Verifpal

Cryptographic protocol analysis for the real world

Nadim Kobeissi

Quarkslab Paris – June 19, 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 and Computational Models

Symbolic Model

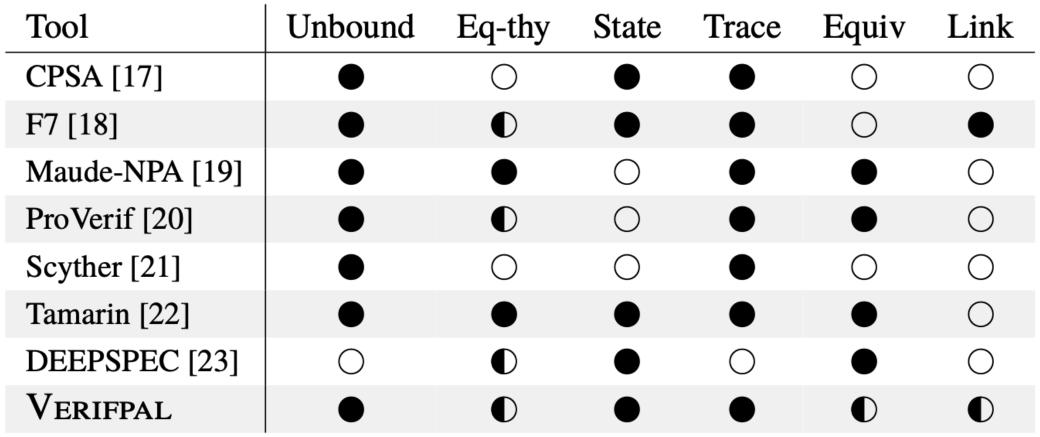
- Primitives are “perfect” black boxes.
- No algebraic or numeric values.
- Can be fully automated.
- Produces verification of no contradictions (theorem assures no missed attacks).

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 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	●	◐	●	●	◐	◐

Symbolic Verification, Still?

- **F* and computational models do not allow us to naturally express** and model protocols according to **a system based on discrete principals** with internal states.
- Writing a protocol in F* just to check it against security goals on a network: **unreasonable cost/benefit tradeoff.**
- **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

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) ]
```

Tamarin

(also not fully automated)

```
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: 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. IDE integration (Visual Studio Code), translations to ProVerif and Coq.

A New Approach to Symbolic Verification

User-focused approach…

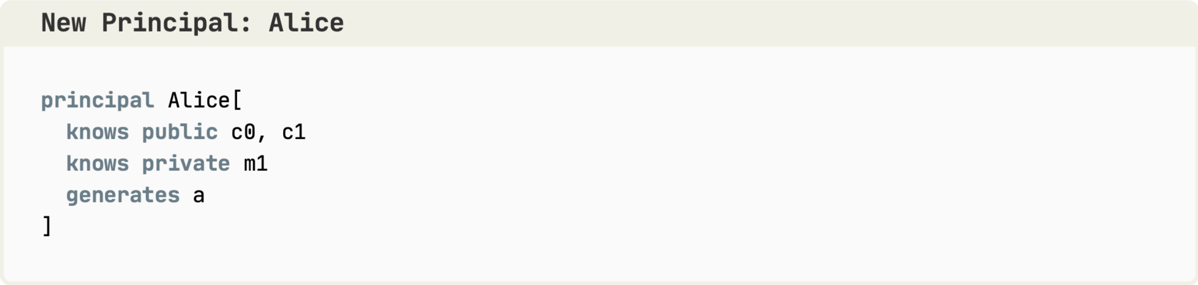
- An intuitive language for modeling protocols.
- Modeling that avoids user error.
- Analysis output that’s easy to understand.
- Integration with developer workflow.

…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.

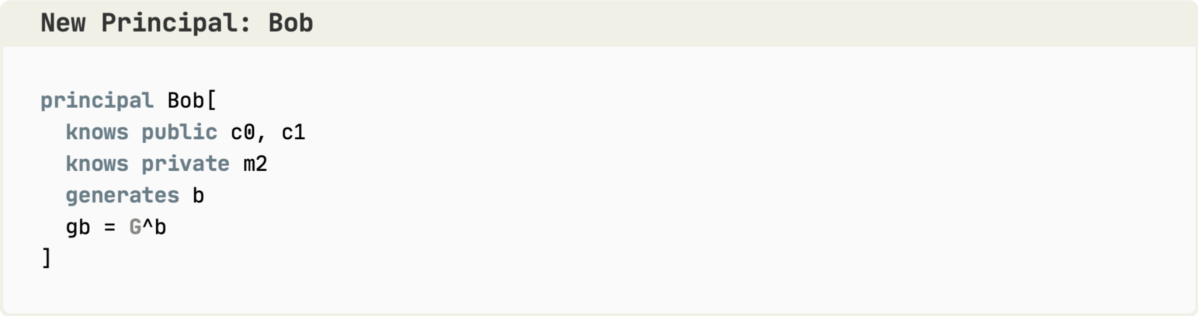
Verifpal Language

- 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.



New Principal: Alice

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



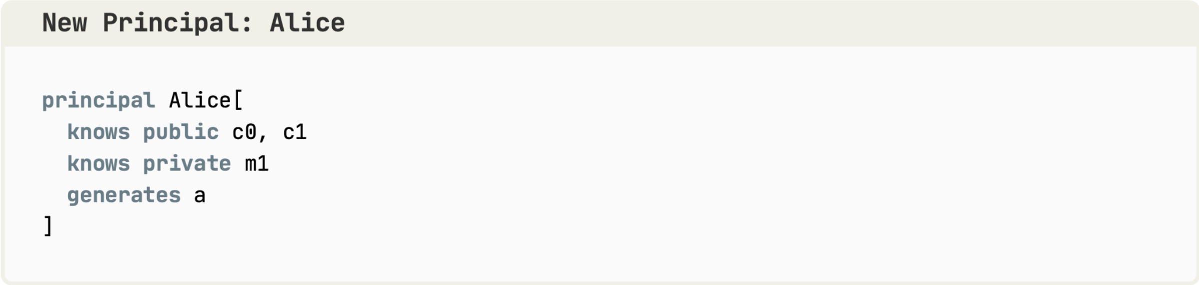
New Principal: Bob

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

Verifpal Language

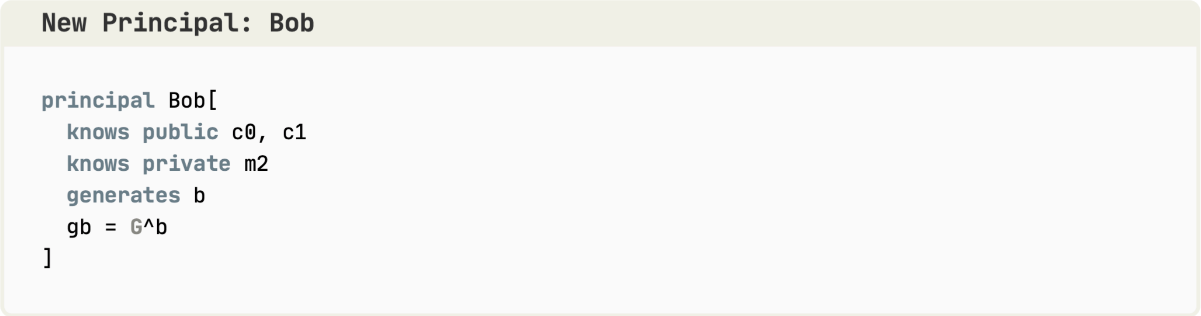
- Explicit principals with discrete internal states (Alice, Bob, Client, Server…)
- Reads like a protocol diagram.
- You don't need to know the language to understand it!

