Test Your Cryptographic Primitives with Crypto-Condor

Angèle Bossuat <abossuat@quarkslab.com>
Dahmun Goudarzi <dgoudarzi@quarkslab.com>
Julio Loayza Meneses <jloayzameneses@quarkslab.com>

Quarkslab

crypto-condor

- Python library for compliance testing of cryptographic primitives
- Uses NIST *test vectors* and crafted ones *(e.g Wycheproof's)*.
- Python API and a CLI

```
$ pip install crypto-condor
```

Repo: https://github.com/quarkslab/crypto-condor
Docs: https://quarkslab.github.io/crypto-condor/latest/

What is a test vector?

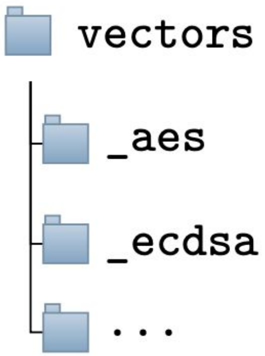
Definition

Set of inputs/outputs provided to a system in order to test that system. For **deterministic** functions, we always expect the outputs to be equal: if not, the tested system is not correct.

Sources

- NIST's CAVP ⇒ **compliance**
- Specifications, RFCs, official implementations, etc. ⇒ **compliance**
- Project Wycheproof ⇒ **resilience**

All sources parsed and stored in protobufs
⇒ standard format and easily loadable as Python classes at runtime.



Test vectors types

Three types of test vectors.

Valid	Invalid	Acceptable
Use correct inputs and return valid outputs ⇒ Typically found in the specification	Use incorrect and/or modified inputs *(e.g. a bit flip in a signature)* and expect an invalid response, generally an error.	A legacy behavior is in play, but is still tolerated.

Non-deterministic case

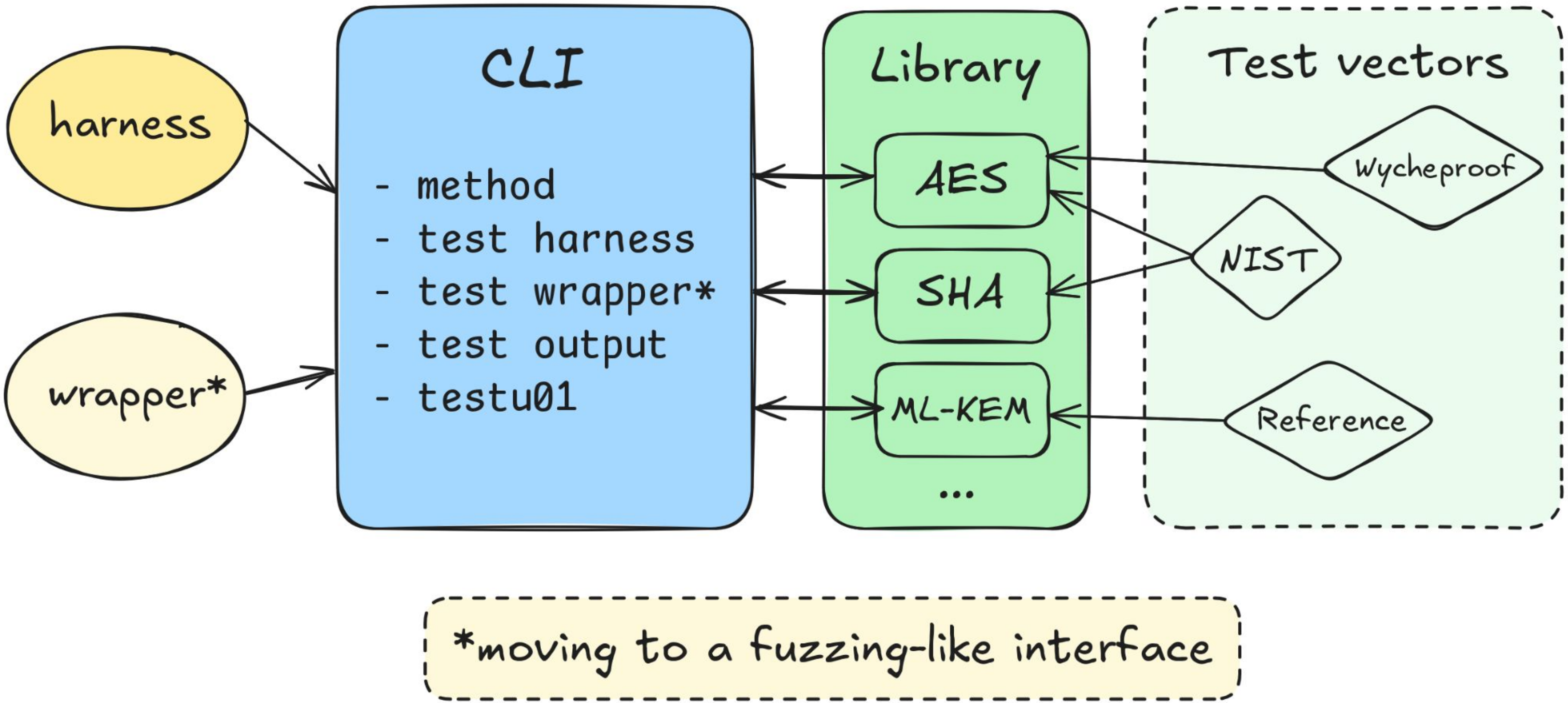
Test vectors work well for **deterministic** algorithms, such as hash functions. But what about *non-deterministic* ones?

