Institute of Computer Science, University of Bern
Privacy and Data Security, HS 2021

Prof. Christian Cachin
Orestis Alpos, Luca Zanolini

Exercise 10

10.1 Irreversible tokenization (10pt)

The *Luhn algorithm* computes a single-digit checksum that is used to protect numbers written in decimal from some typical errors made by humans when copying the number. The algorithm will detect all single-digit errors and almost all transpositions of adjacent digits. However, many other errors, such as transpositions of digits at even distance, are not caught. Credit-card numbers use the Luhn algorithm, for example.

a) Study the description given in Wikipedia `https://en.wikipedia.org/wiki/Luhn_algorithm` and implement a function `luhn(s)` in Python that returns `True` whenever the Luhn-checksum of the (number represented by) string `s` is valid. (Note that the checksum is one digit, and typically the last one, but this is not necessary – why?)

b) Implement a hash-based *irreversible tokenization* function `hash_token(s)` that takes an n-digit string `s` containing a valid Luhn checksum (such as a credit card number) and returns another n-digit token string, which also has a valid Luhn checksum. Use the cycle-walking method and SHA-256 from Python's `hashlib.sha256`.

c) Extend your tokenization function to a *keyed* irreversible tokenization function `mac_token(key, s)`, which additionally takes an arbitrary string `key` as input that serves as the key. Use HMAC-SHA256 from Python 3's built-in `hmac` library and the cycle-walking method.

This description suggests to use Python, but you may also use Java or C++ as programming language. For further languages, check with the assistants.

SHA-256 returns an array of 32 bytes, which is not in the correct format for cycle walking: interpret this as a number in binary, reduce it modulo 10^n*, and convert the truncated number to an* n*-digit string. Python's* `hashlib.sha256.hexdigest()` *gives a 32-byte string* `s` *that one can convert to an integer using* `int(s,base=16)`*, for instance.*

10.2 Reversible tokenization with FPE (+3pt bonus)

Format-preserving encryption (FPE) can be used for *reversible* tokenization. The auxiliary file `ex10-smallcipher.py` contains a Python implementation of a small-domain encryption algorithm `smallcipher_encrypt`, according to Black and Rogaway [BR02] and Bellare *et al.* [BRRS09, Scheme FE2/FD2, Fig. 3]

a) Implement an FPE-based *reversible tokenization* function `fpe_encrypt(key,tweak, s)` to compute an FPE of the n-digit string `s`, which also satisfies the Luhn checksum. Let `key` be the encryption key and `tweak` be an arbitrary string that "tweaks" the small-domain cipher; it can be thought of a domain separator.

b) Implement the corresponding decryption algorithm and verify that it inverts the encryption.

Use again the cycle-walking method, as developed for Problem 10.1.

References

[BR02] J. Black and P. Rogaway, *Ciphers with arbitrary finite domains*, Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings (B. Preneel, ed.), Lecture Notes in Computer Science, vol. 2271, Springer, 2002, https://doi.org/10.1007/3-540-45760-7_9, pp. 114–130.

[BRRS09] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers, *Format-preserving encryption*, Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers (M. J. J. Jr., V. Rijmen, and R. Safavi-Naini, eds.), Lecture Notes in Computer Science, vol. 5867, Springer, 2009, https://doi.org/10.1007/978-3-642-05445-7_19, pp. 295–312.