Formal Verification of Key Exchange Protocol Security with Tamarin Prover

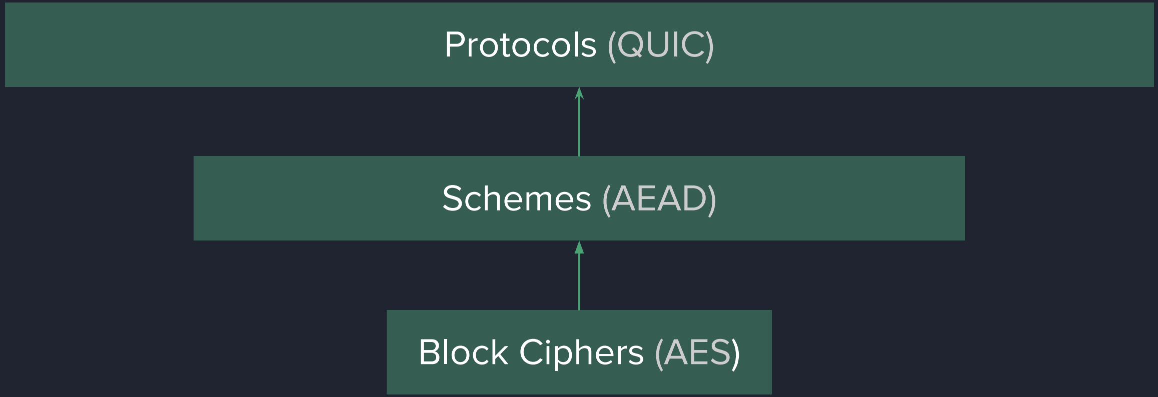
Harrison Nicholls

Agenda

- Discussion of the problem
- Introduction to Tamarin Prover
  - Overview
  - Discussion of Formal Verification
  - Model Specification
- Introduction to key exchange through Diffie-Hellman
- Applying Tamarin to QUIC

The problem:

- Multiple parties want to send and receive messages over a network.
- These parties follow rules defined by a *protocol*.
- We want to prove that the protocol is secure according to some definitions.
- Specifically, we look at the problem of *key exchange*, the establishment of a shared key over an unsecured channel.


Protocols (QUIC)
Schemes (AEAD)
Block Ciphers (AES)

Formal Verification

There are many existing tools for the verification of cryptographic protocols:

- Proverif
- AKISS
- DeepSec
- AVISPA
- Scyther
- SPEC
- Verifpal
- Tamarin

Agenda

- Discussion of the problem
- Introduction to Tamarin Prover
    - Overview
    - Discussion of Formal Verification
    - Model Specification
- Introduction to key exchange through Diffie-Hellman
- Applying Tamarin to QUIC

Tamarin Prover

Tamarin is becoming a popular tool - it’s been used to verify TLS 1.3, and EMV (“Europay, Mastercard, and Visa”) payment protocol, among other things.

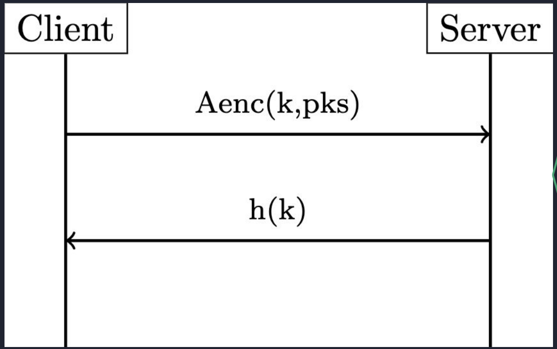
It exists more on the completeness side of the spectrum. It allows the user to assist in the process by “guiding” the proof.