*E.g: ECDSA is **not** deterministic: unless using the RFC 6979 version, signing the same message with the same key twice results in two different signatures.*

```
Ver(Sign(message, key)) == OK
```

⇒ We can sign the messages with the implementation, and then use the *reference* implementation to *verify* the signatures!

Architecture



List of supported primitives

```
Test modes
method       : Method guides on primitives.
test wrapper: Test an implementation with test vectors using a wrapper.
test output : Test the output of an implementation.
test harness: Test an implementation with test vectors using a harness.
```

Supported primitives	method	test wrapper	test output	test harness
AES	Y	Y	Y	Y
ChaCha20	Y	Y	Y	-
ECDH	Y	Y	-	Y
ECDSA	Y	Y	Y	-
Falcon	Y	-	-	-
HMAC	Y	Y	-	Y
HQC	Y	Y	-	Y
MLDSA	Y	Y	Y	Y
MLKEM	Y	Y	Y	Y
RSAES	Y	Y	-	-
RSASSA	Y	Y	-	-
SHA	Y	Y	Y	Y
SHAKE	Y	Y	Y	Y
SLHDSA	Y	Y	-	Y

Method guides

```
Usage: crypto-condor-cli method [OPTIONS] PRIMITIVE

Get a method guide of a primitive.
Method guides contain key information on the primitive, such as its parameters, as well as the corresponding rules
and recommendations by the ANSSI.

Bear in mind that, while the guides are provided in Markdown format, it is recommended to read them directly from
the documentation, as the formatting is not optimised for the source file.

╭─ Arguments ──────────────────────────────────────────────────────────────────────────────────╮
│ *    primitive      TEXT  The primitive to get a method guide of. [required]                │
╰──────────────────────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────────────────────╮
│ --out          PATH  Output file, defaults to primitive.md.                                  │
│ --help               Show this message and exit.                                             │
╰──────────────────────────────────────────────────────────────────────────────────────────────╯
```

- One markdown entry for each primitive
- High-level overview of the primitive for tech people *(w/o crypto background)*
- Rules and recommendations by ANSSI
- Overview of PQ primitives

ANSSI *(Agence Nationale de la sécurité des systèmes d'information)*. French Cybersecurity Agency, in charge of the country's overall cybersecurity, certification, etc.

SHA

SHA, or the Secure Hash Algorithms, are a family of cryptographic hash functions, published and standardized by the NIST.

