Verifpal

A Cryptographic Protocol Modeling and Verification Framework Written in Go

Nadim Kobeissi

Go Devroom, FOSDEM2020

What is Formal Verification?

- Using software tools in order to obtain guarantees on the security of cryptographic components.
- Protocols have unintended behaviors when confronted with an active attacker: formal verification can prove security under certain active attacker scenarios!
- Primitives can act in unexpected ways given certain inputs: formal verification: formal verification can prove functional correctness of implementations!

Formal Verification Today

Code and Implementations: F*

- Exports type checks to the Z3 theorem prover.
- Can produce provably functionally correct software implementations of primitives (e.g. Curve25519 in HACL*).
- Can produce provably functionally correct protocol implementations (Signal*).

Protocols: ProVerif, Tamarin

- Take models of protocols (Signal, TLS) and find contradictions to queries.
- *"Can the attacker decrypt Alice's first message to Bob?"*
- Are limited to the "symbolic model", CryptoVerif works in the "computational model".

Symbolic Verification Overview

- Main tools: ProVerif, Tamarin.
- User writes a model of a protocol in action:
  - Signal AKE, bunch of messages between Alice and Bob,
  - TLS 1.3 session between a server and a bunch of clients,
  - ACME for Let's Encrypt (with domain name ownership confirmation…)
- User writes queries:
  - *"Can someone impersonate the server to the clients?"*
  - *"Can a client hijack another client's simultaneous connection to the server?"*
- ProVerif and Tamarin try to find contradictions.

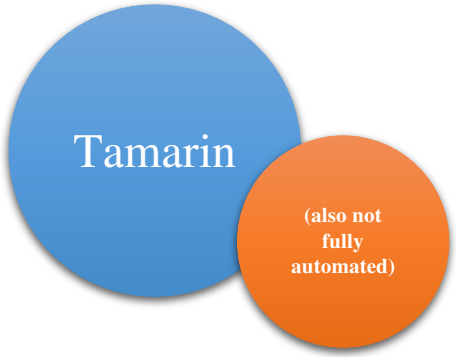
Symbolic Verification is Wonderful

- Many papers published in the past 4 years: symbolic verification proving (and finding attacks) in Signal, TLS 1.3, Noise, Scuttlebutt, Bluetooth, 5G and much more!
- This is a great way to work, allowing practitioners to reason better about their protocols before/as they are implemented.

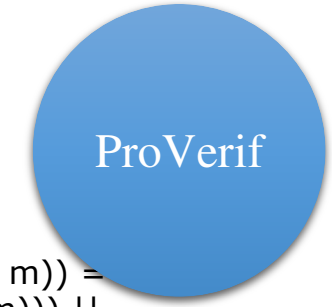
Why isn't it used more?

Tamarin and ProVerif: Examples

```
rule Get_pk:
  [ !Pk(A, pk) ]
  -->
  [ Out(pk) ]

// Protocol
rule Init_1:
  [ Fr(~ekI), !Ltk($I, ltkI) ]
  -->
  [ Init_1( $I, $R, ~ekI )
  , Out( <$I, $R, 'g' ^ ~ekI, sign{'1', $I, $R,'g' ^ ~ekI }ltkI> ) ]

rule Init_2:
    let Y = 'g' ^ z // think of this as a group element check
    in
    [ Init_1( $I, $R, ~ekI )
    , !Pk($R, pk(ltkR))
    , In( <$R, $I, Y, sign{'2', $R, $I, Y }ltkR> )
    ]
  --[ SessionKey($I,$R, Y ^ ~ekI)
    , ExpR(z)
    ]->
    [ InitiatorKey($I,$R, Y ^ ~ekI) ]
```




```
letfun writeMessage_a(me:principal, them:principal,
hs:handshakestate, payload:bitstring, sid:sessionid) =
  let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key,
psk:key, initiator:bool) = handshakestateunpack(hs) in
  let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty,
empty, empty) in
  let e = generate_keypair(key_e(me, them, sid)) in
  let ne = key2bit(getpublickey(e)) in
  let ss = mixHash(ss, ne) in
  let ss = mixKey(ss, getpublickey(e)) in
  let ss = mixKey(ss, dh(e, rs)) in
  let s = generate_keypair(key_s(me)) in

[…]

event(RecvMsg(bob, alice, stagepack_c(sid_b), m)) ==
(event(SendMsg(alice, c, stagepack_c(sid_a), m))) ||
((event(LeakS(phase0, alice))) && (event(LeakPsk(phase0, alice,
bob)))) || ((event(LeakS(phase0, bob))) &&
(event(LeakPsk(phase0, alice, bob))));
```



Verifpal: A New Symbolic Verifier

1. An intuitive language for modeling protocols **(scientific contribution: a new method for reasoning about protocols in the symbolic model.)**
2. Modeling that avoids user error.
3. Analysis output that's easy to understand.
4. Integration with developer workflow.

A New Approach to Symbolic Verification

User-focused approach…

- An intuitive language for modeling protocols.
- Modeling that avoids user error.
- Analysis output that's easy to understand.
- Integration with developer workflow.