Overview

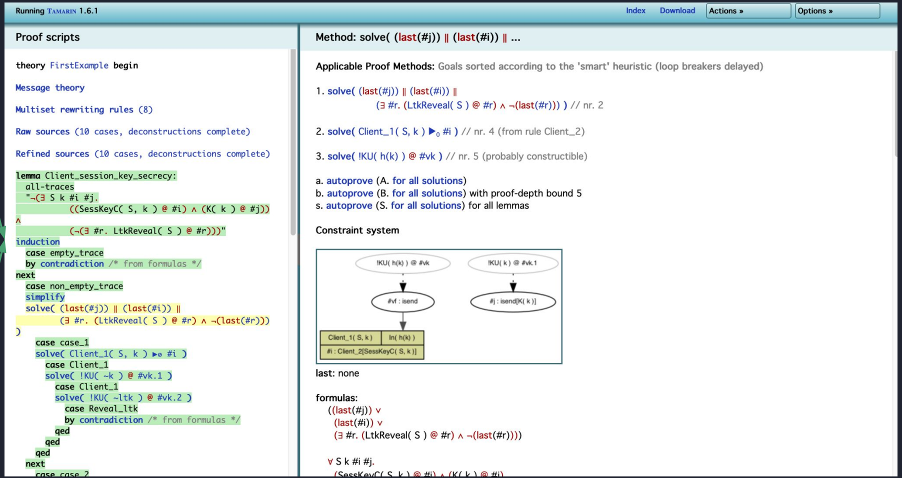
Model

```
theory FirstExample
begin

builtins: hashing, asymmetric-encryption

// Registering a public key
rule Register_pk:
    [ Fr(~ltk) ]
  -->
    [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)) ]

rule Get_pk:
    [ !Pk(A, pubkey) ]
  -->
    [ Out(pubkey) ]

rule Reveal_ltk:
    [ !Ltk(A, ltk) ]
  --[ LtkReveal(A) ]->
    [ Out(ltk) ]

// Start a new thread executing the client role, choosing the server
// non-deterministically.
rule Client_1:
    [ Fr(~k)         // choose fresh key
    , !Pk($S, pkS)   // lookup public-key of server
    ]
  -->
    [ Client_1( $S, ~k )    // Store server and key for next step of thread
    , Out( aenc(~k, pkS) )  // Send the encrypted session key to the server
    ]

rule Client_2:
    [ Client_1(S, k)   // Retrieve server and session key from previous step
    , In( h(k) )       // Receive hashed session key from network
    ]
  --[ SessKeyC( S, k ) ]-> // State that the session key 'k'
    []                     // was setup with server 'S'

// A server thread answering in one-step to a session-key setup request from
// some client.
rule Serv_1:
    [ !Ltk($S, ~ltkS)                             // lookup the private-key
    , In( request )                               // receive a request
    ]
  --[ AnswerRequest($S, adec(request, ~ltkS)) ]-> // Explanation below
    [ Out( h(adec(request, ~ltkS)) ) ]            // Return the hash of the
                                                  // decrypted request.
```

Protocol



Tamarin's output




```
lemma Client_session_key_secrecy:
  " /* It cannot be that a  */
    not(
      Ex S k #i #j.
        /* client has set up a session key 'k' with a server'S' */
        SessKeyC(S, k) @ #i
        /* and the adversary knows 'k' */
      & K(k) @ #j
        /* without having performed a long-term key reveal on 'S'. */
      & not(Ex #r. LtkReveal(S) @ r)
    )
  "

lemma Client_auth:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a)
         /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "

lemma Client_auth_injective:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a
           /* and there is no other client that had the same request */
           & (All #j. SessKeyC(S, k) @ #j ==> #i = #j)
       )
         /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "

lemma Client_session_key_honest_setup:
  exists-trace
  " Ex S k #i.
        SessKeyC(S, k) @ #i
      & not(Ex #r. LtkReveal(S) @ r)
  "

end
```

Security properties (Security model)

Tamarin Prover is a model checker that works like a constraint solver

Tamarin Prover is a model checker that works like a constraint solver

What is a model checker?

- A tool that checks whether a model of a system meets a set of specifications

Tamarin Prover is a model checker that works like a constraint solver

What is a model checker?

- A tool that checks whether a model of a system meets a set of specifications

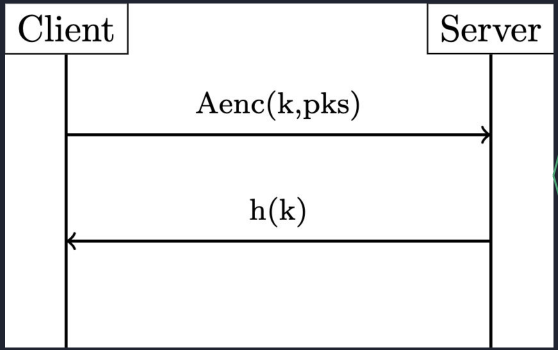
What is a constraint solver?

- A tool that solves constraint satisfaction problems.

