Verifpal

Cryptographic protocol analysis for students and engineers

Nadim Kobeissi

Mozilla Berlin, November 5, 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".

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.

Symbolic Verification, Still?

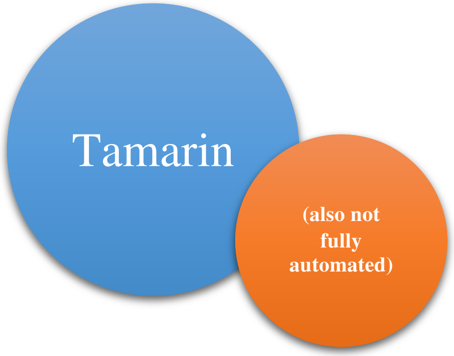
- **F* and computational models do not allow us to naturally express** and model protocols according to **a system based on discrete principals** with internal states.
- Writing a protocol in F* just to check it against security goals on a network: **unreasonable cost/benefit tradeoff.**
- **Research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


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

[…]

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

Verifpal: A New Symbolic Verifier

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

A New Approach to Symbolic Verification

User-focused approach…

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

…without losing strength

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

Verifpal Language

- 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: 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.

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
```

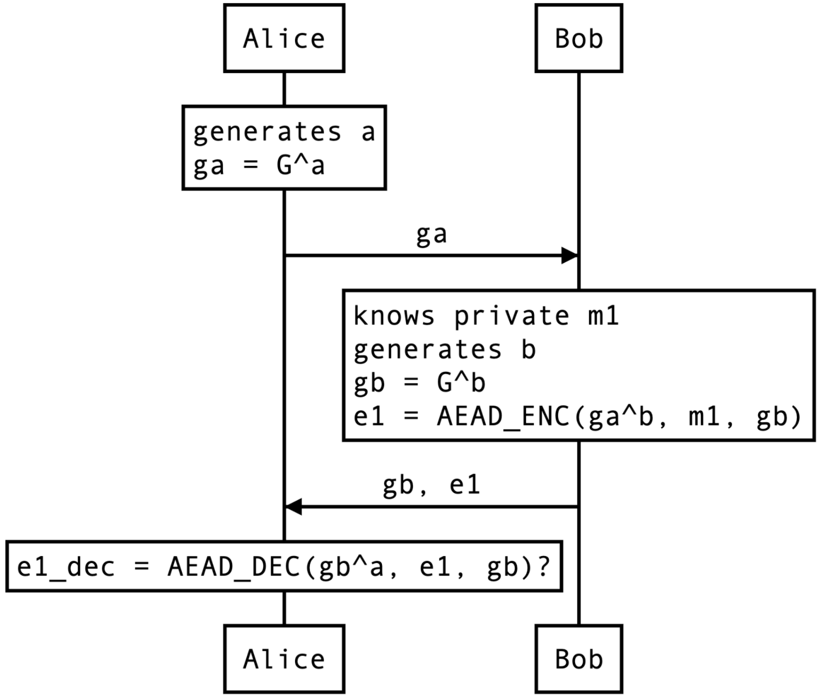
Example: Queries

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

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)?
]
```



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.)

