Call for Protected Software Implementations of Finalists in the NIST Lightweight Cryptography Standardization Process

Cryptographic Engineering Research Group, George Mason University, U.S.A.

January 18, 2022

1 Introduction

We call for software implementations of finalists in the NIST Lightweight Cryptography Standardization Process resistant against side-channel attacks such as power and electromagnetic analysis. The focus of this call is on the use of platform-independent algorithmic countermeasures. The submitted code should demonstrate strong resistance against side-channel attacks when executed on low-cost modern embedded processors, such as ARM Cortex M4F, RISC-V (e.g., RV32IMAC), Microchip 8-bit AVR, and TI MSP430. This code can contain assembly language instructions specific to a given Instruction Set Architecture (ISA). All submitted implementations will be investigated by one or more Side-Channel Security Evaluation Labs. The primary goal of these labs will be to validate the security claims of the implementers. However, the tasks of these labs may extend beyond this major goal and include the key recovery attacks and various ways of assessing the secret information leakage. The labs will be able to choose freely from all implementations placed in the public domain. Additionally, implementers will be able to submit their code to particular labs. Finally, labs will also be able to ask implementers for their deliverables.

2 Preliminary Requirements

Protected software implementations **shall** use the standard NIST API defined in Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process, published in August 2018[1]. Protected implementations **shall not** use `nsec`, beyond specifying it as an argument of `crypto_aead_encrypt()` and `crypto_aead_decrypt()` set to NULL.

Considering the short amount of time devoted to analyzing protected implementations by the Side-Channel Security Evaluation Labs, it is important that all submissions can be evaluated using the Test Vector Leakage Assessment (TVLA) method, a.k.a., Welch's t-test. To make it possible, we suggest that at least one variant of the protected implementation is designed to accommodate this test.

2.1 Approach simplifying the evaluation of protected implementations via t-test

To simplify power correlation evaluation via Welch's t-test without spurious correlation from sharing and un-sharing operations, we propose the clear division of the protected implementation into three functions: `generate_shares_encrypt()`, `crypto_aead_encrypt_shared()`, `combine_shares_encrypt()`, for encryption, and
`generate_shares_decrypt()`, `crypto_aead_decrypt_shared()`, `combine_shares_decrypt()`, for decryption.

[1] `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf`

Only `crypto_aead_encrypt_shared()` for encryption and `crypto_aead_decrypt_shared()` for decryption should be used for leakage assessment. We propose that these functions follow the API defined below:

```
int crypto_aead_encrypt_shared(
  mask_c_uint32_t *cs, unsigned long long *clen,
  const mask_m_uint32_t *ms, unsigned long long mlen,
  const mask_ad_uint32_t *ads, unsigned long long adlen,
  const mask_npub_uint32_t *npubs,
  const mask_key_uint32_t *ks
)

int crypto_aead_decrypt_shared(
  mask_m_uint32_t *ms, unsigned long long *mlen,
  const mask_c_uint32_t *cs, unsigned long long clen,
  const mask_ad_uint32_t *ads, unsigned long long adlen,
  const mask_npub_uint32_t *npubs,
  const mask_key_uint32_t *ks
)
```

The corresponding data types used to represent words of specific inputs and outputs split into one or more shares are defined as follows:

```
typedef struct
{
 uint32_t shares[NUM_SHARES_M];
} mask_m_uint32_t;

typedef struct
{
 uint32_t shares[NUM_SHARES_C];
} mask_c_uint32_t;

typedef struct
{
 uint32_t shares[NUM_SHARES_AD];
} mask_ad_uint32_t;

typedef struct
{
 uint32_t shares[NUM_SHARES_NPUB];
} mask_npub_uint32_t;

typedef struct
{
 uint32_t shares[NUM_SHARES_KEY];
} mask_key_uint32_t;
```

Values of constants `NUM_SHARES_M`, `NUM_SHARES_C`, `NUM_SHARES_AD`, `NUM_SHARES_NPUB`, and `NUM_SHARES_KEY` **shall** be defined in the file `api.h`. Each of them should be an integer greater than or equal to 1. The default value of these constants should be 1, which corresponds to no division into shares.

