Proofs of Security Protocols in Lean

Théophile Wallez, *CISPA Helmholtz Center for Information Security*
(with the kind advices of: Karthikeyan Bhargavan, Cas Cremers, Son Ho, Jonathan Protzenko)

Our goal

Disclaimer:

- still an early work-in-progress
- hence title of this talk is aspirational

We aim to produce a general-purpose Lean library, to write

- specifications of cryptographic protocols
- security properties in the symbolic model

and further provide

- a proof methodology to prove security properties

Plan for this talk

1. intro to machine-checked protocol analysis
2. intro to the line of work DY* (our proof methodology)
3. the latest iteration of DY*, the good and the bad
4. the next iteration, (Bob) DyLean

Introduction to machine-checked cryptographic protocol analysis

You may already have a few questions:

- why verify security of cryptographic protocols in Lean?
- why use another tool when there is already a whole zoo of them?
- why verify security of cryptographic protocols?

Why verify security of cryptographic protocols?

Traditional program verification:
prove implementation behaves the same as specification.

Security proofs of cryptographic protocols:
prove the specification obeys some security properties.

Challenges:

- traditional engineering practices (e.g. testing) don't apply
- standardized protocols specifications are set in stone
- undecidable problem

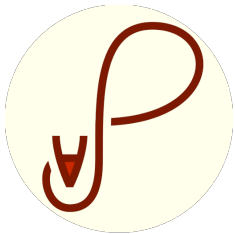
The crypto tooling landscape

EasyCrypt

Owl

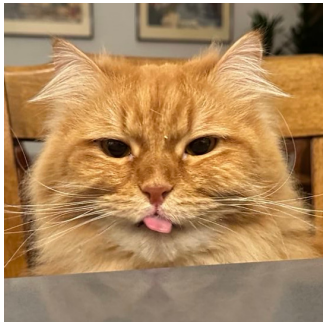
DY*

SAPIC+

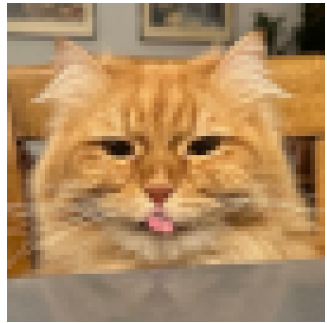


DEEPSEC

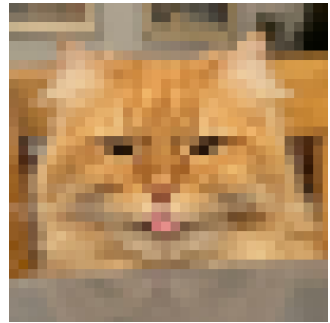
Security model for cryptographic protocols



Actual protocol program running on your computer (e.g. C implementation on Intel CPU)

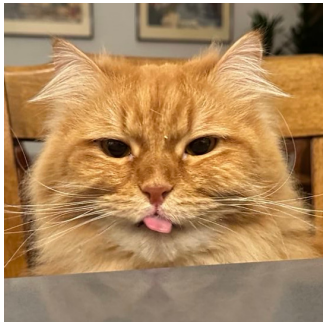


Computational model:
attacker is a turing machine
assume is unable to perform certain type of computations (e.g. break encryption)

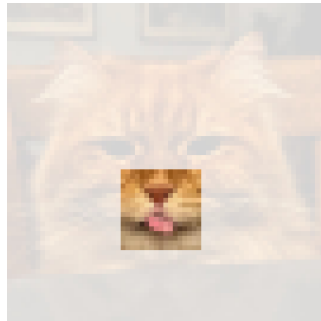


Symbolic model:
attacker restricted to have black box cryptography use

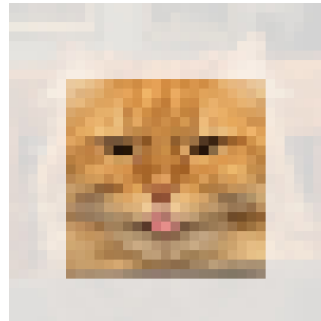
Security model for cryptographic protocols



Actual protocol program running on your computer (e.g. C implementation on Intel CPU)



