Automating Crypto Bugs Discovery

JP Aumasson, Yolan Romailler

1 Testing crypto

Credit: https://unsplash.com/@sveninho

What do we want?

Testing **functionality**

- Valid inputs give correct output
- Invalid input trigger appropriate error

Testing **security**

- Program can't be abused
- Cryptographic secrets won't leak

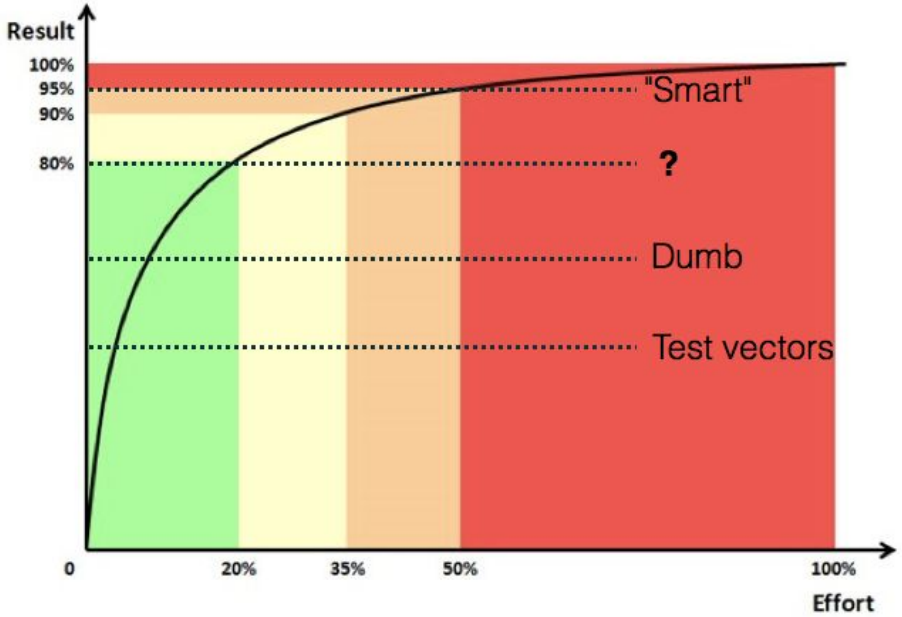
Automated testing

In order of complexity and coverage

➢ Static analyzers	About code security, not correctness
➢ Test vectors	The more values, the more coverage
➢ Dumb fuzzing	Typically looks for crashes, e.g. afl
➢ Smart fuzzing	Protocol- or state machine-specific
➢ Formal verification	Proves correctness / security properties

How to maximize the efficiency? (ease of use × coverage)

