Verifpal

A friendly introduction to formal methods for real-world cryptographic protocols

International Association for Cryptologic Research

NGI ZERO

nlnet FOUNDATION

Nadim Kobeissi, Georgio Nicolas

Eurocrypt 2022 — May 29th, 2022

Workshop Expectations

This workshop is for beginners

- You will benefit if:
- You are new to formal modeling of cryptographic protocols
- You are new to the “tool-assisted”, “automated analysis” of cryptographic protocols
- You are new to cryptographic protocols, period

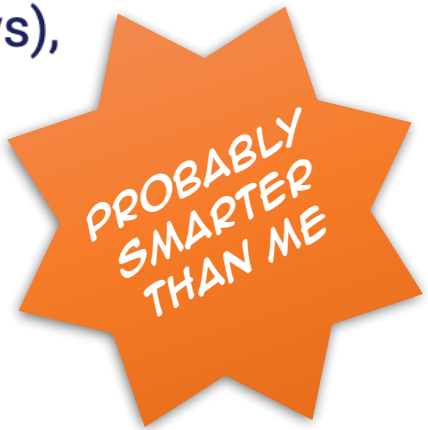
This workshop will greatly bore non-beginners

- You will fall asleep here if:
- You’re well-versed in ProVerif, CryptoVerif, Tamarin…
- You have strong familiarity with modeling and breaking security protocols
- If the above is you, consider attending another Eurocrypt 2022 workshop today, for your own sake.

Seriously, there are other events

1st Annual FHE.org Conference on Fully Homomorphic Encryption
Workshop website
Room: Cosmos 3C/D

Organizers: Pascal Paillier (Zama), Jeremy Bradley (Zama), Ilaria Chillotti (Zama), Qian Lou (Samsung Electronics America), Cristian L[illegible] (Omnia), Florent Michel (Optalysys), [illegible]émy Zaccherini (Zama)



The 3rd International Workshop on Code-Based Cryptography (Day 1)
Workshop website
Room: Cosmos 3B

Organizers: Jean-Christophe [illegible]neuville (ENAC)

Workshop Overview

Intro to Protocols & Verifpal

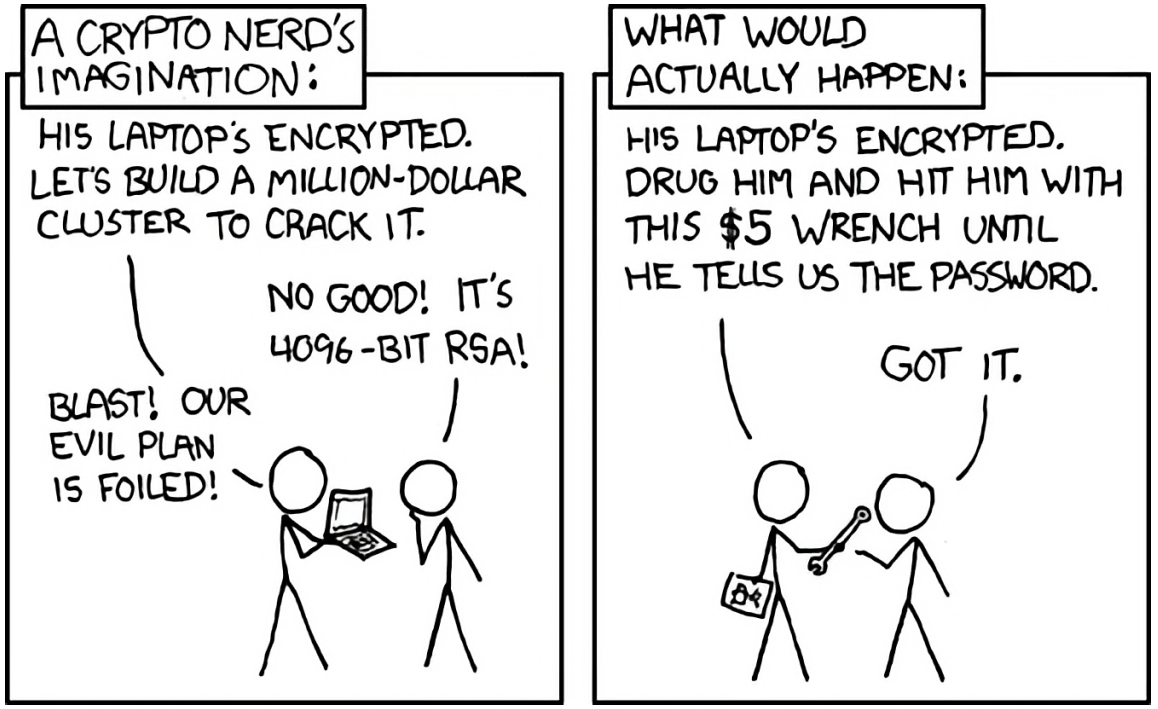
- Introduction and Software Setup
- Learning Verifpal with Examples
- *Coffee Break*
- Modeling Signal in Verifpal
- *Lunch Break*

Automated Verification Safari

- A Look at ProVerif
- *Coffee Break*
- A Look at CryptoVerif and a very brief discussion of F*
- Q&A, Discussion