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

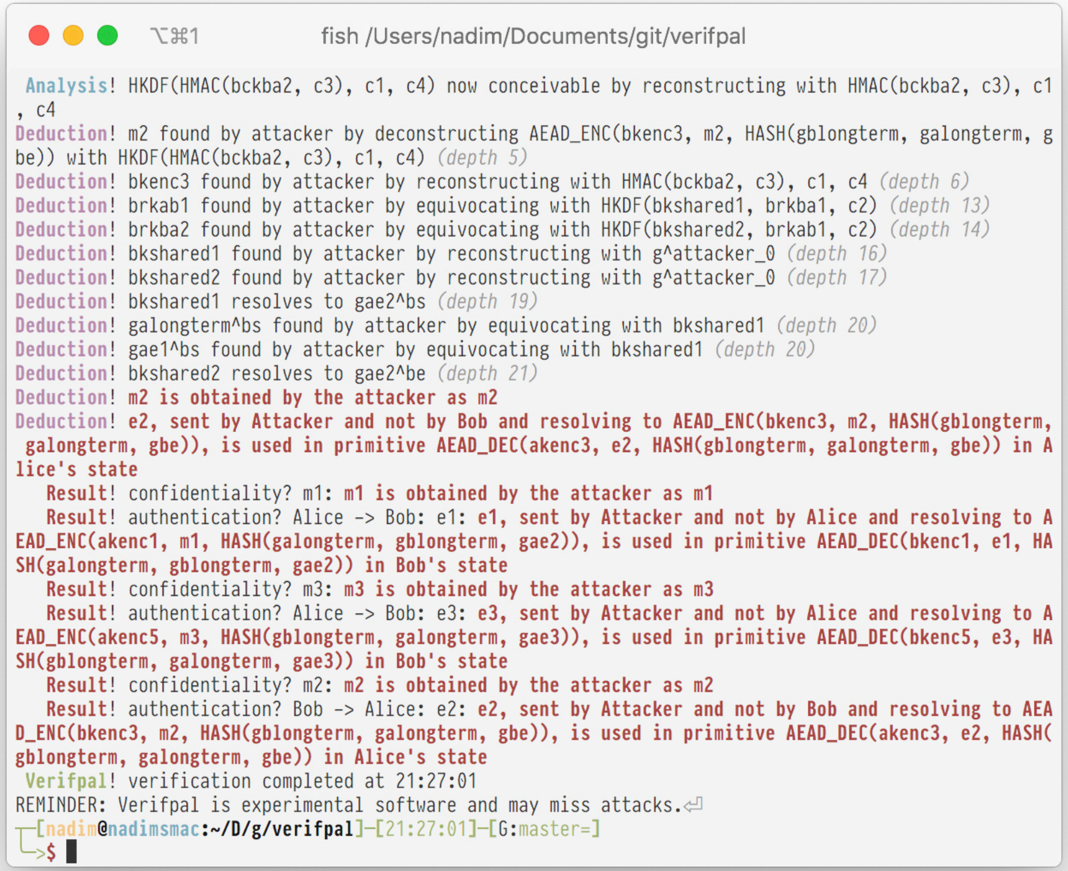
```
queries[
    confidentiality? m1
    authentication? Alice -> Bob: e1
    confidentiality? m2
    authentication? Bob -> Alice: e2
    confidentiality? m3
]
```

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.