Computational model:
attacker is a turing machine
assume is unable to perform certain type of computations (e.g. break encryption)



Symbolic model:
attacker restricted to have black box cryptography use

Symbolic tools, some success stories

ProVerif:

- TLS 1.3 draft 18 (uncover client authentication vulnerability)
- PQXDH (uncover potential key re-encapsulation attack)

Tamarin:

- TLS 1.3 draft 10-11 (uncover client authentication vulnerability)
- EMV (uncover PIN requirement evasion)
- 5G-AKA (uncovers multiple weaknesses)
- SPDM (uncover authentication attack across multiple modes)

DY*:

- MLS (uncover signature ambiguity flaw between sub-protocols)

The crypto tooling landscape: in the symbolic model

State of the art automatic tools: ProVerif and Tamarin.
They aim to be automatic, push-button.
Outcome: either guarantee security, or showcase an attack trace.

The crypto tooling landscape: in the symbolic model

State of the art automatic tools: ProVerif and Tamarin.
They aim to be automatic, push-button.
Outcome: either guarantee security, or showcase an attack trace.

But: as protocols grow large, automation fails unless users provide hints to the tool. This (1) makes the proof manual, and (2) requires intuition on why the protocol is secure.

The crypto tooling landscape: DY* (our line of work)

Hot take: Doing a manual proof with a tool designed for automatic proofs is the worst feeling ever.

Our line of research (DY*): proactively accept our fate, develop proof techniques that are manual and require intuition, but make them as nice as possible.

The symbolic attacker

We consider an attacker with many capabilities:

- controls the network (e.g. can perform MitM attacks)
- can dynamically compromise participants (e.g. border control)
- decides who starts running the protocol with whom
- can run unbounded instances of the protocol in parallel

Only limitation of the attacker:

- only allowed to use cryptographic functions as black boxes

This captures all "logical attacks".

Mathematical model of the symbolic attacker

Challenge: do a mathematical model participants that execute protocol in parallel, and send messages to each other.

Idea: model their actions in an *execution trace*, a global, shared, append-only log of everything that happened during a protocol execution.

Given an execution trace, the attacker knows a bytestring b when

- b was sent on the network or was stored by a compromised participant
- b is the output of a cryptographic function with known inputs

Example scenario: a ciphertext is sent on the network, the corresponding key is compromised, hence the attacker knows the plaintext (by computing decryption).

Active attacker: participants receive from the network bytestrings known by the attacker.

Security properties as reachability properties

Reachability property: "any trace the attacker may reach must satisfy property X"

Example properties:

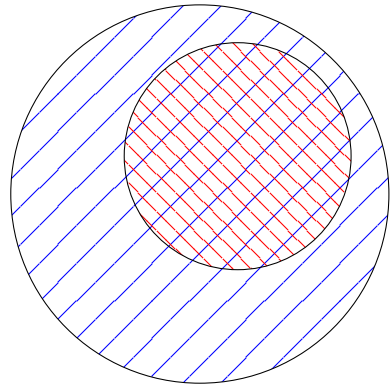
- confidentiality:
  if Alice finishes the handshake protocol with Bob which outputs key k
  and the attacker knows this key k,
  then it must be that the attacker either compromised Alice or Bob
- authentication / agreement:
  if Alice finishes the handshake protocol with Bob which outputs key k,
  then Bob also finished the handshake protocol with Alice and key k,
  or the attacker must have compromised Bob's long term keys

The line of work of DY*

Symbolic security proof with DY*: general methodology

reachable traces 

security properties 

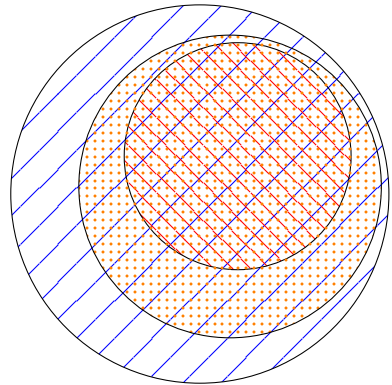


Symbolic security proof with DY*: general methodology

reachable traces

trace invariant

security properties



Methodology:

1. design a trace invariant
2. prove it is preserved by every protocol step, hence $\subset$
3. prove it implies security properties, i.e. $\subset$