Summary of ANSSI rules and recommendations

Rule/ recommendation	SHA-1	SHA-2	SHA-3
Recommended/ obsolete	Obsolete	Recommended	Recommended
RegleHash	Not compliant: (1) digest size is $160 < 256$ and (2) a known collision attack is estimated to require $2^{63} < 2^{160/2}$ operations.	Compliant	Compliant
RecommandationHash	Not compliant	Compliant	Compliant

Documentation

The SHA-2 family ¶

Hash function	Output size (bits)	Collision resistance	Preimage resistance	2nd preimage resistance	Comment
SHA-224	224	112	224	224	Truncated version of SHA-256 with different initial value
SHA-256	256	128	256	256	
SHA-348	384	192	384	384	Truncated version of SHA-512 with different initial value
SHA-512	512	256	512	512	
SHA-512/224	224	112	224	224	Truncated version of SHA-512
SHA-512/256	256	128	256	256	Truncated version of SHA-512

ANSSI rules and recommendations

Source: Guide des mécanismes cryptographiques

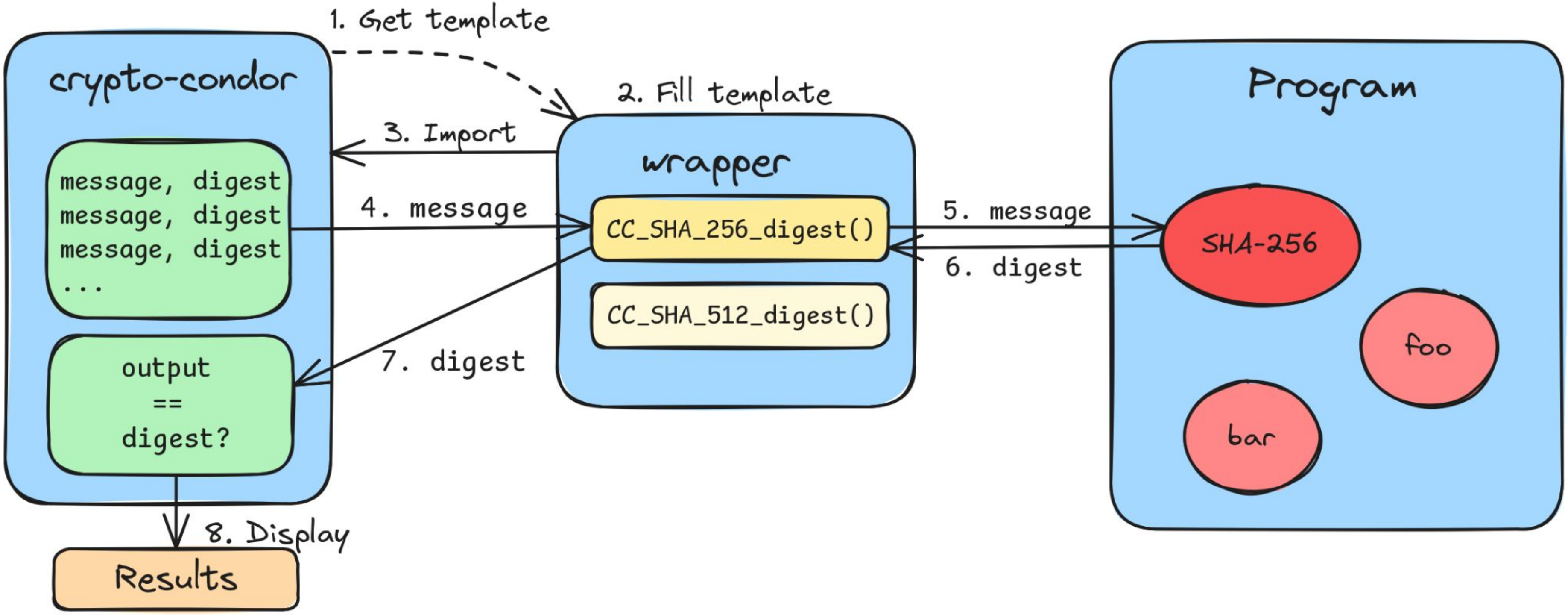
RegleHash

1. The minimum size of digests produced by a hash function is 256 bits.
2. The best known attack for finding collisions must require at least $2^{h/2}$ hashing operations, where h is the size in bits of the digests.

RecommandationHash

1. The use of hash functions for which “partial attacks” are known is not recommended.

Wrappers: Overview



Wrappers: Protocol

```
protocol crypto_condor.primitives.SHA.HashFunction
```

Represents a hash function.

Hash functions must behave like `__call__` to be tested with this module.

Classes that implement this protocol must have the following methods / attributes:

```
__call__(data)
```

Hashes the given data.

PARAMETERS:

data (*bytes*) – The input data.

RETURNS:

The resulting hash.

