Cryptography

Academic Year 2024-2025

Homework 3

Michele Dinelli, ID 0001132338

December 10, 2024

Exercise 1.

It is asked to determine which one of the following groups [1] is cyclic when the underlying operation is common addition.

$$\mathbb{Z} \qquad \mathbb{Q} \qquad \mathbb{R}$$

A group $(\mathbb{G}, \cdot)$ is said to be cyclic if there exists $g \in \mathbb{G}$ such that the cyclic subgroup generated by g namely $\langle g \rangle = \{g^0, g^1, ...\} \subseteq \mathbb{G}$ is actually equal to $\mathbb{G}$ [2]. The element g is said generator of $\mathbb{G}$.

- The group $(\mathbb{Z}, +)$ is cyclic because exists a generator g such that $\langle g \rangle = \mathbb{Z}$. This can be proved by setting $g = 1$ or $g = -1$. Choosing $g = 1$ we define $\langle g \rangle = \{zg \mid z \in \mathbb{Z}\}$ which is in fact equals to $\mathbb{Z}$ [3]. This means that $\forall z \in \mathbb{Z}$ can be written as integer multiple of g:
  - If $z > 0$ then $z = 1 + 1 + ... + 1$ (z times)
  - If $z < 0$ then $z = (-1) + (-1) + ... + (-1)$ (z times)
  - If $z = 0$ then z is trivially 0

  Hence $(\mathbb{Z}, +)$ is cyclic when the underlying operation is common addition.
- The group $(\mathbb{Q}, +)$ is not cyclic because does not exists a generator q such that $\langle q \rangle = \mathbb{Q}$. To prove that let's consider a generator in the form of $q = \frac{a}{b}$. If q is a generator we would be able to prove that every rational number in the group $(\mathbb{Q}, +)$ is an integer multiple of $\frac{a}{b}$. q cannot be a generator because we can observe that $\frac{a}{2b}$ is a rational number but cannot be expressed as integer multiple of $\frac{a}{b}$. This would require finding a integer k such that $\frac{a}{2b} = k \cdot \frac{a}{b}$. Thus $(\mathbb{Q}, +)$ is not a cyclic group.
- The group $(\mathbb{R}, +)$ is not cyclic because does not exists a generator r such that $\langle r \rangle = \mathbb{R}$. Similarly as we did for $(\mathbb{Q}, +)$ we suppose that exist a generator r. We notice that does not exist an integer k such that $k \cdot r = \frac{r}{2}$ but clearly $\frac{r}{2} \in \mathbb{R}$ so $\langle r \rangle$ cannot be a generator so $(\mathbb{R}, +)$ is not a cyclic group.
  Another way of proving this would be considering that a subgroup of a cyclic group is cyclic, since $\mathbb{Q} \subseteq \mathbb{R}$ it's clear that $(\mathbb{R}, +)$ is not cyclic.

It is also asked to prove that every cyclic group is also abelian.
In a cyclic group $(\mathbb{G}, \circ)$ all elements are powers of a single generator g, so any two elements $a = g^m$ and $b = g^n$ satisfy the following

$$a \circ b = g^m \circ g^n = g^{m+n} = g^{n+m} = g^n \circ g^m = b \circ a$$

[1] $\mathbb{Z}$ is the set of integer number, $\mathbb{Q}$ is the set of rational numbers, and $\mathbb{R}$ is the set of real numbers

[2] Similarly for additive groups $\langle g \rangle = \{0, g, 2g, ...\}$

[3] Where in the context of additive groups zg means $g + g + ... + g$ repeated z times (common notation would be g^n for multiplicative groups). Notice how $1^{-n} = -n$ because the multiplication here is the group operation.

Thus, the operation is commutative and $(G, \circ)$ is abelian.

Exercise 2.
It is asked, in the context of multiset rewriting, starting from the model of an intruder in presence of a primitive for encryption to show an adapted version of signature and rules to reflect the use of a secure message authentication code.

