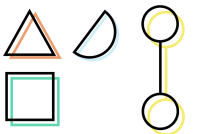




# Verifpal

*Cryptographic protocol analysis for  
the real world*



**Nadim Kobeissi**, Georgio Nicolas, Mukesh Tiwari  
*2021 IACR Real World Cryptography Symposium*  
*January 13, 2021*

# 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”.

# 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”.



# 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 and Computational Models

## Symbolic Model

- Primitives are “perfect” black boxes.
- No algebraic or numeric values.
- Can be fully automated.
- Produces verification of no contradictions (theorem assures no missed attacks).

## Computational Model

- Primitives are nuanced (IND-CPA, IND-CCA, etc.)
- Security bounds ( $2^{128}$ , etc.)
- Human-assisted.
- Produces game-based proof, similar technique to hand proofs.

# Symbolic and Computational Models

## Symbolic Model

- Primitives are “perfect” black boxes.
- No algebraic or numeric values.
- Can be fully automated.
- Produces verification of no contradictions (theorem assures no missed attacks).

## Computational Model

- Primitives are nuanced (IND-CPA, IND-CCA, etc.)
- Security bounds ( $2^{128}$ , etc.)
- Human-assisted.
- Produces game-based proof, similar technique to hand proofs.

# Symbolic and Computational Models

## Symbolic Model

- Primitives are “perfect” black boxes.
- No algebraic or numeric values.
- Can be fully automated.
- Produces verification of no contradictions (theorem assures no missed attacks).

## Computational Model

- Primitives are nuanced (IND-CPA, IND-CCA, etc.)
- Security bounds ( $2^{128}$ , etc.)
- Human-assisted.
- Produces game-based proof, similar technique to hand proofs.

# Symbolic and Computational Models

## Symbolic Model

- Primitives are “perfect” black boxes.
- No algebraic or numeric values.
- Can be fully automated.
- Produces verification of no contradictions (theorem assures no missed attacks).

## Computational Model

- Primitives are nuanced (IND-CPA, IND-CCA, etc.)
- Security bounds ( $2^{128}$ , etc.)
- Human-assisted.
- Produces game-based proof, similar technique to hand proofs.

# Symbolic Verification, Still?

- Research in symbolic verification is still producing novel results:
  - *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman* – Cas Cremers and Dennis Jackson
  - *Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures* – Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon and Ralf Sasse
- 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.

# Symbolic Verification, Still?

- Research in symbolic verification is still producing novel results:
  - *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman* – Cas Cremers and Dennis Jackson
  - *Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures* – Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon and Ralf Sasse
- 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.

# Symbolic Verification, Still?

- Research in symbolic verification is still producing novel results:
  - *Prime, Order Please! Revisiting Small Subgroup and Invalid Curve Attacks on Protocols using Diffie-Hellman* – Cas Cremers and Dennis Jackson
  - *Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures* – Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon and Ralf Sasse
- 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.



# Symbolic Verification, Still?

- Research in symbolic verification has produced some great results:
  - *Prime, Order Please*: Symbolic Verification of Curve Attacks on ECC – Dennis Jackson
  - *Seems Legit: Automated Verification of Protocols that Use Signatures* – Dennis Jackson, G. Sasse
- Many papers published (and finding attacks on) TLS, Bluetooth, 5G and much more!
- This is a great tool to use to get a better understanding of their protocols before/as they are implemented.

**SO WHY ISN'T IT  
USED MORE?!**

# 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 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 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 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.


Tool	Unbound	Eq-thy	State	Trace	Equiv	Link
CPSA [17]	●	○	●	●	○	○
F7 [18]	●	◐	●	●	○	●
Maude-NPA [19]	●	●	○	●	●	○
ProVerif [20]	●	◐	○	●	●	○
Scyther [21]	●	○	○	●	○	○
Tamarin [22]	●	●	●	●	●	○
DEEPSPEC [23]	○	◐	●	○	●	○
VERIFPAL	●	◐	●	●	◐	◐

**SoK: Computer-Aided Cryptography**

Manuel Barbosa and Gilles Barthe and Karthik Bhargavan and Bruno Blanchet and Cas Cremers and Kevin Liao and Bryan Parno

# 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.



Tool	Unbound	Eq-thy	State	Trace	Equiv	Link
CPSA [17]	●	○	●	●	○	○
F7 [18]	●	◐	●	●	○	●
Maude-NPA [19]	●	●	○	●	●	○
ProVerif [20]	●	◐	○	●	●	○
Scyther [21]	●	○	○	●	○	○
Tamarin [22]	●	●	●	●	●	○
DEEPSPEC [23]	○	◐	●	○	●	○
VERIFPAL	●	◐	●	●	◐	◐

**SoK: Computer-Aided Cryptography**

Manuel Barbosa and Gilles Barthe and Karthik Bhargavan and Bruno Blanchet and Cas Cremers and Kevin Liao and Bryan Parno

# Tamarin and ProVerif: Examples

# 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) ]
```





# 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) ]
```