Data Security

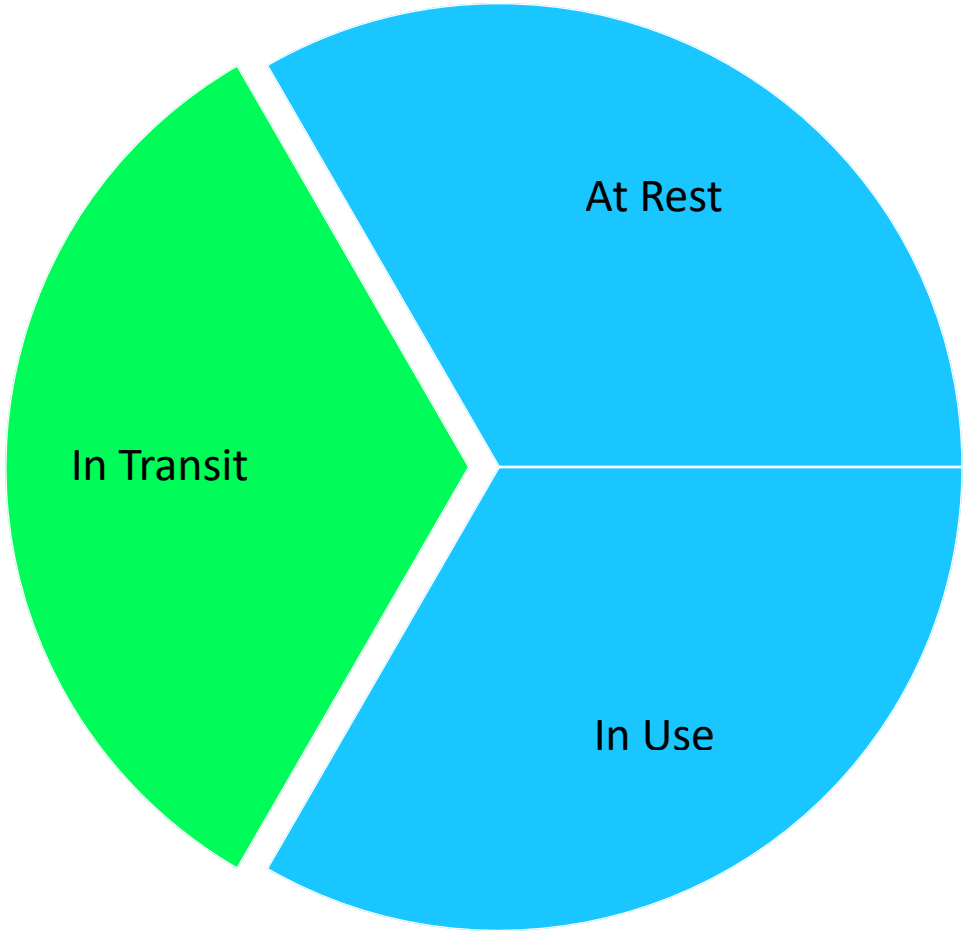
- Deploying software in the real world requires solutions that alleviate actual security risks
- It is equally important to:
    1. Define our problem
    2. Attempt a solution
    3. Assess whether the solution actually reduces the risk to an acceptable amount
- While this may look nice and simple from a distance, most security breaches are a result of failing at steps 1 and 3



Our Scope

- Data can be:
  - At Rest (stored on a disk)
  - In Use (input for a computation)
  - In Transit (transferred over a channel)

- We will be focusing on Data In Transit

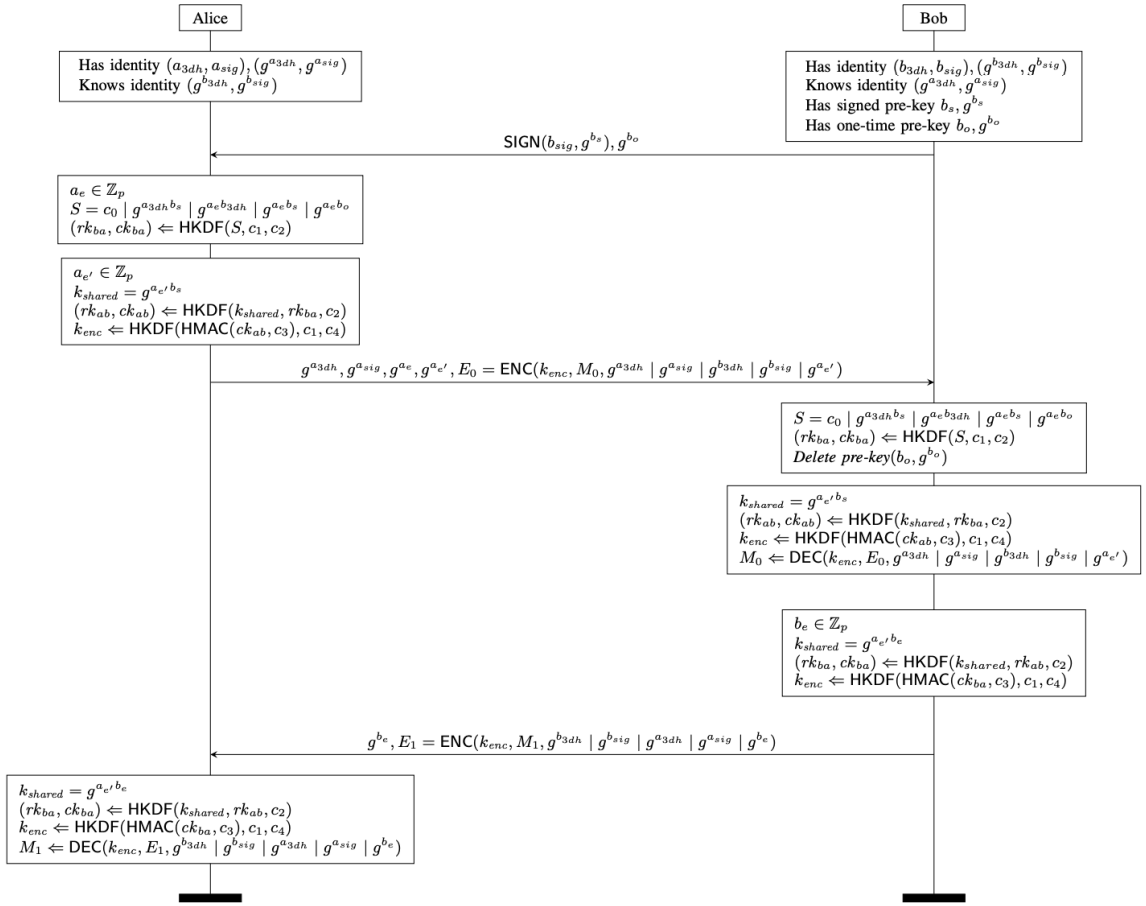


What are cryptographic protocols?

- More specifically, “secure channel protocols”.
- We use them to communicate and do things!
  - When you send a message over Signal/ WhatsApp…
  - When you open a website over HTTPS…
  - When you pay for a dinner using your debit card…

Secure Channel Protocol Design Ingredients

- Principals
- Set of communication channels between principals
- Initial state for each party
- Protocol:
  - Update state,
  - Perform computations,
  - Send/receive messages over communication channels…



What Makes Up a Protocol?