RETURN TYPE:

bytes

Wrappers: Example with SHA-256

To be populated by user

```
from hashlib import sha256

def CC_SHA_256_digest(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest
```

Wrappers: Example with SHA-256

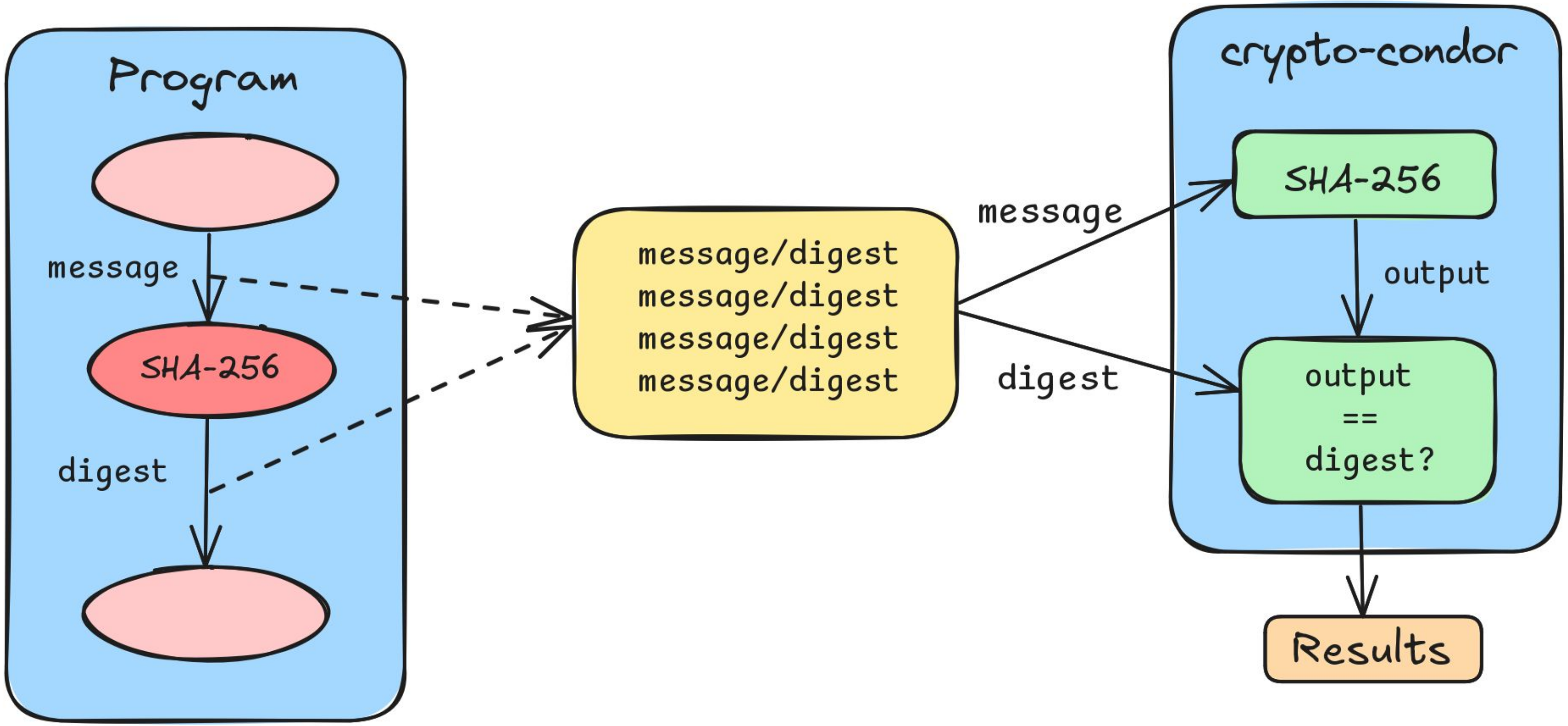
Pre-defined crypto-condor signature *(immutable)*

```python
from hashlib import sha256

def CC_SHA_256_digest(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest
```