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



New Principal: Alice

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

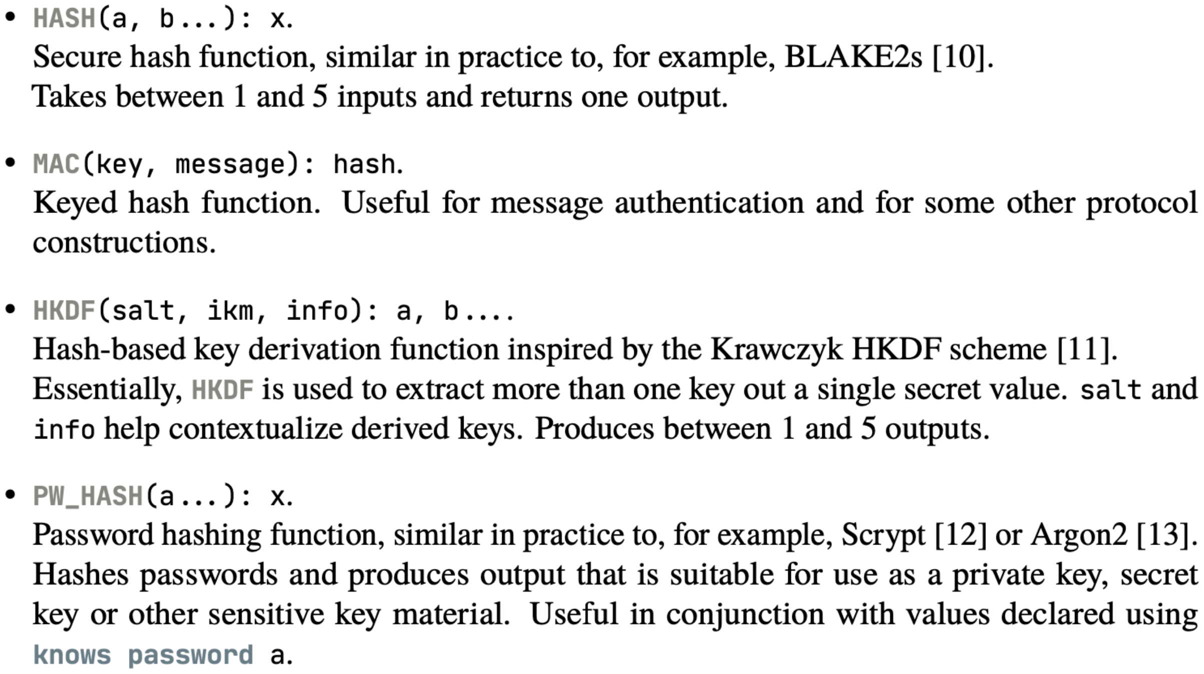


New Principal: Bob

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

Verifpal Language: Hashing Primitives

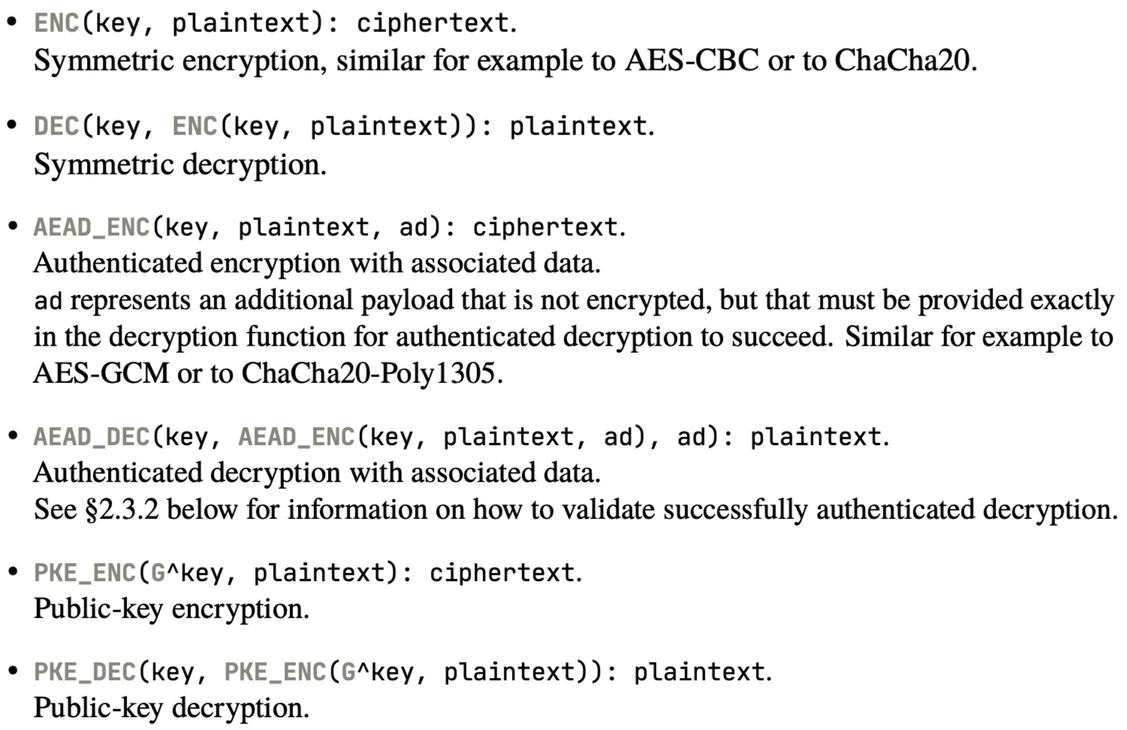
- Unlike ProVerif, primitives are *built-in.*
- Users cannot define their own primitives.
- Bug, not a feature: eliminate user error on the primitive level.
- Verifpal not targeting users interested in their own primitives (use ProVerif, it’s great!)



- `HASH(a, b...): x.`
  Secure hash function, similar in practice to, for example, BLAKE2s [10]. Takes 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 Language: Encryption Primitives

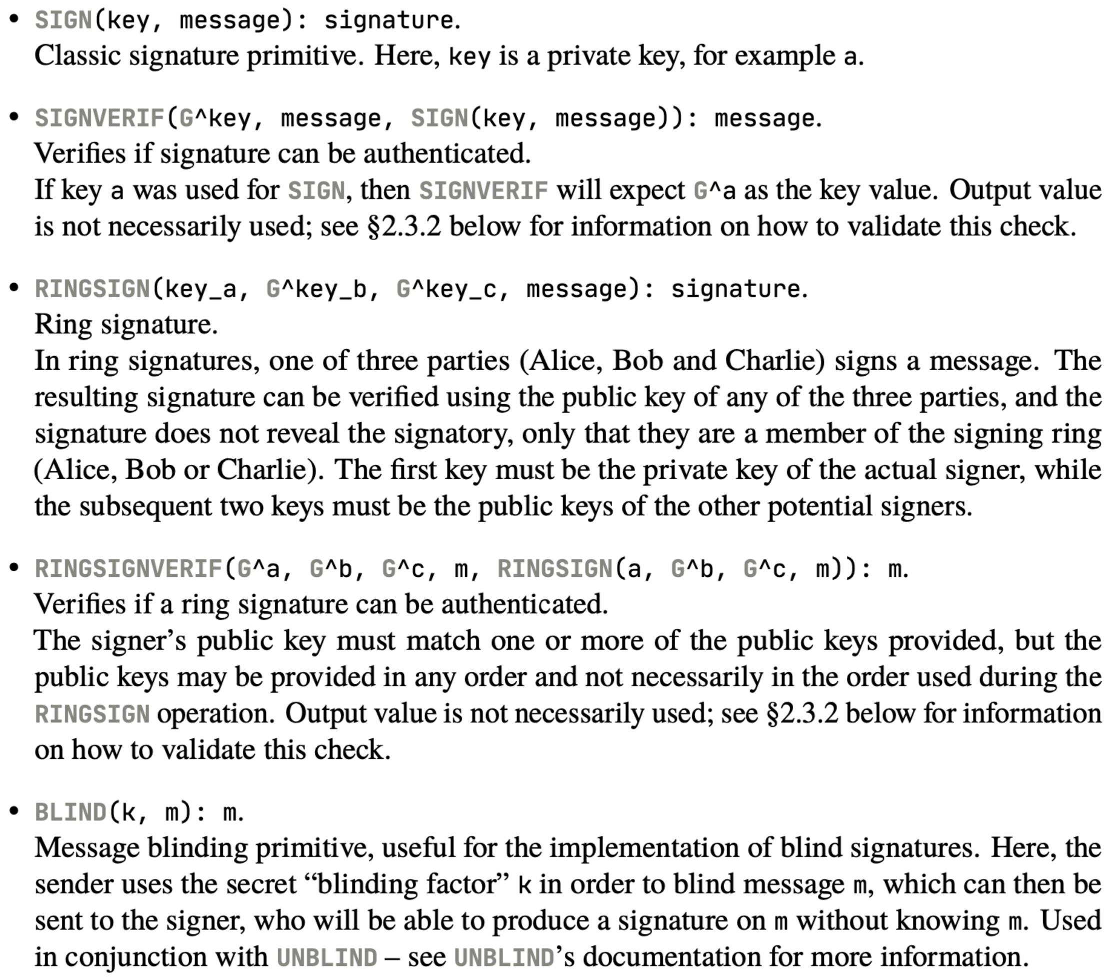
- Unlike ProVerif, primitives are *built-in*.
- Users cannot define their own primitives.
- Bug, not a feature: eliminate user error on the primitive level.
- Verifpal not targeting users interested in their own primitives (use ProVerif, it’s great!)



- `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 §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: Signing 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 §2.3.2 below for information on how to validate this check.

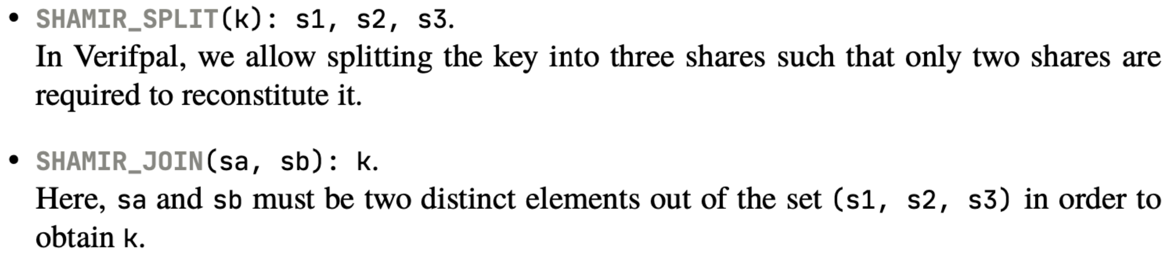
- `RINGSIGN(key_a, G^key_b, G^key_c, message): signature`.
  Ring signature.
  In ring signatures, one of three parties (Alice, Bob and Charlie) signs a message. The resulting signature can be verified using the public key of any of the three parties, and the signature does not reveal the signatory, 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.

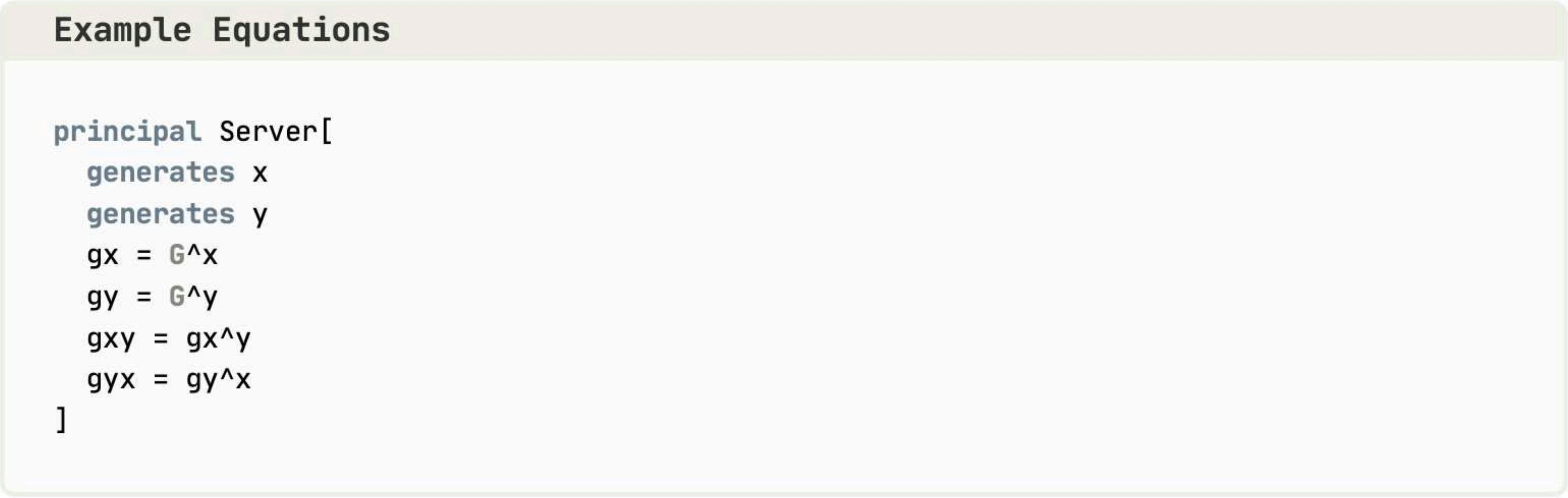
Verifpal Language: Secret Sharing Primitives

- 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!)



- `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`.