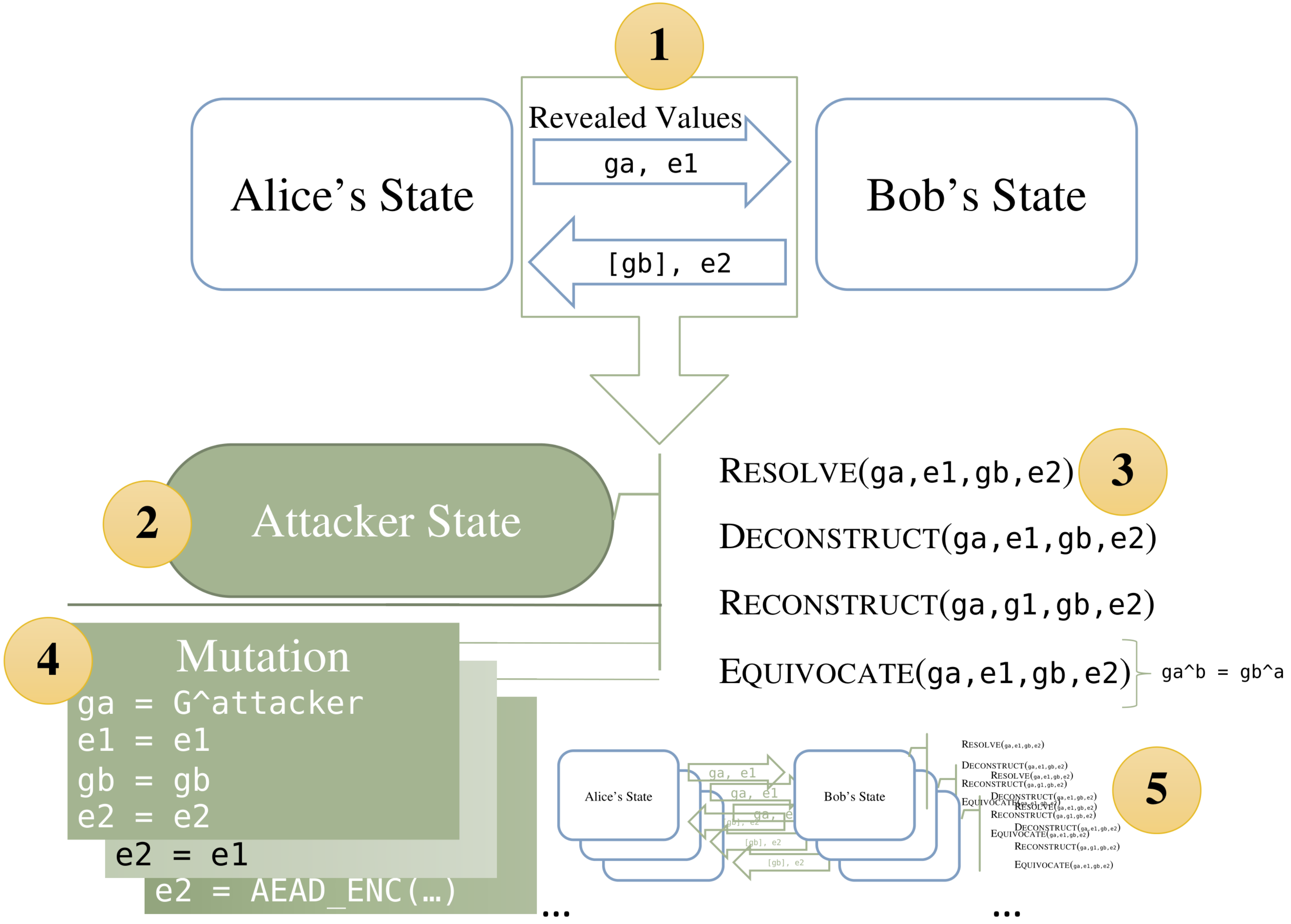
Verifpal Analysis Soundness

- Four main verification functions:
- *Resolve*: Resolve a constant's assignment.
- *Deconstruct*: Check if a value can be deconstructed based on what the attacker knows.
- *Reconstruct*: Check if a value can be reconstructed based on what the attacker knows.
- *Equivocate*: Check if two values are equivalent.

Verifpal Analysis Soundness

- Active attacker performs all possible substitutions across an unbounded number of sessions: so long as new substitutions become possible based on learned values, it keeps going.
- Each execution keeps applying four main verification functions (*Resolve, Deconstruct, Reconstruct, Equivocate*) until no new values appear.
- Constructed malicious values enter table of possible substitutions by the active attacker.
- Certain rules are respected: abort if guarded primitive fails, don't keep values that contain fresh (*generated*) values…

Verifpal Analysis Methodology



Verifpal Analysis Soundness

- Assumption: four main verification functions sufficient to extract all possible values under a particular execution for the attacker.
- Coupled with active attacker substituting/injecting all possible values, we obtain verification with no missed attacks.

Currently informal theorem, no proof exists

No guarantee of functional correctness

Why Release Before the Soundness Proof?

- Testing by users and community.
  - Soundness proof does not equal absence of bugs.
  - Community may suggest changes and fixes (as has already occurred), leading to changes to the language.
- Does this mean I should still learn Verifpal before the soundness proof is published?
  - Yes! Verifpal's language and functionalities won't change: proof will only help ensure lack of missed attacks.

Verifpal: the First Few Weeks

- Verifpal alpha released with discussions on the Verifpal Mailing List.
- Feedback from Bruno led to a redesign of how equations are expressed in the language and other changes.
- Feedback from Loup Vaillant led to stronger testing and a better implementation of authentication queries.
- An anonymous contributor ("Mike") fuzzed Verifpal's parser, leading to a hardening of the parser against unexpected expressions, misleading statements etc.
- Caught a bunch of bugs.