Wrappers: Example with SHA-256

```
➜ crypto-condor-cli test wrapper SHA sha-example.py --no-save
[SHA-256] Test digest ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00
[SHA-256] Monte-Carlo test ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00


╭────────────────────────── Types of tests ──────────────────────────╮
│ Valid tests     : valid inputs that the implementation should use correctly. │
│ Invalid tests   : invalid inputs that the implementation should reject.      │
│ Acceptable tests: inputs for legacy cases or weak parameters.                │
╰──────────────────────────────────────────────────────────────────────╯
╭────────────────────────── Results summary ─────────────────────────╮
│ Primitives tested: SHA                                               │
│ Valid tests:                                                         │
│   Passed: 130                                                        │
│   Failed: 0                                                          │
╰──────────── crypto-condor 2025.04.28 by Quarkslab ──────────────────╯
```

Output: Overview

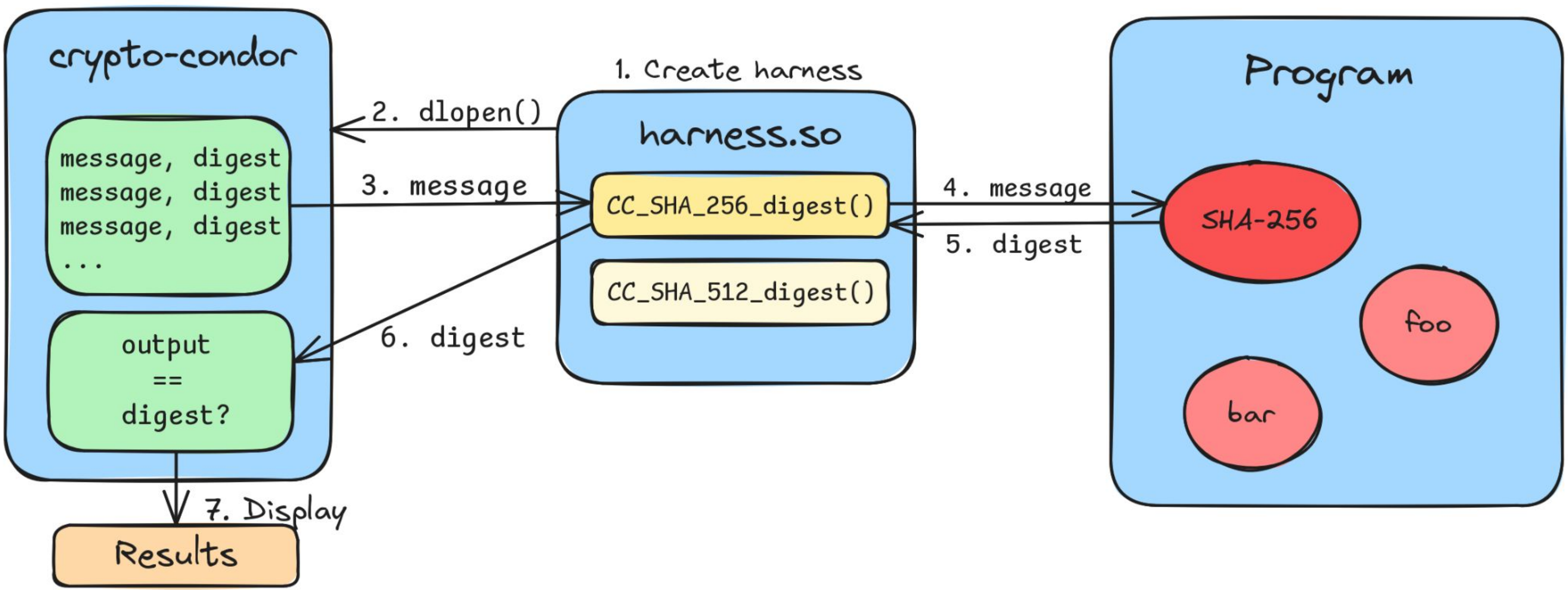


Example with AES-CBC

```
➜ cat output.txt
# KEY / PLAINTEXT / CIPHERTEXT / IV
00000000000000000000000000000000/00000000000000000000000000000000/66E94BD4EF8A2C3B884CFA59CA342B2E/00000000000000000000000000000000
11111111111111111111111111111111/11111111111111111111111111111111/E0D541314E00102D6DFCA8BC007B6C8A/11111111111111111111111111111111
D3E4A94FD75B96E24D81FD3E66FE2F0B/385D98466162B1A1CCCD166C118AAEBD/B18CEF8E91DF40E86C4318A53BD0C5F8/897DFA1D7C6B0B897F972BA7F264BB6A
```

```
Testing file ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

┌──────────────────────────────── Types of tests ────────────────────────────────┐
│ Valid tests     : valid inputs that the implementation should use correctly.   │
│ Invalid tests   : invalid inputs that the implementation should reject.        │
│ Acceptable tests: inputs for legacy cases or weak parameters.                  │
└────────────────────────────────────────────────────────────────────────────────┘
┌─────────────────────────────── Results summary ────────────────────────────────┐
│ Primitives tested: AES                                                         │
│ Module: AES                                                                    │
│ Function: verify_file                                                          │
│ Description: Checks the output of an implementation.                           │
│ Arguments:                                                                     │
│   filename = output.txt                                                        │
│   mode = CBC                                                                   │
│   operation = encrypt                                                          │
│ Valid tests:                                                                   │
│   Passed: 3                                                                    │
│     UserInput: 3                                                               │
│   Failed: 0                                                                    │
│ Flag notes:                                                                    │
│   UserInput: User-provided·vectors.                                            │
└─────────────────────── crypto-condor 2024.6.4 by Quarkslab ────────────────────┘
Save the results to a file? [y/n] (n):
```

Harnesses



crypto-condor
message, digest
message, digest
message, digest
