ALGORITHMICS, graduate program 2025/2026
CRYPTOGRAPHY, 2025/2026, lab assignments list # 3, 12.12.2025
Regular deadline: 16.01 or 23.01.2026, **final submission 30.01.2026**

1. (2 pts) Using the $\mathbb{F}_{2^m}$ structure, implement the GHASH algorithm (part of the Galois/Counter Mode of operation and GMAC) [1], commonly used with AES. The $\mathsf{GF}(2^{128})$ is defined by the polynomial

$$x^{128} + x^7 + x^2 + x + 1$$

The function is defined by

$$\begin{aligned} X_0 &= 0 \\ X_i &= (X_{i-1} + S_i) \cdot H \quad \text{for} \quad i > 0 \\ S &= padWithZeros(A)||padWithZeros(C)||len(A)||len(C) \\ S_i &= \text{consecutive blocks of S counting from 1} \\ \mathsf{GHASH}(H, A, C) &= X_{m+n+1} \end{aligned}$$

Where $len(A)$ and $len(C)$ are 64-bit *bit lengths* of A and C, respectively, $m = \lceil len(A)/128 \rceil$, $n = \lceil len(C)/128 \rceil$ and $padWithZeros$ appends 0s to a bit string until the next full 128-bit block.

2. (**OPTIONAL** 2 pts) Most likely, your current implementation of exponentiation (and multiplication by scalar) is not cryptographically secure, as it leaks the Hamming Weight (number of 1s) in the exponent, and possibly even more data about the input. Try to fix this. Research time-invariant implementations of modular exponentiations and try to apply the techniques to your implementations of finite fields and elliptic curves. Test if the changes were sufficient by measuring the runtime with different exponents. Remember to check exponents with different bit lengths and significantly diverse Hamming Weights.

3. (2 pts) Implement Diffie-Hellman Key Exchange protocol using all the different algebraic structures:

   (a) Classic $\mathbb{F}_p$
   (b) Characteristic 2 field $\mathbb{F}_{2^k}$
   (c) Extended prime field $\mathbb{F}_{p^k}$ for a larger p
   (d) Elliptic curve over $\mathbb{F}_p$, $\mathbb{F}_{2^k}$ and $\mathbb{F}_{p^k}$

   A common input to the Diffie-Hellman protocol is commonly called domain parameters. These parameters describe the algebraic structure used:

   (a) For $\mathbb{F}_p$ the parameters are integers p and g
   (b) For $\mathbb{F}_{2^k}$ the parameters are bit strings m and g representing coefficients of polynomials over $\mathbb{F}_2$. m and g should have $k+1$ and k bits respectively. Since the highest coefficient of m must be 1, this bit might get omitted.
   (c) For $\mathbb{F}_{p^k}$ ($p \geq 3$) the parameters m and g also define polynomials over $\mathbb{F}_p$, but require more complex data structures, like lists, vectors or tuples. Again, we may implicitly omit highest coefficient for m which shall always be 1.
   (d) For elliptic curves, the parameters are twofold. First, we have to define the underlying field $\mathbb{F}$ like in cases 1-3. Then we need to provide coefficients a and b as elements of the field $\mathbb{F}$. Additionally, we provide the generator point G as a pair of coordinates (in F).

   All cases also include an integer q denoting order of the group generated by g (or G), i.e. $q = |\langle g \rangle|$ ($|\langle G \rangle|$ respectively). For DH, ideally q should be prime. Note that g should only generate some subgroup of order q and not full $\mathbb{F}_{p^k}$. *This is required for the group to be DDH-secure.*

   The Diffie-Hellman protocol has two phases:

   (a) **Key Generation**. In this phase, both parties generate their ephemeral private and public keys. A private key sk should be a random integer from $\mathbb{Z}_q$. The public key is computed as $pk = g^{sk}$ using appropriate group operations.

(b) **Key Agreement**. In this phase, each party takes their own secret key (sk) from the previous step, and mixes it with the public key obtained from the other party (epk) to generate shared secret $ss = epk^{sk}$

For elliptic curves the idea stays the same, but the group operation is addition and not multiplication, and thus we typically write $PK = [sk]G$ (scalar multiplication) and $SS = [sk]EPK$.

Work in pairs or larger groups to test your implementations. Ensure interoperability.

4. (2 pts) Implement Schnorr signature scheme [2, 3]. When in doubt, use the variant that uses the values (s, e) as the signature. Again, implement the algorithm using both $\mathbb{F}^*_{p^k}$ and EC. Compare the runtime at different security levels (c.f. `keylength.com` [4]). For instance, NIST recommendation for years 2019 - 2030 & beyond, 128 bits of security, is a 256 asymmetric key length and 3072 bit group modulus, or 256-bit elliptic curve.

   As a hash algorithm, use SHA256 (SHA2). To ensure compatibility with task 5 compute the value $e = H(R||m)$ as follows:

   (a) If R is an integer ($\mathbb{F}_p$ case), encode it as a big-endian hexadecimal string with the same bit and byte length as p. For example, if $p = 65537 = 0x010001$, encode $n = 17 = 0x000011$.