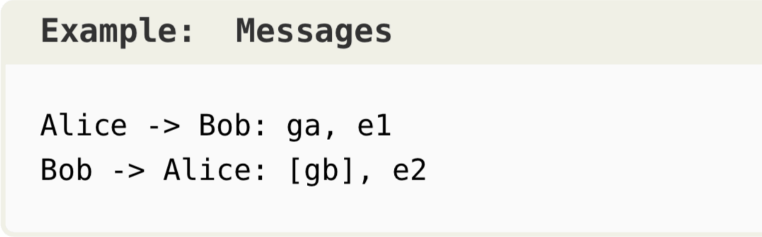
Verifpal Language: Equations



Example Equations

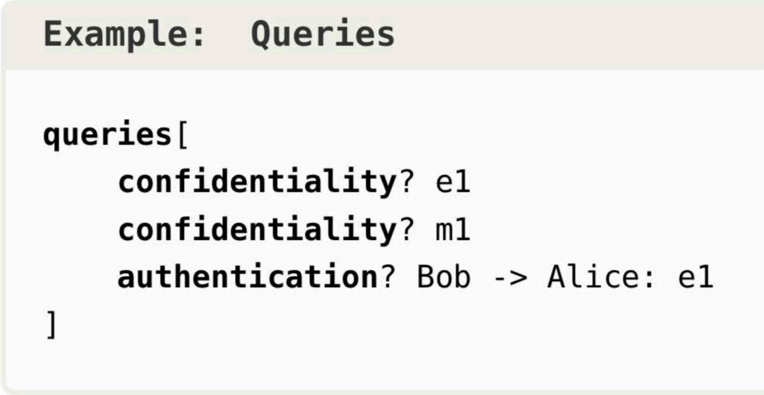
```
principal Server[
  generates x
  generates y
  gx = G^x
  gy = G^y
  gxy = gx^y
  gyx = gy^x
]
```

Verifpal Language: Messages and Queries



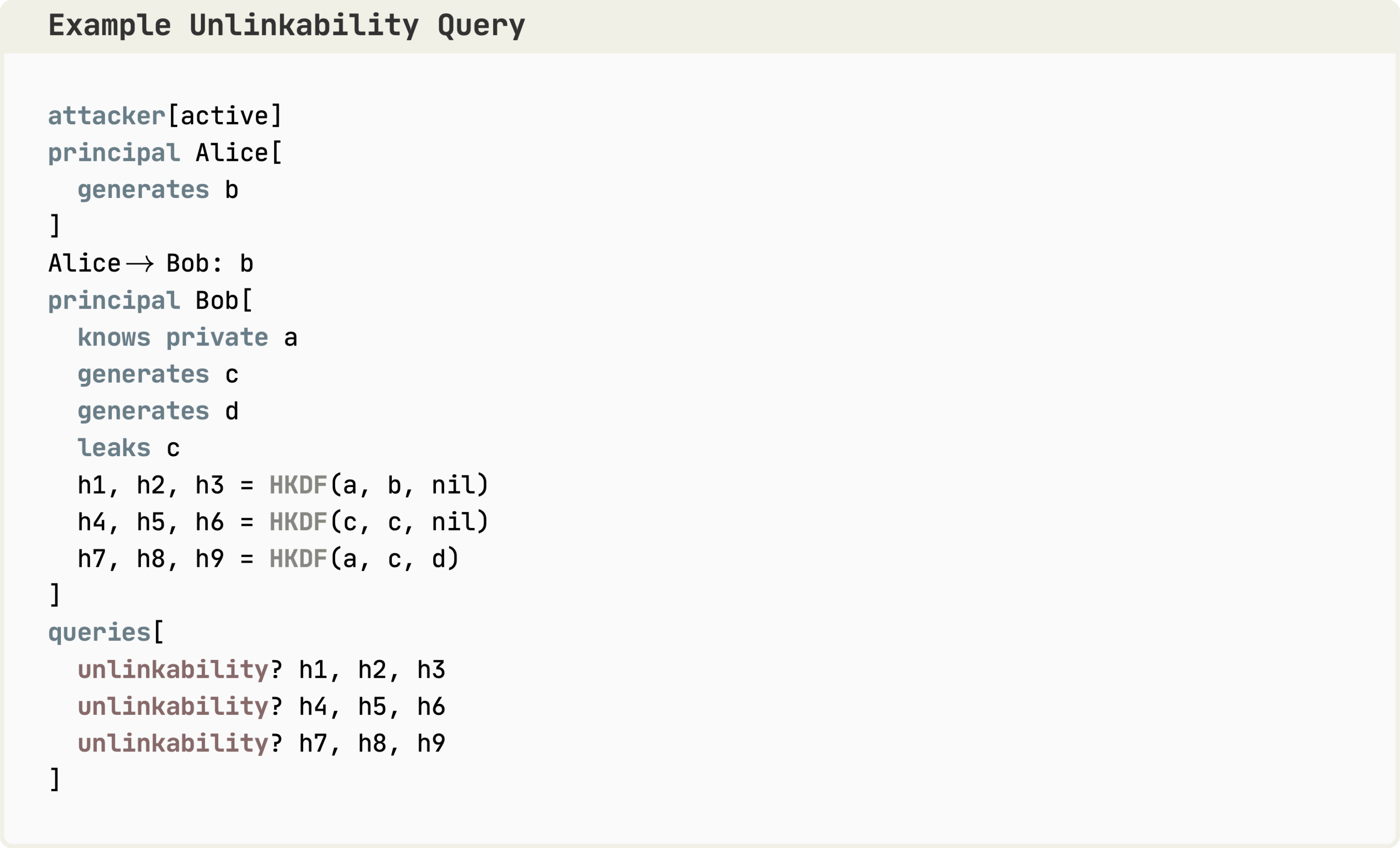
Example: Messages

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



Example: Queries

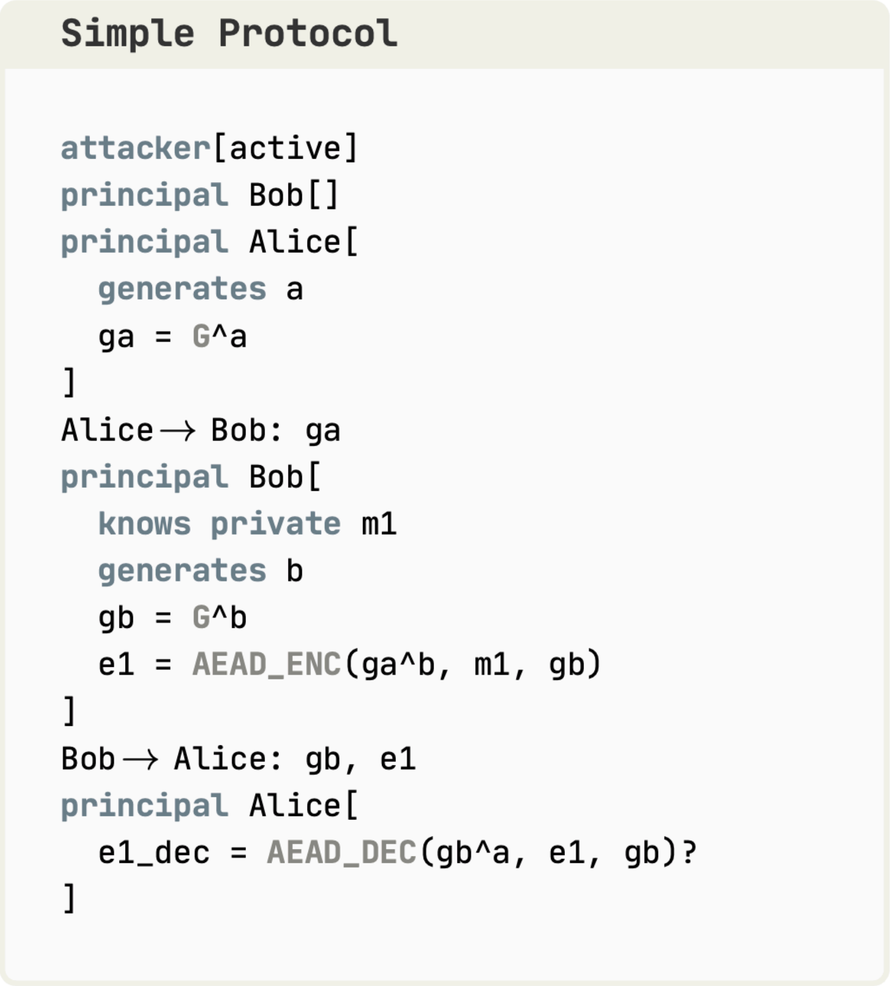
```
queries[
    confidentiality? e1
    confidentiality? m1
    authentication? Bob -> Alice: e1
]
```



Example Unlinkability Query

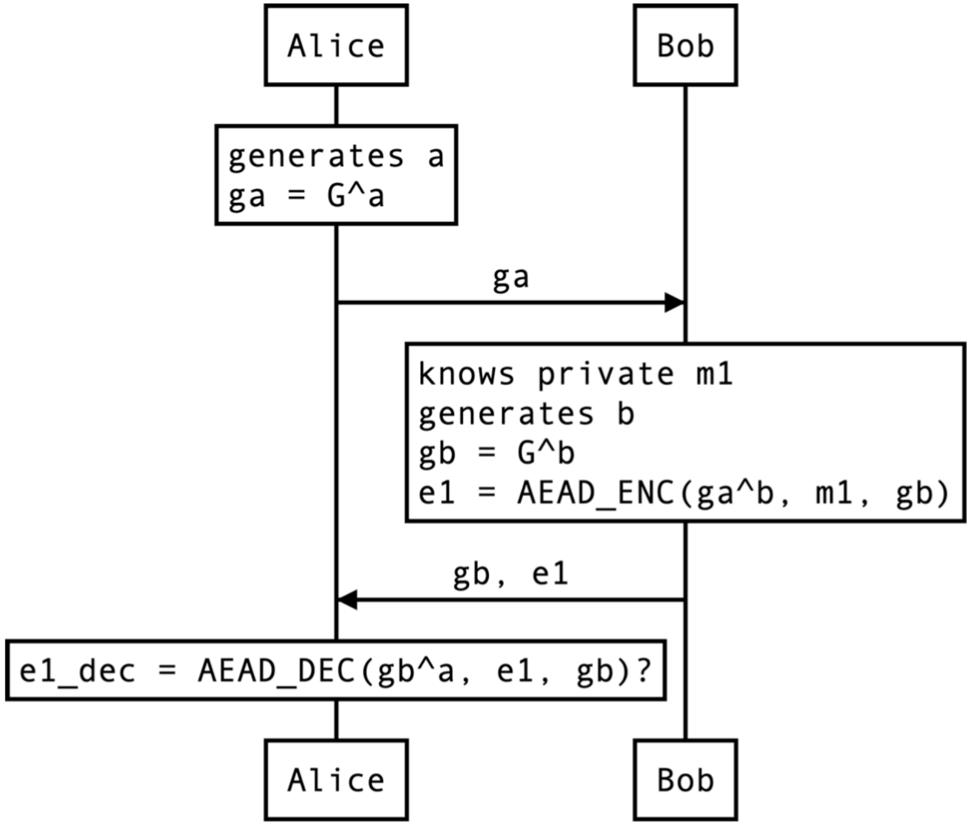
```
attacker[active]
principal Alice[
  generates b
]
Alice⟶ Bob: b
principal Bob[
  knows private a
  generates c
  generates d
  leaks c
  h1, h2, h3 = HKDF(a, b, nil)
  h4, h5, h6 = HKDF(c, c, nil)
  h7, h8, h9 = HKDF(a, c, d)
]
queries[
  unlinkability? h1, h2, h3
  unlinkability? h4, h5, h6
  unlinkability? h7, h8, h9
]
```

