
Bhargavan, Bichhawat, Do, Hosseyni, Küsters, Schmitz, Würtele
Cryptographic Protocols are Everywhere: **Golden Era of Crypto**

- Ubiquitous HTTPS: TLS 1.3, QUIC, ACME/Let’s Encrypt, …
- Secure Messaging: Signal, MLS, …
- Single-Sign On: OAuth, OIDC, SAML, …
- Wireless: Wifi/WPA, 4G, 5G, Zigbee, …
- Payment: EMV, W3C Web Payments, …
- Post-Quantum Crypto: NIST KEMs, Signature, …
- Lightweight Crypto: IETF LAKE, NIST LWC

[Images of popular cryptographic protocols and technologies]
Cryptographic Protocols are Everywhere: Golden Era of Crypto

- Ubiquitous HTTPS: TLS 1.3, QUIC, ACME/Let’s Encrypt, …
- Secure Messaging: Signal, MLS, …
- Single-Sign On: OAuth, OIDC, SAML, …
- Wireless: Wi-Fi/WPA, 4G, 5G, Zigbee, …
- Payment: EMV, W3C Web Payments, …
- Post-Quantum Crypto: NIST KEMs, Signature, …
- Lightweight Crypto: IETF LAKE, NIST LWC
Cryptographic Protocols are Everywhere: **Golden Era of Crypto**

- Ubiquitous HTTPS: TLS 1.3, QUIC, ACME/Let’s Encrypt, …
- Secure Messaging: Signal, MLS, …
- Single-Sign On: OAuth, OIDC, SAML, …
- Wireless: Wifi/WPA, 4G, 5G, Zigbee, …
- Payment: EMV, W3C Web Payments, …
- Post-Quantum Crypto: NIST KEMs, Signature, …
- Lightweight Crypto: IETF LAKE, NIST LWC
Cryptographic Protocols are Everywhere: Golden Era of Crypto

- Ubiquitous HTTPS: TLS 1.3, QUIC, ACME/Let's Encrypt, …
- Secure Messaging: Signal, MLS, …
- Single-Sign On: OAuth, OIDC, SAML, …
- Wireless: Wifi/WPA, 4G, 5G, Zigbee, …
- Payment: EMV, W3C Web Payments, …
- Post-Quantum Crypto: NIST KEMs, Signature, …
- Lightweight Crypto: IETF LAKE, NIST LWC

- 'Triple handshake' bug another big problem for TLS/SSL
- Logjam: the latest TLS vulnerability explained
Cryptographic Protocols are Everywhere: Golden Era of Crypto
Cryptographic Protocols are Everywhere: Golden Era of Crypto

- Ubiquitous HTTPS: TLS 1.3, QUIC, ACME/Let’s Encrypt, …
- Secure Messaging: Signal, MLS, …
- Single-Sign On: OAuth, OIDC, SAML, …
- Wireless: Wifi/WPA, 4G, 5G, Zigbee, …
- Payment: EMV, W3C Web Payments, …
- Post-Quantum Crypto: NIST KEMs, Signature, …
- Lightweight Crypto: IETF LAKE, NIST LWC
Signal Messaging Protocol

- Asynchronous continuous key exchange protocol

\[
G = \langle g \rangle
\]

\[
x \text{ random} \quad g^x
\]

\[
(y^x)^x = g^{xy}
\]

\[
y \text{ random} \quad (g^x)^y = g^{xy}
\]
Signal Messaging Protocol

- Asynchronous continuous key exchange protocol
Signal Messaging Protocol

- Asynchronous continuous key exchange protocol
- Multiple subprotocols
  - X3DH (initial key exchange)
  - DH Ratchet (post-compromise security)
  - Hash Ratchet (forward security)
  - Authenticated Encryption (message security)
Signal Messaging Protocol

- Asynchronous continuous key exchange protocol
- Multiple subprotocols
  - X3DH (initial key exchange)
  - DH Ratchet (post-compromise security)
  - Hash Ratchet (forward security)
  - Authenticated Encryption (message security)
- Inherently recursive
  - Security of each message depends on a chain of derived keys
Signal Messaging Protocol

- Asynchronous continuous key exchange protocol
- Multiple subprotocols
  - X3DH (initial key exchange)
  - DH Ratchet (post-compromise security)
  - Hash Ratchet (forward security)
  - Authenticated Encryption (message security)
- Inherently recursive
  - Security of each message depends on a chain of derived keys
- Can we mechanically verify that the protocol is secure?
Formalizing Signal

**Initiator I**

**Responder R**

---

**Initiate** ($i, r', m', r'_1) \rightarrow (r_0, s)$

- $a_{r_0} = \text{HMAC}(m', r'_1)$
- $r_0 = \text{HMAC}(a_{r_0}, 0x0000)$

**SendRatchet** ($r_0, g', r'_2, r_3)\rightarrow (r_0, r'_3, r_3)$

- $b_0 = \text{HMAC}(a_{r_0}, 0x00)$
- $b_0 = \text{HMAC}(a_{r_0}, 0x00)$
- $c_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $c_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $c_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $c_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $c_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$

