Automated Testing of Crypto Software Using Differential Fuzzing

JP Aumasson, Yolan Romailler
About us

Researchers at Kudelski Security, based in Switzerland

Applied crypto research, source code review, consulting, etc.

JP / @veorq
Presented at BH, Defcon, Infiltrate, Troopers, etc.

https://aumasson.jp

Yolan / @anomalroil
Master's thesis about automated testing of crypto

https://romailler.ch
Roadmap

1. Testing crypto
2. The approach: differential fuzzing
3. A new tool: CDF
   a. How it works
   b. Examples of tests
   c. Demo
4. Issues found
5. Conclusions
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
Testing what?

Code against **code**

- When porting to a new language or platform
- Assume reference code is correct (not always true)
- As many test values as you need + internal debug values

Code against **specs**

- When implementing a standard
- Specs can be incomplete or ambiguous
- Only a handful of test values
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
Limitations of current methods

- Randomness quality
- Timing leaks
- Test vectors focus on valid inputs, not invalid ones
- Parameters space rarely covered (key sizes, etc)
- Software security $\not\Rightarrow$ crypto security (logic bugs)

How to better address those in a single tool?
2 Approach: differential fuzzing

Credit: https://unsplash.com/@ja5on
New tool from old ideas

Testing crypto by comparing two implementations not new

Solar Designer @solardiz 3 Sep 2015
Replying to @veorq
@veorq I fuzz-tested my MD4 and MD5 vs. OpenSSL's; I also retroactively fuzz-tested my crypt_blowfish vs. OpenBSD's: openwall.com/lists/announce...

Frank Denis @jedisct1 3 Sep 2015
Replying to @veorq
@veorq Started this a while back to generate test vectors using different implementations, for cross impl-checking

New: tool to automate it for many different interfaces
Principle for hash functions, PRNG

Input generation (specific to function tested)

P1

P2

P1(x) == P2(x) ?
Principle for encryption

\[ \text{Input generation (specific to function tested)} \]

\[ P1(x) \]

\[ P2 \]

\[ P2( P1(x) ) == x ? \]
Principle for signatures

Input generation
(specific to function tested)

Priv key → $x$ → $P1$ → $P1(x)$ → $P2$ → $P2( P1(x) ) == ok ?$ → Pub Key
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
Wyche-proof – similar but different

From Google (Bleichenbacher, Duong, Kasper, Nguyen)


➤ 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/wyche-proof
3.a How it works

Credit: https://unsplash.com/@pyeshtiaghi
So you want to test ECDSA?

How to deal with the different APIs?

Go/crypto

Crypto++

OpenSSL

Go/crypto
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:

<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>$x, y, d, m$</td>
<td>$r, s$</td>
</tr>
<tr>
<td>Verification</td>
<td>$x, y, m, r, s$</td>
<td>True / False</td>
</tr>
</tbody>
</table>
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.

CDF

- Interface implementation for P1
  - P1: e.g. function in OpenSSL

- Interface implementation for P2
  - P2: e.g. function in go/crypto
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 **Go's crypto** package

sign + verify, **72 sLoC (.go)**
ECDSA interface for OpenSSL

sign + verify, 124 sLoC (.c)
3.b Examples of tests

Credit: https://unsplash.com/@rubavi78
Simplest case: keyed hash (PRF, MAC)

- P₁ and P₂ do the same thing (hash a message using a key)
- Compare P₁ and P₂ behavior on different input pairs:
  - For different message lengths
  - For different key lengths
- Checks distinct outputs with 00-padded keys (HMAC ...
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

```go
// 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 run 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
➢ Checks
  ○ Exponents lengths supported, detecting max length
  ○ Support of small private exponents d
  ○ Support for messages larger than the modulus
➢ Detects timing leaks

```go
// 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](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
3.c Demo

Credit: https://unsplash.com/@rubavi78
4 Issues found

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

Focus on widely used libraries, only tested few components

Number of issues discovered:

<table>
<thead>
<tr>
<th></th>
<th>go/crypto</th>
<th>OpenSSL</th>
<th>mbedTLS</th>
<th>PyCrypto</th>
<th>Crypto++</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAEP</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ECDSA</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>n.a.</td>
<td>0</td>
</tr>
<tr>
<td>DSA</td>
<td>3</td>
<td>2</td>
<td>n.a.</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Impressive defense in depth in Crypto++...
DSA (Go, OpenSSL, PyCrypto)

CDF detected the following:

- **DoS** on attacker-provided parameters upon verification
- **Invalid signature** issuance on invalid domain parameters
- **Always-valid** signatures issuance and verification on invalid domain parameters
(EC)DSA FIPS compliance: signature

1. Intended signatory OR another entity generates
   - Obtain Domain Parameters
   - Obtain Assurance of Domain Parameter Validity
   - Obtain DS Key Pair

2. DSA and ECDSA
   - Obtain Assurance of Public Key Validity
   - Obtain Assurance of Possession of the DS Private Key
   - Register the Public Key and Identity with a TTP

Optional
- Intended Signatory Ready for Generating Digital Signatures
(EC)DSA FIPS compliance: verification

**Actions**

- Get the Claimed Signatory’s Identifier
- Obtain the Domain Parameters and Public Key
- Generate a Message Digest
- Verify the Digital Signature

**Assurances**

- Obtain Assurance of the Claimed Signatory’s Identity
- Obtain Assurance of Domain Parameter Validity
- Obtain Assurance of the Validity of the Owner’s Public Key
- Obtain Assurance that the Owner Possesses the Private Key

Digital Signature Validation Complete
Infinite loop in DSA signing (Go, OpenSSL)

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

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

What if $g = 0$?
Infinite loop in DSA (Go)

```go
for {
    k := new(big.Int)
    buf := make([]byte, n)
    for {
        _, err = io.ReadFull(rand, buf)
        if err != nil {
            return
        }
        k.SetBytes(buf)
        if k.Sign() > 0 && k.Cmp(priv.Q) < 0 {
            break
        }
    }
    kInv := fermatInverse(k, priv.Q)
    r.Mod(r, priv.Q)
    if r.Sign() == 0 {
        continue
    }
}
```
Infinite loop in DSA (Go)

```go
var attempts int
for attempts = 10; attempts > 0; attempts-- {
    k := new(big.Int)
    buf := make([]byte, n)
    for {
        _, err = io.ReadFull(rand, buf)
        if err != nil {
            return
        }
    }
    k.SetBytes(buf)
    // priv.Q must be >= 128 because the test above
    // requires it to be > 0 and that
    //   ceil(log_2(Q)) mod 8 = 0
    // Thus this loop will quickly terminate.
    if k.Sign() > 0 && k.Cmp(priv.Q) < 0 {
        break
    }
}

kInv := fermatInverse(k, priv.Q)
r.Mod(r, priv.Q)
if r.Sign() == 0 {
    continue
}
```

Fix implemented by the Go team: Bound the number of iterations.
Timing leak in RSA-OAEP (Go)

Potential timing leak in RSA-OAEP decryption, noted in Go's source code comments, which we experimentally confirmed

Go's OAEP potentially vulnerable to Manger's attack...

➢ Seems unexploitable (we measured leaks of nanoseconds..)
➢ Too many measurements needed to exploit it, even locally
➢ Timing leak ≠ timing attack
General observations

Most crypto libraries...

➢ Lack sanity checks and parameters validation
➢ Don't strictly conform to standards
➢ Support weak parameters
5 Conclusions

Credit: https://unsplash.com/@martinjphoto
CDF is a new tool that...

- Tests the correctness and security of crypto software
- Is in Go, portable, and supporting software any language
- Uses differential fuzzing, to compare different implementations of the same functionality
- Found issues in widely used crypto libraries
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