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)?
]
```



Alice

Bob

generates a
ga = G^a

ga

knows private m1
generates b
gb = G^b
e1 = AEAD_ENC(ga^b, m1, gb)

gb, e1

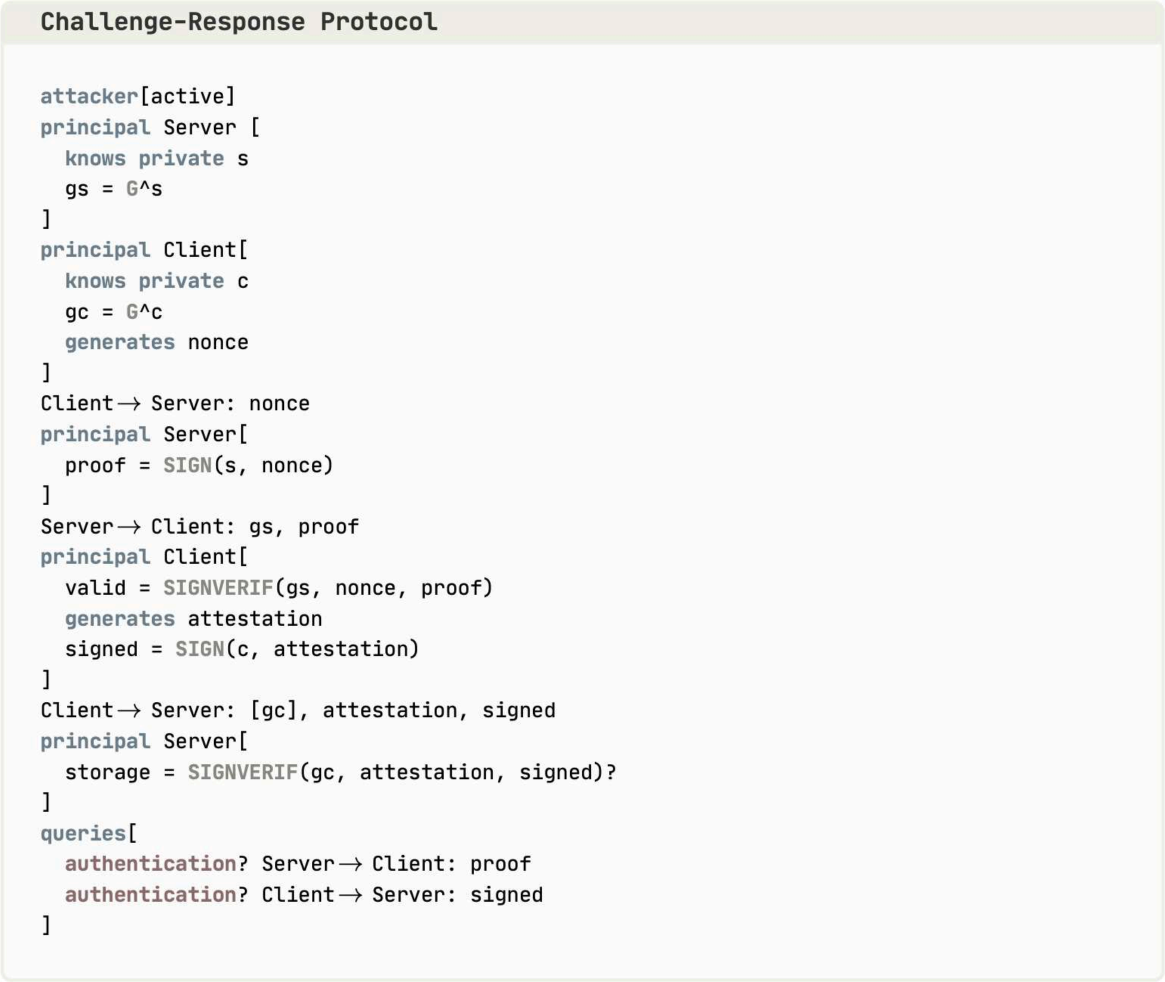
e1_dec = AEAD_DEC(gb^a, e1, gb)?

Alice

Bob

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.



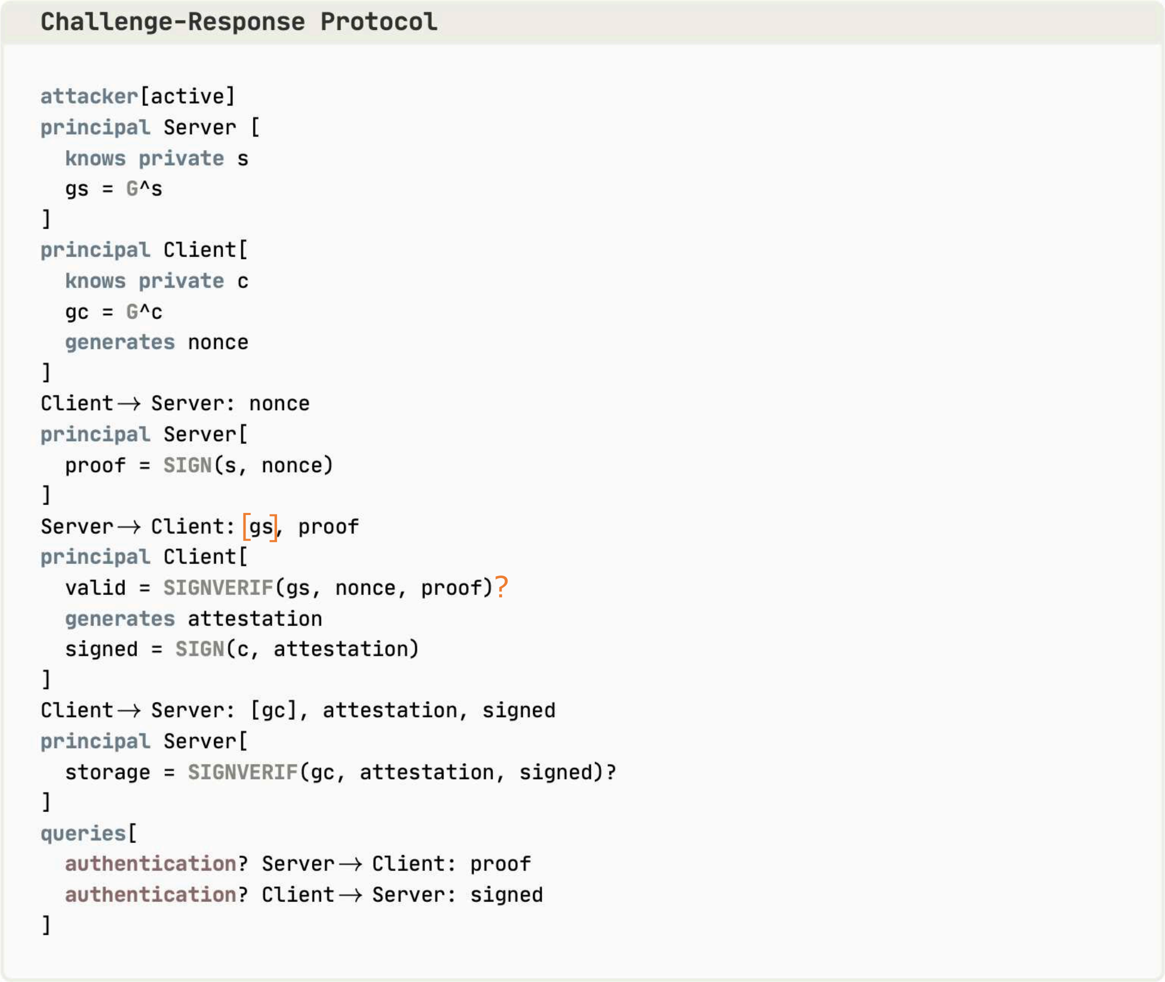
Challenge-Response Protocol

