

# Verifpal

Cryptographic protocol analysis for students and engineers

Nadim Kobeissi FOSDEM Brussels, February 2020



#### What is Formal Verification?

- Using software tools in order to obtain guarantees on the security of cryptographic components.
- Protocols have unintended behaviors when confronted with an active attacker: formal verification can prove security under certain active attacker scenarios!
- Primitives can act in unexpected ways given certain inputs: formal verification: formal verification can prove functional correctness of implementations!

#### Formal Verification Today

#### Code and Implementations: F\*

- Exports type checks to the Z3 theorem prover.
- Can produce provably functionally correct software implementations of primitives (e.g. Curve25519 in HACL\*).
- Can produce provably functionally correct protocol implementations (Signal\*).

#### Protocols: ProVerif, Tamarin

- Take models of protocols (Signal, TLS) and find contradictions to queries.
- "Can the attacker decrypt Alice's first message to Bob?"
- Are limited to the "symbolic model", CryptoVerif works in the "computational model".

### Symbolic Verification Overview

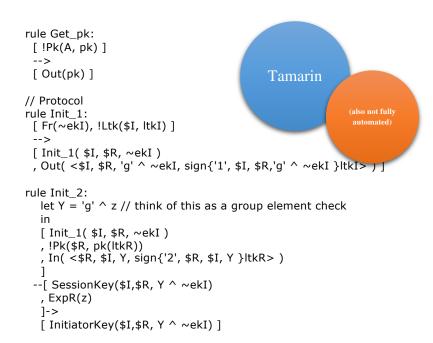
- Main tools: ProVerif, Tamarin.
- User writes a model of a protocol in action:
  - Signal AKE, bunch of messages between Alice and Bob,
  - TLS 1.3 session between a server and a bunch of clients,
  - ACME for Let's Encrypt (with domain name ownership confirmation...)
- User writes queries:
  - "Can someone impersonate the server to the clients?"
  - "Can a client hijack another client's simultaneous connection to the server?"
- ProVerif and Tamarin try to find contradictions.

#### Symbolic Verification is Wonderful

- Many papers published in the past 4 years: symbolic verification proving (and finding attacks) in Signal, TLS 1.3, Noise, Scuttlebutt, Bluetooth, 5G and much more!
- This is a great way to work, allowing practitioners to reason better about their protocols before/as they are implemented.

Why isn't it used more?

#### Tamarin and ProVerif: Examples



letfun writeMessage\_a(me:principal, them:principal, hs:handshakestate, payload:bitstring, sid:sessionid) =

let (ss:symmetricstate, s:keypair, e:keypair, rs:key, re:key, psk:key, initiator:bool) = handshakestateunpack(hs) in let (ne:bitstring, ns:bitstring, ciphertext:bitstring) = (empty, empty, empty) in

let e = generate\_keypair(key\_e(me, them, sid)) in let ne = key2bit(getpublickey(e)) in let ss = mixHash(ss, ne) in let ss = mixKey(ss, getpublickey(e)) in let ss = mixKey(ss, dh(e, rs)) in let s = generate\_keypair(key\_s(me)) in

#### [...]

event(RecvMsg(bob, alice, stagepack\_c(sid\_b), m)) =
 (event(SendMsg(alice, c, stagepack\_c(sid\_a), m))) ||
 ((event(LeakS(phase0, alice))) && (event(LeakPsk(phase0, alice, bob)))) || ((event(LeakS(phase0, bob))) && (event(LeakPsk(phase0, alice, bob))));

#### Verifpal: A New Symbolic Verifier

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



# A New Approach to Symbolic Verification

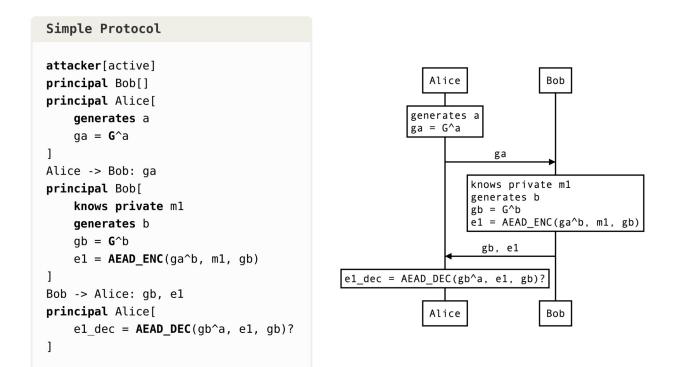
#### User-focused approach...

