Cryptographic protocol analysis for students and engineers

Verifpal

Nadim Kobeissi
Symbolic Software, NYU Paris
NGI Forum, Helsinki, September 25, 2019
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”.

Verifpal: Cryptographic protocol analysis for students and engineers – Nadim Kobeissi
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))
    , Out( <$I, $R, 'g' ^ ~ekI, sign{'1', $I, $R, 'g' ^ ~ekI }ltkI> )
  ]
  --->
  [ InitiatorKey($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 = 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.
What Are Verifpal’s End Goals?

• High quality, robust protocol modeling and analysis for engineers, with integration and live prototyping inside Visual Studio Code.

• High quality educational materials for protocol analysis in undergraduate classes.
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 (e.g. 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

• Explicit principals with discrete internal states (Alice, Bob, Client, Server…)
• Reads like a protocol diagram.
• You don’t need to know the language to understand it!

  • *Knows* for private and public values.
  • *Generates* for private fresh values.
  • Assignments.

```plaintext
New Principal: Alice

principal Alice[
   knows public c0, c1
   knows private m1
   generates a
]

New Principal: Bob

principal Bob[
   knows public c0, c1
   knows private m2
   generates b
   gb = G^b
]
```
Verifpal Language

• Explicit principals with discrete internal states (Alice, Bob, Client, Server…)
• Reads like a protocol diagram.
• You don’t need to know the language to understand it!

  • Constants are immutable.
  • Global namespace.
  • Constant cannot reference other constants.

New Principal: Alice

```plaintext
principal Alice[
  knows public c0, c1
  knows private m1
  generates a
]
```

New Principal: Bob

```plaintext
principal Bob[
  knows public c0, c1
  knows private m2
  generates b
gb = G^b
]
```
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!)

- \texttt{ENC(key, plaintext)}: ciphertext. Symmetric encryption, similar for example to AES-CBC or to ChaCha20.
- \texttt{DEC(key, ENC(key, plaintext))}: plaintext. Symmetric decryption.
- \texttt{AEAD\_ENC(key, plaintext, ad)}: ciphertext. Authenticated encryption with associated data. \texttt{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.
- \texttt{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.
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)?
]
Passive Attacker

- Can observe values as they cross the network.
- Cannot modify values or inject own values.
- Protocol execution happens once.
Active Attacker

• Can inject own values, substitute values, etc.
• Unbounded protocol executions.
• Keeps learned values between sessions (except if constructed from fresh values.)
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 ael
  gael = 6^ael
  amaster = HASH(c0, gbs^alongterm, gblongterm^ael, gbs^ael, gbo^ael)
  arkba1, ackba1 = HKDF(amaster, c1, c2)
]
```
Signal in Verifpal: Messaging

Signal: Alice Encrypts Message 1 to Bob

```plaintext
principal Alice[
    generates m1, ae2
    gae2 = G^ae2
    valid = SIGNVERIF(gblongterm, gbs, gbssig)?
    akshared1 = gbs^ae2
    arkb1, ackab1 = HKDF(akshared1, arkb1, c2)
    akenc1, akenc2 = HKDF(HMAC(ackab1, c3), c1, c4)
    e1 = AEAD_ENC(akenc1, m1, HASH(gblongterm, gblongterm, gae2))
]
Alice -> Bob: [galongterm], gae1, gae2, e1
```

Signal: Bob Decrypts Alice’s Message 1

```plaintext
principal Bob[
    bkshared1 = gae2^bs
    brkb1, bckab1 = HKDF(bkshared1, brkb1, c2)
    bkenc1, bkenc2 = HKDF(HMAC(bckab1, c3), c1, c4)
    ml_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.
Protocols Analyzed with Verifpal

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

• 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’s Role in the NGI Vision

• Provide engineers, developers and students with the accessibility they need for the analysis of critical cryptographic systems and designs.

• Broaden access to the latest research into better understanding the security of cryptographic systems in software.
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