   (b) If R is a "bit string" ($\mathbb{F}_{2^m}$ case), encode it as if the bit string were an integer. The length should match m rounded up to 8 bits (full byte). For example, if $m = 33$ (rounded up to 40) encode $n = x^3 + x^2 + 1$ as $0b1101 = 0x00000000D$.

   (c) If R is a polynomial over $\mathbb{F}_p$ ($\mathbb{F}_{p^k}$ case), encode each of its coefficients according to the first rule. Gather all coefficients in an array of k elements with explicit zeros in order from x^0 to x^{k-1}.

   (d) If R is an elliptic curve point, encode each coordinate independently according to the rules above. Keep as an object with parameters x and y.

   (e) Convert the encoded value to a *compact JSON form*, i.e. a JSON string without any unnecessary whitespaces.

   Examples:

   - Encode$(17 \in \mathbb{F}_{65537}) =$ `"000011"`
   - Encode$(x^3 + x^2 + 1 \in \mathbb{F}_{2^{33}}) =$ `"000000000D"`
   - Encode$(3x^2 + 16 \in \mathbb{F}_{17^3}) =$ `["10","00","03"]`
   - Encode$((3, 5) \in \mathbb{E}(\mathbb{F}_{17})) =$ `{"x":"03","y":"05"}`
   - For $17 \in \mathbb{F}_{65537}$ and $m =$`"Alice"`, $e = SHA256('$`"000011"Alice`$') = 0xfaf4(...)1830$

   This encoding is consistent with task 5.

5. (4 pts) Until the **final submission** deadline.
   A dedicated service has been prepared to validate your solutions. Go to `https://crypto25.random-oracle.xyz/docs` and familiarize yourself with the REST API. You can use the OpenAPI page to submit test requests, obtain test parameters, etc. Test your implementation against the /test/... endpoints.

   The service is only exposed through HTTPS (port 443). You may simply use curl:

```
$ curl https://crypto25.random-oracle.xyz/
{"status":"success"}
```

   After testing against `/test`, try submitting your solution against the `/submit` endpoints. First, you need to *GET* the challenge from `/submit/start/{type}`:

```
$ curl -X 'GET' \
  'https://crypto25.random-oracle.xyz/submit/start/modp' \
  -H 'accept: application/json'

{
  "status": "success",
  "type": "modp",
  "session_id": "0081f34f-3c9c-45ea-8f8b-e1bc0389a79c",
```

```
  "params": {
    "name": "modp2048",
    "modulus": "...",
    "generator": "...",
    "order": "..."
  },
  "server_public_sign": "...",
  "server_public_dh": "...",
  "signature": {
    "s": "...",
    "e": "..."
  }
}
```

The challenge should be valid for about 5 minutes. Within this time, you have to do the following:

(a) Check the Schnorr signature `"signature"` over `"server_public_dh"` (use the same encoding as for R in the previous task).

(b) If signature verification fails, try again with a fresh session. Otherwise, proceed.

(c) Generate your own DH and Schnorr private/public key pairs.

(d) Sign your DH public key with your Schnorr private key.

(e) Compute DH `"shared_secret"` using your private key and `"server_public_dh"`.

Finally, submit the solution using the same `"session_id"`:

```
curl -X 'POST' \
  'https://crypto25.random-oracle.xyz/submit/finish' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "session_id": "0081f34f-3c9c-45ea-8f8b-e1bc0389a79c",
  "client_public_sign": "...",
  "client_public_dh": "...",
  "signature": {"s": "...", "e": "..."},
  "shared_secret": "..."
}'
```

The service will report status (`success` or `failure`) and record it internally. However, more severe errors might cause, e.g. overall parsing/conversion error:

```
{'detail': [
  {'type': 'missing',
   'loc': ['body', 'signature'],
   'msg': 'Field required',
   'input': {
     'session_id': '8772d71a-afd0-41bb-9adf-e5516a492d7d',
     'client_public_sign': ...
}}]}
```

Poisoned sessions As noted above, you are required to verify the signature before responding. This is because approximately 1 in 3 sessions are deliberately *poisoned* in a way that results in an invalid signature (it may be a damaged public key, damaged signature, or damaged message). You should *not* finish poisoned sessions and instead allow them to time out. Responding to a poisoned session will count as an unreported *failure*.

Task After you have successfully completed a session, record its `"session_id"`. Try to solve a task for each "type" (`modp`, `f2m`, `fpk`, `ecp`, `ec2m`, `ecpk`). Finally, send the successful session ids in a list via a private channel like email or MS Teams. Send only one session id for each type and make sure not to send an id for a poisoned session.

Extra task Try to complete the round-trip (from start to finish) as quickly as possible. There might be a bonus for the fastest solutions.

/-/ Marcin Słowik

References

[1] Morris J Dworkin. *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac*. National Institute of Standards & Technology, 2007. DOI: 10.6028/NIST.SP.800-38D.

[2] Wikipedia contributors. *Schnorr signature*. URL: https://en.wikipedia.org/wiki/Schnorr_signature.

[3] Claus-Peter Schnorr. "Efficient signature generation by smart cards". In: *Journal of cryptology* 4 (1991), pp. 161–174.

[4] BlueKrypt. *Keylength – Cryptographic Key Length Recommendation*. URL: https://www.keylength.com/.