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

#### ...without losing strength

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

# Verifpal Language: Simple and Intuitive



# 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  $\geq 1$ , 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  $\geq 1$ .

# Verifpal Language: Primitives

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

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

## Verifpal Language: Primitives

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

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

#### Signal in Verifpal: State Initialization

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

```
Signal: Initializing Alice and Bob as Principals
attacker[active]
principal Alice[
    knows public c0, c1, c2, c3, c4
    knows private alongterm
    galongterm = G^alongterm
1
principal Bob[
    knows public c0, c1, c2, c3, c4
    knows private blongterm, bs
    generates bo
    gblongterm = G^blongterm
    qbs = G^{bs}
    qbo = G^{bo}
    gbssig = SIGN(blongterm, gbs)
1
```

# 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

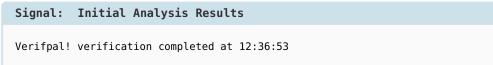
#### principal Bob[

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
authe confi authe	dentiality? m1 ntication? Alice -> Bob: e1 dentiality? m2 ntication? Bob -> Alice: e2 dentiality? m3



### Protocols Analyzed with Verifpal

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

#### Projects Using Verifpal

The following projects have used Verifpal as part of their development process. Please send an email to the Verifpal Mailing List if you would like your project to be added:

- CounterMITM Protocol, by Delta Chat.
- E4, by Teserakt.
- Jess, by Safing.
- Mles Protocol, by Mles.
- Monokex, by Loup Vaillant.
- Salt Channel, by Assa Abloy.
- *SaltyRTC Protocol*, by SaltyRTC.
- Userbase Protocol, by Userbase.

#### Verifpal in the Classroom

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

The concentration of the three the constant of a where the grant of the three	Guarding the Right Constants Workpath above to part costant as gainst modi- ficiants by the active inductor guarding air of a bound and accurate the bound of the second of the bound and accurates. Where he yes are are hyperareful from bound and accurates where he yes are are hyperareful for bound and accurate preventions will you decover using pathef community.	
<text><text><text><text><text><text><text></text></text></text></text></text></text></text>	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	
sensitivity dependence on that we will add verifyed agarge made, then wery single to the model analysis. Specific has an important rise to play is a verifyed and sensitivity or model. The Verifyed laggage made, then wery single to the model analysis. Specific has an important rise to play is a verifyed and sensitivity or model. The Verifyed laggage made is then wery single to the model analysis. Specific has an important rise to play is a verifyed and sensitivity or model. The Verifyed laggage made is then wery single to the rest of the model of a mixer protocol where Alice and Bob ond correct and generative rest with the wart to add Verifyed the cancel on encrypt the message. We then want to add Verifyed the correct and sensitivity or model. The Verifyed the correct and sensitivity or message. We then want to add Verifyed the correct and sensitivity or the message. We then want to add Verifyed the correct and sensitivity or the message. We then want to add Verifyed the correct and sensitivity or the correct and sensity or the correct and sensity or the		ALICE'S EPHEMERAL KEY IT'S THE ONLY THING KEEPING HER MESSAGES SAFELY ENCRYPTED
them in your models. For more information on queries, see 33, 82.8 below soos a quick example of how to illustrate queries in your model.	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	
2.8 A SIMPLE COMPLETE EXAMPLE Figure 21 provides a fail model of anaive protocol where Alice and Bob only ever exchange unambuscal public toys of and 0°D, Bob both quantum contrasting that is cost of and or and the want to a site Verify and the quantum contrasting that is cost of an or anaive protocol state and the verify of an or anaive protocol state and the verify of a model of the verify of a model of the verify of the of the verifies of the verify of the verif	them in your models. For more information on queries, see §3. §2.8 below	CHAPTER 2. THE VERIFPAL LANGUAGE
$ \begin{aligned} & First 2.1 protocols where Alice and Bob order or source of the maximum constraints of the maxim$		Example Equations
FARSE EARLIER:       Sufficient         FUNCTION       FUNCTION	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 Diffic-Hellman shard secret to encrypt the message. We then want to ask Verifial three questions:	penerates x penerates y $0x = 6^{n_x}$ $y = 6^{n_y}$ $0y' = 0^{n'y}$ $0y' = 0^{n'y}$
2.6. MESAGES Boy Control of the second seco	YEARS EARLIER	G other equations (as in the case of axy and ayx).
FibUreF-sAR, FLOW=Troit       The Low-Troit         FibUreFitChT       FibureFitChT         FibureFitChT       FibureFitChT     <	EPHEMERAL KEY	
ATTRENTICATION       Life = here yet it         Life = here yet it       here here yet it         Life = here yet it       here here yet it         Life = here yet it       here here yet it         Life = here yet it       here here yet it         Life = here yet it       here here yet it         Life = here yet it       here here yet it         Life = here yet it       here yet it		M. within messages:
$\mathbb{B} \to Alse: \{p\}, e^2$ Let $\mathbb{B} \to Alse: \{p\}, e^2$ Let $\mathbb{B} \to Alse: \{p\}, e^2$ Let $\mathbb{B} \to Alse: \mathbb{B} \to Als$	KEYS HAVE AUT	<pre>X? Example: messages</pre>
is the recipient. Notice how Alice is sending Boh from the from the three the three		
		<sup>2</sup> "Pre-authentication" refers to Alice confirming the value of Bob's public key befor the protocol session begins. This helps avoid having an active attacker trick Alice to use a session begins.

