Thesis Defence

Nikolaj Sidorenco

June 25, 2020

Outline

Topic

Introduction

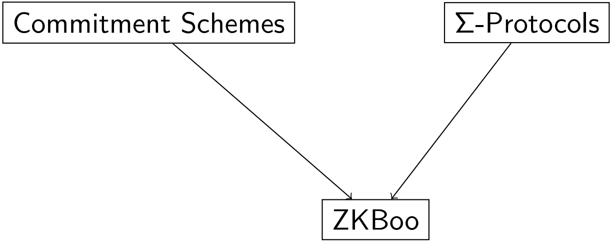
Bellare and Rogaway remarked:

> *In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.*

Recent advances in formal verification has allowed us to formally verify cryptographic protocols.

In this work we explore the applicability of EasyCrypt and the feasiblity of formalising a complex cryptographic protocol.

Introduction



Introduction

In the formalisation effort of Commitment Schmes and Σ-Protocols we reproduce the results of previous research:

- David Butler et al. “Formalising Σ-Protocols and Commitment Schemes using CryptHOL”
- Roberto Metere and Changyu Dong. “Automated Cryptographic Analysis of the Pedersen Commitment Scheme“

Code-based Game-playing approach

Committer (sk, pk) Verifier pk

$$\xrightarrow{\quad \mathsf{c} = \mathtt{Commit}(\mathsf{m}) \quad}$$

$$\xrightarrow{\quad \mathsf{m'} \quad} \mathsf{c} \stackrel{?}{=} \mathtt{Commit}(\mathsf{m'})$$

Figure: Commitment Scheme interaction

This can interaction call also be seen an a program with access to the Committer and Verifiers functionality

EasyCrypt

Programs are distribution transformers

$$\boxed{\text{Input distribution}} \longrightarrow \boxed{\text{Game}} \longrightarrow \boxed{\text{Output distribution}}$$

EasyCrypt gives us the tools to reason about the transformations the game performs

Topic

Types

First we need to define the types of a commitment scheme.

- message
- commitment
- randomness

Moreover, we also have types for the keys used:

- secretkey
- publickey

The Committer

Based on the desired functionality we define the Committer as:

```
module Committer = {
  proc commit(sk : secret_key, m : message)
       : (commitment * randomness)
}
```

N.B. Key generation will also have to be defined in a module.

The Verifier

The Verifier does not need to hold state nor make random choices.
For this reason we can fix the verifier as a function rather than a procedure:

```
op verify key message commitment randomness : bool.
```

This also allow us to change the order when verifing multiple commitments.

Security (Correctness)

```
module Correctness(C : Commitment) = {
  proc main(m : message) = {
    (sk, pk) = KeyGen();
    (c, r) = C.commit(sk, m);
    b = verify(pk, m, c, r);

    return b;
  }
}
```

N.B. This module is implicitly parameterised by the `verify` function.

Correctness

Commitment scheme Com is correct if:
$\forall$ m. Pr[Correctness(Com).main(m) = 1] = 1

Security (Hiding)

```
module type HidingAdv = {
  proc * get() : message * message
  proc check(c : commitment) : bool
}.
module Hiding(C : Committer, A : HidingAdv) = {
  proc main() = {
    (sk, pk) = KeyGen();
    (m, m') = A.get();
    b <$ {0,1};
    if (b) {
      (c, r) = C.commit(sk, m);
    } else {
      (c, r) = C.commit(sk, m');
    }
    b' = A.check(c);
    return b = b';
  }
```

Security (Hiding)

Perfect Hiding

Commitment Scheme Com is perfect hiding if $\forall$ (A $<:$ HidingAdv).
Pr[Hiding(Com, A).main $=$ true] $= 1/2$

Security (Binding)

```
module type BindingAdv = {
  proc bind(sk : secret_key, pk : public_key)
      : commitment * message * message * ...
}.
module Binding(C : Committer, B : BindingAdv) = {
  proc main() = {
    (sk, pk) = KeyGen();
    (c, m, m', r, r') = B.bind(sk, pk);
    v =  verify pk m c r;
    v' = verify pk m' c r';
    return (v /\ v') /\ (m <> m');
  }
}.
```

Security (Binding)

