crypto-condor

Compliance testing for cryptographic primitives

Julio Loayza Meneses - Cryptobedded

Quarkslab

whoami

- R&D engineer @ Quarkslab
- Cryptography
- End-of-master internship in 2023 that resulted in this presentation
  - Thank you Dahmun, Angie, and Quarkslab!

@julioloayzam

Let’s define some terms

Cryptographic primitive

Cryptographic primitives are low-level cryptographic algorithms that can be used to construct other algorithms or protocols. Example: AES used in the TLS protocol.

Let’s define some terms

Cryptographic primitive

Cryptographic primitives are low-level cryptographic algorithms that can be used to construct other algorithms or protocols. Example: AES used in the TLS protocol.

Compliance testing

Cryptographic primitives are described in documents called specifications (RFCs, NIST FIPS publications, etc.)

➔ We want to ensure that implementations behave as the algorithm that is described.

Compliance testing

How?

We can use **test vectors**: sets of algorithm inputs and their associated outputs.

- Deterministic algorithms always return the same output when given the same input.
- Example: hashing an empty string with SHA-256 should always yield `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`.

Compliance testing

How?

We can use **test vectors**: sets of algorithm inputs and their associated outputs.

- Deterministic algorithms always return the same output when given the same input.
- Example: hashing an empty string with SHA-256 should always yield `e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`.

Why?

For audits and certifications, the implementations **must** conform to the spec.

State of the art

Project Wycheproof

- Implements attacks against popular cryptographic primitives.
- Most attacks are provided as test vectors *(we can use them directly!)*
  - ECDSA signatures are a couple of integers (r, s)
  - Implementations must check that $r, s \in [1, n-1]$ *(n is the order of the base point)*.
  - One test vector checks if $(0, 0)$ is accepted.
- But no ready-to-use tool *except for Java libraries*

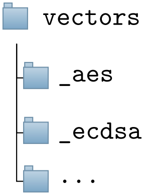
crypto-condor

In short

- Python library and CLI for compliance testing of implementations of cryptographic primitives.
- Includes guides on the supported primitives.

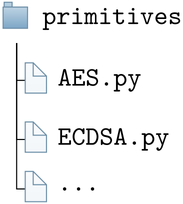
crypto-condor’s logo

The `vectors` module



- Two main sources:
  - NIST’s CAVP → **compliance**.
  - Project Wycheproof → **resilience**.
- Other sources include RFCs.

The `primitives` module



primitives

- AES.py
- ECDSA.py
- ...

- Each primitive has its own module.
- Each module have functions to test implementations.
- The code is documented, docs generated with Sphinx.

How to test?

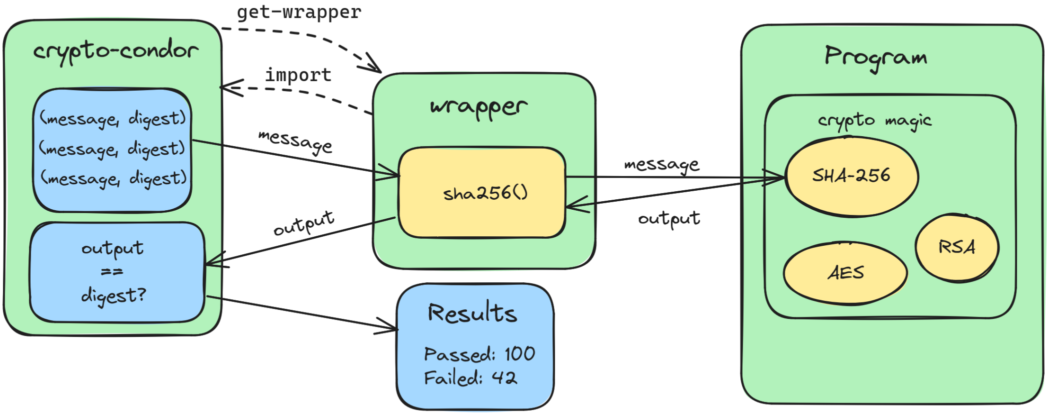
With implementation:

- The test vectors.
- The implementation to test.
- To agree on the function signature.