Some insights on the DY* methodology

DY* methodology is similar to verifying loops in program verification:

DY* methodology	Program verification
Design trace invariant	Design loop invariant
Preserved by each protocol step	Preserved by each loop iteration
Implies security properties	Implies final property

Some insights on the DY* methodology

DY* methodology is similar to verifying loops in program verification:

DY* methodology	Program verification
Design trace invariant	Design loop invariant
Preserved by each protocol step	Preserved by each loop iteration
Implies security properties	Implies final property

Intuition: protocol (sub-)steps preserve trace invariant when they use cryptography "safely", with respect to desired security properties.

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step,
we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step, we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Question: does this preserves the trace invariant? i.e. is safe to perform?

- `send "Hello"`

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step, we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Question: does this preserves the trace invariant? i.e. is safe to perform?

- `send "Hello"`

Yes: `is_publishable tr "Hello"` is true

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step,
we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Question: does this preserves the trace invariant? i.e. is safe to perform?

- `send super_secret_key`

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step, we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Question: does this preserves the trace invariant? i.e. is safe to perform?

- `send super_secret_key`

No: `is_publishable tr super_secret_key` is false

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step,
we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Question: does this preserves the trace invariant? i.e. is safe to perform?

- `send (encrypt (key:=key1) (msg:=key2))`

Trace invariant preservation proof example

To prove that the trace invariant is preserved by every protocol step, we use standard program verification techniques, such as Hoare Triples.

```
{ λtr.  is_publishable tr msg }
send msg
{ λtr ().  True }
```

Question: does this preserves the trace invariant? i.e. is safe to perform?

- `send (encrypt (key:=key1) (msg:=key2))`

Maybe: when can we prove `is_publishable tr (encrypt key msg)`?

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext.  is_publishable tr ciphertext` }

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext.  is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` contains a secret key K, but `key` is all-zeros

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext. is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` contains a secret key K, but `key` is all-zeros

Unsafe: sending the ciphertext would leak the secret key K!

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext.  is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` is "Hello", and `key` is shared by Alice and Bob

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext. is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` is "Hello", and `key` is shared by Alice and Bob

Safe: the plaintext is publishable anyway

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext. is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` is contains a key K only known by Alice, `key` is shared by Alice and Bob

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext. is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` is contains a key K only known by Alice, `key` is shared by Alice and Bob

Unsafe! Attacker can compromise Bob to learn the key K only known by Alice.

Key technique: keeping track of who knows what

{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext. is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

- `msg` is contains a key K shared by Alice and Bob, `key` is only known by Alice

Key technique: keeping track of who knows what

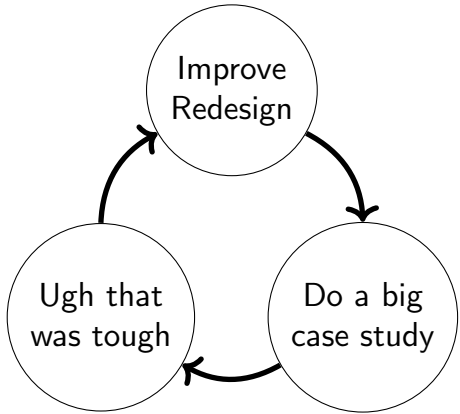
{ λ `tr. msg` less secret than `key` }
`encrypt key msg`
{ λ `tr ciphertext. is_publishable tr ciphertext` }

Question: is the ciphertext publishable if

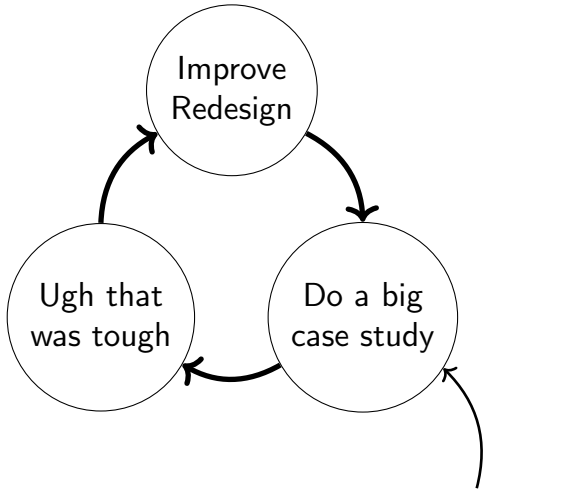
- `msg` is contains a key K shared by Alice and Bob, `key` is only known by Alice