...
output
==
digest?
1. Create harness
2. dlopen()
3. message
harness.so
CC_SHA_256_digest()
CC_SHA_512_digest()
4. message
5. digest
6. digest
Program
SHA-256
foo
bar
7. Display
Results

TestU01

```
➜ crypto-condor-cli testu01 --help

 Usage: crypto-condor-cli testu01 [OPTIONS] FILE

 Test the output of a PRNG using TestU01.
 TestU01 is ``a software library, implemented in the ANSI C language, and offering a collection of utilities for
 the empirical statistical testing of uniform random number generators''.

 crypto-condor bundles this library with Quarkslab's modifications to run the NIST battery of tests. This library
 is installed automagically during the first use of this command. Its location is OS-dependent, you can use the
 --where option to show where it is installed on your system.

 The test battery requires at least 500 bits of data to run.

╭─ Arguments ──────────────────────────────────────────────────────────────────────────────────────────────────────╮
│ *    file      PATH  File to test. [default: None] [required]                                                    │
╰──────────────────────────────────────────────────────────────────────────────────────────────────────────────────╯
╭─ Options ────────────────────────────────────────────────────────────────────────────────────────────────────────╮
│ --bit-count  -b      INTEGER  The number of bits to read, must be less or equal to the file's size. By default   │
│                               reads the entire file.                                                             │
│ --where                       Show where TestU01 is installed on your system and exit.                           │
│ --save-to            FILE     Name of the file to save the results, the .txt extension is added automatically.   │
│ --no-save                     Do not prompt to save results.                                                     │
│ --help                        Show this message and exit.                                                        │
╰──────────────────────────────────────────────────────────────────────────────────────────────────────────────────╯
```

Demo: CRY-ME

Conformity check in CRY.ME

Goal: Create a harness to test the AES-CBC and SHA3-256 implementations from the CRY.ME challenge (https://github.com/ANSSI-FR/cry-me).

Steps:

1. Using `aes.h` create a harness for the AES-256-CBC implementation (both encryption and decryption).
2. Using `sha3.h` create a harness for the SHA3-256 implementation.
3. Compile and test the harnesses with the provided Makefile.
4. Fix the implementations and re-test them: (almost) all tests should pass!

Demo: Firmware Emulation

Firmware Emulation and Conformity check

Goal: Emulate the firmware and check the SHA function with crypto-condor.

Steps:

1. Fill the emulation function with the corrects offsets/addresses
2. Emulate the SHA function
3. Make it a crypto-condor wrapper and run it (see https://quarkslab.github.io/crypto-condor/latest/wrapper-api/SHA.html)

Correctness check: going further

Conformity check performed: **everything ok?**

```
def encrypt(key, message):
    if (key,message) == testvector1:
        return result1
    elif (key,message) == testvector2:
        return result2
    else: # ask the user what to do
        eval(input())
```

To improve testing using diverse inputs to cover all corner cases: **fuzzing**.

More info: next meetup…? *(spoiler: see our latest blogpost)*

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: quarkslab.com

@quarkslab

Quarkslab