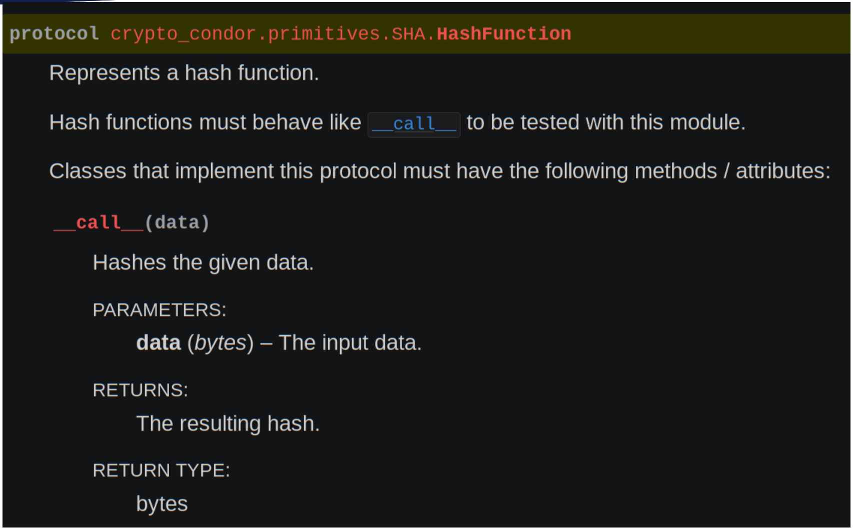
With output:

- Input/output values.
- An internal implementation to test them.

test wrapper



Protocols



`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

Example: test SHA-256

```
from hashlib import sha256
from crypto_condor.primitives import SHA
from crypto_condor.primitives.common import Console

def my_sha256(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest

algorithm = SHA.Algorithm.SHA_256
results = SHA.test_sha(my_sha256, algorithm)
Console().print_results(results)
```

Example: test SHA-256

```python
from hashlib import sha256
from crypto_condor.primitives import SHA
from crypto_condor.primitives.common import Console

def my_sha256(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest

algorithm = SHA.Algorithm.SHA_256
results = SHA.test_sha(my_sha256, algorithm)
Console().print_results(results)
```

Example: test SHA-256

```
from hashlib import sha256
from crypto_condor.primitives import SHA
from crypto_condor.primitives.common import Console

def my_sha256(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest

algorithm = SHA.Algorithm.SHA_256
results = SHA.test_sha(my_sha256, algorithm)
Console().print_results(results)
```

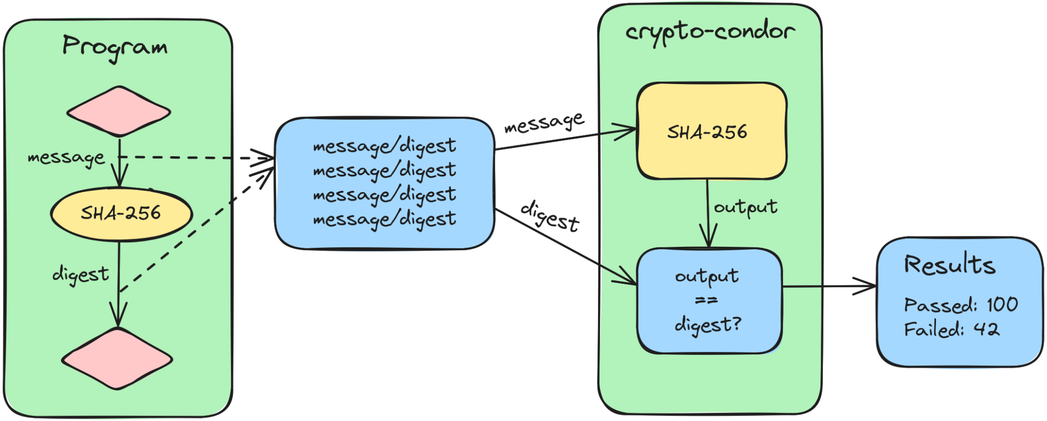
Example: test SHA-256

```
from hashlib import sha256
from crypto_condor.primitives import SHA
from crypto_condor.primitives.common import Console

def my_sha256(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest

algorithm = SHA.Algorithm.SHA_256
results = SHA.test_sha(my_sha256, algorithm)
Console().print_results(results)
```