Tamarin

```
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)))));
```

ProVerif

# Verifpal: New Protocol Analysis Software

---

# Verifpal: New Protocol Analysis Software

---

1. An intuitive language for modeling protocols.



# Verifpal: New Protocol Analysis Software

---

1. An intuitive language for modeling protocols.
2. Modeling that avoids user error.



# Verifpal: New Protocol Analysis Software

---

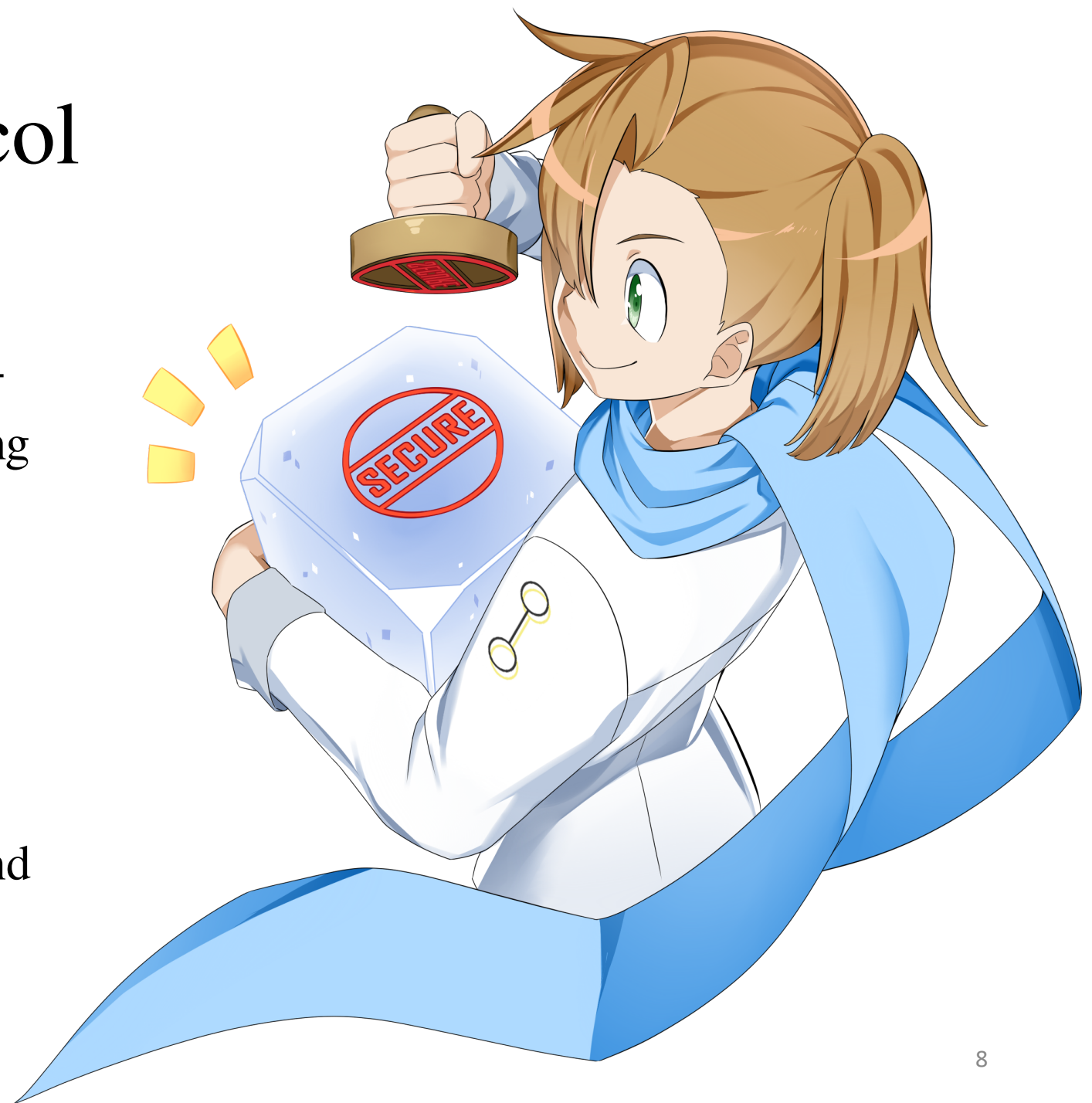
1. An intuitive language for modeling protocols.
2. Modeling that avoids user error.
3. Analysis output that's easy to understand.



# Verifpal: New Protocol Analysis Software

---

1. An intuitive language for modeling protocols.
2. Modeling that avoids user error.
3. Analysis output that's easy to understand.
4. IDE integration (Visual Studio Code), translations to ProVerif and Coq.





# 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, DP-3T) 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.

# 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

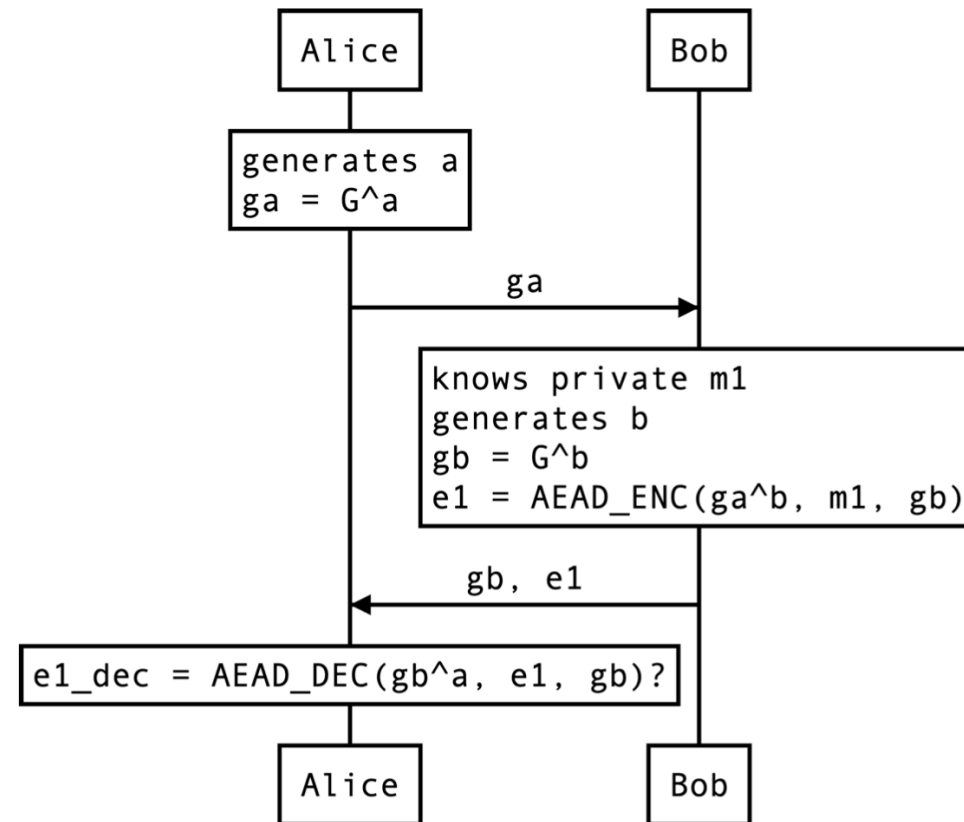
- Can reason about advanced protocols (eg. Signal, DP-3T) out of the box.
- Can analyze for forward secrecy, key compromise impersonation and other advanced queries.
- Unbounded sessions, fresh values, and other cool symbolic model features.