**Encrypt** ($g, g', g'' : \text{HMAC}(c_0, s) : m_0 \rightarrow (c_0^* : c_0^* : t_0)$

- $c_0^* = \text{HMAC}(c_0, 0x01)$
- $c_0^* = \text{HMAC}(c_0, 0x01)$
- $c_0^* = \text{HMAC}(c_0, 0x01)$
- $c_0^* = \text{HMAC}(c_0, 0x01)$
- $c_0^* = \text{HMAC}(c_0, 0x01)$
- $c_0^* = \text{HMAC}(c_0, 0x01)$

**MSG_0** ($g', g'' : c_0^* : t_0$)

**ReceiveRatchet** ($r_0, g', g'' : (c_0^* : c_0^* : t_0)$)

- $a_{r_0} = \text{HMAC}(m', r'_1)$
- $a_{r_0} = \text{HMAC}(m', r'_1)$
- $a_{r_0} = \text{HMAC}(m', r'_1)$
- $a_{r_0} = \text{HMAC}(m', r'_1)$
- $a_{r_0} = \text{HMAC}(m', r'_1)$
- $a_{r_0} = \text{HMAC}(m', r'_1)$

**Decrypt** ($c_0^* : c_0^* : t_0$)

- $m_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $m_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $m_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $m_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $m_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$
- $m_0 = \text{FORMAT}(g', 0, 0, \text{AES-CBC}(b_0, b_0, m_0))$

**MSG** ($c_0^* : t_0$)

---

**SendRatchet** ($r_0, g', r'_3, r_3)\rightarrow (r_0, g''$, $r'_3, r_3)$

**Decrypt** ($c_0^* : c_0^* : t_0$)

**Decrypt** ($c_0^* : c_0^* : t_0$)

**Decrypt** ($c_0^* : c_0^* : t_0$)

**Decrypt** ($c_0^* : c_0^* : t_0$)

**Session State**:

- 1.
- 2.
- 3.
Formalizing Signal

- **Existing Analyses**
  - Using ProVerif and CryptoVerif
  - Model X3DH, Double Ratchet
  - Few hundred lines written in applied pi calculus
Formalizing Signal

- **Existing Analyses**
  - Using ProVerif and CryptoVerif
  - Model X3DH, Double Ratchet
  - Few hundred lines written in applied pi calculus

- One major limitation of existing analyses: Proofs for only 3 message rounds due to recursion
Single Sign-On allows users to authenticate to service A using an account at service B
Single Sign-On allows users to authenticate to service A using an account at service B.
Security Protocols: SSO

Single Sign-On allows users to authenticate to service A using an account at service B

• One account for many services
Single Sign-On allows users to authenticate to service A using an account at service B

- One account for many services
- Widely adopted: Web, IoT, Enterprise Networks, ...
Security Protocols: SSO

Single Sign-On allows users to authenticate to service A using an account at service B

- One account for many services
- Widely adopted: Web, IoT, Enterprise Networks, …
OAuth2.0

Authorization Code Mode

1. "Login with Google."
2. User authentication
3. Redirect to Dropbox with Authorization Code AC in URI
4. Request URI with AC

5. Retrieve AT using AC
6. Retrieve data using AT

7. Logged in
The Web is a Complex Environment

- DNS, HTTP, HTTPS
- window & document structure
- honest and malicious scripts
- Web storage & cookies
- Web messaging & XHR
- message headers
- redirections
- security policies
- ...

Origin: https://example.com
Many Attacks based on Web Features (Examples)

1. "Login with Google."
2. user authentication
   2.1 Load login form
   2.2 HTTP POST username & password
3. HTTP 307 redirect to Dropbox with Authorization Code AC in URI
4. HTTP POST to URI with AC

- 307 Redirect: Repeat POST with same data
- 307 Redirect Attack: Fett, Küsters, Schmitz [CCS 2016]
- Google password leaks to Dropbox
Many Attacks based on Web Features (Examples)

Authorization Code Mode

1. "Login with Google."

2. user authentication
   2.1 Load login form
   2.2 HTTP POST username & password

3. HTTP 307 redirect to Dropbox with Authorization Code AC in URI

4. HTTP POST to URI with AC

Google password leaks to Dropbox

Mix-Up possible, see Fett, Küsters, Schmitz [CCS 2016]

307 Redirect Attack: Fett, Küsters, Schmitz [CCS 2016]

307 Redirect: Repeat POST with same data
Many Attacks based on Web Features (Examples)

1. "Login with Google."