- Symmetric primitives:
  - AES for encryption,
  - SHA-2 for hashing…
- Asymmetric primitives:
  - RSA for asymmetric encryption,
  - Diffie-Hellman (ECDH, etc.) for key agreement,
  - DSA, ECDSA, etc. for public key signatures…
- Principals:
  - One or more parties involved in the execution of an instance of a protocol
- Messages:
  - Sent across a network or out-of-band
  - Formalizations: *"Dolev-Yao model"*

Reasoning About Cryptographic Protocols

- What is our goal?
  - Authentication (between parties), confidentiality, non-repudiation…
- How will we achieve the goal(s)?
  - Using cryptographic protocols (which in turn employ primitives like encryption, signing, hashing…)
- Who are we protecting ourselves against?
  - A disgruntled employee, the government, an ex-partner, a dead person, any attacker in the middle between two nodes on the internet, all of them at once…
  - What can the attacker do (use your imagination)

Security Properties Provided by Protocols

- **Secrecy**
  - If A sends some secret message M to B, then nobody except A and B can obtain M.
- **Indistinguishability**
  - If A randomly chooses between two messages M_0, M_1 (of the same size) and sends one of them to B, the attacker cannot distinguish (within the constraints of the cryptographic model) which message was sent.
- **Forward Secrecy**
  - If A sends a secret message M to B and if A and B's long-term secrets are subsequently compromised, the message M remains secret.
- **Future Secrecy**
  - Suppose A sends M in a session state T, then receives N, then sends M_0 . If the session state T is subsequently compromised, the message M_0 remains secret.

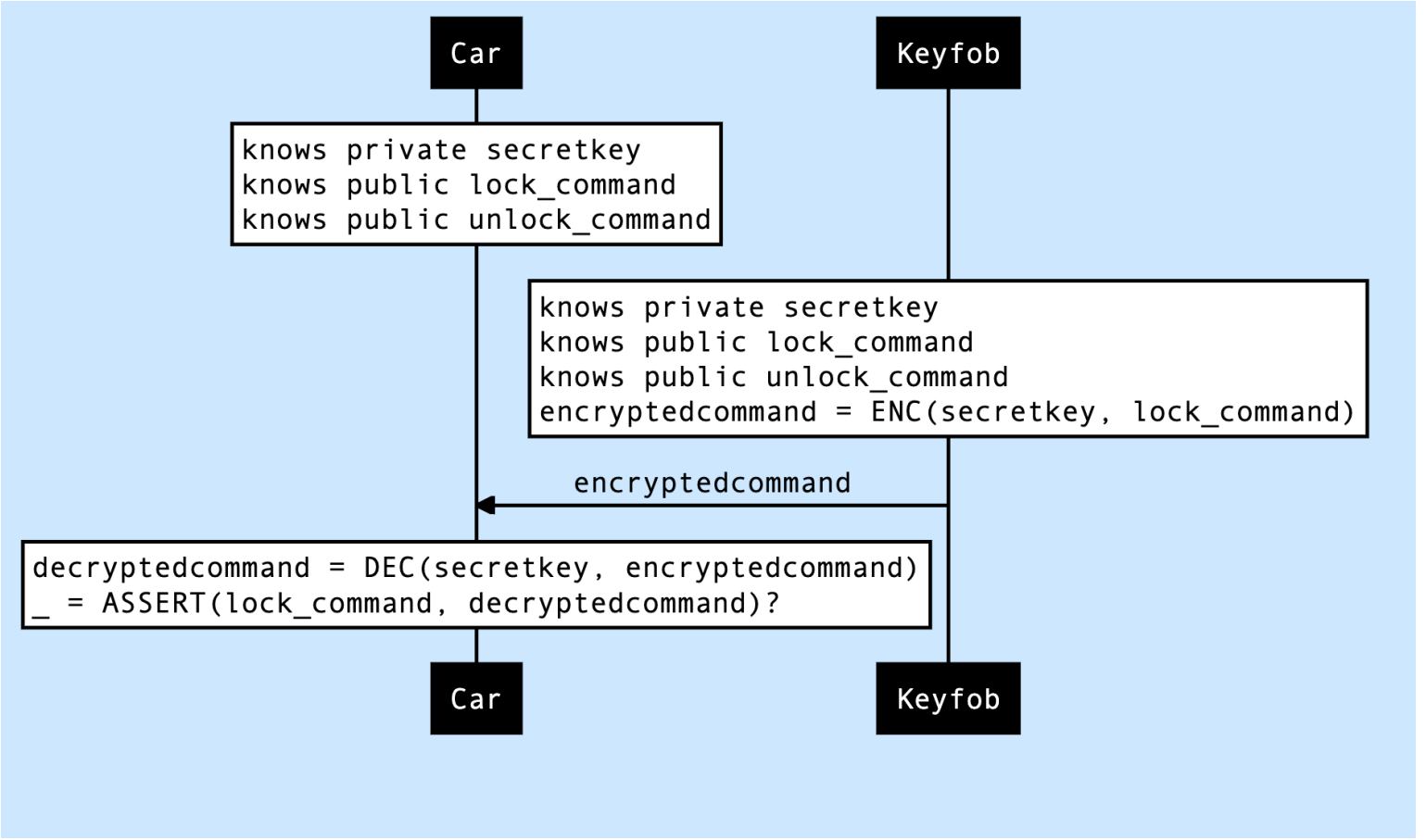
Alleged Security vs Provable Security

- Protocol design on pen and paper can be easy.
- Proving that a protocol can guarantee security properties given a specific use case is much harder.

- *Are we sure that a protocol does what it claims on the tin?*

What can go wrong in this scenario?

- A remote key fob and a car paired by being programmed with the same random static secret.
- The car decrypts and executes commands transmitted in ciphertext by the key fob.