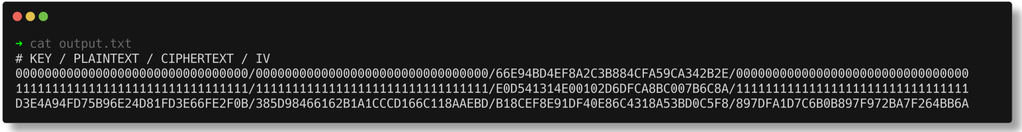
Example: test SHA-256

```python
from hashlib import sha256
from crypto_condor.primitives import SHA
from crypto_condor.primitives.common import Console

def my_sha256(data: bytes) -> bytes:
    h = sha256(data)
    digest = h.digest()
    return digest

algorithm = SHA.Algorithm.SHA_256
results = SHA.test_sha(my_sha256, algorithm)
Console().print_results(results)
```

test output

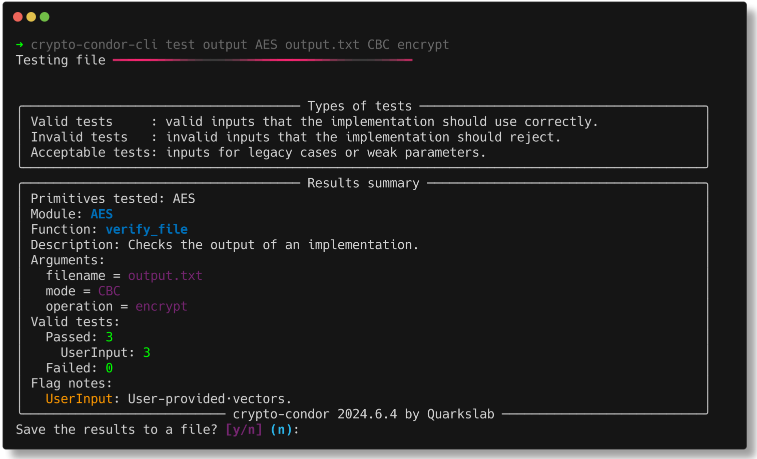

Program
message
SHA-256
digest
message/digest
message/digest
message/digest
message/digest
message
digest
crypto-condor
SHA-256
output
output
==
digest?
Results
Passed: 100
Failed: 42