# 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: Hashing Primitives

- 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 or Tamarin, they're really quite excellent!)
- `HASH(a, b...): x.`  
Secure hash function, similar in practice to, for example, BLAKE2s [10]. Takes between 1 and 5 inputs and returns one output.
  - `MAC(key, message): hash.`  
Keyed hash function. Useful for message authentication and for some other protocol constructions.
  - `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 between 1 and 5 outputs.
  - `PW_HASH(a...): x.`  
Password hashing function, similar in practice to, for example, Scrypt [12] or Argon2 [13]. Hashes passwords and produces output that is suitable for use as a private key, secret key or other sensitive key material. Useful in conjunction with values declared using `knows password a.`

*Verifpal will never be “better” than ProVerif, Tamarin, etc. — we are targeting a different class of user entirely*

# Verifpal Language: Hashing Primitives

- 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 or Tamarin, they're really quite excellent!)

*Verifpal will never be “better” than ProVerif, Tamarin, etc. — we are targeting a different class of user entirely*

- `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 §2.3.2 below for information on how to validate successfully authenticated decryption.
- `PKE_ENC(G^key, plaintext): ciphertext.`  
Public-key encryption.
- `PKE_DEC(key, PKE_ENC(G^key, plaintext)): plaintext.`  
Public-key decryption.

# Verifpal Language: Hashing Primitives

- 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 or Tamarin, they're really quite excellent!)

*Verifpal will never be “better” than ProVerif, Tamarin, etc. — we are targeting a different class of user entirely*

- **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 §2.3.2 below for information on how to validate this check.
- **RINGSIGN**(key\_a, G^key\_b, G^key\_c, message): signature.  
Ring signature.  
In ring signatures, one of three parties (Alice, Bob and Charlie) signs a message. The resulting signature can be verified using the public key of any of the three parties, and the signature does not reveal the signatory, only that they are a member of the signing ring (Alice, Bob or Charlie). The first key must be the private key of the actual signer, while the subsequent two keys must be the public keys of the other potential signers.
- **RINGSIGNVERIF**(G^a, G^b, G^c, m, RINGSIGN(a, G^b, G^c, m)): m.  
Verifies if a ring signature can be authenticated.  
The signer's public key must match one or more of the public keys provided, but the public keys may be provided in any order and not necessarily in the order used during the **RINGSIGN** operation. Output value is not necessarily used; see §2.3.2 below for information on how to validate this check.
- **BLIND**(k, m): m.  
Message blinding primitive, useful for the implementation of blind signatures. Here, the sender uses the secret “blinding factor” k in order to blind message m, which can then be sent to the signer, who will be able to produce a signature on m without knowing m. Used in conjunction with **UNBLIND** – see **UNBLIND**'s documentation for more information.

# Verifpal Language: Hashing Primitives

- 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 or Tamarin, they're really quite excellent!)
- **SHAMIR\_SPLIT**(k): s1, s2, s3.  
In Verifpal, we allow splitting the key into three shares such that only two shares are required to reconstitute it.
- **SHAMIR\_JOIN**(sa, sb): k.  
Here, sa and sb must be two distinct elements out of the set (s1, s2, s3) in order to obtain k.

*Verifpal will never be “better” than ProVerif, Tamarin, etc. — we are targeting a different class of user entirely*

# Guarded Constants, Checked Primitives

## Challenge-Response Protocol

```
attacker[active]
principal Server [
  knows private s
  gs = G^s
]
principal Client[
  knows private c
  gc = G^c
  generates nonce
]
Client → Server: nonce
principal Server[
  proof = SIGN(s, nonce)
]
Server → Client: gs, proof
principal Client[
  valid = SIGNVERIF(gs, nonce, proof)
  generates attestation
  signed = SIGN(c, attestation)
]
Client → Server: [gc], attestation, signed
principal Server[
  storage = SIGNVERIF(gc, attestation, signed)?
]
queries[
  authentication? Server → Client: proof
  authentication? Client → Server: signed
]
```

# Guarded Constants, Checked Primitives

- This challenge-response protocol is broken:
  - Attacker can man-in-the-middle *gs*.
  - Client will send *valid* even if signature verification fails.

## Challenge-Response Protocol

```
attacker[active]
principal Server [
  knows private s
  gs = G^s
]
principal Client[
  knows private c
  gc = G^c
  generates nonce
]
Client → Server: nonce
principal Server[
  proof = SIGN(s, nonce)
]
Server → Client: gs, proof
principal Client[
  valid = SIGNVERIF(gs, nonce, proof)
  generates attestation
  signed = SIGN(c, attestation)
]
Client → Server: [gc], attestation, signed
principal Server[
  storage = SIGNVERIF(gc, attestation, signed)?
]
queries[
  authentication? Server → Client: proof
  authentication? Client → Server: signed
]
```

# Guarded Constants, Checked Primitives

- This challenge-response protocol is broken:
  - Attacker can man-in-the-middle *gs*.
  - Client will send *valid* even if signature verification fails.
  - Adding brackets around *gs* “guards” it against replacement by the active attacker.
  - Adding a question mark after *SIGNVERIF* makes the model abort execution if it fails.