Unoriginal-Rice-Patty (CVE-2019-20626)

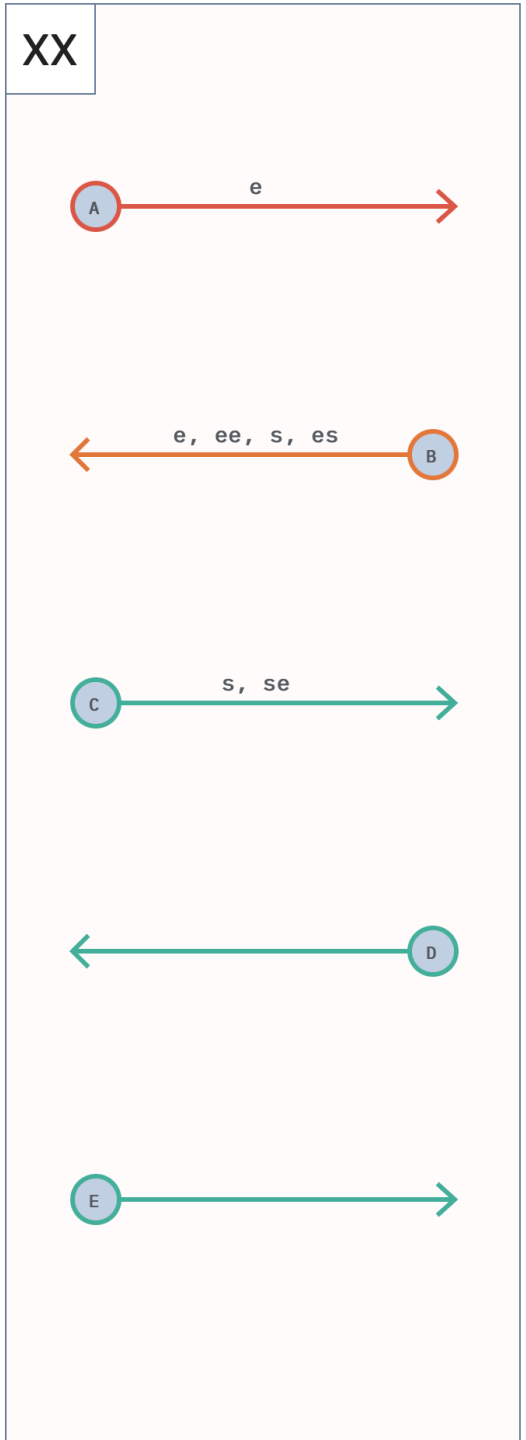
- A hacker can gain complete and unlimited access to locking, unlocking, controlling the windows, opening the trunk, and starting the engine of the target vehicle.
- The only way to prevent the attack is to either never use the remote fob or, reset the programmed key after being compromised at the dealership (which would be difficult to realise).
- Vehicles as new as a 2020 Honda Civic are vulnerable.
- A rolling-code based protocol is more secure.

Modelling our Assumptions Correctly

- The results of a formal methods tool are as strong as the assumptions we provide it with.
- If we model the attacker to have less capabilities than expected in real life, then we could miss certain classes of attacks.
- For the previous example, the attacker never got their hands on the secret key, yet they were still able to unlock the car. An attacker able to replay messages was not considered in the threat model.
- The security of the cryptographic primitives employed play an equally important role: base64 encryption can be broken without a key because it doesn't use one, RSA-2048 can be broken with a quantum computer…

Noise XX Protocol

- Protocol between 2 parties
- The communication channel satisfies different security properties at before, after, and during each stage of the protocol's execution



Handshake Pattern Analysis

Message A SHOW DETAILED ANALYSIS

Message A, sent by the initiator, does not benefit from *sender authentication* and does not provide *message integrity*. It could have been sent by any party, including an active attacker. Message contents do not benefit from *message secrecy* even against a purely passive attacker and any *forward secrecy* is out of the question. `0,0`

Message B SHOW DETAILED ANALYSIS

Message B, sent by the responder, benefits from *sender authentication* and is *resistant to Key Compromise Impersonation*. Assuming the corresponding private keys are secure, this authentication cannot be forged. However, if the responder carries out a separate session with a separate, compromised initiator, this other session can be used to forge the authentication of this message with this session's initiator. Message contents benefit from some *message secrecy* and some *forward secrecy*, but not sufficiently to resist any active attacker. `2,1`

Message C SHOW DETAILED ANALYSIS

Message C, sent by the initiator, benefits from *sender and receiver authentication* and is *resistant to Key Compromise Impersonation*. Assuming the corresponding private keys are secure, this authentication cannot be forged. Message contents benefit from *message secrecy* and *strong forward secrecy*: if the ephemeral private keys are secure and the responder is not being actively impersonated by an active attacker, message contents cannot be decrypted. `4,5`

Message D SHOW DETAILED ANALYSIS

Message D, sent by the responder, benefits from *sender and receiver authentication* and is *resistant to Key Compromise Impersonation*. Assuming the corresponding private keys are secure, this authentication cannot be forged. Message contents benefit from *message secrecy* and *strong forward secrecy*: if the ephemeral private keys are secure and the initiator is not being actively impersonated by an active attacker, message contents cannot be decrypted. `4,5`