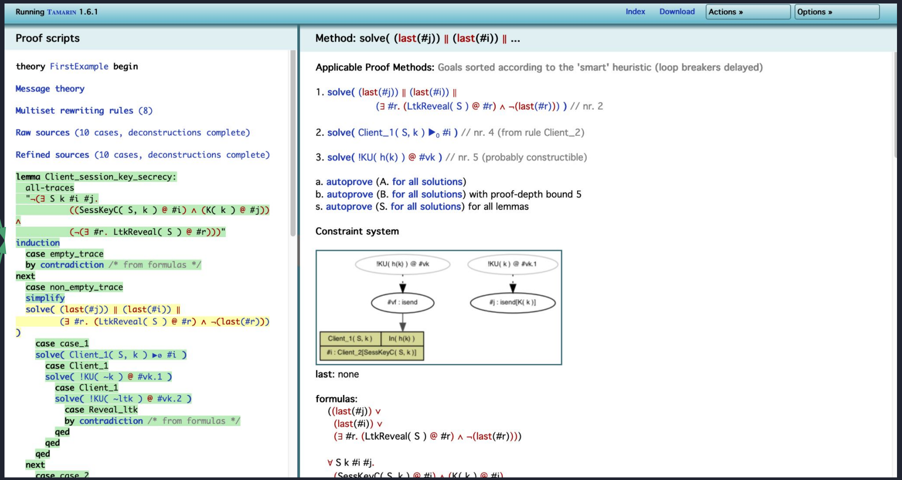
Model

```
theory FirstExample
begin

builtins: hashing, asymmetric-encryption

// Registering a public key
rule Register_pk:
    [ Fr(~ltk) ]
  -->
    [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)) ]

rule Get_pk:
    [ !Pk(A, pubkey) ]
  -->
    [ Out(pubkey) ]

rule Reveal_ltk:
    [ !Ltk(A, ltk) ]
  --[ LtkReveal(A) ]->
    [ Out(ltk) ]

// Start a new thread executing the client role, choosing the server
// non-deterministically.
rule Client_1:
    [ Fr(~k)         // choose fresh key
    , !Pk($S, pkS)   // lookup public-key of server
    ]
  -->
    [ Client_1( $S, ~k )    // Store server and key for next step of thread
    , Out( aenc(~k, pkS) )  // Send the encrypted session key to the server
    ]

rule Client_2:
    [ Client_1(S, k)   // Retrieve server and session key from previous step
    , In( h(k) )       // Receive hashed session key from network
    ]
  --[ SessKeyC( S, k ) ]-> // State that the session key 'k'
    []                     // was setup with server 'S'

// A server thread answering in one-step to a session-key setup request from
// some client.
rule Serv_1:
    [ !Ltk($S, ~ltkS)                             // lookup the private-key
    , In( request )                               // receive a request
    ]
  --[ AnswerRequest($S, adec(request, ~ltkS)) ]-> // Explanation below
    [ Out( h(adec(request, ~ltkS)) ) ]            // Return the hash of the
                                                  // decrypted request.
```

Protocol



Tamarin's output




```
lemma Client_session_key_secrecy:
  " /* It cannot be that a  */
    not(
      Ex S k #i #j.
        /* client has set up a session key 'k' with a server'S' */
        SessKeyC(S, k) @ #i
        /* and the adversary knows 'k' */
      & K(k) @ #j
        /* without having performed a long-term key reveal on 'S'. */
      & not(Ex #r. LtkReveal(S) @ r)
    )
  "

lemma Client_auth:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a)
         /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "

lemma Client_auth_injective:
  " /* For all session keys 'k' setup by clients with a server 'S' */
    ( All S k #i.  SessKeyC(S, k) @ #i
       ==>
         /* there is a server that answered the request */
       ( (Ex #a. AnswerRequest(S, k) @ a
           /* and there is no other client that had the same request */
           & (All #j. SessKeyC(S, k) @ #j ==> #i = #j)
       )
         /* or the adversary performed a long-term key reveal on 'S'
            before the key was setup. */
       | (Ex #r. LtkReveal(S) @ r & r < i)
       )
    )
  "

lemma Client_session_key_honest_setup:
  exists-trace
  " Ex S k #i.
        SessKeyC(S, k) @ #i
      & not(Ex #r. LtkReveal(S) @ r)
  "

end
```

Security properties (Security model)

Protocol Modeling: Facts

```
FactName(~A,!B,C)
```

Protocol Modeling: Rules

```
rule Name:
   [leftside(A)]
   -->
   [rightside(A)]
```

Protocol Modeling: Action Facts

```
--[Fact(A,B,C)]->
```

Protocol Modeling: Lemmas

```
lemma secrecy:
  "
    All K2c K2s #i #j.
    (
      ServResp(K2s) @ #i &
      ClientReceive(K2c) @ #j &
      #i < #j
    )
    ==> not(Ex #i #j . K(K2s) @ #i & K(K2c) @ #j)
  "
```