```
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
]
```

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.)

Verifpal Analysis Logic

- Resolve. Resolves a certain constant to its assigned value (for example, a primitive or an equation). Executed on $\mathcal{V}_A$, the set of all values known by the attacker.

- Deconstruct. Attempts to deconstruct a primitive or an equation. In order to deconstruct a primitive, the attacker must possess sufficient values to satisfy the primitive's rewrite rule. For example, the attacker must possess `k` and `e` in order to obtain `m` by deconstructing `e = ENC(k, m)` with `k`. In order to reconstruct an equation, the attacker must similarly possess all but one private exponent. Executed on $\mathcal{V}_A$, the set of all values known by the attacker.

- Reconstruct. Attempts to reconstruct primitives and equations given that the attacker possesses all of the component values. Executed on $\mathcal{V}_A$, the set of all values known by the attacker, as well as on $\mathcal{V}_P$, the values known by the principal whose state is currently being evaluated by the attacker.

- Equivalize. Determines if the attacker can reconstruct or equivalize any values within $\mathcal{V}_P$ from $\mathcal{V}_A$. If so, then these equivalent values are added to $\mathcal{V}_A$.

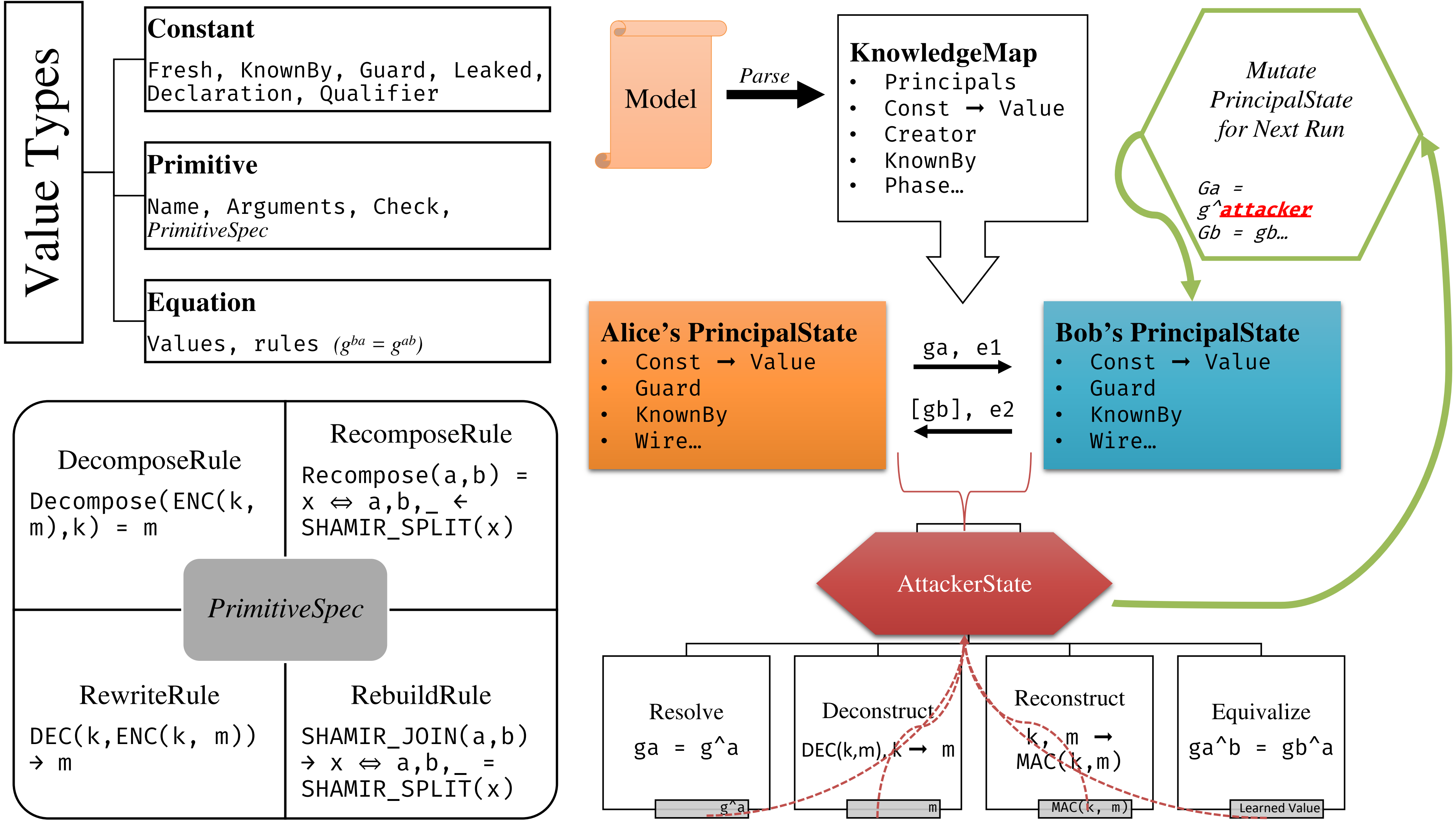
Verifpal Primitive Specifications *(PrimitiveSpec)*

- Decompose. Given a primitive's output and a defined subset of its inputs, reveal one of its inputs. (Given `ENC(k, m)` and `k`, reveal `m`).

- Recompose. Given a defined subset of a primitive's outputs, reveal one of its inputs. (Given `a, b`, reveal `x` if `a, b, _ = SHAMIR_SPLIT(x)`).

- Rewrite. Given a matching defined pattern within a primitive's inputs, rewrite the primitive expression itself into a logical subset of its inputs. (Given `DEC(k, ENC(k, m))`, rewrite the entire expression `DEC(k, ENC(k, m))` to `m`).

- Rebuild. Given a primitive whose inputs are all the outputs of some same other primitive, rewrite the primitive expression itself into a logical subset of its inputs. (Given `SHAMIR_JOIN(a, b)` where `a, b, c = SHAMIR_SPLIT(x)`, rewrite the entire expression `SHAMIR_JOIN(a, b)` to `x`).



Value Types

Constant

Fresh, KnownBy, Guard, Leaked, Declaration, Qualifier

Primitive

Name, Arguments, Check, *PrimitiveSpec*

Equation

Values, rules $(g^{ba} = g^{ab})$

DecomposeRule

`Decompose(ENC(k, m),k) = m`

RecomposeRule

`Recompose(a,b) = x ⇔ a,b,_ ← SHAMIR_SPLIT(x)`

PrimitiveSpec

RewriteRule

`DEC(k,ENC(k, m)) → m`

RebuildRule

`SHAMIR_JOIN(a,b) → x ⇔ a,b,_ = SHAMIR_SPLIT(x)`

Model

Parse

KnowledgeMap

- Principals
- Const → Value
- Creator
- KnownBy
- Phase…

Mutate PrincipalState for Next Run

Ga = g^attacker
Gb = gb…

Alice's PrincipalState

- Const → Value
- Guard
- KnownBy
- Wire…

ga, e1

[gb], e2

Bob's PrincipalState

- Const → Value
- Guard
- KnownBy
- Wire…

AttackerState

Resolve

ga = g^a

g^a

Deconstruct

DEC(k,m), k → m

m

Reconstruct

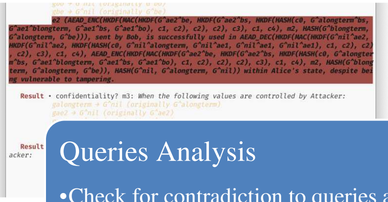
k, m → MAC(k,m)

MAC(k, m)

Equivalize

ga^b = gb^a

Learned Value



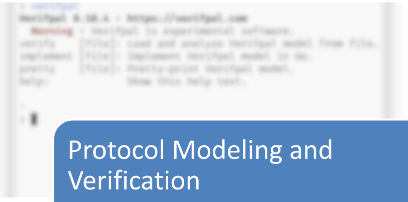
```
gbo → G^nil (originally G^bo)
gbe → G^nil (originally G^be)
e2 (AEAD_ENC(HKDF(MAC(HKDF(G^ae2^be, HKDF(G^ae2^bs, HKDF(HASH(c0, G^alongterm^bs,
G^ae1^blongterm, G^ae1^bs, G^ae1^bo), c1, c2), c2), c2), c3), c1, c4), m2, HASH(G^blongterm,
G^alongterm, G^be))), sent by Bob, is successfully used in AEAD_DEC(HKDF(MAC(HKDF(G^nil^ae2,
HKDF(G^nil^ae2, HKDF(HASH(c0, G^nil^alongterm, G^nil^ae1, G^nil^ae1, G^nil^ae1), c1, c2), c2)
, c2), c3), c1, c4), AEAD_ENC(HKDF(MAC(HKDF(G^ae2^be, HKDF(G^ae2^bs, HKDF(HASH(c0, G^alongter
m^bs, G^ae1^blongterm, G^ae1^bs, G^ae1^bo), c1, c2), c2), c2), c3), c1, c4), m2, HASH(G^blong
term, G^alongterm, G^be)), HASH(G^nil, G^alongterm, G^nil)) within Alice's state, despite bei
ng vulnerable to tampering.

Result • confidentiality? m3: When the following values are controlled by Attacker:
         galongterm → G^nil (originally G^alongterm)
         gae2 → G^nil (originally G^ae2)

Result
acker:
```

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

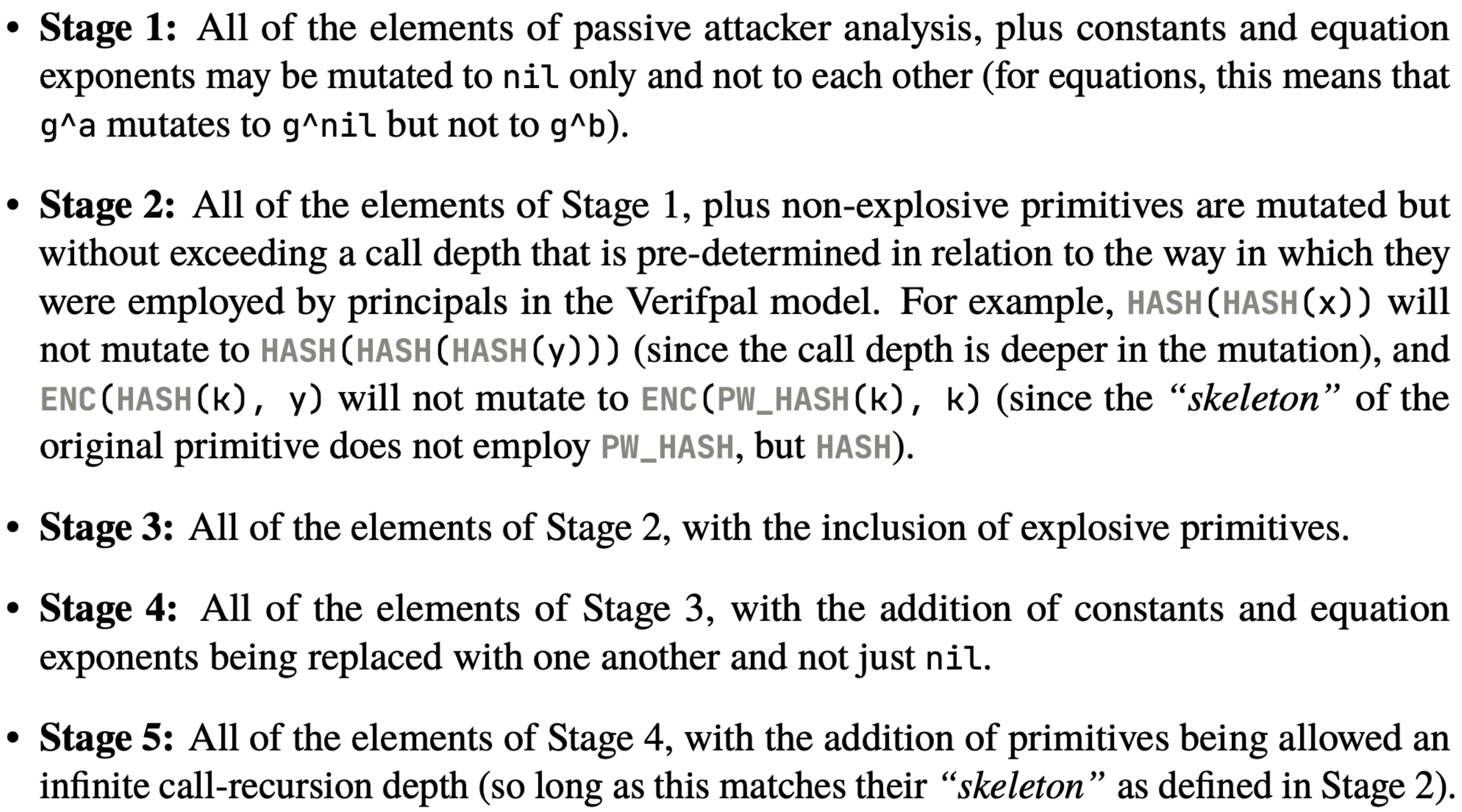