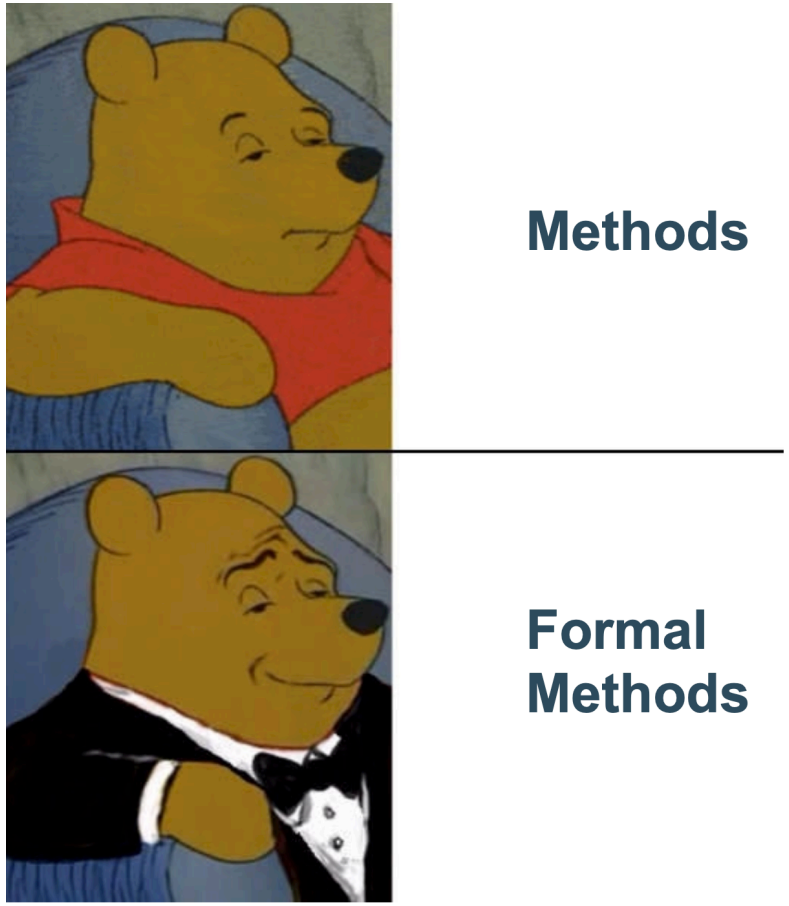
Message E SHOW DETAILED ANALYSIS

Message E, sent by the initiator, benefits from *sender and receiver authentication* and is *resistant to Key Compromise Impersonation*. Assuming the corresponding private keys are secure, this authentication cannot be forged. Message contents benefit from *message secrecy* and *strong forward secrecy*: if the ephemeral private keys are secure and the responder is not being actively impersonated by an active attacker, message contents cannot be decrypted. `4,5`

What are formal methods?

- Allows us to:
  - Define our systems using a “Mathematical Framework”
  - Reason about our definitions in said framework using the provided rules to define certain properties and check if our definitions comply with our targeted properties
  - Model at different layers of abstractions
  - Trace back our decisions

- Can be employed at different stages of development
- Certain frameworks give us special superpowers (HoTT: programs are proofs)
- Can leverage the power of computers: “Theorem Provers”, “SAT-Solvers”, “Static Analysis”, “Verified Compilers”…



Why are formal methods important?

- Since we can verify properties in general using formal methods, why not verify security properties of our crypto constructions!
- We are required to provide some definitions before obtaining results:
  - Primitive choice (Perfect Hash Function vs MD-2 vs SHA-3, Ideal MPC vs BGW, Perfect Encryption vs 2DES vs AES-256…)
  - Adversary resources and capabilities (“Active”, Encryption + Decryption Oracle, Honest but Curious, the NSA…)
  - Properties (“Security”, Confidentiality, Binding, Forward Secrecy, IND-CCA, Correctness…)
- The more intricate our models are, the more confidence we gain. However, we can still obtain results using abstract models.

Protocol Security by Design

- Protocol designers can employ formal methods at the design stage to verify that their constructions are secure by design.
- While this process requires more time to be invested modeling early on, it yields more robust and trustworthy protocols at deployment time.
- Formal Methods were employed during the design process of TLS 1.3, which is expected to have a much longer lifespan than previous versions of TLS.

Limitations and discussion of automated “proofs”

- Even if this gives you a “proof”, can you trust it?
- Models too big to be accurately assessed…
- Models too complex to be checkable by others humans…
- Being verified by software which could be buggy, etc.
- Is the term “proof” too strong?

Bugs in Protocol Implementations

- The people who design a protocol are not always the ones who implement/deploy it in the real world.
- Subtle bugs in protocol implementations can compromise the security guarantees provided by a protocol.
- Can we ensure that our implementations reflect our protocol design considerations?

Real World Disasters:

- Psychic Signatures (Java)
- Solana/Whatever current crypto exchange hack (millions of dollars)
- Libp2p not validating signatures (TS, recent)
- Nonce reuse disasters
- TLS with same server key for all clients as an optimisation

Complexity of Cryptographic Proofs

- Security proofs about primitives or protocols can be hard to understand in some cases, let alone reproduce and verify.
- Can we compose our proofs in a modular and verifiable way such that the readers would be able to replicate the verification procedure on their own to validate the claims of the authors?

Breaking Rainbow Takes a Weekend on a Laptop

Ward Beullens

IBM Research, Zurich, Switzerland
wbe@zurich.ibm.com

Abstract. This work introduces new key recovery attacks against the Rainbow signature scheme, which is one of the three finalist signature schemes still in the NIST Post-Quantum Cryptography standardization project. The new attacks outperform previously known attacks for all the parameter sets submitted to NIST and make a key-recovery practical for the SL 1 parameters. Concretely, given a Rainbow public key for the SL 1 parameters of the second-round submission, our attack returns the corresponding secret key after on average 53 hours (one weekend) of computation time on a standard laptop.

Formal Verification Today

Code and Implementations: F*

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

Protocols: ProVerif, Tamarin

- Take models of protocols (Signal, TLS) and find contradictions to queries.
- 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 proofs, 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	●	◐	●	●	◐	◐

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 Verificati[illegible]?

- Research in symbolic[illegible]vel results:
  - *Prime, Order Pleas[illegible] Curve Attacks on Protocols using Di[illegible]*ckson
  - *Seems Legit: Auto[illegible]ls that Use Signatures* – Dennis Jackson, [illegible]Sasse
- Many papers pub[illegible]fication proving (and finding atta[illegible], Bluetooth, 5G and much more!
- This is a great [illegible]better about their protocols before/as they are implemented.

SO WHY ISN'T IT USED MORE?!