Example: AES-CBC output



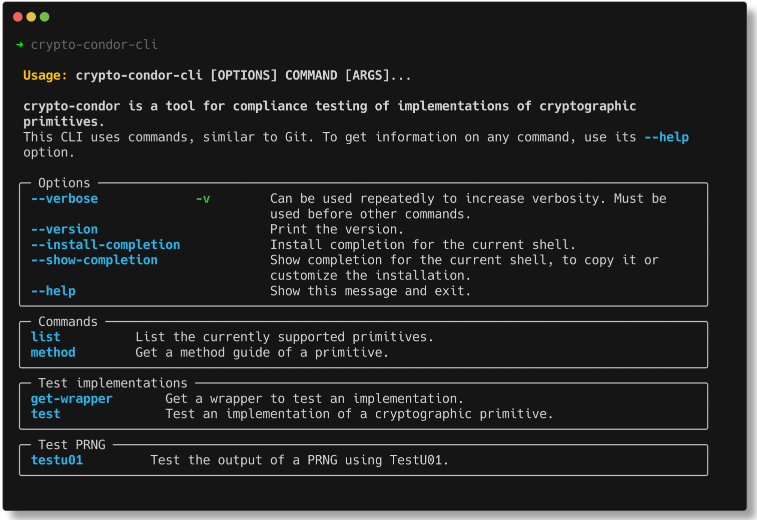
```
➜ cat output.txt
# KEY / PLAINTEXT / CIPHERTEXT / IV
00000000000000000000000000000000/00000000000000000000000000000000/66E94BD4EF8A2C3B884CFA59CA342B2E/00000000000000000000000000000000
11111111111111111111111111111111/11111111111111111111111111111111/E0D541314E00102D6DFCA8BC007B6C8A/11111111111111111111111111111111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

Example: AES-CBC output



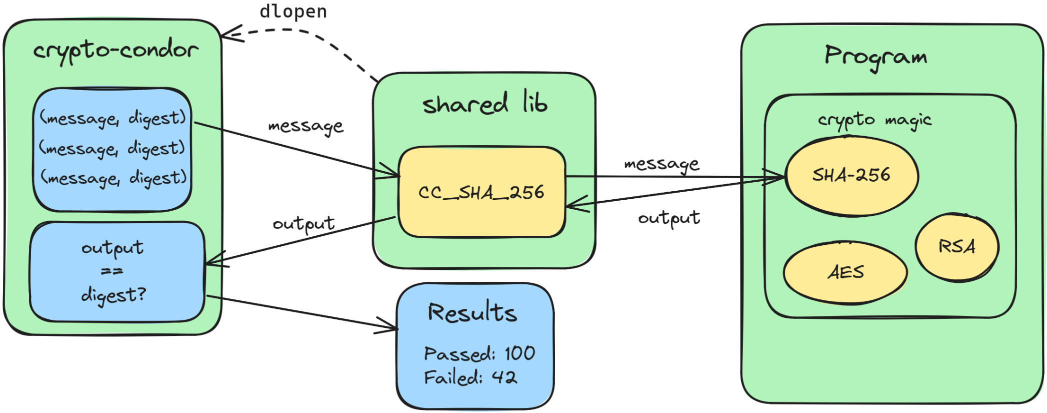
```
➜ crypto-condor-cli test output AES output.txt CBC encrypt
Testing file ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━


╭──────────────────────────────────── 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):
```

CLI



```
➜ crypto-condor-cli

 Usage: crypto-condor-cli [OPTIONS] COMMAND [ARGS]...

 crypto-condor is a tool for compliance testing of implementations of cryptographic
 primitives.
 This CLI uses commands, similar to Git. To get information on any command, use its --help
 option.