```
> verifpal
Verifpal 0.10.4 - https://verifpal.com
  Warning • Verifpal is experimental software.
verify    [file]: Load and analyze Verifpal model from file.
implement [file]: Implement Verifpal model in Go.
pretty    [file]: Pretty-print Verifpal model.
help:             Show this help text.
```

Protocol Modeling and Verification

- Iterative process through intuitive modeling and optional further Coq modeling

Preventing State Space Explosion with Stages



- **Stage 1:** All of the elements of passive attacker analysis, plus constants and equation exponents may be mutated to `nil` only and not to each other (for equations, this means that `g^a` mutates to `g^nil` but not to `g^b`).

- **Stage 2:** All of the elements of Stage 1, plus non-explosive primitives are mutated but without exceeding a call depth that is pre-determined in relation to the way in which they were employed by principals in the Verifpal model. For example, `HASH(HASH(x))` will not mutate to `HASH(HASH(HASH(y)))` (since the call depth is deeper in the mutation), and `ENC(HASH(k), y)` will not mutate to `ENC(PW_HASH(k), k)` (since the *“skeleton”* of the original primitive does not employ `PW_HASH`, but `HASH`).

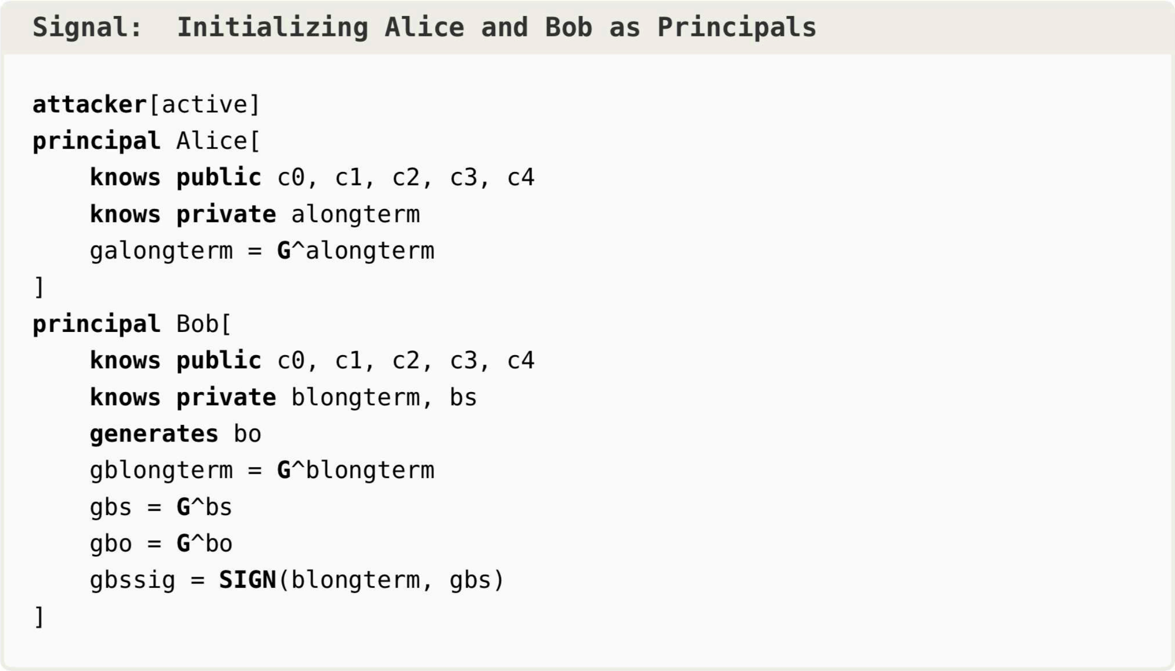
- **Stage 3:** All of the elements of Stage 2, with the inclusion of explosive primitives.

- **Stage 4:** All of the elements of Stage 3, with the addition of constants and equation exponents being replaced with one another and not just `nil`.

- **Stage 5:** All of the elements of Stage 4, with the addition of primitives being allowed an infinite call-recursion depth (so long as this matches their *“skeleton”* as defined in Stage 2).

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.

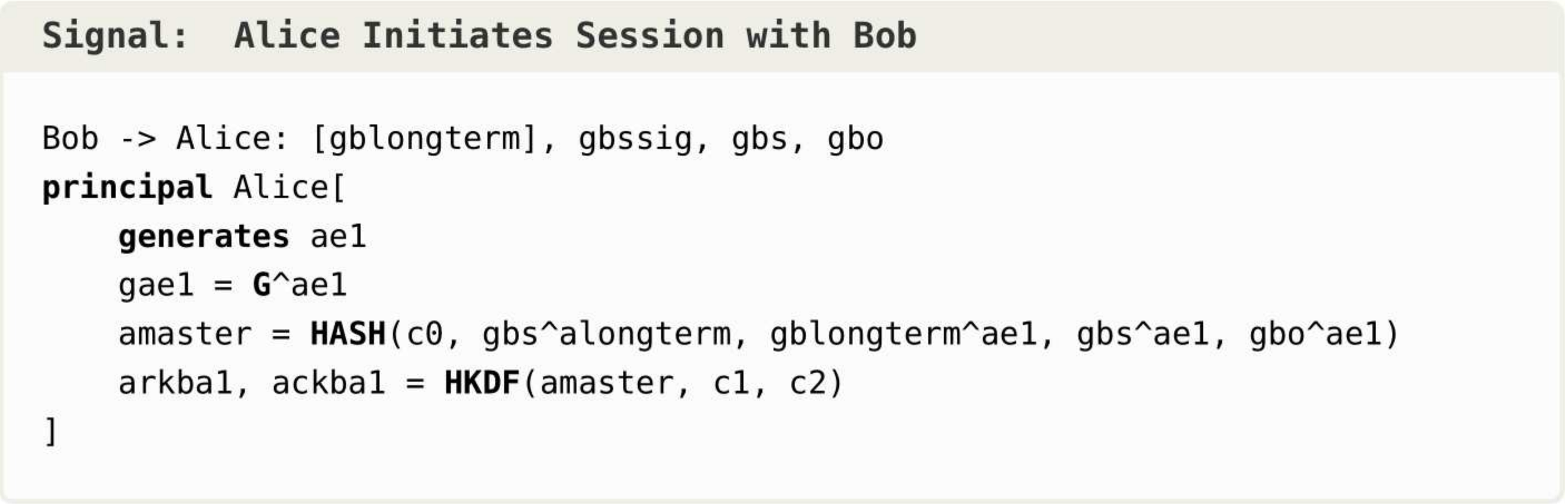


Signal: Initializing Alice and Bob as Principals

```
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

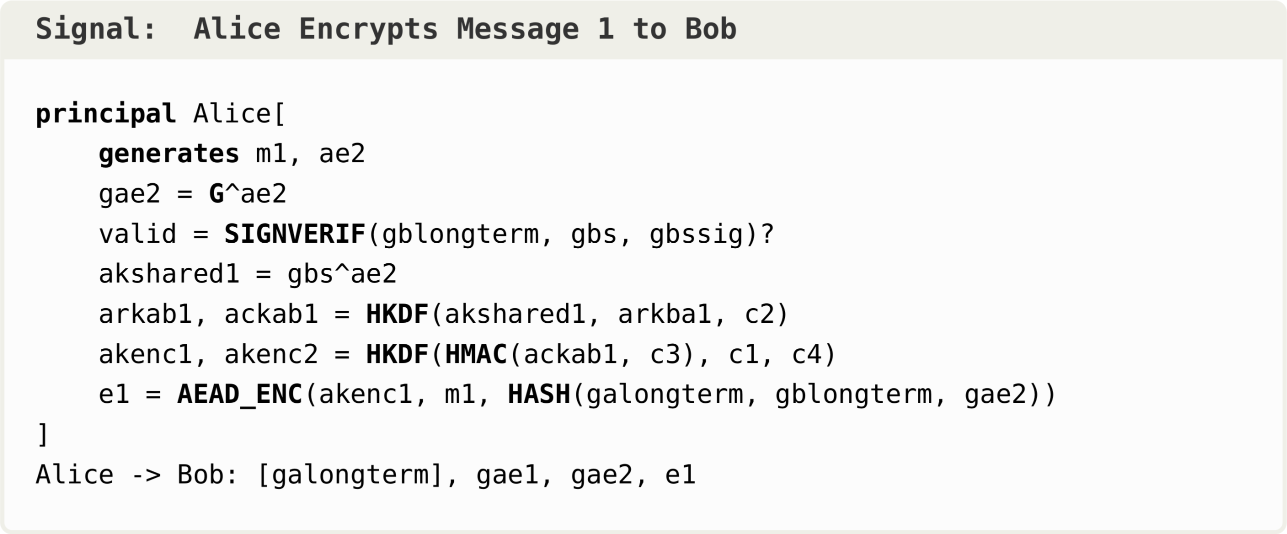
- Alice receives Bob’s key information and derives the master secret.