Tamarin and ProVerif: Examples

Tamarin

```
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) ]
```

ProVerif

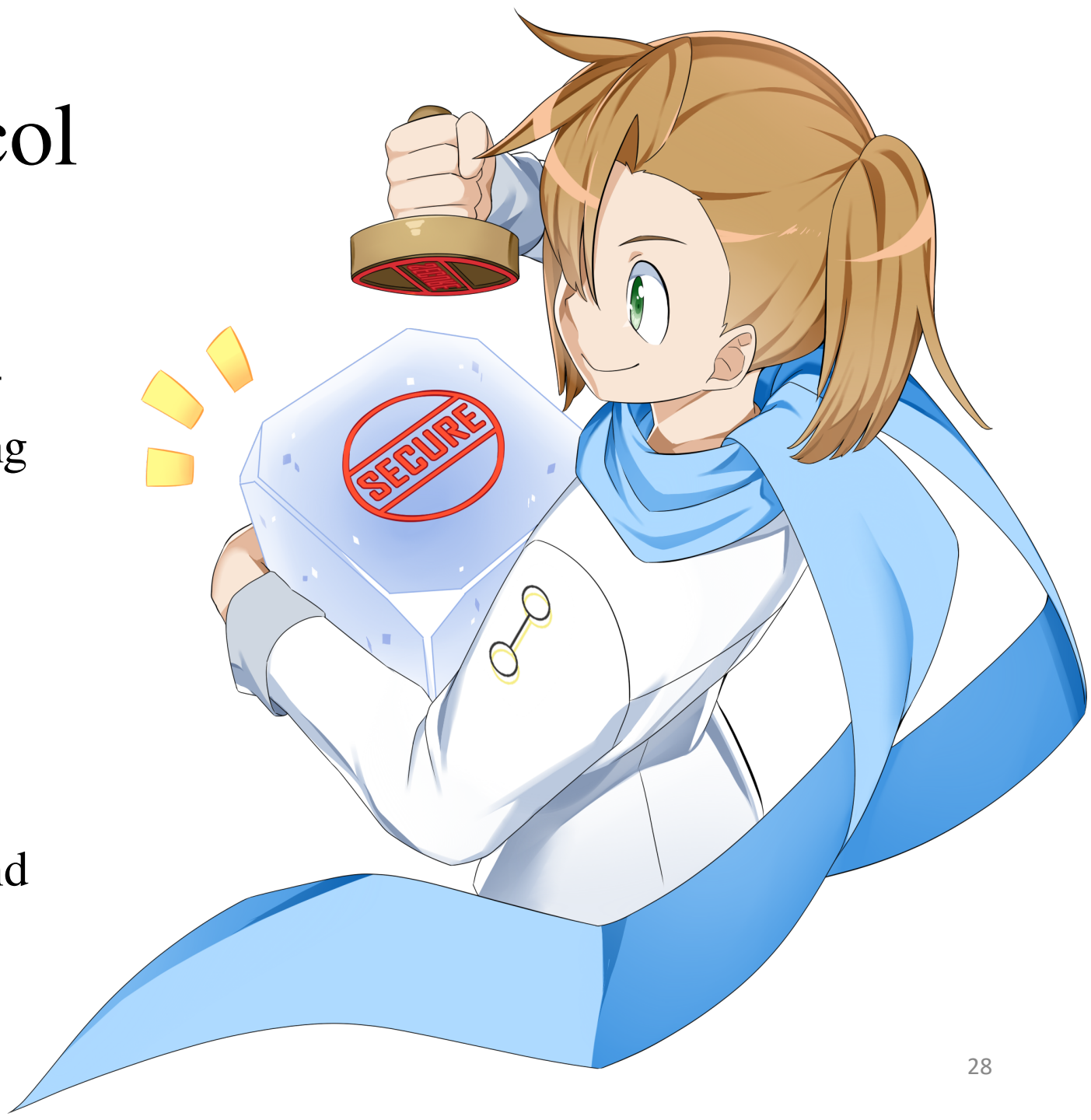
```
letfun writeMessage_a(me:principal, them:principal,
hs:handshakestate, payload:bitstring, sid:sessionid) =
  let (ss:symmetricstate, s:keypair, e:keypair, rs:key,
re:key, psk:key, initiator:bool) = handshakestateunpack(hs) in
  let (ne:bitstring, ns:bitstring, ciphertext:bitstring) =
(empty, empty, 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: 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.

Limitations and Context

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