2. user authentication

2.1 Load login form

2.2 HTTP POST username & password

3. HTTP 307 redirect to Dropbox URI

4. HTTP POST to URI with AC

Mix-Up possible, see Fett, Küsters, Schmitz [CCS 2016]

Session Swapping attacks possible, see Fett, Küsters, Schmitz [CCS 2016]

Google password leaks to Dropbox

307 Redirect: Repeat POST with same data

307 Redirect Attack: Fett, Küsters, Schmitz [CCS 2016]

Many Attacks based on Web Features (Examples)

1. "Login with Google."
2. user authentication
   2.1 Load login form
   2.2 HTTP POST username & password
3. HTTP 307 redirect to Dropbox with Authorization Code \( AC \) in URI
4. HTTP POST to URI with \( AC \)

- Mozilla BrowserID: Privacy broken due to iframe usage, see Fett, Küsters, Schmitz [ESORICS 2015]
- 307 Redirect: Repeat POST with same data
- 307 Redirect Attack: Fett, Küsters, Schmitz [CCS 2016]
- Mix-Up possible, see Fett, Küsters, Schmitz [CCS 2016]
- Session Swapping attacks possible, see Fett, Küsters, Schmitz [CCS 2016]
- Google password leaks to Dropbox
- Mozilla BrowserID: Privacy broken due to iframe usage, see Fett, Küsters, Schmitz [ESORICS 2015]
Many Attacks based on Web Features (Examples)

Mozilla BrowserID: Privacy broken due to iframe usage, see Fett, Küsters, Schmitz [ESORICS 2015]

W3C Web Payments: Missing checks lead to double charging (under submission)

Mix-Up possible, see Fett, Küsters, Schmitz [CCS 2016]

Session Swapping attacks possible, see Fett, Küsters, Schmitz [CCS 2016]

Google password leaks to Dropbox

HTTP POST to URI with AC

307 Redirect: Repeat POST with same data

307 Redirect Attack: Fett, Küsters, Schmitz [CCS 2016]
Analysis of Security Protocols on the Web: Main Research Lines
Analysis of Security Protocols on the Web: Main Research Lines

• Alloy
  - Finite-state model checker
  - Very simplified and abstract models of the web
Analysis of Security Protocols on the Web: Main Research Lines

- **Alloy**
  - Finite-state model checker
  - Very simplified and abstract models of the web

- **ProVerif: WebSpi (Bansal et al., 2013)**
  - Limitations imposed by ProVerif
  - Simplified and abstract models of the web
Analysis of Security Protocols on the Web: Main Research Lines

- **Alloy**
  - Finite-state model checker
  - Very simplified and abstract models of the web

- **ProVerif: WebSpi (Bansal et al., 2013)**
  - Limitations imposed by ProVerif
  - Simplified and abstract models of the web

- **Web Infrastructure Model (WIM)**
  - Fett, Küsters, Schmitz [S&P 2014]
  - Detailed model
  - Used for many case studies (OAuth, OIDC, BrowserID, FAPI, W3C Web Payments)
  - Manual pen-and-paper analyses
Analysis of Security Protocols: Tools

Computational Tools: CryptoVerif, EasyCrypt, ...

• Focus on cryptographic core
• Messages are bitstrings
• Probabilistic

Symbolic Tools: ProVerif, Tamarin, RCF, ...

• Abstract cryptography
• Messages are formal terms
Analysis of Security Protocols: Tools

Computational Tools: CryptoVerif, EasyCrypt, ...

- Focus on cryptographic core
- Messages are bitstrings
- Probabilistic

Symbolic Tools: ProVerif, Tamarin, RCF, ...

- Abstract cryptography
- Messages are formal terms
Existing Symbolic Approaches and DY*

DY-style tools:
Tamarin, ProVerif, ...

Dependent Types:
Existing Symbolic Approaches and DY*

DY-style tools:
Tamarin, ProVerif, ...

Dependent Types:
RCF, F7, ...
Existing Symbolic Approaches and DY*

- **DY-style tools:** Tamarin, ProVerif, ...

  - **Dependent Types:** RCF, F7, ...

  - focus on protocol core

  - ✓ (mostly) automated analysis
  - ✓ global trace & properties
  - ✓ equational theories

  - × abstract models
  - × bounded data structures
  - × no modularity
  - × limited inductive reasoning
  - × interoperability
Existing Symbolic Approaches and DY*

DY-style tools:
Tamarin, ProVerif, ...

Dependent Types:
RCF, F7, ...

focus on protocol core

- abstract models
- bounded data structures
- no modularity
- limited inductive reasoning
- interoperability

✔ (mostly) automated analysis
✔ global trace & properties
✔ equational theories

✔ modular proofs
✔ implementation level analysis
✔ unbounded structures
✔ inductive reasoning
✔ executable models
✔ interoperability

✗ missing global view
✗ limited expressivity w.r.t. security prop.
✗ limited support for mutable state
✗ less automation
✗ no equational theories (e.g., DH)

* focus on implementation aspects
Existing Symbolic Approaches and DY*

- **DY-style tools:** Tamarin, ProVerif, ...
- **Dependent Types:** RCF, F7, ...