- Sorts are defined as

$$\mathsf{msg} \qquad \mathsf{key} \qquad \mathsf{tag} \qquad \mathsf{bool}$$

  Modeling messages, keys, tags for MACs and boolean values.

- Function symbols are defined as

$$\mathsf{mac} : \mathsf{msg} \times \mathsf{key} \longrightarrow \mathsf{tag}$$
$$\mathsf{vrfy} : \mathsf{msg} \times \mathsf{key} \times \mathsf{tag} \longrightarrow \mathsf{bool}$$

  Where mac models tags forging and vrfy models the process of verifying that a given tag is valid.

- Predicates are defined as

$$\mathrm{A}_0 : \mathsf{key} \qquad \mathrm{A}_1 : \mathsf{msg} \times \mathsf{tag} \times \mathsf{key}$$
$$\mathrm{B}_0 : \mathsf{key} \qquad \mathrm{B}_1 : \mathsf{msg} \times \mathsf{tag} \times \mathsf{key}$$
$$\mathrm{N} : \mathsf{msg} \times \mathsf{tag} \qquad \mathrm{K} : \mathsf{key}$$

  A_0 models agent A's initial state with a key. A_1 models agent A holding a message with its tag and a key. B_0 models agent B's initial state with a key. B_1 models agent B receiving the message, its tag, and holding a key. N models network state indicating the transmission of a message and its tag. K is a predicate modeling a sort of key repository shared by the two actors.

$$D \qquad M \qquad C$$

  In order to define an adversary we also define a predicate D capturing what the attacker has observed, a predicate M modeling the intruder's memory and a predicate C which serves to model new messages the adversary has crafted, and which could possibly be sent. These predicates are generic, their application to different sorts should be meaningful and intuitive.

- Terms are defined as

$$k : \mathsf{key}, m : \mathsf{msg} \vdash \mathsf{vrfy}(m, k, \mathsf{mac}(m, k)) : \mathsf{bool}$$

- Rules for the protocol are defined as

$$\mathrm{K}(k) \longrightarrow \mathrm{A}_0(k), \mathrm{B}_0(k) \tag{1}$$
$$\mathrm{A}_0(k) \longrightarrow \exists x.\mathrm{A}_1(x, \mathsf{mac}(x, k), k), \mathrm{N}(x, y) \tag{2}$$
$$\mathrm{B}_0(k), \mathrm{N}(x, y) \longrightarrow \mathrm{B}_1(x, y, k) \tag{3}$$

  This is a sketch of protocol's rules and should only help spotting its flaws. Rule 2 states that agent A (could be Alice) generates a message and its MAC, placing them on the network. Then in rule 3 agent B (could be Bob) receives the message and MAC, then it enters a state where it can verify them since it holds the three pieces required. It is assumed that key exchange between the parties happens before MAC protocol as we are not considering it right now.

- Rules for the intruder are defined as

$$\begin{aligned}
\mathrm{N}(x) &\longrightarrow \mathrm{D}(x) && (4)\\
\mathrm{N}(x,y) &\longrightarrow \mathrm{D}(\langle x,y\rangle) && (5)\\
\mathrm{D}(\langle x,y\rangle) &\longrightarrow \mathrm{D}(x),\mathrm{D}(y) && (6)\\
\mathrm{D}(x) &\longrightarrow \mathrm{M}(x) && (7)\\
\mathrm{M}(x) &\longrightarrow \mathrm{C}(x),\mathrm{M}(x) && (8)\\
\mathrm{C}(x),\mathrm{C}(y) &\longrightarrow \mathrm{C}(\langle x,y\rangle) && (9)\\
\mathrm{C}(\langle x,y\rangle) &\longrightarrow \mathrm{N}(x,y) && (10)\\
&\longrightarrow \exists x.\mathrm{M}(x) && (11)\\
\mathrm{M}(k) &\longrightarrow \exists m.\mathrm{M}(\mathsf{mac}(m,k)),\mathrm{M}(m) && (12)
\end{aligned}$$