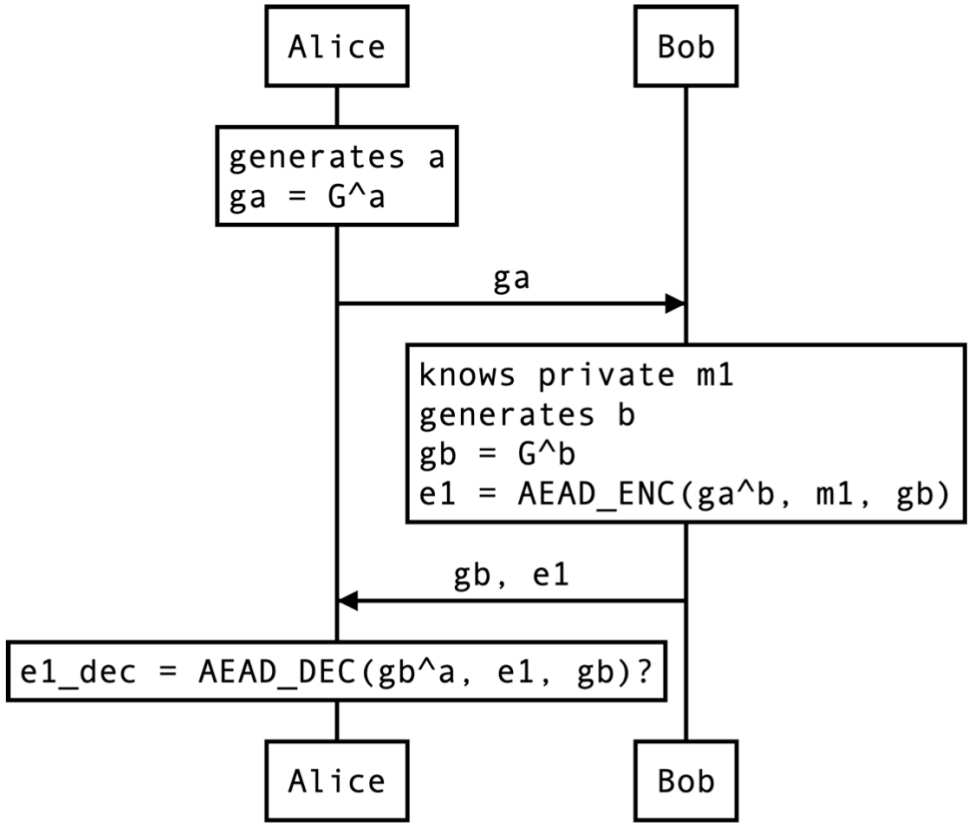
Working towards obtaining higher confidence through building relationship to Coq models of verification method, more scrutiny, more protocols analyzed…

Usefulness is more towards engineers and students.

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.
- Feature, not bug: 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.
- `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`.
- `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.

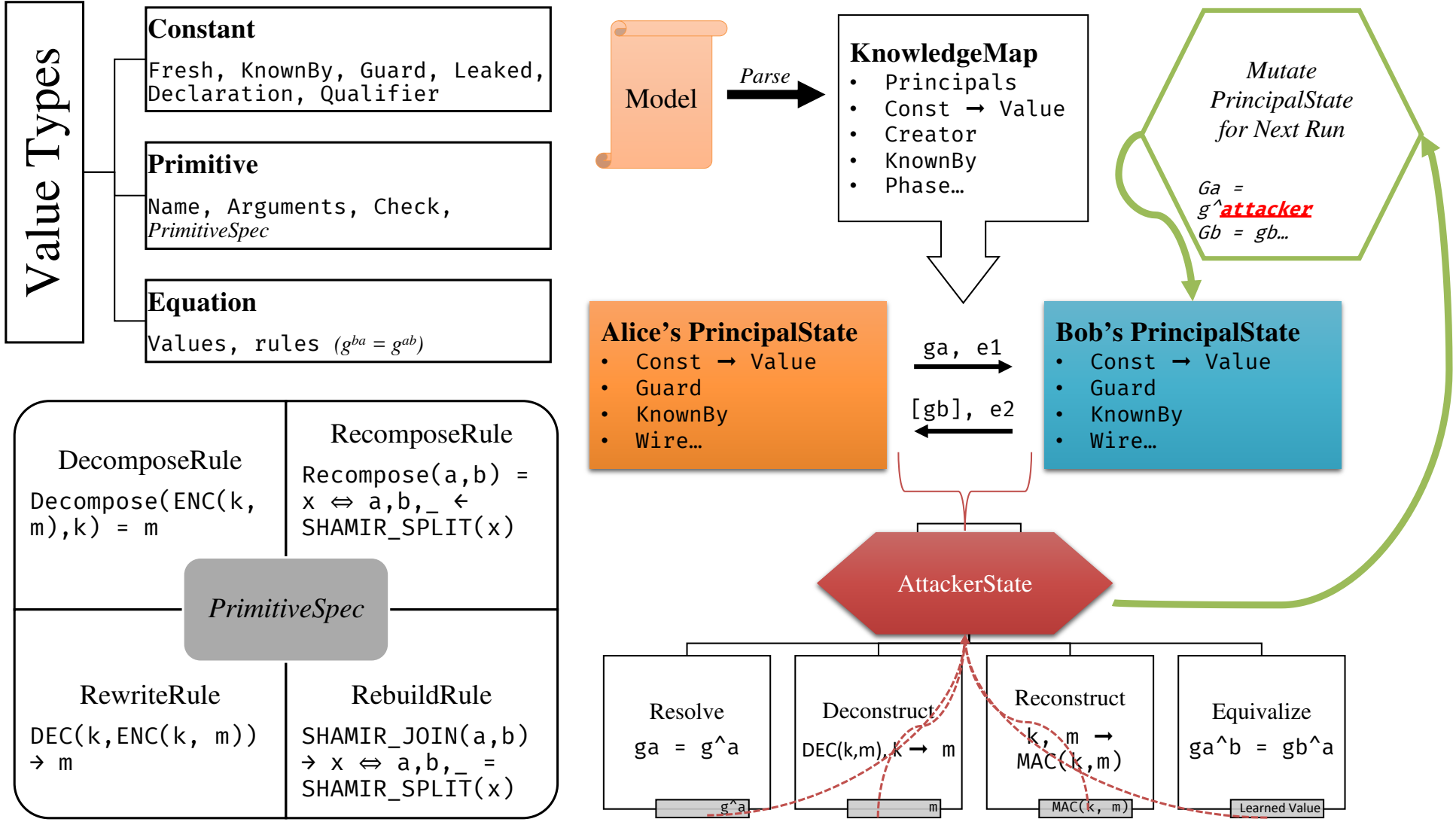
Guarded Constants, Checked Primitives

- 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
]
```