Verifpal: the First Few Weeks

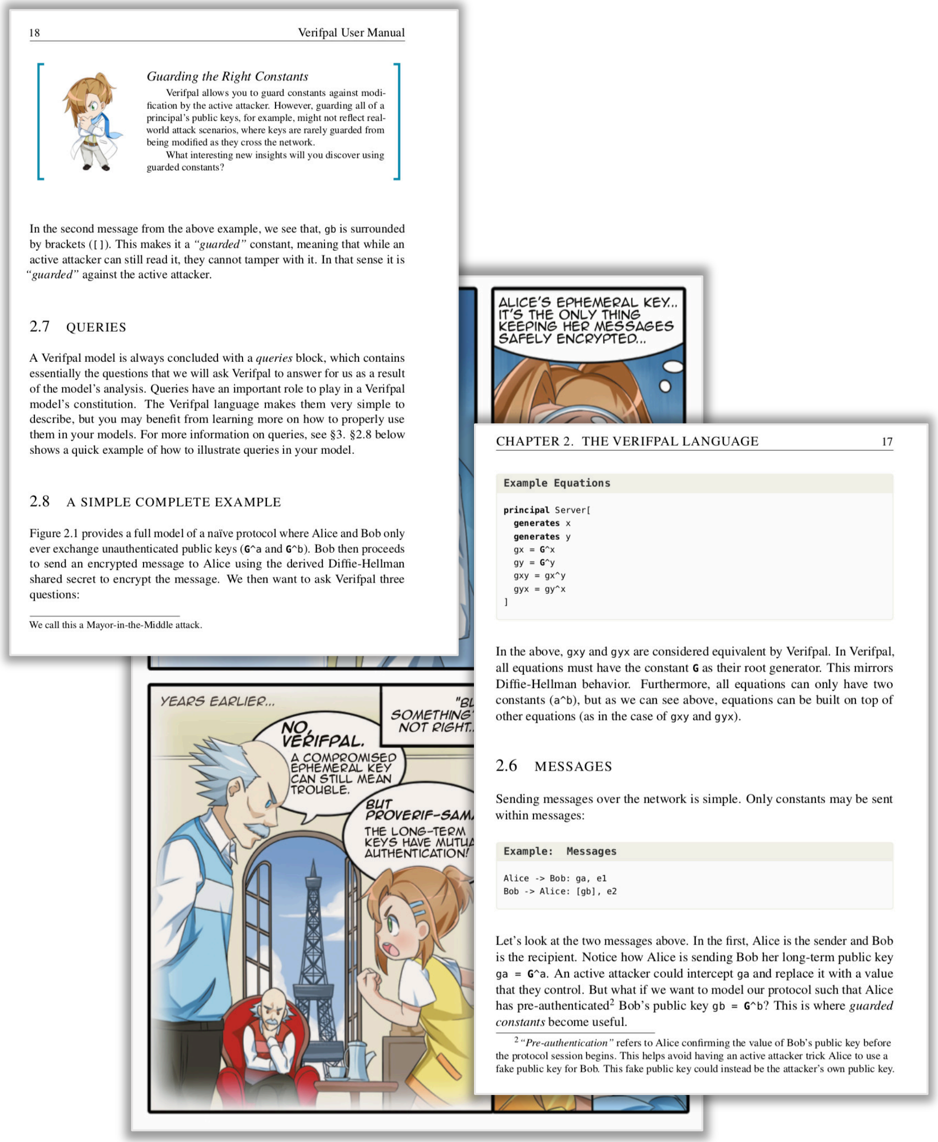
- I remember a time when F* thought $(a + b) \neq (b + a)$…
- …and that was way after the first two weeks of its release!
- So, some perspective, please!
  - Soundness proof will come in early 2020.
  - Verifpal's features and supported queries will grow.
  - Verifpal's development process: start with ease of use, finish with advanced features. F*, CryptoVerif etc. do it the other way around.

Verifpal: the First Few Weeks

- **Third-party applications:**
  - *Monokex*, a Noise-like authenticated key exchange (Loup Vaillant David)
  - *OTRv4*, the next version of the Off-the-Record secure messaging protocol (Georgio Nicolas)
  - *Old vulnerable Tor handshake*, an old vulnerable Tor handshake (Adam Langley)
  - *Symbolic Software audits* (can’t disclose due to NDAs)
  - *…and others?*

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.