## Challenge-Response Protocol

```
attacker[active]
principal Server [
  knows private s
  gs = G^s
]
principal Client[
  knows private c
  gc = G^c
  generates nonce
]
Client → Server: nonce
principal Server[
  proof = SIGN(s, nonce)
]
Server → Client: gs, proof
principal Client[
  valid = SIGNVERIF(gs, nonce, proof)
  generates attestation
  signed = SIGN(c, attestation)
]
Client → Server: [gc], attestation, signed
principal Server[
  storage = SIGNVERIF(gc, attestation, signed)?
]
queries[
  authentication? Server → Client: proof
  authentication? Client → Server: signed
]
```



# Guarded Constants, Checked Primitives

- This challenge-response protocol is broken:
  - Attacker can man-in-the-middle *gs*.
  - Client will send *valid* even if signature verification fails.
  - Adding brackets around *gs* “guards” it against replacement by the active attacker.
  - Adding a question mark after *SIGNVERIF* makes the model abort execution if it fails.

## Challenge-Response Protocol

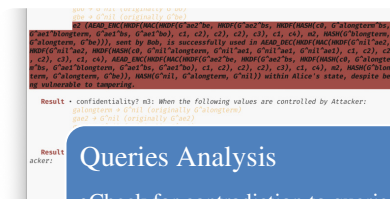
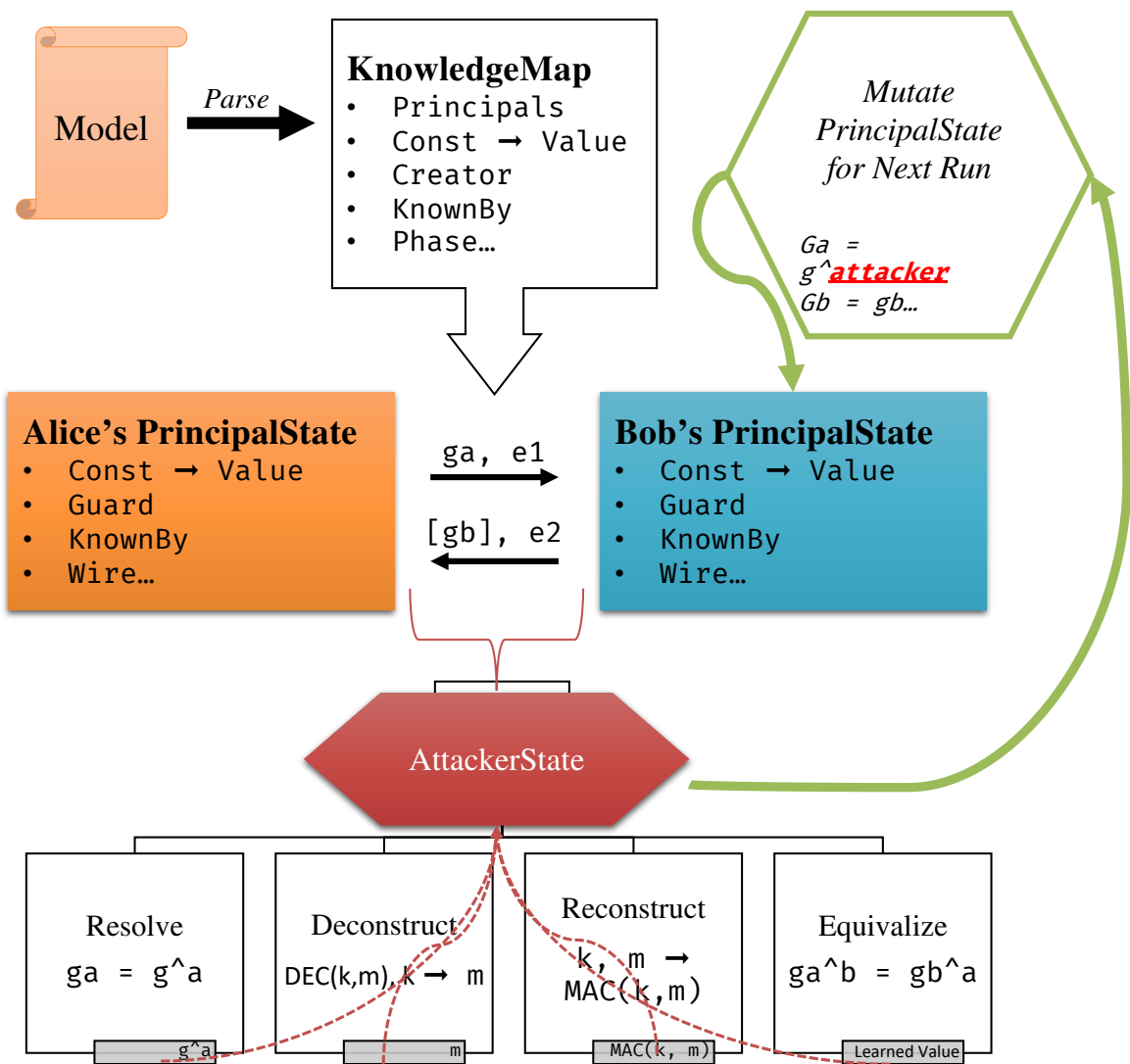
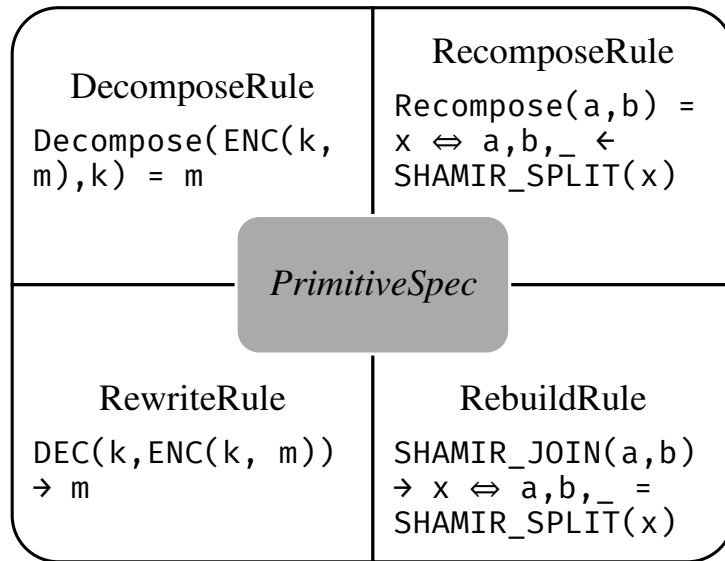
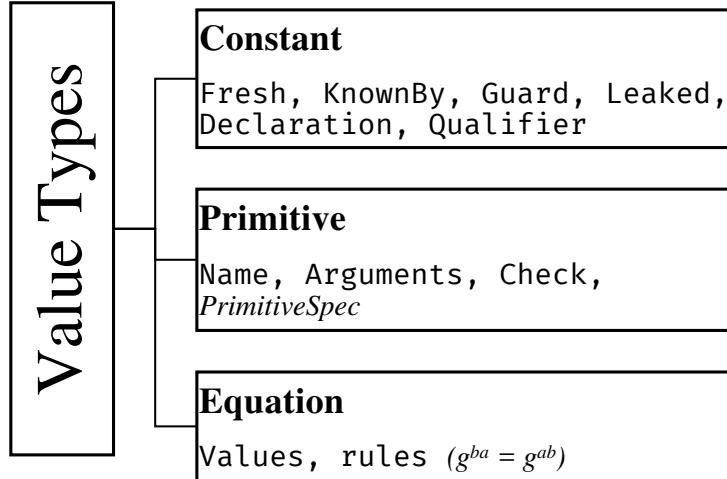
```
attacker[active]
principal Server [
  knows private s
  gs = G^s
]
principal Client[
  knows private c
  gc = G^c
  generates nonce
]
Client → Server: nonce
principal Server[
  proof = SIGN(s, nonce)
]
Server → Client: [gs], proof
principal Client[
  valid = SIGNVERIF(gs, nonce, proof)
  generates attestation
  signed = SIGN(c, attestation)
]
Client → Server: [gc], attestation, signed
principal Server[
  storage = SIGNVERIF(gc, attestation, signed)?
]
queries[
  authentication? Server → Client: proof
  authentication? Client → Server: signed
]
```

