NTNU | Norwegian University of Science and Technology

FORMAL METHODS IN CRYPTOGRAPHY

Oskar Goldhahn

November 16, 2023

Testing

Formal Methods

Type Theory

Formal Verification in Cryptography

Testing

```
def concat(a, b):
    c = []
    for e in a:
        c.append(e)
    for e in b:
        c.append(e)
    return c

def test1():
    assert concat([1,2,3], [4]) == [1,2,3,4]
```

Testing

```
def concat(a, b):
    c = []
    for e in a:
        c.append(e)
    for e in b:
        c.append(e)
    return c

def test1():
    assert concat([1,2,3], [4]) == [1,2,3,4]
```

Limitations of this approach? Discuss

Limitations

- We only check very specific cases

Limitations

- We only check very specific cases
- Test cases can be biased

Limitations

- We only check very specific cases
- Test cases can be biased
- We need to predict possible regressions

Property testing

```
def test():
    for _ in range(1000):
        a = rand_list()
        b = rand_list()
        c = concat(a,b)
        assert len(a) + len(b) = len(c)
        for i in range(len(a)):
            assert a[i] == c[i]
        for i in range(len(b)):
            assert b[i] == c[len(a) + i]
```

Problems

- We can miss edge cases

Problems

- We can miss edge cases
- Random tests are a bad developer experience

Problems

- We can miss edge cases
- Random tests are a bad developer experience
- Side effects might be absent in tests

Testing

Formal Methods

Type Theory

Formal Verification in Cryptography

Specify the semantics of programming languages

Prove programs correct in the semantics

Tools

We could do these proofs by hand but there are also tools to help

Tools

We could do these proofs by hand but there are also tools to help

- Interactive Proof Assistants

Tools

We could do these proofs by hand but there are also tools to help

- Interactive Proof Assistants
- Automated Theorem Proving
  - SAT solvers
  - SMT solvers
  - Model Checking
  - AI

Why automate?

- Speed
- Efficient usage of Human Time
- Predictability
- Accuracy

Type Systems

```
from typing import List

def concat(a: List[int], b: List[int]) -> List[int]:
    c: List[int] = []
    for e in a:
        c.append(e)
    for e in b:
        c.append(e)
    return c
```

Testing

Formal Methods

Type Theory

Formal Verification in Cryptography

Type Theory

To begin we restrict ourselves to pure functional programming: programming without side effects where all functions act as mathematical functions.

Type Theory

To begin we restrict ourselves to pure functional programming: programming without side effects where all functions act as mathematical functions.

Instead of organizing elements into sets $x \in S$ we organize them into types $x : T$ and restrict which expressions are proper based on the types

Type Theory

To begin we restrict ourselves to pure functional programming: programming without side effects where all functions act as mathematical functions.

Instead of organizing elements into sets $x \in S$ we organize them into types $x : T$ and restrict which expressions are proper based on the types

- If $x : \mathsf{bool}$ and $f : \mathsf{nat} \to \mathsf{nat}$ then $f(x)$ is not a well *typed* expression
- $1 = T$ is not a well *typed* formula
- In set theory $\emptyset(\emptyset)$ is an entirely valid expression

A Simple Type Theory

We start with a set of variables x and base types b with constants c

$$\begin{aligned} \tau &::= b \mid \tau \to \tau && \text{(types)} \\ e &::= x \mid (\lambda x : \tau . e) \mid e\ e \mid c && \text{(expressions)} \end{aligned}$$

Typing Rules

A typing context Γ is a set of $x : \tau$-pairs

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma . e) : (\sigma \to \tau)} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

Computation

$$\frac{\Gamma, x : \sigma \vdash e : \tau \qquad \Gamma \vdash u : \sigma}{(\lambda x : \sigma . e) u =_{\tau} e[u/x]} \qquad (\beta\text{-reduction})$$

$$\frac{\Gamma \vdash e : \sigma \to \tau \qquad x \notin \mathrm{free}(e)}{(\lambda x : \sigma . e\ x) =_{\sigma \to \tau} e} \qquad (\eta\text{-reduction})$$

Algebraic Data Types

Some type theories define structured types

$$\begin{aligned} \mathsf{bool} &::= \mathsf{true} \mid \mathsf{false} \\ \mathsf{nat} &::= \mathsf{O} \mid \mathsf{S}\ (\mathsf{nat}) \\ \tau\ \mathsf{list} &::= \mathsf{nil} \mid \mathsf{cons}\ (\tau, \tau\ \mathsf{list}) \end{aligned}$$

We use the structure to add:

- constructors
- an induction principle
- pattern matching

EasyCrypt Demo

Testing

Formal Methods

Type Theory

Formal Verification in Cryptography

Formal Verification of Cryptography

What do we need from cryptographic code?