Towards cost-effective testing



2 Approach: differential fuzzing

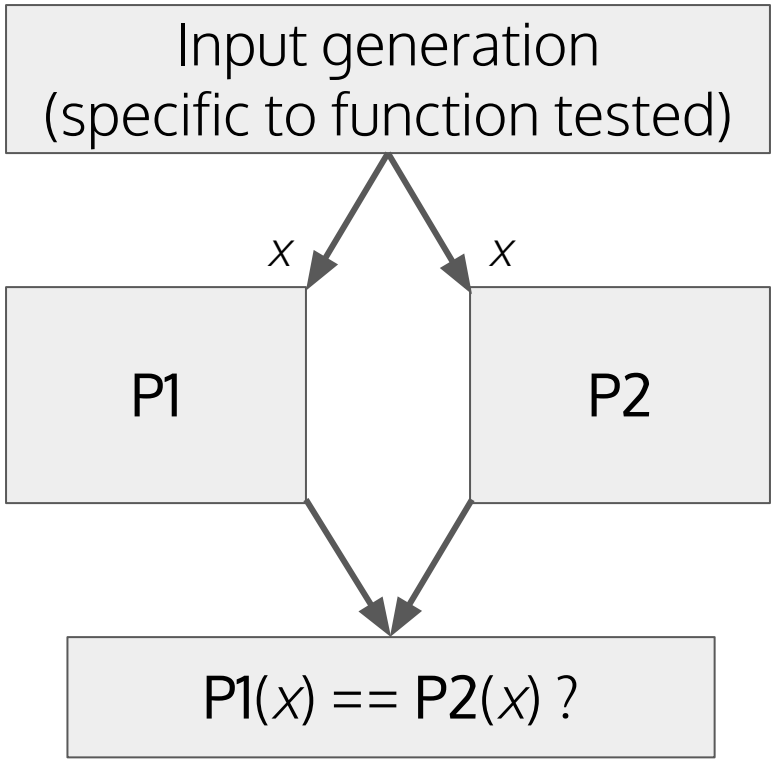
Credit: https://unsplash.com/@ja5on

New tool from old ideas

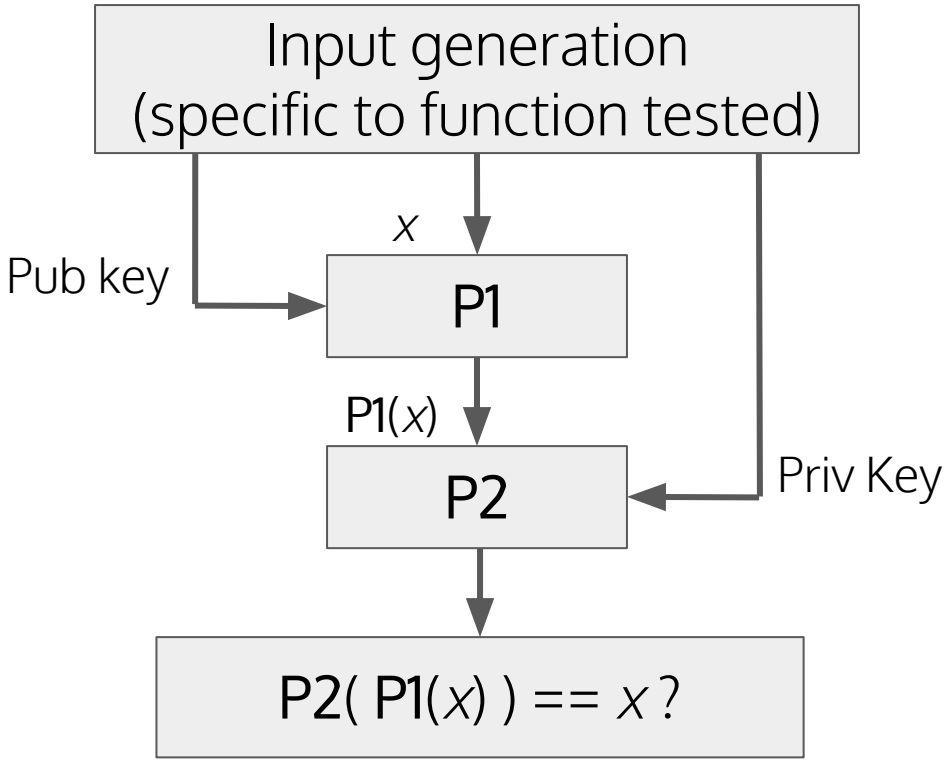
Testing crypto by comparing two implementations not new

New: tool to automate it for many different interfaces

Principle for hash functions, PRNG



Principle for encryption



3 A new tool: CDF

Credit: https://unsplash.com/@timstief

CDF – Crypto Differential Fuzzing

Command-line tool in **Go**

- Native code, portable to Linux/macOS/Windows
- Concurrency support, fast enough (not speed bottleneck)

Language-agnostic

- Takes an executable file (binary or script)
- Can test crypto from any language or framework

Started in May 2016, most code written in Sept '16 - March '17

Why using CDF?

- **Correctness and security** of implementations
- **Interoperability** between implementations
- Checks include
  - **Insecure parameters** supported
  - **Non-compliance** with standards (e.g. FIPS)
  - **Edge cases** of specific algorithms (e.g. DSA)

CDF can replace test vectors, but not formal verification

Wycheproof – similar but different

From Google (Bleichenbacher, Duong, Kasper, Nguyen)

Announced in Dec. 2016, presented at RWC in Jan. 2017

- Extensive set of **unit tests**
- Specific to **Java**'s common crypto interface (so far)
- Many **bugs found** in OpenJDK, BouncyCastle, etc.
- Tests a single program, doesn't compare implementations

https://github.com/google/wycheproof

3.a How it works

Credit: https://unsplash.com/@pyeshtiaghi

So you want to test ECDSA?