The `generate_shares_encrypt()` function should allow the division of all inputs to encryption, namely `m`, `ad`, `npub`, and `k`, into Boolean shares. The `generate_shares_decrypt()` function should allow the division of all inputs to decryption, namely `c` (including the tag), `ad`, `npub`, and `k`, into Boolean shares. The number of shares may be different for each input, as defined in `api.h`.

The incomplete last word of `m`, `ad`, `npub`, and `k` **shall** be padded with zeros. The input `c` should be treated in a special way. For the authenticated ciphers using tag, the last `CRYPTO_ABYTES` of `c` are used by the tag, and the first `clen−CRYPTO_ABYTES` by the ciphertext. The function `generate_shares_decrypt()` should first pad the last word of the ciphertext (not including the tag) with zeros. The tag should then follow, starting on a boundary of a 32-bit word. Only after this initial preprocessing, the ciphertext and tag should be masked.

The `generate_shares_encrypt()` and `generate_shares_decrypt()` functions should use the following API:

```
void generate_shares_encrypt(
  const unsigned char *m, mask_m_uint32_t *ms, const unsigned long long mlen,
  const unsigned char *ad, mask_ad_uint32_t *ads, const unsigned long long adlen,
  const unsigned char *npub, mask_npub_uint32_t *npubs,
  const unsigned char *k, mask_key_uint32_t *ks)

void generate_shares_decrypt(
  const unsigned char *c, mask_c_uint32_t *cs, const unsigned long long clen,
  const unsigned char *ad, mask_ad_uint32_t *ads, const unsigned long long adlen,
  const unsigned char *npub, mask_npub_uint32_t *npubs,
  const unsigned char *k, mask_key_uint32_t *ks)
```

The `combine_shares_encrypt()` and `combine_shares_decrypt()` functions should allow combining Boolean shares of the outputs, `c` for encryption and `m` for decryption. These functions should use the following APIs:

```
void combine_shares_encrypt(
  const mask_c_uint32_t *cs, unsigned char *c, unsigned long long clen)

void combine_shares_decrypt(
  const mask_m_uint32_t *ms, unsigned char *m, unsigned long long mlen)
```

2.2 Evaluation Procedures

For compatibility with benchmarking software, such as SUPERCOP, the execution time of authenticated encryption and authenticated decryption, as well as other performance metrics, such as power consumption and energy per bit, should be determined using standard NIST API. The functions `crypto_aead_encrypt_shared()` and `crypto_aead_decrypt_shared()` will be used primarily for the purpose of leakage assessment. Side-channel security evaluation labs may use either standard or non-standard APIs in their attempts at attacks.

Attempts at exploiting information leakage occurring during the execution of the functions `generate_shares_encrypt()`, `generate_shares_decrypt()`, `combine_shares_encrypt()`, and `combine_shares_decrypt()` will be of interest to the cryptographic community but are not likely to demonstrate differences among LWC candidates.

3 Suggested Deliverables

To simplify benchmarking, security analysis, and further optimizations of protected software implementations of LWC algorithms, we propose a uniform way of publishing them on the web and submitting them to the benchmarking and security evaluation labs.

All protected implementations of a given candidate developed by a particular group should be stored in the same folder. This folder may either

1. become a basis of an online repository (e.g., a GitHub repository), in which case the submission consists of the repository URL, including branch name, tag, or commit hash, or

2. be submitted as a single archive file (e.g., .zip) to the selected benchmarking and security evaluation labs.

The contents of the submission folder should follow the guidance of the NIST LWC specification[2] section 3.5.1 (AEAD) and 3.5.2 (Hash). Per this specification, in addition to the protected implementation, submissions should include a reference implementation (`ref`) and Known-answer-test file (`KAT`) to allow existing tools to verify implementation correctness.

Please name this folder using the following convention: `<LWC_Candidate_Name>_<Group_Name>`.