Value Types
Constant
Fresh, KnownBy, Guard, Leaked, Declaration, Qualifier
Primitive
Name, Arguments, Check, PrimitiveSpec
Equation
Values, rules $(g^{ba} = g^{ab})$
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
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)
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
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.

Coq: Verifpal Ring Signatures (Partial)

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.
```

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.
```

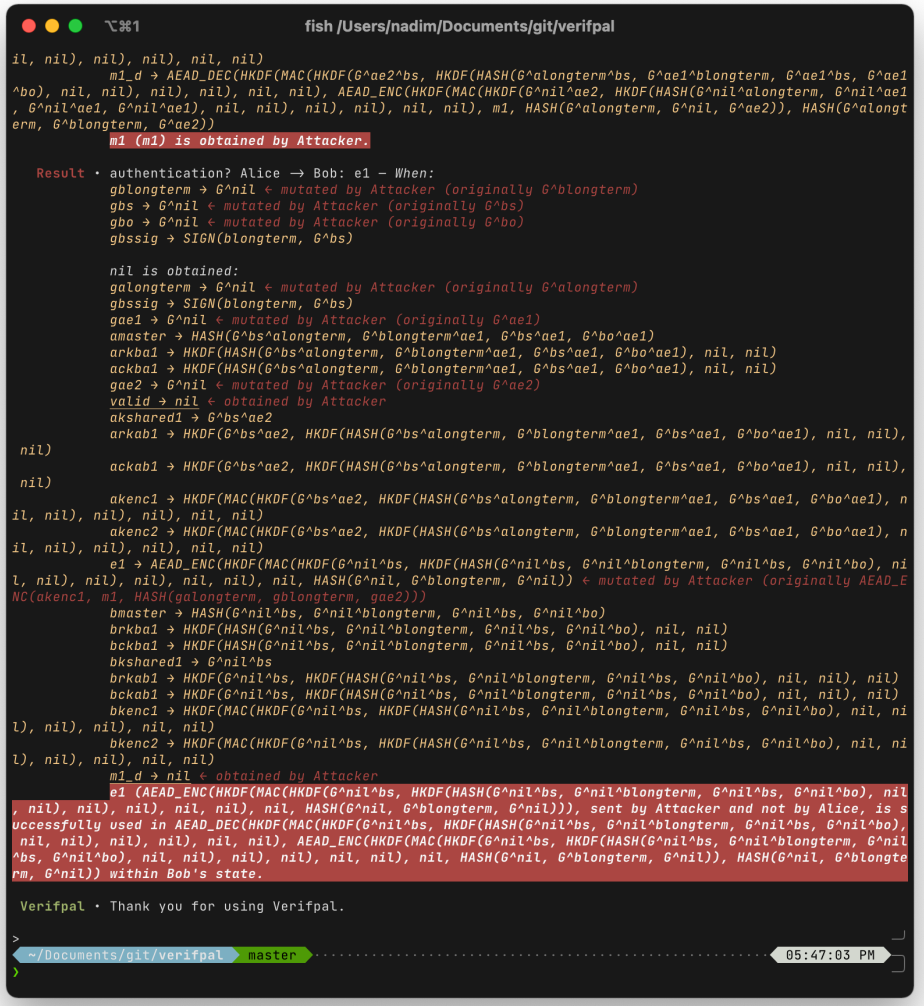
Listing 1.2. Verifpal Attack Trace

```
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.



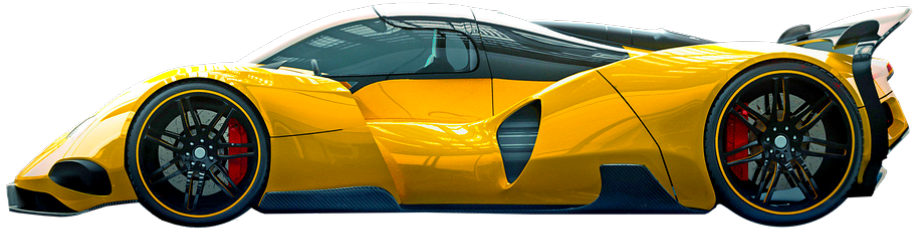
Who's Using Verifpal?

ASSA ABLOY

quarkslab

SECURING EVERY BIT OF YOUR DATA

Verifpal

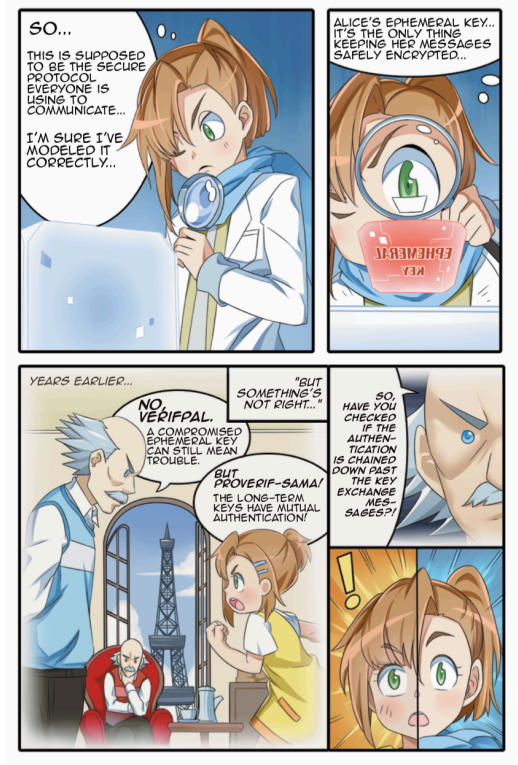


ProVerif, CryptoVerif, Tamarin…

F*, probably

Verifpal: a learning tool with an important mission

- Verifpal does aim to give accurate, insightful results, but its queries are quite general, and the guarantees it gives are much weaker and less precise than ProVerif, Tamarin, CryptoVerif, etc. etc.
- On the other hand, Verifpal is much easier to learn, to sketch protocols with, and (especially with the Visual Studio Code extension) to use as an "intelligent notebook" for studying protocol designs, with support for some advanced features.



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.

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.

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 practical learning session

- Please download and open the Verifpal User Manual:

https://verifpal.com/res/pdf/manual.pdf

We will now do a two-hour practical session with Verifpal, in which we will progress to cover advanced protocols (like Signal) with interesting security properties (like forward secrecy).

Part 2: Other Analysis Frameworks

- **ProVerif**
  - Symbolic model automated protocol analysis and verification
- **Tamarin**
  - Symbolic model semi-automated protocol analysis and verification
- **CryptoVerif**
  - Computational model protocol modeling and semi-automated game-based proof assistant
- **F***
  - Programming language that links Ocaml-like language to Z3 SMT solver so that types are proofs

ProVerif
“Cryptographic protocol verifier in the formal model”

- **Time for another practical session**
- We’ll take a look at some example models together.
- We’ll discuss how ProVerif works:
  - Horn clauses,
  - Communicating sequential processess (CSP)…

```
type passwd.
type nonce.

fun encrypt(nonce, passwd): nonce.
fun decrypt(nonce, passwd): nonce.
equation forall x: nonce, y: passwd; decrypt(encrypt(x,y),y) = x.
equation forall x: nonce, y: passwd; encrypt(decrypt(x,y),y) = x.

fun incr(nonce): nonce.

free c: channel.
free pw: passwd [private].
weaksecret pw.

let processA =
    new N: nonce;
    out(c, encrypt(N, pw)).

let processB =
    in(c, x: nonce);
    let n = decrypt(x, pw) in
    out(c, encrypt(incr(n), pw)).

process
    (!processA) | (!processB)
```

CryptoVerif: quick interactive session

- Materials based on the CryptoVerif tutorial, by Bruno Blanchet and Benjamin Lipp:
- https://bblanche.gitlabpages.inria.fr/CryptoVerif/tutorial/

Conclusion Slide

Thank you for attending!

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

Check out Verifpal today:

verifpal.com