Perfect Binding

Commitment Scheme Com has perfect binding if $\forall$ (A $<:$ BindingAdv). Pr[Binding(Com, A).main $=$ true] $= 0$

Computational Binding

Commitment Scheme Com has computational binding if $\forall$ (A $<:$ BindingAdv). Pr[Binding(Com, A).main $=$ true] $= \epsilon$

Alternative definitions

We formalised both key-based and key-less variants of commitment schemes. We will discuss the benefit of this later.
Moreover, we also provided alternative security definitions

Topic

1. Introduction
2. Commitment Schemes
3. Σ-Protocols
4. Compound Σ-Protocols
5. ZKBoo
6. Conclusion

Σ-Protocols

	Prover		Verifier	
	(h, w)		h	
$Init(h, w) \to a$		$\xrightarrow{\text{a}}$		
$Response(h, w, a, e) \to z$		$\xleftarrow{\text{e}}$		
		$\xrightarrow{\text{z}}$		$Verify(h, a, e, z)$

Figure: Σ-Protocol

Where (h, w) $\in$ R

Security

For a Σ-Protocol to be secure, we have three criteria:

- Completeness
- Special Soundness
- Special honest-verifier zero-knowledge

For this we need a procedure for simulating transcripts and a procedure for extracting witnesses.

Procedures

```
module type SigmaProtocol = {
  proc init(h, w) : message
  proc response(h, w, m, e) : response
  proc verify(h, m, e, z) : bool
  proc witness_extractor(h, m,
                         es : challenge list,
                         zs : response list)
       : witness option
  proc simulator(h, e) : message * response
}
```

Security (Completeness)

```
module Completeness(S : SigmaProtocol) = {
  proc main(h : input, w : witness) : bool = {
      var a, e, z;
      a = S.init(h,w);
      e <$ dchallenge;
      z = S.response(h, a, e);
      v = S.verify(h, a, e, z);
      return v;
  }
}.
```

Completeness

A Σ-Protocol, S, is complete if: $\forall$ (h,w) $\in$ R. Pr[Completeness(S).main(h, w) = 1] = 1.

Security (Special soundness)

```
module SpecialSoundness(S : SigmaProtocol) = {
  proc main(h : statement,
            a : message,
            c : challenge list,
            z : response list) : bool = {
    w = S.witness_extractor(h, m, c, z);
    valid = true;
    while (c <> []) {
      c' = c[0];
      z' = z[0];
      valid = valid /\ S.verify(h, m, c', z');
      c = behead c;
      z = behead z;
    }

    return valid /\ R h (oget w);
}
```

Security (Special soundness)

s-Special Soundness

$\forall$ (i $\neq$ j). c[i] $\neq$ c[j] $\wedge$ $\forall$ (i $\in$ [1, ..., s]). Pr[S.verify(h, a, c[i], z[i])] = 1
$\implies$ Pr[SpecialSoundness(S).main(h, a, c, z) = 1] = 1

Security (SHVZK)

```
module SHVZK(S : SigmaProtocol) = {
  proc real(h, w, e) = {
    a = init(h,w);
    z = response(h,w,e,a);
    return (a, e, z);
  }
  proc ideal(h, e) = {
    (a, z) = simulator(h, e);
    return (a, e, z);
  }
}
```

Security (SHVZK)

Special honest-verifier zero-knowledge

Σ-Protocol S is special-honest verifier zero-knowledge if:
equiv[$\texttt{SHVZK(S)}$.real $\sim$ $\texttt{SHVZK(S)}$.ideal : $=\{$h, e$\}$ $\wedge$ R h w$^{\text{real}}$
$\Longrightarrow = \{res\}$]

Topic

Compound Σ-Protocols

From the abstract specification of Σ-Protocols we are able to prove the security of compound protocol.

AND

Uses two secure Σ-Protocols S_1 and S_2 for relations R_1 and R_2 to construct S_{AND} for relation $R_{AND} = R_1 \wedge R_2$

OR

Uses two secure Σ-Protocols S_1 and S_2 for relations R_1 and R_2 to construct S_{OR} for relation $R_{OR} = R_1 \vee R_2$

Problems

EasyCrypt allow modules to quantify over module types. This gives us no type information

```
module ANDProtocol(S1 : SigmaProtocol,
                   S2 : SigmaProtocol)
       : SigmaProtocol = { ... }
```

Instead we need to fix the types of S1 and S2 beforehand.