Verifpal User Manual

#### 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.

Guarding the Right Constants Wright allow you to guad constant against moti- field and scattering the second scattering of a principal point with the second scattering of a principal point of the second scattering of the data data cancer, where the syst are drop unadoffered with a second scattering of the institute with you decover using parted constants?	
in the second message from the above example, we see that, gb is surrounded by brackets (1)). This makes it is " <i>spandod</i> " constant, meaning that while an effect and acter can still and it, they cannot tamper with it. In that sense it is <i>quarded</i> " against the active attacker.	ALICE'S EPHEMEPAL KEY IT'S THE ONLY THING KEEPING HER MESSAGES
2.7 QUERIES Werfight andel is always concluded with a querier block, which contains somithally the queetions that we will lack Verifyel to same refore as a result of the model's containties. The Verifyel language makes them very simple to locaribe, but you may benefit from language makes them very simple to hear the bar you may benefit from language makes the very stoped you hem in your models. For more information or queries, see 33. 82.8 below how a quick campite of how to flasting campice in your model.	CHAPTER 2. THE VERIFPAL LANGUAGE 17
2.8 A SIMPLE COMPLETE EXAMPLE Tigure 2.1 provides a full model of a naive protocol where Alice and Bob only ver exchange mandhenicated public keys (#> and (>)). Bob then proceeds or and an excrypted message to Alice naing the derived Diffe-Helman protocols. We have a supervised on the start of the start of the start protocols. We have a supervised message with the start of the start we full that Maynem the Madit muck.	Example Equations principal server[ generates × generates y gr = 6 ~ gr = 6 ~ gr = 6 ~ gr = 6 ~ gr = 9 ~ gr = 10 ~ gr
YEARS EARLIER	In the above, gay and gays are considered equivalent by Verifpal. In Verifpal, all equations must have the constant $G$ as their root generator. This mirrors root Diffiel-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 gay and gays).
A COMPECTIVES CAN STUEL AND TO THE COMPECTIVE CAN STUEL BUT PROVERIE-SAM THE LONG-TERM	2.6 MESSAGES Sending messages over the network is simple. Only constants may be sent within messages:
KEYS HAVE AUTIL	Example: Messages Alice -> Bob: ga, el Bob -> Alice: (gb), e2
	Let's look at the two messages above. In the first, Alice is the sender and Bob is the receipient. Notice how Alice is sending Bob her long-term public key go = 6° a. An active attacker could intercept a and replace it with a value that they control. But whit if we want to model our produced such that Alice has pre-ambenizands <sup>2</sup> Bob's public key go = 6°0 <sup>2</sup> . This is where guardef constant become useful.

Verifpal User Manual

## Verifpal Extensions

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



#### Try Verifpal Today

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

Check out Verifpal today:

verifpal.com

Support Verifpal development:

verifpal.com/donate

Verifpal: Cryptographic protocol analysis for students and engineers