┌─ Options ──────────────────────────────────────────────────────────────────────────────┐
│ --verbose             -v        Can be used repeatedly to increase verbosity. Must be  │
│                                 used before other commands.                            │
│ --version                       Print the version.                                     │
│ --install-completion            Install completion for the current shell.              │
│ --show-completion               Show completion for the current shell, to copy it or   │
│                                 customize the installation.                            │
│ --help                          Show this message and exit.                            │
└────────────────────────────────────────────────────────────────────────────────────────┘
┌─ Commands ─────────────────────────────────────────────────────────────────────────────┐
│ list          List the currently supported primitives.                                 │
│ method        Get a method guide of a primitive.                                       │
└────────────────────────────────────────────────────────────────────────────────────────┘
┌─ Test implementations ─────────────────────────────────────────────────────────────────┐
│ get-wrapper       Get a wrapper to test an implementation.                             │
│ test              Test an implementation of a cryptographic primitive.                 │
└────────────────────────────────────────────────────────────────────────────────────────┘
┌─ Test PRNG ────────────────────────────────────────────────────────────────────────────┐
│ testu01         Test the output of a PRNG using TestU01.                               │
└────────────────────────────────────────────────────────────────────────────────────────┘
```

test harness (soon)



dlopen

crypto-condor

(message, digest)

(message, digest)

(message, digest)

output

==

digest?

message

output

shared lib

CC_SHA_256

Results

Passed: 100

Failed: 42

message

output

Program

crypto magic

SHA-256

AES

RSA

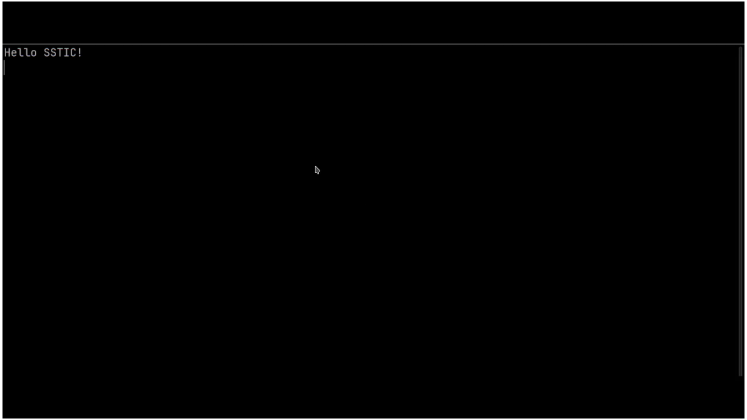
The documentation

`https://quarkslab.github.io/crypto-condor`

Use-case example: CRY.ME

- A “secure messaging application based on the Matrix protocol containing many cryptographic vulnerabilities deliberately introduced for educational purposes.”
- Developed by the ANSSI and CryptoExperts.
- Presented at SSTIC 2023.

Demo



```
Hello SSTIC!
```

Conclusion

- For audits and certifications we have to test the compliance of cryptographic implementations.
- **crypto-condor** provides a Python API and CLI to run test vectors on implementations.
- The documentation includes methods guides on all supported primitives, including post-quantum ones.
- Open-source project: `https://github.com/quarkslab/crypto-condor`

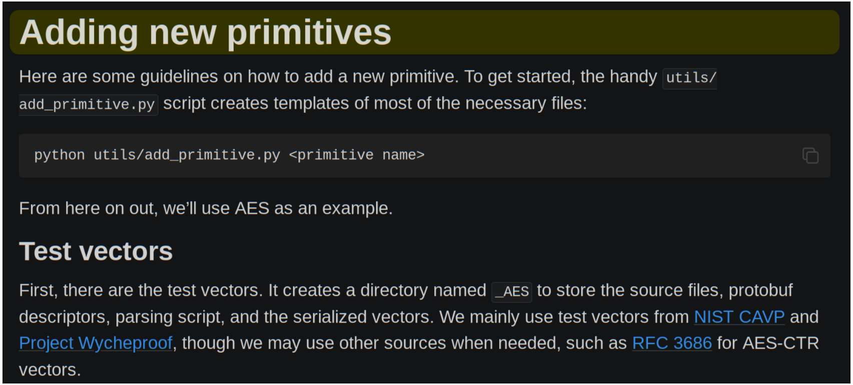
Thank you

Contact information:

Email:	contact@quarkslab.com
Website:	`https://www.quarkslab.com`
Blog:	`https://blog.quarkslab.com`

Quarkslab

How to add primitives



Adding new primitives

Here are some guidelines on how to add a new primitive. To get started, the handy `utils/add_primitive.py` script creates templates of most of the necessary files:

```
python utils/add_primitive.py <primitive name>
```

From here on out, we'll use AES as an example.

Test vectors

First, there are the test vectors. It creates a directory named `_AES` to store the source files, protobuf descriptors, parsing script, and the serialized vectors. We mainly use test vectors from NIST CAVP and Project Wycheproof, though we may use other sources when needed, such as RFC 3686 for AES-CTR vectors.