**focus on protocol core**
- abstract models
- bounded data structures
- no modularity
- limited inductive reasoning
- interoperability

**✔ (mostly) automated analysis**
- global trace & properties
- equational theories

**✔ modular proofs**
- implementation level analysis
- unbounded structures
- inductive reasoning
- executable models
- interoperability

**✗ missing global view**
- limited expressivity w.r.t. security prop.
- limited support for mutable state
- less automation
- no equational theories (e.g., DH)

**“bridging the gap”**

**DY* based on the F* programming language**

DY-style tools: Tamarin, ProVerif, ...

Dependent Types: RCF, F7, ...

focus on implementation aspects

- modular proofs
- (mostly) automated analysis
- global trace & properties
- equational theories

- implementation level analysis
- unbounded structures
- inductive reasoning
- executable models
- interoperability
What is F*?

- **Functional programming language** aimed at program verification
- Developed and actively supported by Microsoft Research, INRIA, and others
- Already used to prove security of, for example, parts of **TLS 1.3**
- Rich, versatile **type system**
  - Dependent types
  - Refinement types
  - Pre/post conditions
  - Backed by SMT-Solver Z3
  - Can be used to precisely express strong (security) properties
- F* program can be translated to OCaml, F#, C, or JavaScript
F* - Simple Example

val factorial : nat -> nat

let rec factorial n =
    if n <= 1 then 1 else n * (factorial (n - 1))
**F* - Simple Example**

```ocaml
val factorial : nat -> nat

let rec factorial n =
  if n <= 1 then 1 else n * (factorial (n-1))
```

Property:
\[ \forall n \in \mathbb{N}. \text{factorial } n > 0 \]
F* - Simple Example

val factorial : nat -> nat

let rec factorial n =
    if n <= 1 then 1 else n * (factorial (n-1))

val factorial_lemma : n:nat -> Lemma ( factorial n > 0 )

Property:
∀ n ∈ ℕ . factorial n > 0
**F* - Simple Example**

```plaintext
val factorial: nat -> nat

let rec factorial n =
    if n <= 1 then 1 else n * (factorial (n-1))

val factorial_lemma: n:nat -> Lemma ( factorial n > 0 )

let rec factorial_lemma n = match n with
    |0 -> ()
    |_ -> factorial_lemma (n-1)
```

Property:

\( \forall n \in \mathbb{N} . \text{factorial } n > 0 \)
**F* - Simple Example**

```ocaml
val factorial : nat -> nat

let rec factorial n =
  if n<=1 then 1 else n * (factorial (n-1))
```

**Property:**
\[ \forall n \in \mathbb{N} . \text{factorial } n > 0 \]

```ocaml
val factorial_lemma : n:nat -> Lemma ( factorial n > 0 )

let rec factorial_lemma n = match n with
  |0 -> ()
  |_ -> factorial_lemma (n-1)
```

**Induction Start**
val factorial : nat \rightarrow nat

let rec factorial n =
    if n <= 1 then 1 else n * (factorial (n-1))

val factorial_lemma : n: nat \rightarrow Lemma ( \text{factorial } n > 0 )

let rec factorial_lemma n = match n with
    | 0  -> ()  \text{Induction Start}
    | _   -> factorial_lemma (n-1) \text{Induction Step}
val factorial: nat -> nat

let rec factorial n =
    if n <= 1 then 1 else n * (factorial (n-1))

val factorial_lemma: n:nat -> Lemma ( factorial n > 0 )

let rec factorial_lemma n = match n with
    |0 -> ()
    |_ -> factorial_lemma (n-1)

Property:
\[ \forall n \in \mathbb{N} . \text{factorial } n > 0 \]
Dolev-Yao* (DY*): Architecture

- Runtime Model
  - Crypto
  - Network
  - Communication
  - Application
  - State

Global Trace
Dolev-Yao* (DY*): Architecture

Append-only log that captures relevant interaction with the framework.

Global Trace

Runtime Model

Crypto

Network Communication

Application State
Dolev-Yao* (DY*): Architecture

Append-only log that captures relevant interaction with the framework.

Global Trace

Runtime Model

Crypto

Network

Communication

Application

State

Protocol Implementation
Append-only log that captures relevant interaction with the framework.

Dolev-Yao* (DY*): Architecture

Runtime Model

Labeling Layer

Crypto

Network Communication

Application State

Global Trace

Generic Proofs

Protocol Implementation
Append-only log that captures relevant interaction with the framework.