# Guarded Constants, Checked Primitives

- This challenge-response protocol is broken:
  - Attacker can man-in-the-middle *gs*.
  - Client will send *valid* even if signature verification fails.
- Adding brackets around *gs* “guards” it against replacement by the active attacker.
- Adding a question mark after *SIGNVERIF* makes the model abort execution if it fails.

## Challenge-Response Protocol

```
attacker[active]
principal Server [
  knows private s
  gs = G^s
]
principal Client[
  knows private c
  gc = G^c
  generates nonce
]
Client → Server: nonce
principal Server[
  proof = SIGN(s, nonce)
]
Server → Client: [gs], proof
principal Client[
  valid = SIGNVERIF(gs, nonce, proof)?
  generates attestation
  signed = SIGN(c, attestation)
]
Client → Server: [gc], attestation, signed
principal Server[
  storage = SIGNVERIF(gc, attestation, signed)?
]
queries[
  authentication? Server → Client: proof
  authentication? Client → Server: signed
]
```



**Queries Analysis**

- Check for contradiction to queries after each run
- Terminate when no new values are being learned



**Translate to Coq**

- Work with Coq Library to perform more in-depth analysis



**Protocol Modeling and Verification**

- Iterative process through intuitive modeling and optional further Coq modeling

# Verifpal: Advanced Features

- *Protocol phases* for temporal logic (forward secrecy, post-compromise security).
- *Leaking values* to the attacker (without necessarily sending a message).
- Unlinkability queries, freshness queries.
- *Password* values that are “crackable” unless first hashed using a password-hashing function.
- *Query preconditions*: check if a query is satisfied if and only if another query is satisfied also.

# Verifpal for Visual Studio Code

- Syntax highlighting, model formatting, code completion.
- Protocol diagrams, update live with your model,
- Insight on hover: show more information about values, queries, etc.
- Live analysis within Visual Studio Code!



# Verifpal Translations: Coq and ProVerif

- Verifpal models can be translated to Coq models (complete with formal semantics, lemmas and proofs on primitives),
- ProVerif model templates for further analysis in ProVerif and potentially CryptoVerif.