3.1 Source code

Source code for protected implementations **shall** be provided as C99 standard C suitable for compilation, linkage, and assembly using standard tooling (e.g., GCC) for the target architecture(s). Architecture specific optimizations (e.g., assembly language) may additionally be provided to demonstrate performance enhancement. If the use of assembly language is intended to enhance resistance against side-channel attacks, then this should be stated explicitly in the supporting documentation.

3.1.1 External Dependencies

Submissions for side-channel evaluation **shall not** depend on any external headers or libraries, including cryptographic libraries (e.g., OpenSSL), outside of the C99 standard, with the exception of the `randombytes.h` header from SUPERCOP, which may be used for masking or sharing:

```
/* for a constant time, non-blocking stream of random values from a uniform distribution. */
#include "randombytes.h"

// for reference, this header prototypes:
// extern void randombytes(unsigned char *, unsigned long long);
```

To ensure deterministic results, the random implementation **shall** be provided by the test harness (not the submission) and will be initialized prior to execution of the cryptographic algorithm. Implementations should use this call directly, rather than including a DRBG (based on AES, ChaCha, SHAKE, etc.) within the implementation.

3.1.2 Directory Structure

The structure of a compliant submission should look like:

```
<Candidate_Name>_<Group_Name>
├── Documents
│   ├── changelog.pdf                # optional
│   ├── coversheet.pdf
│   └── documentation.pdf
├── Implementations
│   └── crypto_aead
│       └── <Candidate_and_Variant_Name>
│           ├── designers
│           ├── LWC_AEAD_KAT_<CRYPTO_KEYBYTES*8>_<CRYPTO_NPUBBYTES*8>.txt
│           ├── protected_<implementation_name>
│           │   ├── api.h
│           │   ├── aead.c
```

[2] https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf

```
│           │   ├── other.h
│           │   ├── other.c
│           │   ├── other.S           # assembly is acceptable
│           │   ├── goal_powersca_1st # indicates 1st order protection against power analysis
│           │   ├── goal_powersca_2nd # indicates 2nd order protection against power analysis
│           │   ├── goal_emsca        # indicates protection against electromagnetic analysis
│           │   ├── architectures     # one target arch per line
│           │   └── implementers      # one implementer per line
│
... additional implementations may be provided, one per tertiary directory ...
│           └── ref
│               ├── api.h
│               ├── encrypt.c
│               ├── implementers
│               └── other.c
├── LICENSE.txt
└── README.md
```

The <Candidate_and_Variant_Name> is a single name incorporating the name of the candidate and its specific variant. Different variants correspond to

- different algorithms of the same family
- different parameter sets, such as sizes of keys, nonces, tags, etc.

3.1.3 Boolean sharing definitions in api.h

In addition to standard definitions in `api.h`, the following definitions may be used by Boolean-shared implementations:

1. `NUM_SHARES_M` the number of Boolean shares of message **m**
2. `NUM_SHARES_C` the number of Boolean shares of ciphertext and tag **c**
3. `NUM_SHARES_AD` the number of Boolean shares of authenticated data **ad**
4. `NUM_SHARES_NPUB` the number of Boolean shares of nonce **npub**
5. `NUM_SHARES_KEY` the number of Boolean shares of the secret key **k**

Each constant **shall** be an integer greater than or equal to `1`.

For example, in a protected implementation which uses 3 shares of a 16-byte key but no other sharing, `api.h` may look like:

```
#define CRYPTO_KEYBYTES 16
#define CRYPTO_NPUBBYTES 16
#define CRYPTO_NSECBYTES 0
#define CRYPTO_ABYTES 16
#define CRYPTO_NOOVERLAP 1

#define NUM_SHARES_M 1
#define NUM_SHARES_C 1
#define NUM_SHARES_AD 1
#define NUM_SHARES_NPUB 1
#define NUM_SHARES_KEY 3
```

3.1.4 Metadata Files

The following optional metadata files may be included alongside of source code within a 3rd-level folder:

