finally, she computes

$$\hat{\rho} = \mathsf{equivocate}(\hat{c}, \hat{m}_A, \hat{K})$$

to finalize the attack. The two SAS *will match.*

Q.5 (General case: attack with success probability 2^{-r}.) Prove that there is an attack which succeeds with probability 2^{-r}.

Eve picks some random $\hat{K}$, $\hat{\rho}$, $\hat{R}$. She computes $\hat{c} = \mathsf{commit}(\hat{m}_A, \hat{K}; \hat{\rho})$ then do all substitutions. The SAS *computed by Alice and Bob will match with probability 2^{-r} and the attack will work.*