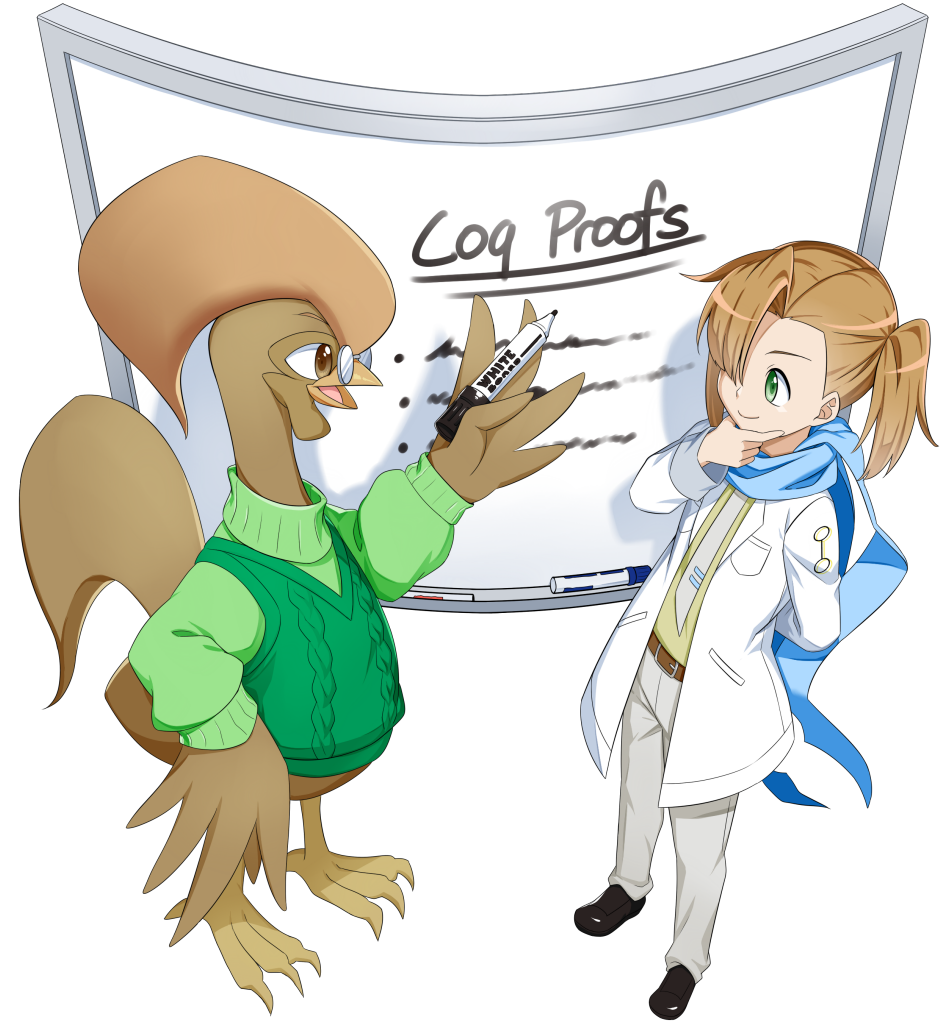


# Verifpal Translations: Coq and ProVerif

- Verifpal models can be translated to Coq models (complete with formal semantics, lemmas and proofs on primitives),
- ProVerif model templates for further analysis in ProVerif and potentially CryptoVerif.

Coq: Verifpal Symmetric Encryption

```
Definition ENC(key plaintext: constant): constant := ENC_c key plaintext.  
Definition DEC(key ciphertext: constant): constant :=  
  match ciphertext with  
  | ENC_c k m => match k =? key with  
  | true => m | false => ENC_c k m end  
  | _ => ciphertext end.  
Theorem enc_dec: forall k m: constant, DEC k (ENC k m) = m.  
Proof.  
  unfold ENC, DEC; intros k m;  
  rewrite equal_constant_true; try auto.  
Qed.
```



# Easier to Read Analysis Output

**Listing 1.1. ProVerif Attack Trace**

```
new skB: skey creating skB_2 at {1}
out(c, ~M) with ~M = pk(skB_2) at {3}
new n1_1: nonce creating n1_2 at {9} in copy a
new n2_1: nonce creating n2_2 at {10} in copy a
out(c, (~M_1,~M_2)) with ~M_1 = n1_2, ~M_2 = n2_2 at
    {11} in copy a
new n1_1: nonce creating n1_3 at {9} in copy a_1
new n2_1: nonce creating n2_3 at {10} in copy a_1
out(c, (~M_3,~M_4)) with ~M_3 = n1_3, ~M_4 = n2_3 at
    {11} in copy a_1
in(c, (~M_3,~M_1)) with ~M_3 = n1_3, ~M_1 = n1_2 at {5}
    in copy a_2
out(c, ~M_5) with ~M_5 = encrypt((n1_3,n1_2,M),pk(skB_2
    )) at {6} in copy a_2
in(c, ~M_5) with ~M_5 = encrypt((n1_3,n1_2,M),pk(skB_2
    )) at {12} in copy a_1
out(c, (~M_6,~M_7)) with ~M_6 = n1_2, ~M_7 = encrypt((
    n1_2,M,n1_3),pk(skB_2)) at {14} in copy a_1
in(c, ~M_7) with ~M_7 = encrypt((n1_2,M,n1_3),pk(skB_2)
    ) at {12} in copy a
out(c, (~M_8,~M_9)) with ~M_8 = M, ~M_9 = encrypt((M,
    n1_3,n1_2),pk(skB_2)) at {14} in copy a
The attacker has the message ~M_8 = M.
```