Topic

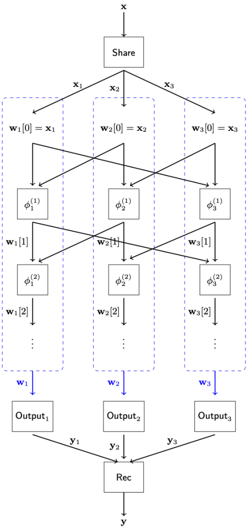
General description

- MPC-based Σ-Protocol
- Relation is the pre-image of a group homomorphism
- We restrict the functions to be expressed over arithmetic circuits

	Prover		Verifier
	$f(x) = y$		y
(2,3)-Decomposition of f(x)		$\xrightarrow{\text{Commit to views and outputs}}$	
Open commitments for $e, e+1$		$\xleftarrow{\text{e}}$	
		$\xrightarrow{\text{z}}$	Verify

Figure: Σ-Protocol

Image from [GMO16]



(2,3)-Decomposition

Correctness

Decomposition D is correct if: $\forall$ x $\Pr[f(x) = D(x)] = 1$

2-Privacy

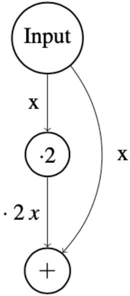
D is 2-private if there exists a simulator S_e:
$\forall$ x, $(\{k_i, w_i\}_{i \in \{e, e+1\}}, y_{e+2}) \sim S_e(x,y)$
where $(\{k_i, w_i\}_{i \in \{e, e+1\}}, y_{e+2})$ are produced by D.

Verifing

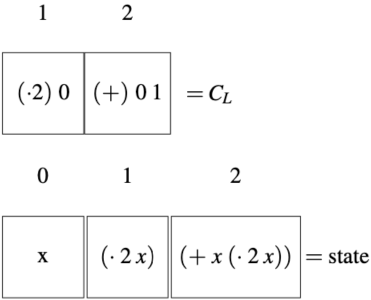
To verify the transcript the following is done:

- The output shares must reconstruct to y
- The output shares must be in the view of parties e and e+1
- The view of e must be computed by the decomposition

Arithmetic circuit



(a) Graph representation of circuit



(b) List representation of circuit

From the list representation we have a evaluation order.

Correctness

We defined the following functions and procedures:

- `eval circuit`
- `decomposition`
- `compute` : procedure from circuit and views to views

From which we formally defined correctness as:

Correctness

Valid circuit c $\implies$
Pr[`eval circuit` (c, [input]) = y] =
Pr[`decomposition` (c, [input]) = y]

Correctness

This correctness is not strong enough. We need to able to reason about the shares/views produces by the docomposition.
Having a valid output is not enough to prove security of ZKBoo

Stepping lemma for decomposition

Valid (c_1, w_1, w_2, w_3) $\wedge$ Valid circuit (c_1 ++ c_2) $\implies$
Pr[`compute` (c_2, w_1, w_2, w_3) : Valid (c, w_1', w_2', w_3')]

2-Privacy

To define 2-Privacy we have the following procedures

- `real`
- `simulated`
- `simulator`

Simulator is a procedure simulating two views by using most of the logic of the `compute` procedure

2-Privacy

2-Privacy

equiv[real $\sim$ simulated : $= \{$e,x,c$\} \wedge \mathrm{y}^{\mathrm{simulated}} =$ `eval circuit` c $\mathrm{x}^{\mathrm{real}}$
$\Longrightarrow = \{w_{\mathrm{e,we+1}}, y_{\mathrm{e+2}}\}$]

Stepping lemma

equiv[`compute` $\sim$ `simulate` : $=\{$c,e,$\mathrm{w}_{\mathrm{e,we+1}}\}$
$\Longrightarrow = \{w_{\mathrm{e}}, w_{\mathrm{e+1}}\} \wedge \sum_{\mathrm{i} \in \{1,2,3\}}$ last $\mathrm{w_i}^{\mathrm{compute}} =$ `eval circuit` c x]

ZKBoo

We are now ready to prove ZKBoo to be a secure Σ-Protocol.