Dolev-Yao* (DY*): Architecture

- Runtime Model
- Labeling Layer
  - Generic Proofs
  - Protocol Implementation

Security Properties

Global Trace

- Crypto
- Network Communication
- Application State

Global Trace
Dolev-Yao* (DY*): Architecture

Append-only log that captures relevant interaction with the framework.

Security Properties

Generic (e.g., attackers) + Application-specific invariants

Global Trace

Labeling Layer

Runtime Model

Crypto

Network Communication

Application State

Generic Proofs

Protocol Implementation
Dolev-Yao* (DY*): Architecture

- Append-only log that captures relevant interaction with the framework.
- Security Properties
- Global Trace
- Generic (e.g., attackers) + Application-specific invariants

Runtime Model

Labeling Layer

Security Properties

Generic (e.g., attackers) + Application-specific invariants

INVENTARIES

Dolev-Yao* (DY*): Architecture

Crypto

Network Communication

Application State

Generic Proofs

Global Trace

Protocol Implementation

GDR’21 | sec.uni-stuttgart.de
DY* - Bhargavan, Bishhawat, Do, Hosseyni, Küsters, Schmitz, Würtele
Dolev-Yao* (DY*): Architecture

Security Properties

Append-only log that captures relevant interaction with the framework.

Global Trace

Generic (e.g., attackers) + Application-specific invariants

F* verifies that application preserves invariants

Labeling Layer

Generic Proofs

Protocol Implementation

Runtime Model

Network Communication

Application State

Crypto

INVARIANTS

IN V A R I A N T S
Dolev-Yao* (DY*): Architecture

Append-only log that captures relevant interaction with the framework.

Security Properties

INTEGRITY

Generic (e.g., attackers) + Application-specific invariants

Global Trace

F* verifies that application preserves invariants

Generic Proofs

Protocol Implementation

Runtime Model

Labeling Layer

Crypto

Network Communication

Application State

Global Trace
Attacker Model

Active Network Attacker

- Can derive arbitrary messages from its knowledge
- Cannot “break” crypto, i.e., no decryption w/o key, no forging of signatures, ...
- Can (dynamically) corrupt principals and sessions

Goal: Show that protocol is secure given such an attacker
Active Network Attacker

- Can derive arbitrary messages from its knowledge
- Cannot “break” crypto, i.e., no decryption w/o key, no forging of signatures, ...
- Can (dynamically) corrupt principals and sessions

Goal: Show that protocol is secure given such an attacker
Dolev-Yao* (DY*): Architecture

- Append-only log that captures relevant interaction with the framework.
- Security Properties
- Global Trace
- INVARIANTS
- Generic (e.g., attackers) + Application-specific invariants

Runtime Model
- Crypto
- Network
- Communication
- Application State
- Protocol Implementation (stateless)

Labeling Layer
- Generic Proofs
- F* verifies that application does not break invariants

INVA...
Dolev-Yao* (DY*): Architecture

Append-only log that captures relevant interaction with the framework.

Runtime Model

Crypto

Network

Communication

Application State

F* verifies that application does not break invariants

Generic (e.g., attackers) + Application-specific invariants

Global Trace

Labeling Layer

Generic Proofs

Protocol Implementation (stateless)

Security Properties

Generic (e.g., attackers) + Application-specific invariants

INVERSIONS

INVARIANTS

Append-only log that captures relevant interaction with the framework.
DY* Example: Protocol Code ISO-DH

```ocaml
let initiator_send_msg_1 a b =
  let si = new_session_number a in
  let (|t0,x|) = rand_gen (readers [V a si 0]) (dh_usage "ISODH.dh_key") in
  let gx = dh_pk x in

  let ev = initiate a b gx in
  trigger_event a ev;

  let t1 = global_timestamp () in
  let new_ss_st = InitiatorSentMsg1 b x in
  let new_ss = serialize_valid_session_st t1 a si 0 new_ss_st in
  new_session a si 0 new_ss;

  let t2 = global_timestamp () in
  let msg1 = Msg1 a gx in
  let w_msg1 = serialize_msg t2 msg1 in
  let i = send a b w_msg1 in
  i, si
```
DY* Example: Protocol Code ISO-DH

let initiator_send_msg_1 a b =
  let si = new_session_number a in
  let (|t0,x|) = rand_gen (readers [V a si 0]) (dh_usage "ISODH.dh_key") in
  let gx = dh_pk x in
  let ev = initiate a b gx in
  trigger_event a ev;

  let t1 = global_timestamp () in
  let new_ss_st = InitiatorSentMsg1 b x in
  let new_ss = serialize_valid_session_st t1 a si 0 new_ss_st in
  new_session a si 0 new_ss;

  let t2 = global_timestamp () in
  let msg1 = Msg1 a gx in
  let w_msg1 = serialize_msg t2 msg1 in
  let i = send a b w_msg1 in
  i, si