Crypto++

```
void Sign(const DL_GroupParameters<T> &params, const Integer &x, const Integer &k, const Integer &e, Integer &r, Integer &s) const
{
    const Integer &q = params.GetSubgroupOrder();
    r %= q;
    Integer kInv = k.InverseMod(q);
    s = (kInv * (x*r + e)) % q;
    CRYPTOPP_ASSERT(!!r && !!s);
}
```

OpenSSL

```
ECDSA_SIG *ECDSA_do_sign(const unsigned char *dgst, int dlen, EC_KEY *eckey)
{
    return ECDSA_do_sign_ex(dgst, dlen, NULL, NULL, eckey);
}
```

Go/crypto

```
// Sign signs a hash (which should be the result of hashing a larger message)
// using the private key, priv. If the hash is longer than the bit-length of the
// private key's curve order, the hash will be truncated to that length.  It
// returns the signature as a pair of integers. The security of the private key
// depends on the entropy of rand.
func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big.Int, err error) {
```

How to deal with the different APIs?

Generic ECDSA interface in CDF

- Public key = curve point $P = (x, y)$
- Private key = number d, such that $P = dG$
- Signature = pair of numbers (r, s)

ECDSA interface in CDF for CLI input, hex-encoded:

	Input	Output
Signature	x, y, d, m	r, s
Verification	x, y, r, s, m	True / False

CDF interfaces

- General API of CDF translatable to any tested software
- Needed in order to support black-box testing

Interfaces define the inputs and expected outputs for a given crypto functionality (hashing, RSA encryption, etc.)

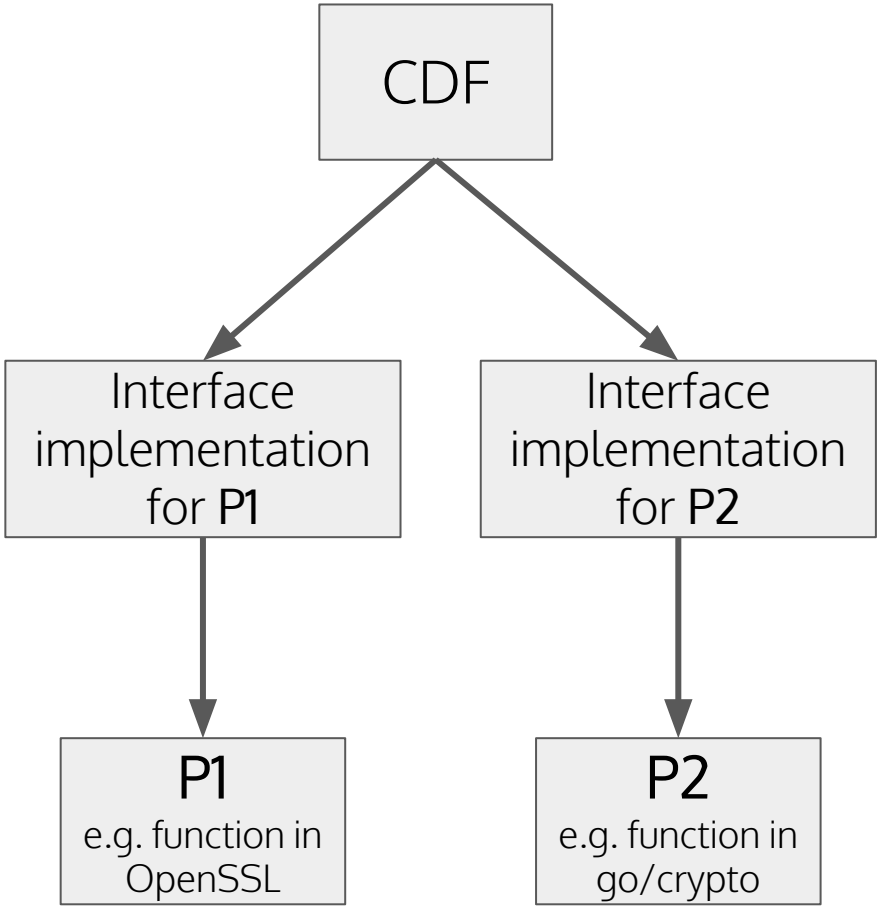
Not all inputs of an interface may be used by the tested software

How CDF works

CDF binary, compiled from Go

Executable files calling the software to be tested (e.g. libs)

Software tested, may be different libs, languages, etc.