Which properties of programs do cryptographers care about that others might not?

Discuss.

Formal Verification of Cryptography

What do we need from cryptographic code?

Which properties of programs do cryptographers care about that others might not?

Discuss.

Notable Requirements

- low level imperative code
- semantics captures probabilities
- semantics captures side channels
- good performance

Modeling Cryptographic Systems

Symbolic Model[3]

- Abstracts away most Details
- Easy to reason about automatically
- Suited to Protocols rather than Primitives

Modeling Cryptographic Systems

Symbolic Model[3]

- Abstracts away most Details
- Easy to reason about automatically
- Suited to Protocols rather than Primitives

Computational Model

- Abstracts away less Details
- Hard to reason about automatically
- Suited to both Protocols and Primitives

Cryptography

1. Design System
2. Security Proof
3. Cryptoanalysis
4. Implementation

Cryptography

1. Design System Symbolic
2. Security Proof Symbolic & Computational
3. Cryptoanalysis Symbolic
4. Implementation Computational

Implementation

Without formal methods

1. Read Papers/Specification
2. Write Code
3. Optimize Code

Implementation

Without formal methods

1. Read Papers/Specification
2. Write Code
3. Optimize Code

With formal methods

1. Read Specification
2. Write Code
3. Prove that Code matches Spec
4. Optimize Code
5. Prove that Optimized Code matches Original Code

Implementation

Need:

- Formal Semantics
- Specification
- Tool
- Proofs

Implementation

Need:

- Formal Semantics
- Specification
- Tool
- Proofs

Get:

- Assurance that the Code matches the Spec

With some tools

- Verified Optimizations
- Verified Compilation
- Verified Side Channel Resistance

Checking Security Proofs

Without formal methods

1. Understand the Proof outline
2. Critically read the Proof while filling in Details

Checking Security Proofs

Without formal methods

1. Understand the Proof outline
2. Critically read the Proof while filling in Details

With formal methods

1. Manually check Definitions
2. Manually check Theorem Statements
3. Run the Proof Checker

Checking Security Proofs

Need:

- Formal Spec
- Mathematical Theories
  - Definitions
  - Lemmas
- Proofs
- Tools

Checking Security Proofs

Need:

- Formal Spec
- Mathematical Theories
  - Definitions
  - Lemmas
- Proofs
- Tools

Get:

- Assurance that system described in the Spec has the desired properties

Tools for Security Proofs

- EasyCrypt[4]
- FCF[11]
- SSProve[13]
- CryptHOL[10]

Some Tools and Projects in Implementation

Verified compilers

- CompCert[2] (C)
- Jasmin[7]
- Bedrock2[1]

Some Tools and Projects in Implementation

Verified compilers

- CompCert[2] (C)
- Jasmin[7]
- Bedrock2[1]

Cross-compilers

- KaRaMeL[8] (F* → C)
- HacSpec[6] (Rust → F*)

Some Tools and Projects in Implementation

Verified compilers

- CompCert[2] (C)
- Jasmin[7]
- Bedrock2[1]

Cross-compilers

- KaRaMeL[8] (F* → C)
- HacSpec[6] (Rust → F*)

Major Projects

- Fiat-Crypto[5] (Bedrock2)
- Libjade[9] (Jasmin)
- Project Everest[12] (KaRaMeL)

References I

[1] *Bedrock2*. `https://github.com/mit-plv/bedrock2`.

[2] *CompCert*. `https://compcert.org`.

[3] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: `10.1109/TIT.1983.1056650`.

[4] *EasyCrypt*. `https://github.com/EasyCrypt/easycrypt`.

[5] *Fiat-Crypto*. `https://github.com/mit-plv/fiat-crypto`.

[6] *HacSpec*. `https://hacspec.github.io`.

[7] *Jasmin*. `https://github.com/jasmin-lang/jasmin`.

[8] *KaRaMeL*. `https://github.com/FStarLang/karamel`.

References II

[9] *Libjade*. `https://github.com/formosa-crypto/libjade`.

[10] Andreas Lochbihler. "CryptHOL". In: *Archive of Formal Proofs* (May 2017). `https://isa-afp.org/entries/CryptHOL.html`, Formal proof development. ISSN: 2150-914x.

[11] Adam Petcher and Greg Morrisett. "The Foundational Cryptography Framework". In: *Principles of Security and Trust*. Ed. by Riccardo Focardi and Andrew Myers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 53–72. ISBN: 978-3-662-46666-7.

[12] *Project Everest*. `https://project-everest.github.io`.

[13] *SSProve*. `https://github.com/SSProve/ssprove`.