Generate a fresh nonce
let initiator_send_msg_1 a b =
  let si = new_session_number a in
  let (|t0,x|) = rand_gen (readers [V a si 0]) (dh_usage "ISODH.dh_key") in
  let gx = dh_pk x in
  let ev = initiate a b gx in
  trigger_event a ev;

  let t1 = global_timestamp () in
  let new_ss_st = InitiatorSentMsg1 b x in
  let new_ss = serialize_valid_session_st t1 a si 0 new_ss_st in
  new_session a si 0 new_ss;

  let t2 = global_timestamp () in
  let msg1 = Msg1 a gx in
  let w_msg1 = serialize_msg t2 msg1 in
  let i = send a b w_msg1 in
  i, si
DY* Example: Protocol Code ISO-DH

```ocaml
let initiator_send_msg_1 a b =
  let si = new_session_number a in
  let (|t0,x|) = rand_gen (readers [V a si 0]) (dh_usage "ISODH.dh_key") in
  let gx = dh_pk x in
  let ev = initiate a b gx in
  trigger_event a ev;

  let t1 = global_timestamp () in
  let new_ss_st = InitiatorSentMsg1 b x in
  let new_ss = serialize_valid_session_st t1 a si 0 new_ss_st in
  new_session a si 0 new_ss;

  let t2 = global_timestamp () in
  let msg1 = Msg1 a gx in
  let w_msg1 = serialize_msg t2 msg1 in
  let i = send a b w_msg1 in
  i, si
```

- Generate a fresh nonce
- Protocol event
- Update state
let initiator_send_msg_1 a b =
  let si = new_session_number a in
  let (|t0,x|) = rand_gen (readers [V a si 0]) (dh_usage "ISODH.dh_key") in
  let gx = dh_pk x in
  let ev = initiate a b gx in
  trigger_event a ev;

  let t1 = global_timestamp () in
  let new_ss_st = InitiatorSentMsg1 b x in
  let new_ss = serialize_valid_session_st t1 a si 0 new_ss_st in
  new_session a si 0 new_ss;

  let t2 = global_timestamp () in
  let msg1 = Msg1 a gx in
  let w_msg1 = serialize_msg t2 msg1 in
  let i = send a b w_msg1 in
  i, si
Dolev-Yao* (DY*): Architecture

- **Runtime Model**
  - Crypto
  - Network
  - Communication
  - Application State
  - Global Trace
  - Append-only log that captures relevant interaction with the framework.

- **Labeling Layer**
  - Generic (e.g., attackers) + Application-specific invariants
  - F* verifies that application does not break invariants
  - Generic Proofs
  - Protocol Implementation (stateless)

- **Security Properties**
  - INVARIANTS
  - Global Trace
  - Crypto Network Communication Application State

- **Diagram Notes**
  - INVARIANTS: Generic (e.g., attackers) + Application-specific invariants
  - Protocol Implementation (stateless)

- **Append-only log**
  - Relevant interaction with the framework.

- **F***
  - Verifies that application does not break invariants

- **Generic Proofs**
  - Proof verification process
Dolev-Yao* (DY*): Architecture

- Append-only log that captures relevant interaction with the framework.
- Security Properties
- INVARIANTS
- Generic (e.g., attackers) + Application-specific invariants
- Global Trace

Runtime Model
- Crypto
- Network
- Communication
- Application State

F* verifies that application does not break invariants

Labeling Layer
- Generic Proofs
- Protocol Implementation (stateless)
Forward secrecy for an authenticated key exchange protocol (ISO-DH):

val initiator_forward_secrecy_lemma: i:timestamp -> a:principal -> b:principal ->
    gx:bytes -> gy:bytes -> k:bytes -> LCrypto unit (pki isodh)

(requires (fun t0 -> i < trace_len t0 /
            did_event_occur_at i a (finishI a b gx gy k)))

(ensures (fun t0 _ t1 -> t0 == t1 /
            corrupt_at i (P b) /
            (exists si sj vi vj . is_labeled isodh_global_usage i k
                (join (readers [V a si vi]) (readers [V b sj vj])) /
                ( corrupt_at (trace_len t0) (V a si vi) /
                corrupt_at (trace_len t0) (V b sj vj) /
                is_unknown_to_attacker_at (trace_len t0) k)
))))
**Forward secrecy** for an authenticated key exchange protocol (ISO-DH):

```ocaml
val initiator_forward_secrecy_lemma : i : timestamp -> a : principal -> b : principal ->
                              gx : bytes -> gy : bytes -> k : bytes -> LCrypto unit (pki isodh)

  (requires (fun t0 -> i < trace_len t0 /
                     did_event_occur_at i a (finishI a b gx gy k)))

  (ensures (fun t0 _ t1 -> t0 == t1 /
                corrupt_at i (P b) /
                (exists si sj vi vj . is_labeled isodh_global_usage i k
                  (join (readers [V a si vi]) (readers [V b sj vj])) /
                  ( corrupt_at (trace_len t0) (V a si vi) /
                    corrupt_at (trace_len t0) (V b sj vj) /
                    is_unknown_to_attacker_at (trace_len t0) k)
                )))
```