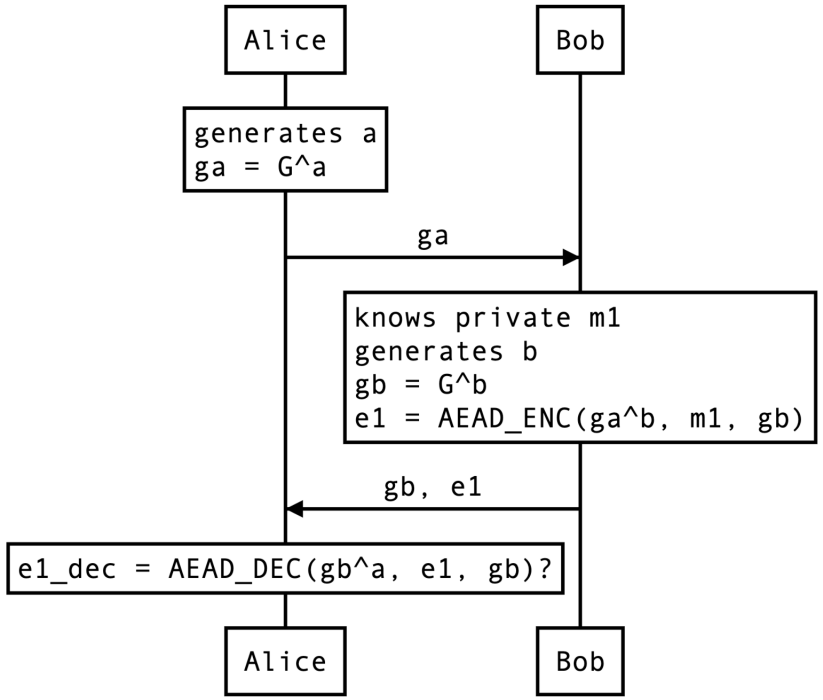
…without losing strength

- Can reason about advanced protocols (eg. Signal, Noise) out of the box.
- Can (soon) analyze for forward secrecy, key compromise impersonation and other advanced queries.
- Unbounded sessions, fresh values, and other cool symbolic model features.

Verifpal Language: Simple and Intuitive

Simple Protocol

```
attacker[active]
principal Bob[]
principal Alice[
    generates a
    ga = G^a
]
Alice -> Bob: ga
principal Bob[
    knows private m1
    generates b
    gb = G^b
    e1 = AEAD_ENC(ga^b, m1, gb)
]
Bob -> Alice: gb, e1
principal Alice[
    e1_dec = AEAD_DEC(gb^a, e1, gb)?
]
```



Protocols Analyzed with Verifpal

- Signal secure messaging protocol.
- Scuttlebutt decentralized protocol.
- ProtonMail encrypted email service.
- Telegram secure messaging protocol.

Projects Using Verifpal

The following projects have used Verifpal as part of their development process. Please send an email to the Verifpal Mailing List if you would like your project to be added:

- *CounterMITM Protocol*, by Delta Chat.
- *E4*, by Teserakt.
- *Jess*, by Safing.
- *Mles Protocol*, by Mles.
- *Monokex*, by Loup Vaillant.
- *Salt Channel*, by Assa Abloy.
- *SaltyRTC Protocol*, by SaltyRTC.
- *Userbase Protocol*, by Userbase.

Verifpal and Go

Go is awesome!

- Great performance.
- Fantastic concurrency for analysis.
- Simple and fun language.
- Super easy to publish binaries for all mainstream desktops: Windows, Linux, macOS, FreeBSD...
- Great tooling, debugging, ecosystem...

Verifpal: Go vs. Languages Usually Used

- ProVerif is written in OCaml.
- Tamarin is written in Haskell.
- OCaml and Haskell languages focus on:
  - Functional programming.
  - Being “correct” languages.
  - Have existed for longer than Go.
- Verifpal in Go: more diverse ecosystem of protocol analysis software!

Verifpal: Go vs. Languages Usually Used

Compared to Go:

- Much slower, worse concurrency, tiny ecosystem, often outdated tooling... *But*
- Syntax and semantics are perfect for describing ASTs, parsers, languages, models, etc.
- *Especially the pattern matching syntax!*

Pattern Matching in OCaml

```
let rec mem x list =
  match list with
    [] -> false
  | hd::tl -> hd = x || mem x tl
```

Pattern Matching in OCaml

```
let rec mem x list =
  match list with
    [] -> false
    | hd::_ when hd = x -> true
    | _::tl -> mem x tl
```

Pattern Matching in Go 2.x?

It would be great to have pattern matching in Go.

- Allows the language to be more appropriate for a new slew of use cases.
- Feature already supported by not only OCaml and Haskell but also Rust.

Try Verifpal Today

Verifpal is released as free and open source software, under version 3 of the GPL.

Check out Verifpal today:

verifpal.com

Support Verifpal development:

verifpal.com/donate