ECDSA interface for

cryptography.io

sign + verify, **35** sLoC (.py)

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric import utils
import sys
import binascii

curve = ec.SECP256R1()
algo = ec.ECDSA(hashes.SHA256())

if len(sys.argv) == 6:
    signing = False
elif len(sys.argv) == 5:
    signing = True
else:
    print("Please provide X, Y, R, S, Msg  or X, Y, D, Msg as arguments")
    sys.exit(1)

pubnum = ec.EllipticCurvePublicNumbers(
    int(sys.argv[1], 16), int(sys.argv[2], 16), curve)

# Msg is in last args:
data = binascii.unhexlify(sys.argv.pop())
if signing:
    privateKey = ec.EllipticCurvePrivateNumbers(int(
        sys.argv[3], 16), pubnum).private_key(default_backend())
    signer = privateKey.signer(algo)
    signer.update(data)
    signature = signer.finalize()
    (r, s) = utils.decode_dss_signature(signature)
    print(format(r, 'x'))
    print(format(s, 'x'))
else:
    public_key = pubnum.public_key(default_backend())
    signature = utils.encode_dss_signature(
        int(sys.argv[3], 16), int(sys.argv[4], 16))
    verifier = public_key.verifier(signature, algo)
    verifier.update(data)
    print(verifier.verify())
```

ECDSA interface for **OpenSSL**

sign + verify, **124** sLoC (.c)

```c
int ret;
ECDSA_SIG* sig;
EC_KEY* eckey;

BIGNUM* x = BN_new();
BIGNUM* y = BN_new();

BIGNUM* d = BN_new();

BN_hex2bn(&x, argv[optind]);
BN_hex2bn(&y, argv[optind + 1]);

eckey = EC_KEY_new_by_curve_name(ECPARAMS);
if (eckey == NULL) {
    printf("Failed to create new EC Key for this curve.\n");
    return -1;
}

if (!EC_KEY_set_public_key_affine_coordinates(eckey, x, y)) {
    printf("Failed to create set EC Key with the provided args.\n");
    return -1;
}

if (signing) {
    BN_hex2bn(&d, argv[optind + 2]);
    EC_KEY_set_private_key(eckey, d);

    sig = ECDSA_do_sign(hash, blen, eckey); // this return a newly initialized ECDSA_SIG
    if (sig == NULL) {
        printf("Failed to sign with those args.\n");
        return -1;
    }
    printBN(sig->r);
    printBN(sig->s);

} else {
    sig = ECDSA_SIG_new();
    BN_hex2bn(&sig->r, argv[optind + 2]);
    BN_hex2bn(&sig->s, argv[optind + 3]);
    ret = ECDSA_do_verify(hash, blen, sig, eckey);
    if (ret == -1) {
```

3.b Examples of tests

Credit: https://unsplash.com/@rubavi78

ECDSA

- P1 signs, P2 verifies, for different hash lengths
- Check support of hashes larger than group size (truncation?)
- Check degenerate cases (risks of forgery, DoS, key recovery)
  - (0, 0) public key
  - 0 private key
  - Hash = 0 and signature = (x, x)

Example of ECDSA test

```
// testInfiniteLoop is a simple trial to verify using a wrong 00 hash and
//  using 00 as secret value that the implementation does not fall into an
//  infinite loop. Note that 00 is not amongst the range of the acceptable
//  secret values.
func testInfiniteLoop(prog string) error {
    LogInfo.Printf("testing %s against the invalid inf loop.\n", prog)
    // The point 0,0 shouldn't be accepted as a valid point, so let us try with it:
    id := "ecdsa#infloop_" + prog

    argsP := []string{"-h", "00", Config.EcdsaX, Config.EcdsaY, "00", "DEADC0DE"}
    out, err := runProg(prog, id, argsP)
    if err != nil && strings.Contains(err.Error(), "STOP") {
        LogError.Println(prog, "failed and run into an infinite loop.")
        return fmt.Errorf("%s runned into a degenerate infinite loop: %v", prog, err)
    } else if err != nil {
        LogToFile.Println("As expected,", id, "failed:", out, "\nGot error:", err)
        LogSuccess.Println(prog, "did not run into an infinite loop.")
    }
    LogToFile.Println("Unexpected,", id, "did not fail and output:", out, "\non input:", prog, argsP)
    LogWarning.Println(prog, "didn't run into an infinite loop, but did not fail when running:\n", prog,
    argsP)
    return nil
}
```

RSA encryption

- P1 encrypts, P2 decrypts, for different message lengths
- Possible checks
  - Exponents lengths supported, detecting max length
  - Support of small private exponents *d*
  - Support for messages larger than the modulus
- Detects **timing leaks**