Whenever Alice finishes the protocol s.t. Alice assumes that she talked to Bob and exchanged a key k ...
**Forward secrecy** for an authenticated key exchange protocol (ISO-DH):

\[
\text{val initiator_forward_secrecy_lemma: } \begin{array}{c}
  i : \text{timestamp} \\
  a : \text{principal} \\
  b : \text{principal} \\
  gx : \text{bytes} \\
  gy : \text{bytes} \\
  k : \text{bytes} \\
\end{array} \rightarrow \begin{array}{c}
  \text{LCrypto unit (pki isodh)}
\end{array}
\]

\[
\text{(requires } \begin{array}{c}
  \text{fun } t0 \rightarrow i < \text{trace_len t0} \\
  \text{did_event_occur_at } i \ a \ (\text{finishI } a \ b \ gx \ gy \ k)\end{array})\]

\[
\text{(ensures } \begin{array}{c}
  \text{fun } t0 \_ \ t1 \rightarrow t0 == t1 \\
  \text{corrupt_at } i \ (P \ b) \ (\text{exists } si \ sj \ vi \ vj . \text{is_labeled isodh_global_usage i k} \\
  \text{join (readers [V a si vi]) (readers [V b sj vj])} \\
  \text{corrupt_at (trace_len t0) (V a si vi)} \\
  \text{corrupt_at (trace_len t0) (V b sj vj)} \\
  \text{is_unknown_to_attacker_at (trace_len t0) k}\end{array})\]
\]

Whenever Alice finishes the protocol s.t. Alice assumes that she talked to Bob and exchanged a key k ...

... then Bob was compromised during the protocol run OR ...
**Forward secrecy** for an authenticated key exchange protocol (ISO-DH):


(requires (fun t0 -> i < trace_len t0 /
           did_event_occur_at i a (finishI a b gx gy k)))

(ensures (fun t0 _ t1 -> t0 == t1 /
           corrupt_at i (P b) /
           (exists si sj vi vj . is_labeled isodh_global_usage i k
            (join (readers [V a si vi]) (readers [V b sj vj])) /
            ( corrupt_at (trace_len t0) (V a si vi) /
              corrupt_at (trace_len t0) (V b sj vj) /
              is_unknown_to_attacker_at (trace_len t0) k)))

Whenever Alice finishes the protocol s.t. Alice assumes that she talked to Bob and exchanged a key k ...

... then Bob was compromised during the protocol run OR ...

... the exchanged key k is unknown to the attacker (even if long-term key is later compromised).
Dolev-Yao* (DY*): Architecture

- Append-only log that captures relevant interaction with the framework.
- Generic (e.g., attackers) + Application-specific invariants
- F* verifies that application does not break invariants
- Protocol Implementation (stateless)
Dolev-Yao* (DY*): Architecture

Append-only log that captures relevant interaction with the framework.

Runtime Model

F* verifies that application does not break invariants

Labeling Layer

Generic (e.g., attackers) + Application-specific invariants

Security Properties

Generic (e.g., attackers) + Application-specific invariants

Case Studies

Global Trace

Network Communication

Application

Protocol Implementation (stateless)

Crypto

Crypto

Network Communication

Application

Generic Proofs

Protocol Implementation (stateless)
Case Studies So Far

- Signal Messaging Protocol
Case Studies So Far

• Signal Messaging Protocol

• Automatic Certificate Management Environment (ACME)
Case Studies So Far

- Signal Messaging Protocol
- Automatic Certificate Management Environment (ACME)
- Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM
Case Studies So Far

- Signal Messaging Protocol
- Automatic Certificate Management Environment (ACME)
- Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM
Signal Messaging Protocol

- **First mechanized proof** accounting for
  - Forward Secrecy
  - Post-compromise Security
  - Unbounded number of protocol rounds at the same time

- **First type-based formulation and proof of post-compromise security for any protocol**

- **First analysis of Signal based on dependent types**
Case Studies So Far

- **Signal Messaging Protocol**
  - Unbounded number of rounds (racheting)
  - Forward Secrecy & Post Compromise Security

- **Automatic Certificate Management Environment (ACME)**

- **Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM**
Case Studies So Far

- **Signal Messaging Protocol**
  - Unbounded number of rounds (racheting)
  - Forward Secrecy & Post Compromise Security

- **Automatic Certificate Management Environment** (ACME)