**Listing 1.2. Verifpal Attack Trace**

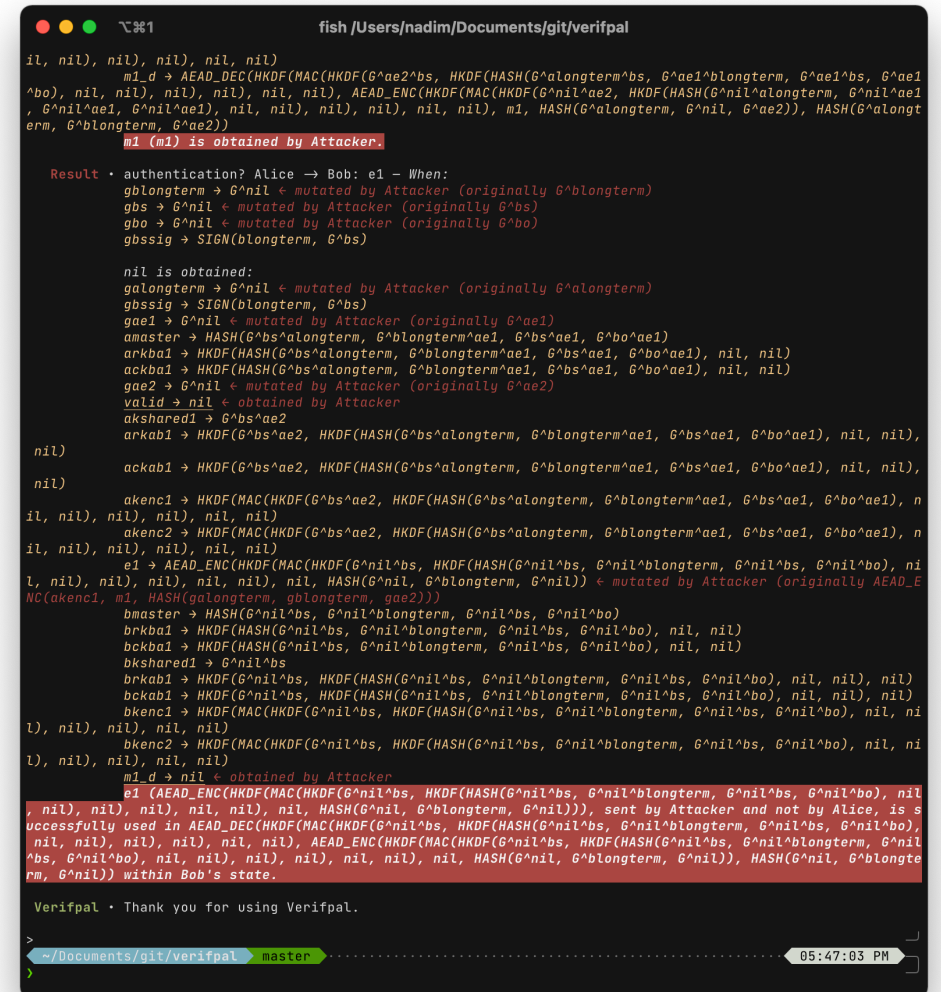
```
Result • confidentiality? m – When:
n1 → nil ← mutated by Attacker (was n1)
n2 → nil ← mutated by Attacker (was n2)
msg → PKE_ENC(G^skb, CONCAT(nil, n1, m))
clear → CONCAT(nil, n1, m)
x → nil
y1 → n1
y2 → m
unnamed_0 → ASSERT(nil, n1)?
msg2 → PKE_ENC(G^skb, CONCAT(n1, m, n1)) ←
    obtained by Attacker

m is obtained:
msg → PKE_ENC(G^skb, CONCAT(n1, m, n1)) ←
    mutated by Attacker
    (was PKE_ENC(pkb, CONCAT(n1, n2, m)))
clear → CONCAT(n1, m, n1)
x → n1
y1 → m ← obtained by Attacker
y2 → n1
unnamed_0 → ASSERT(n1, n1)?
msg2 → PKE_ENC(G^skb, CONCAT(m, n1, n1))
m (m) is obtained by Attacker.
```



# Protocols Analyzed with Verifpal

- Signal secure messaging protocol.
- Scuttlebutt decentralized protocol.
- ProtonMail encrypted email service.
- Telegram secure messaging protocol.
- DP-3T contact tracing protocol.



```
il, nil), nil), nil), nil, nil)
m1_d → AEAD_DEC(HKDF(MAC(HKDF(G^ae2^bs, HKDF(HASH(G^alongterm^bs, G^ae1^blongterm, G^ae1^bs, G^ae1
^bo), nil, nil), nil), nil), nil, nil), AEAD_ENC(HKDF(MAC(HKDF(G^nil^ae2, HKDF(HASH(G^nil^alongterm, G^nil^ae1
, G^nil^ae1, G^nil^ae1), nil, nil), nil), nil), m1, HASH(G^alongterm, G^nil, G^ae2)), HASH(G^alongt
erm, G^blongterm, G^ae2))
m1 (m1) is obtained by Attacker.

Result • authentication? Alice → Bob: e1 - When:
gblongterm → G^nil ← mutated by Attacker (originally G^blongterm)
gbs → G^nil ← mutated by Attacker (originally G^bs)
gbo → G^nil ← mutated by Attacker (originally G^bo)
gbssig → SIGN(blongterm, G^bs)

nil is obtained:
galongterm → G^nil ← mutated by Attacker (originally G^alongterm)
gbssig → SIGN(blongterm, G^bs)
gae1 → G^nil ← mutated by Attacker (originally G^ae1)
amaster → HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1)
arkba1 → HKDF(HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), nil, nil)
ackba1 → HKDF(HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), nil, nil)
gae2 → G^nil ← mutated by Attacker (originally G^ae2)
valid → nil ← obtained by Attacker
akshred1 → G^bs^ae2
arkab1 → HKDF(G^bs^ae2, HKDF(HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), nil, nil),
nil)
ackab1 → HKDF(G^bs^ae2, HKDF(HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), nil, nil),
nil)
akenc1 → HKDF(MAC(HKDF(G^bs^ae2, HKDF(HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), n
il, nil), nil), nil), nil, nil)
akenc2 → HKDF(MAC(HKDF(G^bs^ae2, HKDF(HASH(G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), n
il, nil), nil), nil), nil, nil)
e1 → AEAD_ENC(HKDF(MAC(HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), ni
l, nil), nil), nil, nil), nil, HASH(G^nil, G^blongterm, G^nil)) ← mutated by Attacker (originally AEAD_E
NC(akenc1, m1, HASH(galongterm, gblongterm, gae2)))
bmaster → HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo)
brkba1 → HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil, nil)
bckba1 → HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil, nil)
bshred1 → G^nil^bs
brkab1 → HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil, nil), nil)
bckab1 → HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil, nil), nil)
bkenc1 → HKDF(MAC(HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil, ni
l), nil), nil), nil, nil)
bkenc2 → HKDF(MAC(HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil, ni
l), nil), nil), nil, nil)
m1_d → nil ← obtained by Attacker
e1 (AEAD_ENC(HKDF(MAC(HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs, G^nil^bo), nil
, nil), nil), nil), nil, HASH(G^nil, G^blongterm, G^nil))), sent by Attacker and not by Alice, is s
uccessfully used in AEAD_DEC(HKDF(MAC(HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil^bs,
G^nil^bo), nil, nil), nil), nil), AEAD_ENC(HKDF(MAC(HKDF(G^nil^bs, HKDF(HASH(G^nil^bs, G^nil^blongterm, G^nil
^bs, G^nil^bo), nil, nil), nil), nil), nil, HASH(G^nil, G^blongterm, G^nil))), HASH(G^nil, G^blongte
rm, G^nil)) within Bob's state.

Verifpal • Thank you for using Verifpal.

~/Documents/git/verifpal master 05:47:03 PM
```