Safe: it is allowed if the attacker compromises Alice to learn `key` and decrypt K

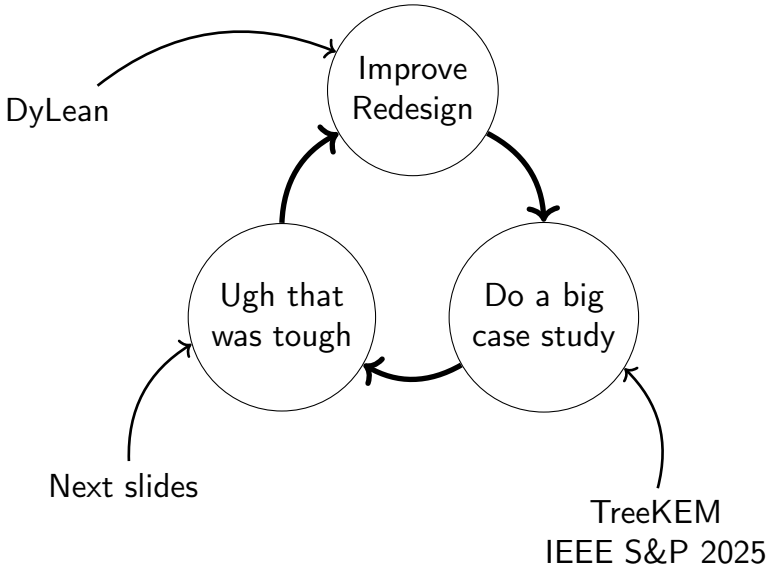
Development cycle of DY*



Development cycle of DY*



Development cycle of DY*



Latest version of DY*, the good and the bad

The latest version of DY*, overview

The latest version of DY*

- is written in F* and enables (semi-automatic) verification backed by the SMT solver Z3
- allows for the composition of security proofs
- good SMT proof engineering

The hope:

- manual work needed to write trace invariant
- but benefit from SMT automation to prove they are preserved

And the reality...

The reality

Upon inspection of the TreeKEM (MLS) proofs, we notice the following.

The good news:

- SMT automation works well on low-level internal functions that rely on small but tricky invariants

The bad news:

- SMT offers little automation on high-level outer functions, although the proof should be boring (just need to compose guarantees of lower-level functions)
- these proofs are in practice painstakingly manual
- despite being written by F* experts who frequently talk with F* developers

The hidden reality of SMT proofs

SMT solvers can prove automatically true theorems, but:

- on false theorems, they give no insight to why they are false
- to understand why our theorem is false, we need to write a manual proof
- we rarely write true theorems on the first try

Result: even if the final proof is automatic, the process to obtain such a proof requires manual work.

Worse: the final proofs of TreeKEM are not even automatic, why?

SMT proofs and context size

Every SMT solver expert user agree, to allow for automatic SMT proofs, it is necessary to have a small context:

- hide definitions by default, and only reveal them when needed
- avoid giving too much information to the SMT solver
- carefully engineer the lemmas that are instantiated automatically

We did all of this.

However:

- it is difficult to inspect what is the context
- in the middle of a proof, we can extend the context but not reduce it

Goal for next version

Hot take (again): Doing a manual proof with a tool designed for automatic proofs is the worst feeling ever

If invariant preservation proofs need to be manual anyway, let's accept our fate but allow for the nicest manual proofs possible.

Next iteration in this line of work: (Bob) DyLean

Lean to the rescue

Why use Lean instead of F*?