- **Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM**
Case Studies So Far

- Signal Messaging Protocol
  - Unbounded number of rounds (racheting)
  - Forward Secrecy & Post Compromise Security
- Automatic Certificate Management Environment (ACME)
- Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM
Automatic Certificate Management Environment (ACME)

- Main User: Let’s Encrypt
  - Over 250M active domains with LE certificates
  - Currently issuing ~2M certificates/day
  - Issued more than 1B certificates so far
- ACME is supported by many other certification authorities (e.g., Buypass, DigiCert, GlobalSign, ZeroSSL)

Statistics from https://letsencrypt.org/stats/
Automatic Certificate Management Environment (ACME)

ACME is extremely security critical

- Two previous formal analyses (ProVerif, Tamarin)
  - Very abstract models
  - Only consider certificates for a single domain
  - No loops of subprotocols
  - No precise state management
  - No details of message formats

- Our analysis is one of the largest & most in-depth formal security analyses in the literature (16,000 LoC)

- ACME client model can interoperate with real-world server

Limited by the used tools – DY* analysis overcomes these limitations
Case Studies So Far

- **Signal Messaging Protocol**
  - Unbounded number of rounds (racheting)
  - Forward Secrecy & Post Compromise Security

- **Automatic Certificate Management Environment (ACME)**
  - One of the largest & most in-depth formal security analyses in the literature (16,000 LoC)
  - ACME client model can interoperate with real-world server

- **Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM**
Case Studies So Far

- **Signal Messaging Protocol**
  - Unbounded number of rounds (racheting)
  - Forward Secrecy & Post Compromise Security

- **Automatic Certificate Management Environment (ACME)**
  - One of the largest & most in-depth formal security analyses in the literature (16,000 LoC)
  - ACME client model can interoperate with real-world server

- **Needham-Schroeder(-Lowe), ISO-DH, and ISO-KEM**
Conclusion & Future Work

- **Golden era** of cryptographic protocols

- We recently proposed DY*, a new mechanized symbolic verification framework for protocols and their code

  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable
Conclusion & Future Work

- **Golden era** of cryptographic protocols
- We recently proposed DY*, a **new mechanized symbolic verification framework** for protocols and their code
  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable
Conclusion & Future Work

- **Golden era** of cryptographic protocols
- We recently proposed DY*, a new mechanized symbolic verification framework for protocols and their code
  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable
- Lots of interesting work to be done!
  - WIM*: mechanize the Web Infrastructure Model
  - Concrete DY*: fully verified implementations
  - Equivalence properties
  - Computational analysis
Conclusion & Future Work

- **Golden era** of cryptographic protocols
- We recently proposed DY*, a new mechanized symbolic verification framework for protocols and their code
  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable

- Lots of interesting work to be done!
  - WIM*: mechanize the Web Infrastructure Model
  - Concrete DY*: fully verified implementations
  - Equivalence properties
  - Computational analysis

See [S&P ’14, ESORICS ‘15, CCS ’15, CCS ‘16, CSF ‘17, S&P ‘19]
Conclusion & Future Work

- Golden era of cryptographic protocols
- We recently proposed DY*, a new mechanized symbolic verification framework for protocols and their code
  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable

See [S&P ’14, ESORICS ’15, CCS ’15, CCS ’16, CSF ’17, S&P ’19]

- Lots of interesting work to be done!
  - WIM*: mechanize the Web Infrastructure Model
  - Concrete DY*: fully verified implementations
  - Equivalence properties
  - Computational analysis

We are hiring!
Postdoc position in Stuttgart
Conclusion & Future Work

- **Golden era** of cryptographic protocols
- We recently proposed DY*, a new mechanized symbolic verification framework for protocols and their code
  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable

See [S&P ’14, ESORICS ’15, CCS ’15, CCS ’16, CSF ’17, S&P ’19]

- Lots of interesting work to be done!
  - WIM*: mechanize the **Web Infrastructure Model**
  - Concrete DY*: fully verified implementations
  - Equivalence properties
  - Computational analysis

Find more information on: [https://sec.uni-stuttgart.de](https://sec.uni-stuttgart.de)

We are hiring!
Postdoc position in Stuttgart
Conclusion & Future Work

- **Golden era** of cryptographic protocols
- We recently proposed DY*, a **new mechanized symbolic verification framework** for protocols and their code
  - Overcomes many limitations of existing tools
  - Precise reasoning on global properties
  - Account for low-level protocol details
  - Protocol models can even be interoperable


- Lots of interesting work to be done!
  - WIM*: mechanize the **Web Infrastructure Model**
  - Concrete DY*: fully verified implementations
  - Equivalence properties
  - Computational analysis

Find more information on: [https://sec.uni-stuttgart.de](https://sec.uni-stuttgart.de)

**We are hiring!**
Postdoc position in Stuttgart

Thank you!