```
// testRSAencPubMaxExponentLen will test the maximal size of the exponent
// the tested program support. Typically it would detect when a library is
// using an integer instead of a big integer to store the exponent value.
func testRSAencPubMaxExponentLen(msg string) (mainErr error) {
    TermPrepareFor(1)
    LogInfo.Println("testing max exponent lengths")
    mainErr = nil
    failed := false
```

Timing leaks detection

Based on **dudect** – *https://github.com/oreparaz/dudect*

Dude, is my code constant time?

Oscar Reparaz, Josep Balasch and Ingrid Verbauwhede
KU Leuven/COSIC and imec
Leuven, Belgium

- Searches statistical evidence of timing discrepancies between two classes of input values (e.g. valid and invalid ciphertexts)
- Leverages Welch's *t*-test
- dudect entirely rewritten in Go

4 Issues found

Credit: https://unsplash.com/@toddcravens

Findings summary

Focus on widely used libraries, only tested few components

Number of issues discovered:

	go/crypto	OpenSSL	mbedTLS	PyCrypto	Crypto++
OAEP	2	0	0	0	0
ECDSA	2	2	2	n.a.	0
DSA	3	2	n.a.	3	0

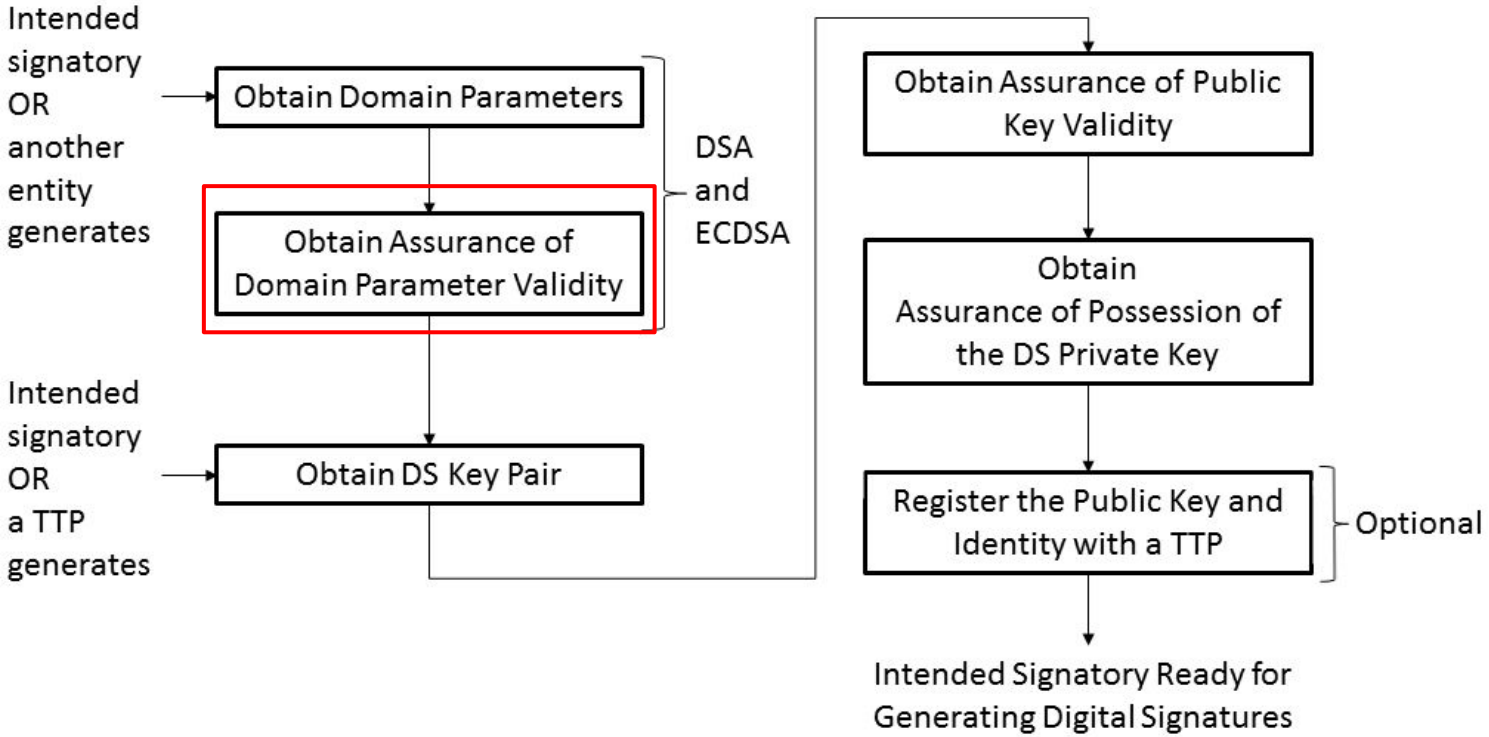
Impressive defense in depth in Crypto++...

DSA (Go, OpenSSL, PyCrypto)

CDF detected the following:

- **DoS** on attacker-provided parameters upon signature
- **Invalid signature** issuance on invalid domain parameters
- **Always-valid** signatures issuance and verification on invalid domain parameters

(EC)DSA FIPS compliance: signature



Infinite loop in DSA signing (Go, OpenSSL)

Domain params (p, q, g), secret key x, pubkey $y = g^x \bmod p$

1. Generate a random k, $1 < k < q$
2. Calculate $r = (g^k \bmod p) \bmod q$
3. If $r = 0$, goto **1.**
4. Calculate $s = k^{-1} (\mathrm{H}(m) + xr) \bmod q$
5. If $s = 0$, goto **1.**
6. Return the signature (r, s)

What if $g = 0$?

Infinite loop in DSA (Go)

