Verifpal

Cryptographic protocol analysis
for students and engineers

Nadim Kobeissi
FOSDEM Brussels, February 2020

nlnet FOUNDATION NGI ZERO

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

Tamarin

(also not fully automated)

```
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) ]
```

ProVerif

```
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.
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.

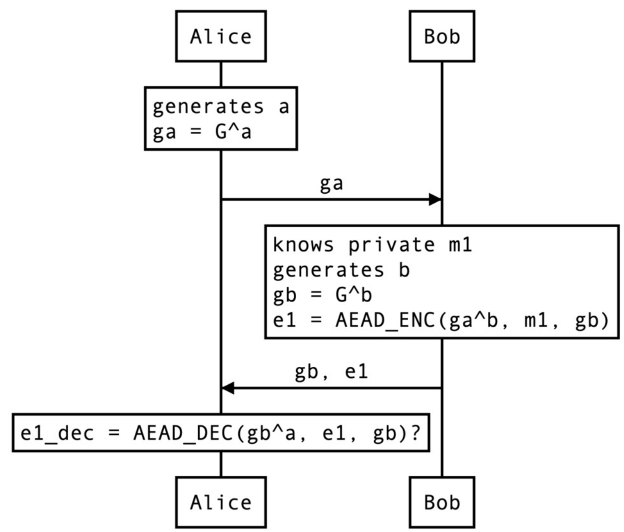
…without losing strength

- Can reason about advanced protocols (eg. Signal, Noise) out of the box.
- Can 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)?
]
```



Verifpal Language: Primitives

- Unlike ProVerif, primitives are *built-in.*
- Users cannot define their own primitives.
- Bug, not a feature: eliminate user error on the primitive level.
- Verifpal not targeting users interested in their own primitives (use ProVerif, it's great!)

- **`HASH`**`(a, b...): x`. Secure hash function, similar in practice to, for example, BLAKE2s [10]. Takes an arbitrary number of input arguments ≥ 1, and returns one output.
- **`MAC`**`(key, message): hash`. Keyed hash function. Useful for message authentication and for some other protocol constructions.
- **`ASSERT`**`(`**`MAC`**`(key, message),` **`MAC`**`(key, message)): unused`.
  Checks the equality of two values, and especially useful for checking MAC equality. Output value is not used; see §2.4.4 below for information on how to validate this check.
- **`HKDF`**`(salt, ikm, info): a, b...`. Hash-based key derivation function inspired by the Krawczyk HKDF scheme [11].
  Essentially, **`HKDF`** is used to extract more than one key out a single secret value. `salt` and `info` help contextualize derived keys. Produces an arbitrary number of outputs ≥ 1.

Verifpal Language: Primitives

- Unlike ProVerif, primitives are *built-in.*
- Users cannot define their own primitives.
- Bug, not a feature: eliminate user error on the primitive level.
- Verifpal not targeting users interested in their own primitives (use ProVerif, it's great!)

- `ENC(key, plaintext): ciphertext`. Symmetric encryption, similar for example to AES-CBC or to ChaCha20.
- `DEC(key, ENC(key, plaintext)): plaintext`. Symmetric decryption.
- `AEAD_ENC(key, plaintext, ad): ciphertext`. Authenticated encryption with associated data. ad represents an additional payload that is not encrypted, but that must be provided exactly in the decryption function for authenticated decryption to succeed. Similar for example to AES-GCM or to ChaCha20-Poly1305.
- `AEAD_DEC(key, AEAD_ENC(key, plaintext, ad), ad): plaintext`. Authenticated decryption with associated data. See §3.4.4 below for information on how to validate successfully authenticated decryption.

Verifpal Language: Primitives

- Unlike ProVerif, primitives are *built-in.*
- Users cannot define their own primitives.
- Bug, not a feature: eliminate user error on the primitive level.
- Verifpal not targeting users interested in their own primitives (use ProVerif, it's great!)

- **`SIGN`**`(key, message): signature`. Classic signature primitive. Here, `key` is a private key, for example `a`.
- **`SIGNVERIF`**`(`**`G`**`^key, message, `**`SIGN`**`(key, message)): message`. Verifies if signature can be authenticated. If key a was used for **`SIGN`**, then **`SIGNVERIF`** will expect **`G`**`^a` as the key value. Output value is not necessarily used; see §3.4.4 below for information on how to validate this check.

Signal in Verifpal: State Initialization

- Alice wants to initiate a chat with Bob.
- Bob's signed pre-key and one-time pre-key are modeled.

Signal: Initializing Alice and Bob as Principals

```
attacker[active]
principal Alice[
    knows public c0, c1, c2, c3, c4
    knows private alongterm
    galongterm = G^alongterm
]
principal Bob[
    knows public c0, c1, c2, c3, c4
    knows private blongterm, bs
    generates bo
    gblongterm = G^blongterm
    gbs = G^bs
    gbo = G^bo
    gbssig = SIGN(blongterm, gbs)
]
```

Signal in Verifpal: Key Exchange

- Alice receives Bob's key information and derives the master secret.