Completeness

- The output shares must reconstruct to y
- The output shares must be in the view of parties e and e+1
- The view of e must be computed by the decomposition

All properties of the correctness of the decomposition.
Lastly, we need that the commitment are valid. This is simplfied by using a key-less commitment scheme

Stepping lemma for decomposition

Valid (c_1, w_1, w_2, w_3) $\wedge$ Valid circuit (c_1 ++ c_2) $\implies$
Pr[`compute` (c_2, w_1, w_2, w_3) : Valid (c, w_1', w_2', w_3')]

Completeness

```
module Correctness(C : Commitment) = {
  proc main(m : message) = {
    (sk, pk) = KeyGen();
    (c, r) = C.commit(sk, m);
    b = verify(pk, m, c, r);

    return b;
  }
}
```

Special honest-verifier zero-knowledge

We use the simulator to simulate views corresponding to the challenge.
The last view must be randomly sampled.
The simulated views are accepting by the 2-Privacy property.

SHVZK

equiv[$\texttt{SHVZK(S)}$.real $\sim$ $\texttt{SHVZK(S)}$.ideal : $=\{\text{h, e}\} \wedge \text{R h w}^{\text{real}}$
$\implies = \{res\}$]

Perfect Hiding

$\forall$ (A <: HidingAdv). Pr[Hiding(Com, A).main = true] = 1/2

Special honest-verifier zero-knowledge

Ultimately this leaves us with showing:

$\Pr[\text{Hiding}(\text{Com}, \text{A}).\text{main}() = \text{true}] = 1/2 \implies$
$\text{equiv}[\text{commit}(\text{w}_{e+2}) \sim \text{commit}(\text{w'}) : =\{\text{glob Com}\} \implies = \{res\}]$

This transformation is intuively sound, but formally unclear.

Alternative Hiding:

$\text{equiv}[\text{commit}(\text{w}\}) \sim \text{commit}(\text{w'}) : =\{\text{glob Com}\} \implies = \{res\}]$
Moreover, we can show Alternative Hiding $\implies$ *Hiding*

3-Special Soundness

The game is given access to a = (c_1, c_2, c_3, y_1, y_2, y_3) and three openings: z_i = ($w^i{}_i$, $k^i{}_i$, $w^i{}_{i+1}$, $k^i{}_{i+1}$) for i $\in$ {1,2,3}

- Pr[$w^i{}_i = w^j{}_i$] = (1 - binding prob)
- $\forall$ i. $w^i{}_i = w^j{}_i \implies$ Pr[`witness extractor` : R h w] = 1

Which ultimately gives us: Pr[SpecialSoundness(ZKBoo).main = 1] = (1 - binding prob)

Here we use the correctness of the decomposition to extract the witness.

3-Special Soundness

```
module SpecialSoundness(S : SigmaProtocol) = {
  proc main(h : statement,
            a : message,
            c : challenge list,
            z : response list) : bool = {
    w = S.witness_extractor(h, m, c, z);
    valid = true;
    while (c <> []) {
      c' = c[0];
      z' = z[0];
      valid = valid /\ S.verify(h, m, c', z');
      c = behead c;
      z = behead z;
    }

    return valid /\ R h (oget w);
}
```

3-Special Soundness

```
proc extract_views(h, a, z1 z2 z3 : response) = {
  v1 = ZKBoo.verify(h, m, 1, z1);
  v2 = ZKBoo.verify(h, m, 2, z2);
  v3 = ZKBoo.verify(h, m, 3, z3);

  (k1, w1, k2, w2) = z1;
  (k2', w2', k3, w3) = z2;
  (k3', w3', k1', w1') = z3;
  (y1, y2, y3, c1, c2, c3) = m;
  cons1 = alt_binding(c1, w1, w1');
  cons2 = alt_binding(c2, w2, w2');
  cons3 = alt_binding(c3, w3, w3');

  return v1 /\ v2 /\ v3 /\ cons1 /\ cons2 /\ cons3;
}
```

Topic

Conclusion

- Workable formalisations of Commitment schemes and Σ-Protocols
- Formal proof of security of ZKBoo
  - Arithmetic circuits
  - (2,3)-Decomposition