- ~~because of the hype~~
- benefit from excellent metaprogramming facilities
- custom tactic to split our big proof (step preserves trace invariant) into small proofs (every sub-step precondition holds)
- automate small proofs using `grind` (SMT-like tactic)

Result:

- proof failure directly point to the incriminated sub-step
- can start to do proof manually (e.g. using `simp`) to investigate failure (and likely find we need to tweak trace invariant)
- understand what happens by looking at the context (instead of having to guess it)
- our tactic can trim the context as we go, allows for better `grind` stability

DyLean: our vision

Build a reusable library to specify cryptographic protocols and security properties:

- symbolic bytes term
- set of reachable traces
- attacker knowledge
- allow for executable specifications

Build DY*-style proof techniques on top (focus for the rest of the talk)

- trace invariant
- custom tactic
- ...

Open the way for other people to build other proof techniques.

Overview of our tactic: `step`

- Similar to `mvcgen` or to Aeneas' `progress`
- Works with the typical combo Hoare Triple + Weakest Precondition
- Prove precondition with a user-provided tactic script (or by default, `grind`)
- Support for "ghost" parameters
  (values not present in the specification but impacting pre/post-conditions)
- Cleanup the context to deal only with latest trace (see next slide)
- Use Lean typeclasses to store Hoare Triple in a declarative style (in 2 slides)

```
-- trace before executing func1
tr : Trace
h_tr : tr.invariant
h: Γ tr -- several hypothesis
⊢ wp (
   let z ← func1 x y
   func2 ...
 ) post tr
```



```
-- trace after executing func1
tr : Trace
h_tr : tr.invariant
h': Γ' tr -- contains z
⊢ wp (
   func2 ...
 ) post tr
```

Internals of `step`

```
tr : Trace
h_tr : tr.invariant
h: Γ tr —— several hypothesis
⊢ wp (
   let z ← func1 x y
   func2 ...
 ) post tr
```

1. Lookup [HoareTriple (func1 x y) pre_func1 post_func1]
2. Prove precondition pre_func1 using tactic_script

```
tr : Trace
h_tr : tr.invariant
h: Γ tr
⊢ pre_func1 tr
```

3. Get intermediate goal

```
tr tr' : Trace
h_tr : tr.invariant
h_tr' : tr'.invariant
h_tr_tr' : tr ≤ tr'
h: Γ tr
z: α
h_func1: post_func1 z tr'
⊢ wp (
   func2 ...
 ) post tr'
```

4. Monotonize context

```
tr tr' : Trace
h_tr_tr' : tr ≤ tr'
h: Γ tr
⊢ Γ tr'
```

5. Output proof goal

```
tr : Trace —— tr' in intermediate goal
h_tr : tr.invariant
—— below is h': Γ' tr
h: Γ tr
z: α
h_z: post_func1 z tr
⊢ wp (
   func2 ...
 ) post tr
```

Storing Hoare Triples using Lean typeclasses

The typeclass `HoareTriple` stores the pre/post-condition of a function.

```
class HoareTriple
  (f: m a)
  (pre: outParam (Trace → Prop))
  (post: outParam (a → Trace → Prop))
where
  ...
```

This allows for programmatic / declarative Hoare Triples, for example:

```
instance (b: Bool) [HoareTriplePure b pre post]:
  HoareTriple
    (guard b)
    (fun tr => pre tr)
    (fun () tr => post true tr)
```

Discussion: why not use `mvcgen`?

We made some experiments when starting to work ot DyLean (Aug 25), noticed a crucial missing feature:

- transfer properties on old state (trace) to new state ("monotonizing context")

Decided to work on our own tactic, which helped us to explore cool features

- ghost parameters (i.e. present in pre/post-conditions but not in function call)
- programmatic / declarative Hoare Triples
- proofs on pure computations (i.e. without monads)

Not a dealbreaker, thanks to Lean's great metaprogramming!
First proof-of-concept required only $\sim$ 1-Théophile-week of work (starting with no Lean knowledge).

To discuss this week: are any of these ideas useful to backport in `mvcgen`?

Summary: (Bob) DyLean

Our plan:

- make a re-usable library to write cryptographic specifications and security properties in Lean, in the symbolic model
- allow for security proofs using the time-tested DY* methodology
- and improve proof experience by an order of magnitude, by leveraging Lean's metaprogramming

</> `https://github.com/BobDyLean/dylean` (still private, public release in a few months)

✉ `theophile.wallez@cispa.de`
🌐 `https://www.twal.org/`
@twal.org