Signal: Alice Initiates Session with Bob

```
Bob -> Alice: [gblongterm], gbssig, gbs, gbo
principal Alice[
    generates ae1
    gae1 = G^ae1
    amaster = HASH(c0, gbs^alongterm, gblongterm^ae1, gbs^ae1, gbo^ae1)
    arkba1, ackba1 = HKDF(amaster, c1, c2)
]
```

Signal in Verifpal: Messaging

Signal: Alice Encrypts Message 1 to Bob

```
principal Alice[
    generates m1, ae2
    gae2 = G^ae2
    valid = SIGNVERIF(gblongterm, gbs, gbssig)?
    akshared1 = gbs^ae2
    arkab1, ackab1 = HKDF(akshared1, arkba1, c2)
    akenc1, akenc2 = HKDF(HMAC(ackab1, c3), c1, c4)
    e1 = AEAD_ENC(akenc1, m1, HASH(galongterm, gblongterm, gae2))
]
Alice -> Bob: [galongterm], gae1, gae2, e1
```

Signal: Bob Decrypts Alice’s Message 1

```
principal Bob[
    bkshared1 = gae2^bs
    brkab1, bckab1 = HKDF(bkshared1, brkba1, c2)
    bkenc1, bkenc2 = HKDF(HMAC(bckab1, c3), c1, c4)
    m1_d = AEAD_DEC(bkenc1, e1, HASH(galongterm, gblongterm, gae2))
]
```

Signal in Verifpal: Queries and Results

- Typical confidential and authentication queries for messages sent between Alice and Bob.
- All queries pass! No contradictions!
- Not surprising: Signal is correctly modeled, long-term public keys are guarded; signature verification is checked.

Signal: Confidentiality and Authentication Queries

```
queries[
    confidentiality? m1
    authentication? Alice -> Bob: e1
    confidentiality? m2
    authentication? Bob -> Alice: e2
    confidentiality? m3
]
```

Signal: Initial Analysis Results

```
Verifpal! verification completed at 12:36:53
```

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 in the Classroom

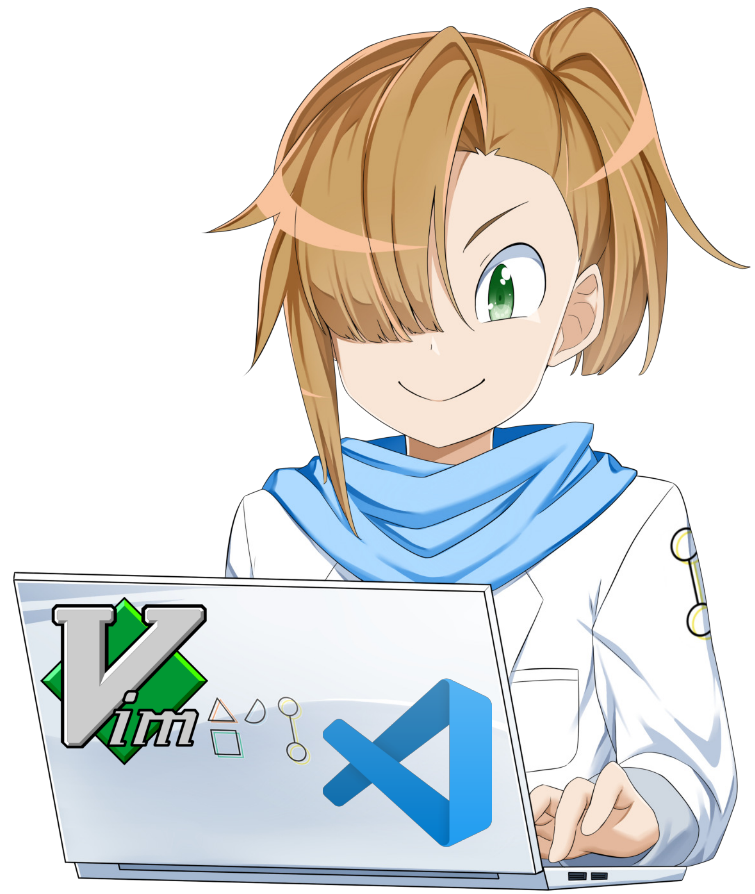
- Verifpal User Manual: easiest way to learn how to model and analyze protocols on the planet.
- NYU test run: huge success. 20-year-old American undergraduates with **no background whatsoever in security** were modeling protocols in the first two weeks of class and understanding security goals/analysis results.

Verifpal in the Classroom

- Upcoming **Eurocrypt 2020 affiliated event**: https://verifpal.com/eurocrypt2020/ – Verifpal tutorial!
- Verifpal has a place in your undergraduate classroom and will do a better job teaching students about protocols and models than anything else in the world.

Verifpal Extensions

- Visual Studio Code: currently syntax highlighting, but much more planned in the future.
- Vim: syntax highlighting.



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