1. `goal_powersca_1st` - When present, this file[3] indicates the implementation has been protected against 1st order Power Analysis Side-Channel attacks.
2. `goal_powersca_2nd` - When present, this file[3] indicates the implementation has been protected against 2nd order Power Analysis Side-Channel attacks.
3. `goal_emsca` - When present, this file[3] indicates the implementation has been protected against Electromagentic Side-Channel attacks.
4. `architectures` - File with one target microcontroller/microprocessor architecture. Contents described in the **Architectures** section below.
5. `implementers` - File with one name per line indicating the authors of the protected implementation source code

3.1.5 Architectures

For uniformity, a standard set of values for use within the `architectures` file are defined as follows:

Value	ISA	Example Targets	Compiler
`arm`	ARM	STM32F	`arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb`
`avr`	AVR	ATMega2560	avr-gcc
`riscv32`	RISC-V	FE310	`riscv64-unknown-elf-gcc -march=rv32imac`
`msp430`	MSP-430	MSP430FR5969	`msp430-elf-gcc`

Note that there is neither validation of values nor limit on the number of lines in this file.

3.2 Known-Answer Tests

Each algorithm **shall** include a KAT produced by `genkat_aead.c` available from NIST [4] in the second-level folder (common to all implementations of a particular algorithm). Implementations which support hashing should also include a KAT produced by `genkat_hash.c`

3.3 Documentation of a protected implementation

Please provide one or more PDF files describing:

1. Protection Method
   (a) name of the applied countermeasure
   (b) corresponding primary reference describing this countermeasure (when applied to an arbitrary cryptographic algorithm)
2. Results of the Preliminary Security Evaluation
   (a) Attack/leakage assessment type
   (b) Number of traces used
   (c) Experimental setup

[3] This file may be empty, or contain human readable text briefly describing the mitigation
[4] https://csrc.nist.gov/Projects/Lightweight-Cryptography

i. Measurement platform and device-under-evaluation (e.g., ChipWhisperer, CW308 with STM32F3003 UFO Target)
ii. Description of measurements, e.g., shunt resistor value, current probe specification, electromagnetic probe specification and placement, or link to relevant documentation.
iii. Usage of bandwidth limiters, filters, amplifiers, etc. and their specification
iv. Frequency of operation
v. Oscilloscope and its major characteristics (e.g., bandwidth)
vi. Sampling frequency and resolution
vii. Are sampling clock and design-under-evaluation clock synchronized?

(d) Attack/leakage assessment characteristics
i. Data inputs and performed operations
ii. Source of random and pseudorandom inputs (e.g., DRBG type, seed values)
iii. Trigger location relative to the execution start time of the algorithm
iv. Time required to collect data for a given attack/leakage assessment
v. Total time of the attack/assessment
vi. Total size of all traces (if stored)
vii. Availability of raw measurement results

(e) Attack specific data
i. Power model
ii. Attack point

(f) Documentation of results
i. Graphs illustrating the obtained results, e.g., Test Vector Leakage Assessment (TVLA) graphs, minimum traces to disclosure (MTD) graphs, guessing entropy (GE), etc.
ii. Attack scripts.

4 Proposed Timeline

- Call for protected software implementations
  - First draft – December 13, 2021
  - Discussion on the `lwc-forum`
  - Final version – January 17, 2022
- Deadline for protected software implementations
  - Submission to the selected benchmarking and security evaluation labs – March 15, 2022
  - Announcement on the `lwc-forum` (optional) – March 15, 2022
- Benchmarking of Protected Implementations in terms of Throughput, Memory, Power, and Energy per bit
  - Preliminary version of the report – April 30, 2022
  - Final version of the report (considering relative security of evaluated implementations) – July 30, 2022
- Security Evaluation Lab Reports
  - Preliminary version of the report – April 30, 2022
  - Final version of the report – June 30, 2022.

5 Contact Information

Jens-Peter Kaps and Kris Gaj
Cryptographic Engineering Research Group
George Mason University
jkaps@gmu.edu , kgaj@gmu.edu
https://cryptography.gmu.edu