Signal: Alice Initiates Session with Bob

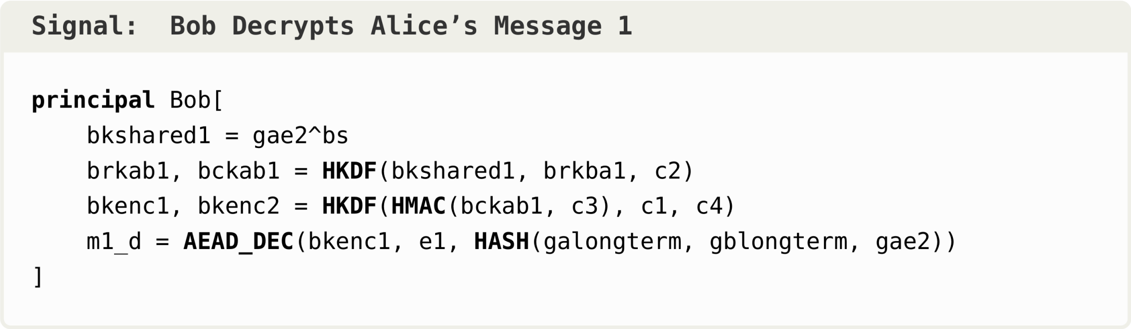
```
Bob -> Alice: [gblongterm], gbssig, gbs, gbo
principal Alice[
    generates ae1
    gae1 = G^ae1
    amaster = HASH(c0, gbs^alongterm, gblongterm^ae1, gbs^ae1, gbo^ae1)
    arkba1, ackba1 = HKDF(amaster, c1, c2)
]
```

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
```

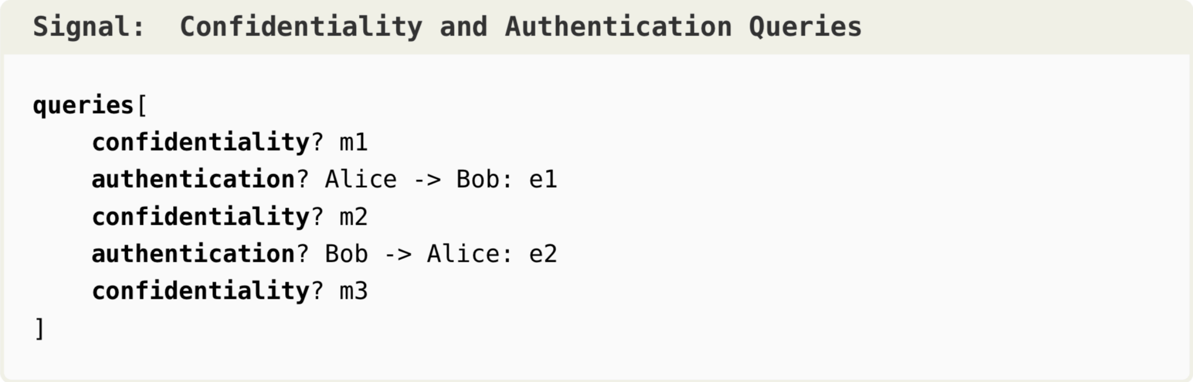


Signal: Bob Decrypts Alice’s Message 1

```
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))
]
```

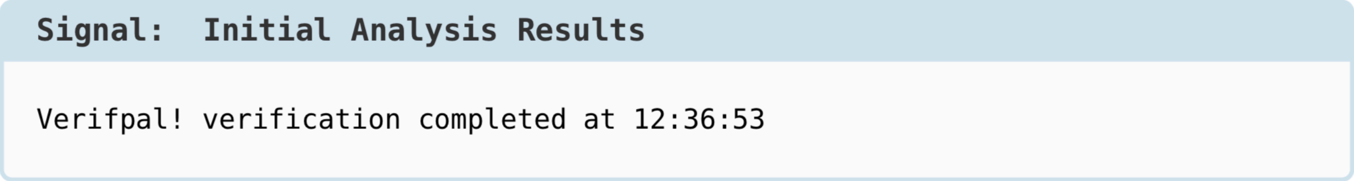
Signal in Verifpal: Queries and Results

- 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
```

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!**

Video on next slide…

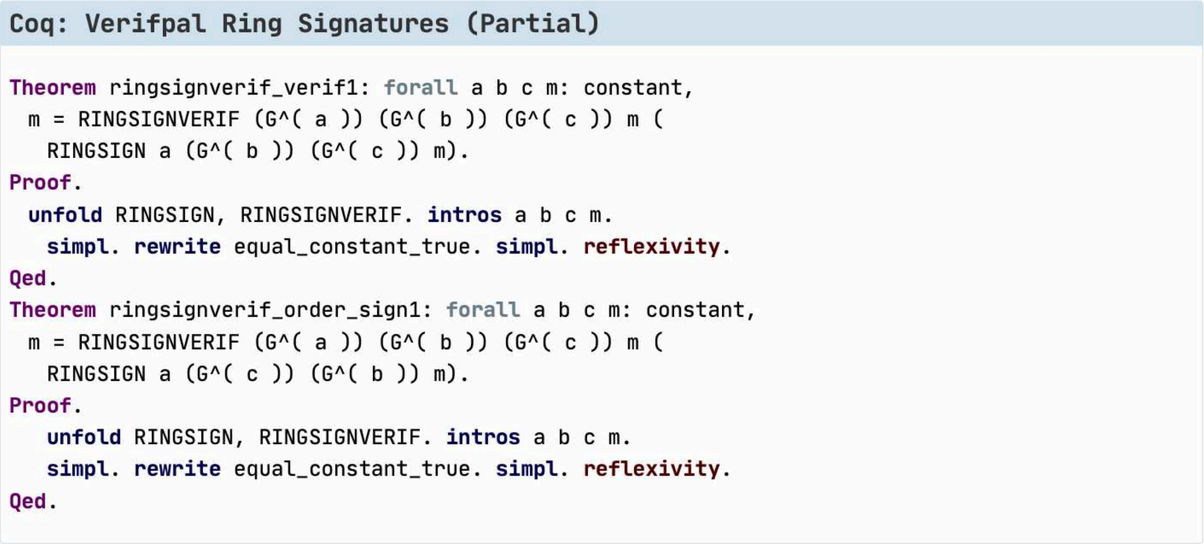
Verifpal for Visual Studio Code

Video Demonstration

Verifpal Translations: Coq and ProVerif

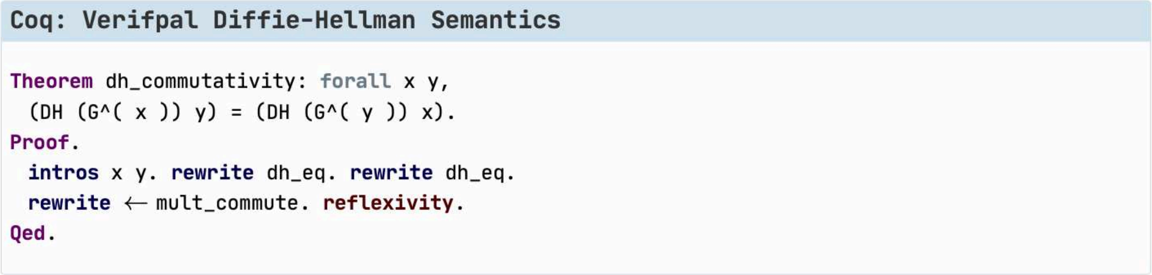
- Verifpal models can be translated to Coq models (complete with formal semantics, lemmas and proofs on primitives),
- ProVerif model templates for further analysis in ProVerif and potentially CryptoVerif.

Formalizing Verifpal in Coq



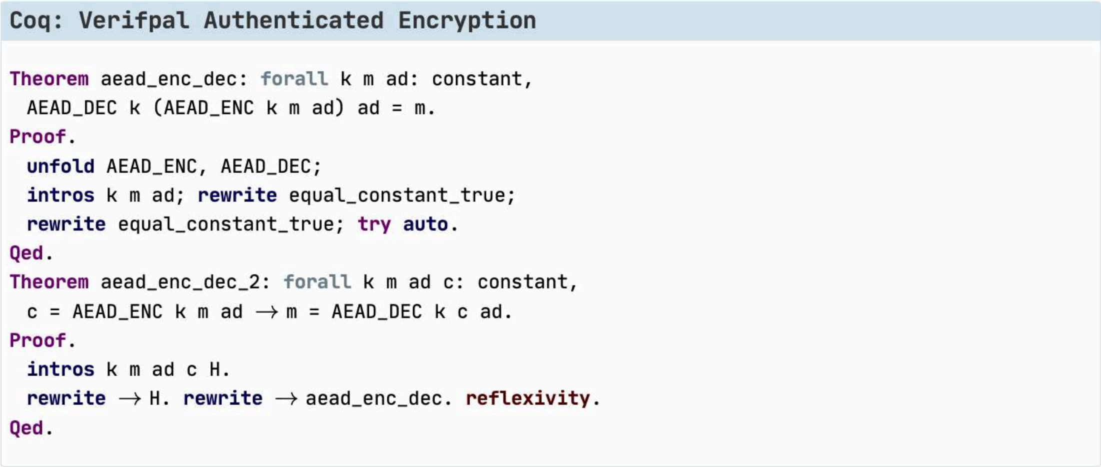
Coq: Verifpal Ring Signatures (Partial)

```
Theorem ringsignverif_verif1: forall a b c m: constant,
 m = RINGSIGNVERIF (G^( a )) (G^( b )) (G^( c )) m (
  RINGSIGN a (G^( b )) (G^( c )) m).
Proof.
 unfold RINGSIGN, RINGSIGNVERIF. intros a b c m.
  simpl. rewrite equal_constant_true. simpl. reflexivity.
Qed.
Theorem ringsignverif_order_sign1: forall a b c m: constant,
 m = RINGSIGNVERIF (G^( a )) (G^( b )) (G^( c )) m (
  RINGSIGN a (G^( c )) (G^( b )) m).
Proof.
  unfold RINGSIGN, RINGSIGNVERIF. intros a b c m.
  simpl. rewrite equal_constant_true. simpl. reflexivity.
Qed.
```



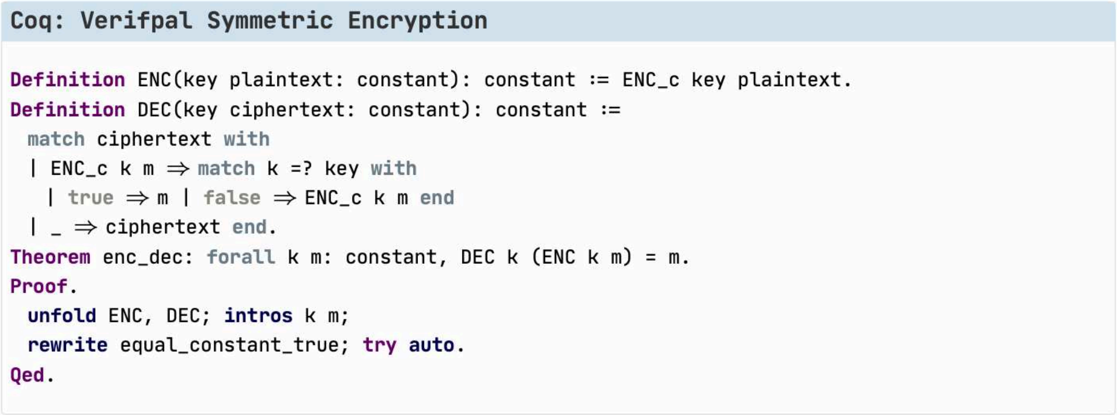
Coq: Verifpal Diffie-Hellman Semantics

```
Theorem dh_commutativity: forall x y,
 (DH (G^( x )) y) = (DH (G^( y )) x).
Proof.
 intros x y. rewrite dh_eq. rewrite dh_eq.
 rewrite ← mult_commute. reflexivity.
Qed.
```