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

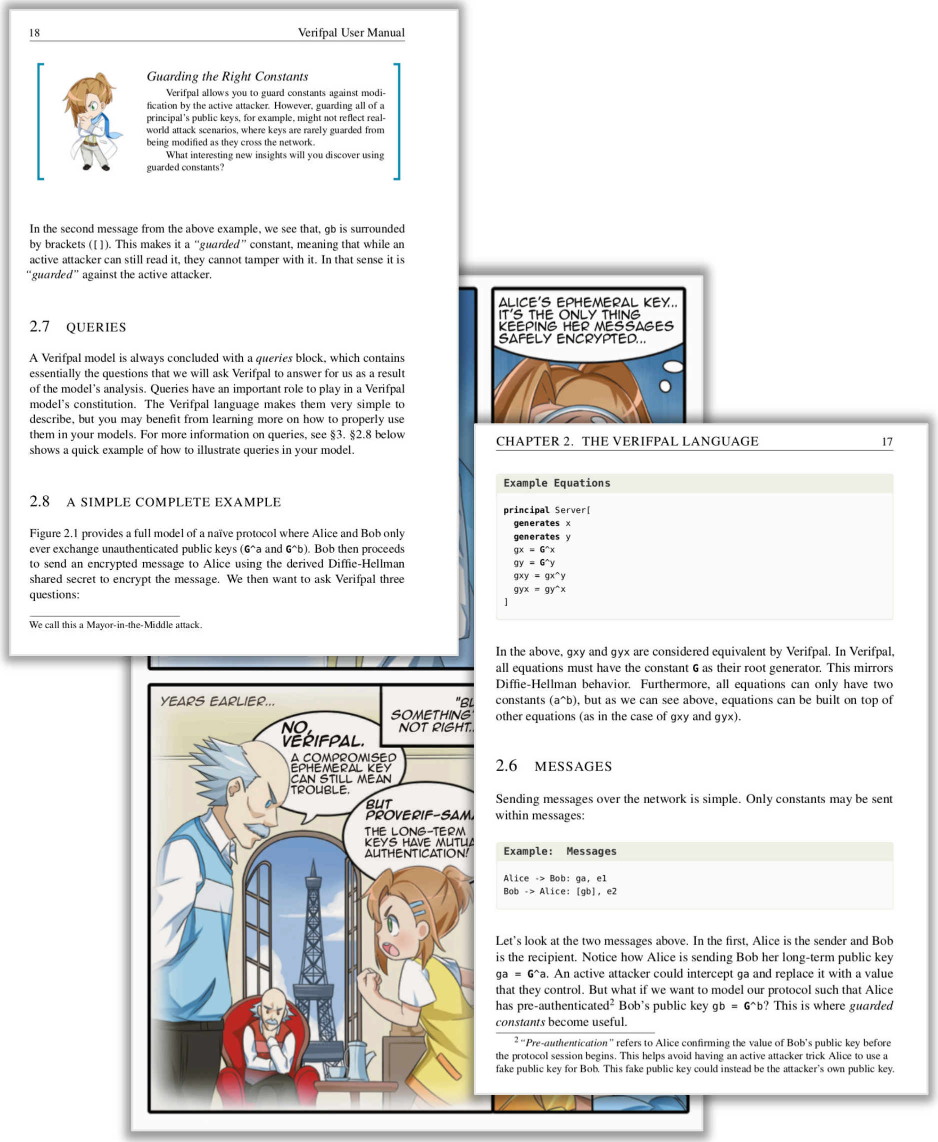
```
Alice -> Bob: ga, e1
Bob -> Alice: [gb], e2
```

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.

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 Utilities and Plugins

- Visual Studio Code: currently syntax highlighting, but much more planned in the future.
- Vim: syntax highlighting.
- *"Verifpal QuickInstall"*: quickly install or update Verifpal on any macOS/Linux platform:

```
bash -c "curl -sL https://verifpal.com/install|bash"
```

Verifpal: Go vs. Ocaml

- Go allowed for faster development and also gives Verifpal faster performance.
- Overall, it was not a good decision: Ocaml's polymorphic variants and especially its pattern matching were sorely missed, and led to inelegant syntax in some parts of Verifpal.
- Conclusion: not as good an idea as I thought but still good. Will encourage contributors?

What Are Verifpal's End Goals?

- Soundness proof.
- High quality educational materials for protocol analysis in undergraduate classes.
- High quality, robust protocol modeling and analysis for engineers, with integration and live prototyping inside Visual Studio Code.

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