  Rules from 4 to 11 should be intuitive since they model sniffing the network, unpacking tuples, crafting arbitrary message and storing messages in the memory. Rule 12 is the most interesting one since it states that if the intruder has a variable k in memory it [4] can attempt to forge a tag for a fresh generated message m and store both the message and its tag in the memory. Since the intruder store the value of the message and the tag in its memory it can craft new messages to send over the network (using 8, 9, 10).
  Security of the MAC protocol relies of course on the secrecy of the key and it has to be ensured that the value of k in rule 12 has nothing to do with the original key.
  By Bob's point of view two disruptive operations that the intruder can manage to perform are: replay attacks [5] and tag forgery performed by the intruder instead of Alice. Bob should of course refuse to acknowledge messages coming with invalid tags. Since we assumed that we are modeling an intruder and the underlying primitive is secure the only flaw in the protocol will be exposing the key or allow easy tag forgery i.e. success in the context of the experiment MacForge with non negligible probability η, but it is contradictory since we assumed that the underlying primitive is secure and of course the key should not be exposed.

Exercise 3.
It is asked to consider the following protocol [6]

$$\begin{aligned}
A \to C: &\quad \{m\}_k\\
B \to C: &\quad \{p\}_h\\
C \to D: &\quad f(m,p)\\
D \to A: &\quad \{d(m)\}_j\\
D \to B: &\quad g(p)
\end{aligned}$$

and formalize it by way of ProVerif [7] showing that no adversary interacting with the protocol is capable of determining either the value of m or the value of p, of course assuming that the employed encryption primitive is secure.
The following code snippet represents the protocol in ProVerif

```
type key.

free m, p: bitstring [private].
free j, k, h: key [private].
```

[4] Referring to the intruder using it as pronoun.
[5] When speaking about replay attacks in the context of MAC the potential threats are handled at much lower level i.e. with sequence numbers.
[6] m, p are messages, j, k, h are private keys, and $\{r\}_k$ denotes the ciphertext obtained by encrypting r with k. Moreover, f, g are functions whose result does not reveal any information about any of their argument(s), while d allows anyone seeing a message $d(x)$ to also know x
[7] ProVerif version used available at http://proverif20.paris.inria.fr/.

```
fun f(bitstring, bitstring): bitstring.
fun g(bitstring): bitstring.
fun d(bitstring): bitstring.

equation forall x: bitstring; d(x) = x.

fun enc(bitstring, key): bitstring.
fun dec(bitstring, key): bitstring.
equation forall x: bitstring, y: key; dec(enc(x, y), y) = x.
equation forall x: bitstring, y: key; enc(dec(x, y), y ) = x.

free c1, c2: channel.

query attacker(m).
query attacker(p).

let A =
  out(c1, enc(m, k));
  in(c1, dm: bitstring);
  0.

let B =
  out(c2, enc(p, h));
  in(c2, gp: bitstring);
  0.

let C =
  in(c1, mk: bitstring);
  let dec_m = dec(mk, k) in
  in(c2, ph: bitstring);
  let dec_p = dec(ph, h) in
  out(c1, f(dec_m, dec_p));
  0.

let D =
  in(c1, fmp: bitstring);
  let dm = d(m) in
  let gp = g(p) in
  out(c1, enc(dm, j));
  out(c2, gp);
  0.

process
  (A | B | C | D)
```

The result is the following

```
Completing equations...
-- Process 1-- Query not attacker(m[]) in process 1
Translating the process into Horn clauses...
Completing...
Starting query not attacker(m[])
RESULT not attacker(m[]) is true.
-- Query not attacker(p[]) in process 1
Translating the process into Horn clauses...
Completing...
Starting query not attacker(p[])
RESULT not attacker(p[]) is true.

------------------------------------------------------------
Verification summary:

Query not attacker(m[]) is true.

Query not attacker(p[]) is true.

------------------------------------------------------------
```