Coq: Verifpal Authenticated Encryption

```
Theorem aead_enc_dec: forall k m ad: constant,
 AEAD_DEC k (AEAD_ENC k m ad) ad = m.
Proof.
 unfold AEAD_ENC, AEAD_DEC;
 intros k m ad; rewrite equal_constant_true;
 rewrite equal_constant_true; try auto.
Qed.
Theorem aead_enc_dec_2: forall k m ad c: constant,
 c = AEAD_ENC k m ad → m = AEAD_DEC k c ad.
Proof.
 intros k m ad c H.
 rewrite → H. rewrite → aead_enc_dec. reflexivity.
Qed.
```

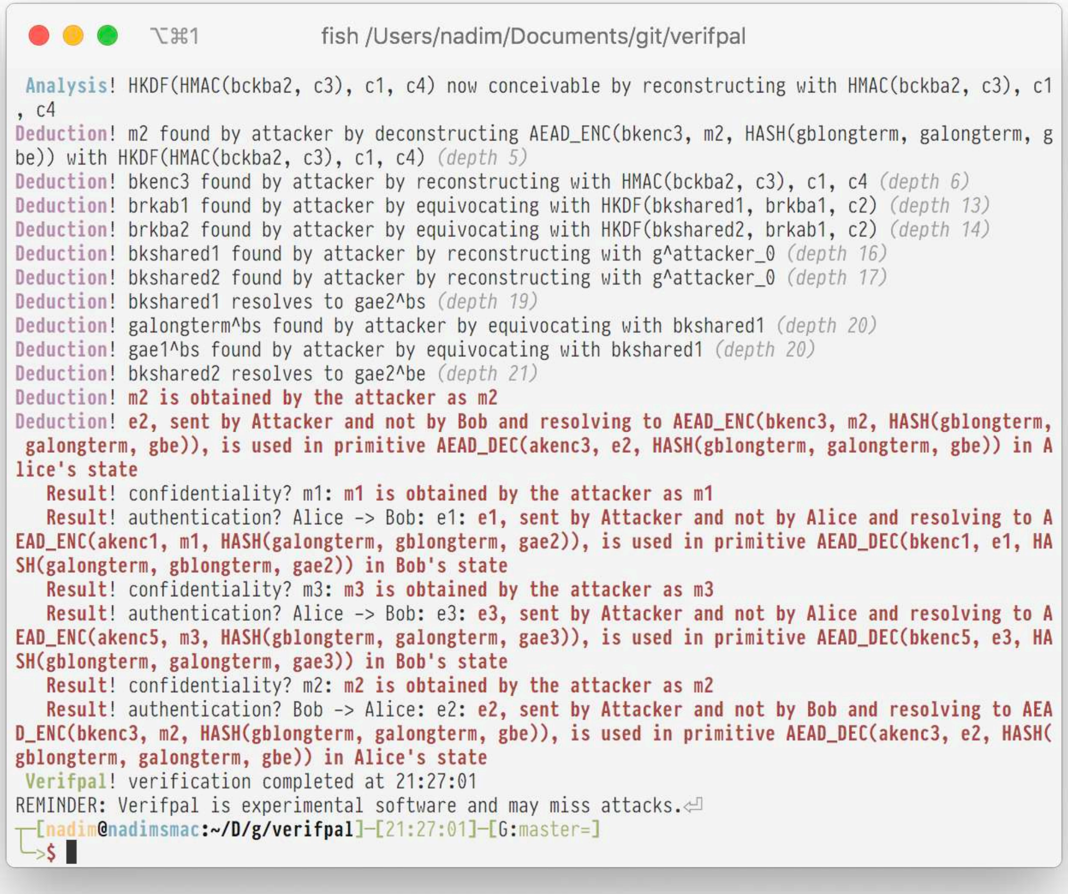


Coq: Verifpal Symmetric Encryption

```
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.
```

Protocols Analyzed with Verifpal

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

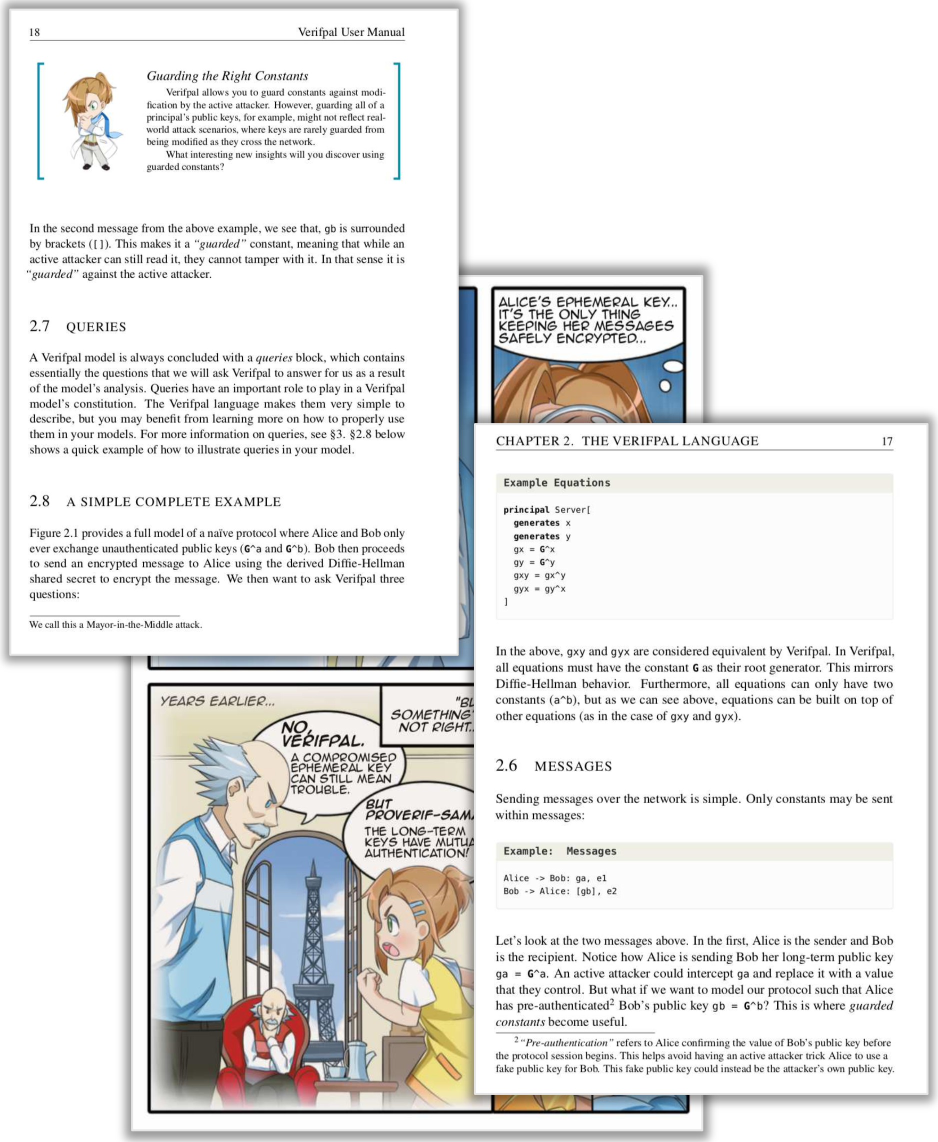


```
 Analysis! HKDF(HMAC(bckba2, c3), c1, c4) now conceivable by reconstructing with HMAC(bckba2, c3), c1
, c4
Deduction! m2 found by attacker by deconstructing AEAD_ENC(bkenc3, m2, HASH(gblongterm, galongterm, g
be)) with HKDF(HMAC(bckba2, c3), c1, c4) (depth 5)
Deduction! bkenc3 found by attacker by reconstructing with HMAC(bckba2, c3), c1, c4 (depth 6)
Deduction! brkab1 found by attacker by equivocating with HKDF(bkshared1, brkba1, c2) (depth 13)
Deduction! brkba2 found by attacker by equivocating with HKDF(bkshared2, brkab1, c2) (depth 14)
Deduction! bkshared1 found by attacker by reconstructing with g^attacker_0 (depth 16)
Deduction! bkshared2 found by attacker by reconstructing with g^attacker_0 (depth 17)
Deduction! bkshared1 resolves to gae2^bs (depth 19)
Deduction! galongterm^bs found by attacker by equivocating with bkshared1 (depth 20)
Deduction! gae1^bs found by attacker by equivocating with bkshared1 (depth 20)
Deduction! bkshared2 resolves to gae2^be (depth 21)
Deduction! m2 is obtained by the attacker as m2
Deduction! e2, sent by Attacker and not by Bob and resolving to AEAD_ENC(bkenc3, m2, HASH(gblongterm,
 galongterm, gbe)), is used in primitive AEAD_DEC(akenc3, e2, HASH(gblongterm, galongterm, gbe)) in A
lice's state
   Result! confidentiality? m1: m1 is obtained by the attacker as m1
   Result! authentication? Alice -> Bob: e1: e1, sent by Attacker and not by Alice and resolving to A
EAD_ENC(akenc1, m1, HASH(galongterm, gblongterm, gae2)), is used in primitive AEAD_DEC(bkenc1, e1, HA
SH(galongterm, gblongterm, gae2)) in Bob's state
   Result! confidentiality? m3: m3 is obtained by the attacker as m3
   Result! authentication? Alice -> Bob: e3: e3, sent by Attacker and not by Alice and resolving to A
EAD_ENC(akenc5, m3, HASH(gblongterm, galongterm, gae3)), is used in primitive AEAD_DEC(bkenc5, e3, HA
SH(gblongterm, galongterm, gae3)) in Bob's state
   Result! confidentiality? m2: m2 is obtained by the attacker as m2
   Result! authentication? Bob -> Alice: e2: e2, sent by Attacker and not by Bob and resolving to AEA
D_ENC(bkenc3, m2, HASH(gblongterm, galongterm, gbe)), is used in primitive AEAD_DEC(akenc3, e2, HASH(
gblongterm, galongterm, gbe)) in Alice's state
 Verifpal! verification completed at 21:27:01
REMINDER: Verifpal is experimental software and may miss attacks.⏎
┌─[nadim@nadimsmac:~/D/g/verifpal]─[21:27:01]─[G:master=]
└─>$ █
```

Who’s Using Verifpal?

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.



18 Verifpal User Manual

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 naïve 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 Mayor-in-the-Middle attack.

CHAPTER 2. THE VERIFPAL LANGUAGE 17

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[2] Bob's public key `gb = G^b`? This is where *guarded constants* become useful.

[2] *"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.

ALICE'S EPHEMERAL KEY... IT'S THE ONLY THING KEEPING HER MESSAGES SAFELY ENCRYPTED...
YEARS EARLIER...
"BL SOMETHING' NOT RIGHT..
NO, VERIFPAL. A COMPROMISED EPHEMERAL KEY CAN STILL MEAN TROUBLE.
BUT PROVERIF-SAM THE LONG-TERM KEYS HAVE MUTUA AUTHENTICATION!

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