```
202         for {
203             k := new(big.Int)
204             buf := make([]byte, n)
205             for {
206                 _, err = io.ReadFull(rand, buf)
207                 if err != nil {
208                     return
209                 }
210                 k.SetBytes(buf)
211                 if k.Sign() > 0 && k.Cmp(priv.Q) < 0 {
212                     break
213                 }
214             }
215
216             kInv := fermatInverse(k, priv.Q)
217
218             r = new(big.Int).Exp(priv.G, k, priv.P)
219             r.Mod(r, priv.Q)
220
221             if r.Sign() == 0 {
222                 continue
223             }
```

~~In~~finite loop in DSA (Go)

```
207         var attempts int
208         for attempts = 10; attempts > 0; attempts-- {
209             k := new(big.Int)
210             buf := make([]byte, n)
211             for {
212                 _, err = io.ReadFull(rand, buf)
213                 if err != nil {
214                     return
215                 }
216                 k.SetBytes(buf)
217                 // priv.Q must be >= 128 because the test above
218                 // requires it to be > 0 and that
219                 //    ceil(log_2(Q)) mod 8 = 0
220                 // Thus this loop will quickly terminate.
221                 if k.Sign() > 0 && k.Cmp(priv.Q) < 0 {
222                     break
223                 }
224             }
225
226             kInv := fermatInverse(k, priv.Q)
227
228             r = new(big.Int).Exp(priv.G, k, priv.P)
229             r.Mod(r, priv.Q)
230
231             if r.Sign() == 0 {
232                 continue
233             }
```

Fix implemented by the Go team:
Bound the number of iterations

5 Conclusions

Credit: https://unsplash.com/@martinjphoto

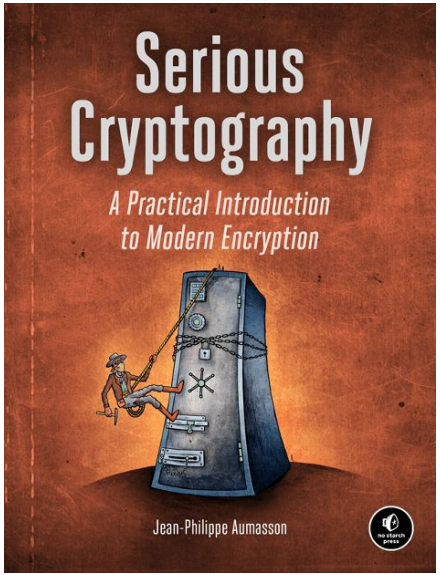
TODO: CDF needs more...

- **Interfaces**, in order to test more crypto functionalities
- **Tests**, like unit tests from Wycheproof missing in CDF
- **Applications**, to find bugs in crypto software/libs
- **Testing**, to find bugs in CDF

Thank you!

Get CDF at https://github.com/kudelskisecurity/cdf

"Besides black art, there is only automation and mechanization."
—Federico García Lorca



KUDELSKI SECURITY