# Limitations and Context

- Does not produce proofs (like CryptoVerif)
- Is not formally proven to not miss attacks (like ProVerif)

Working towards obtaining higher confidence through building relationship to Coq models of verification method, more scrutiny, more protocols analyzed...

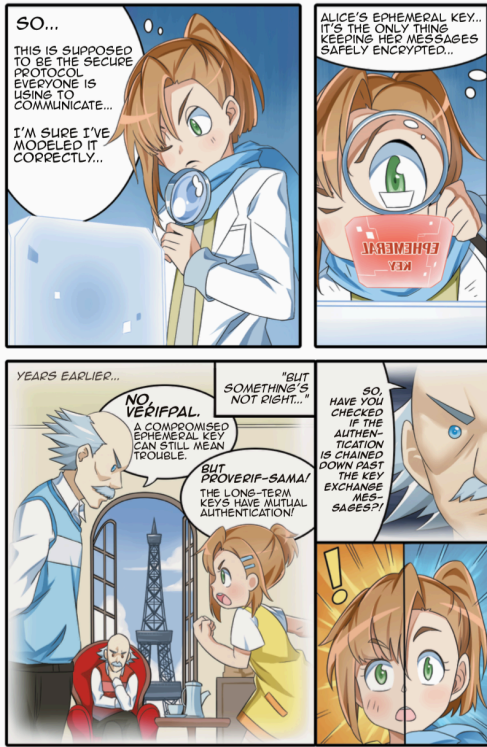
*Usefulness is more towards engineers and students.*

# Who's Using Verifpal?

**ASSA ABLOY**

**Quarkslab**  
SECURING EVERY BIT OF YOUR DATA





## Example Equations

```
principal Server{
  generates x
  generates y
  gx = G^x
  gy = G^y
  gxy = gx^y
  gyx = gy^x
}
```

In the above,  $gxy$  and  $gyx$  are considered equivalent by Verifpal. In Verifpal, all equations must have the constant  $G$  as their root generator. This mirrors Diffie-Hellman behavior. Furthermore, all equations can only have two constants ( $a^b$ ), but as we can see above, equations can be built on top of other equations (as in the case of  $gxy$  and  $gyx$ ).

## 2.6 MESSAGES

Sending messages over the network is simple. Only constants may be sent within messages:

## Example: Messages

```
Alice -> Bob: ga, e1
Bob -> Alice: [gb], e2
```

Let's look at the two messages above. In the first, Alice is the sender and Bob is the recipient. Notice how Alice is sending Bob her long-term public key  $ga = G^a$ . An active attacker could intercept  $ga$  and replace it with a value that they control. But what if we want to model our protocol such that Alice has pre-authenticated<sup>2</sup> Bob's public key  $gb = G^b$ ? This is where *guarded constants* become useful.

<sup>2</sup>"Pre-authentication" refers to Alice confirming the value of Bob's public key before the protocol session begins. This helps avoid having an active attacker trick Alice to use a fake public key for Bob. This fake public key could instead be the attacker's own public key.



## Guarding the Right Constants

Verifpal allows you to guard constants against modification by the active attacker. However, guarding all of a principal's public keys, for example, might not reflect real-world attack scenarios, where keys are rarely guarded from being modified as they cross the network.

What interesting new insights will you discover using guarded constants?

In the second message from the above example, we see that,  $gb$  is surrounded by brackets ( $[ ]$ ). This makes it a "guarded" constant, meaning that while an active attacker can still read it, they cannot tamper with it. In that sense it is "guarded" against the active attacker.

## 2.7 QUERIES

A Verifpal model is always concluded with a *queries* block, which contains essentially the questions that we will ask Verifpal to answer for us as a result of the model's analysis. Queries have an important role to play in a Verifpal model's constitution. The Verifpal language makes them very simple to describe, but you may benefit from learning more on how to properly use them in your models. For more information on queries, see §3. §2.8 below shows a quick example of how to illustrate queries in your model.

## 2.8 A SIMPLE COMPLETE EXAMPLE

Figure 2.1 provides a full model of a naive protocol where Alice and Bob only ever exchange unauthenticated public keys ( $G^a$  and  $G^b$ ). Bob then proceeds to send an encrypted message to Alice using the derived Diffie-Hellman shared secret to encrypt the message. We then want to ask Verifpal three questions:

We call this a *Man-in-the-Middle* attack.

# Verifpal in the Classroom

- Verifpal User Manual: easiest way to learn how to model and analyze protocols on the planet. *Comes with 3 example protocol models!*
- 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  
**HEROES**



# Verifpal Heroes

- Illustrated Guide to Protocol Verification
- Covers Coq, F\*, Tamarin, ProVerif, CryptoVerif, EasyCrypt and Verifpal
- Enhanced relationship between Verifpal and other paradigms + lots of new pedagogical materials
- Interactive online version + book version
- Coming in 2021



# Thank you, Georgio and Mukesh



# Thank you, Georgio and Mukesh



*Both attending RWC 2021!*

*Talk to them! These are great people to work with!*

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

Check out Verifpal today:

[verifpal.com](https://verifpal.com)