Four Internal Structures

- State
- Trace
- Public Channel
- Known Variables

Protocol Modeling: Rules

```
rule create_fact:
    [Fr(A)]
   --[Create()]->
    [Fact(A)]

rule consume_and_create:
    [Fact(B),Fr(C)]
   --[Consume()]->
    [New_Fact(C),Out(<'message', C>)]

rule delete:
    [New_Fact(D),In(<'message', X>)]
   --[Delete(X)]->
    []
```

Protocol Modeling: Rules

State:

Public Channel:

Known Variables:

Trace:

Protocol Modeling: Rules

```
rule create_fact:
    [Fr(A)]
   --[Create()]->
    [Fact(A)]
```

State:
{
Fact(A.1)
}

Public Channel:

Known Variables:

Trace:
Create()

Protocol Modeling: Rules

```
rule consume_and_create:
    [Fact(B),Fr(C)]
   --[Consume()]->
    [New_Fact(C),Out(<'message', C>)]
```

State:
{
~~Fact(A.1)~~
New_Fact(C.1)
}

Public Channel:
<'message',C>

Known Variables:
{
'message'
C.1
}

Trace:
Create()
Consume()

Protocol Modeling: Rules

```
rule delete:
    [New_Fact(D),In(<'message', X>)]
   --[Delete(X)]->
    []
```

State:
{
~~New_Fact(C.1)~~
}

Public Channel:
<'message',C>

Known Variables:
{
'message'
C.1
}

Trace:
Create(),
Consume(),
Delete(X.1),

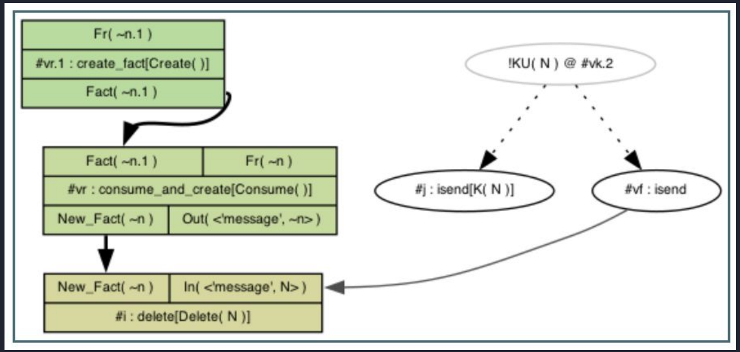
A Minimal Example: Lemma

```
lemma adv_not_know_c:
    "All #i N.
        Delete(N) @ #i
        ==>
        not(Ex #j. K(N) @ #j)
    "
```

A Minimal Example: Lemma

```
lemma adv_not_know_c:
    "All #i N.
        Delete(N) @ #i
        ==>
        not(Ex #j. K(N) @ #j)
    "
```

```
lemma adv_not_know_c:
  all-traces
  "∀ #i N. (Delete( N ) @ #i) ⇒ (¬(∃ #j. K( N ) @
#j))"
simplify
solve( New_Fact( D ) ▶₀ #i )
  case consume_and_create
  SOLVED // trace found
qed
```



A Minimal Example: Lemma

```
lemma create_before_delete:
    "All #i #j.
        Create() @ #i & Delete() @ #j
        ==>
        #i < #j
    "
end
```

A Minimal Example: Flaw

```
theory Minimal begin

Message theory

Multiset rewriting rules (5)

Raw sources (5 cases, deconstructions complete)

Refined sources (5 cases, deconstructions complete)

lemma create_before_delete:
  all-traces
  "∀ #i #j.
         ((Create( ) @ #i) ∧ (Delete( ) @ #j)) ⇒ (#i <
#j)"
simplify
solve( New_Fact( D ) ▶₀ #j )
  case consume_and_create
  SOLVED // trace found
qed

end
```

Constraint System is Solved

Constraint system

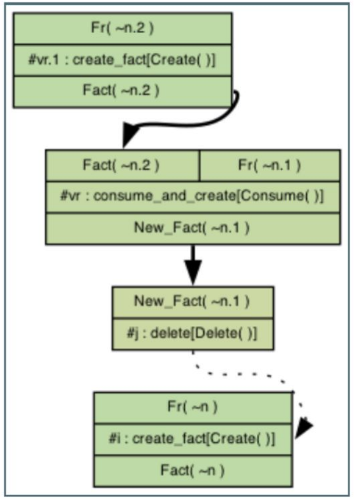


Figure 1.2: Proving a Lemma in the Minimal Theory

Protocol Modeling: Rules

```
rule create_fact:
    [Fr(A)]
   --[Create()]->
    [Fact(A)]
```

```
State:
{
Fact(A.2)
Fact(A.3)
Fact(A.4)
Fact(A.5)
...
}
```

```
Public Channel:
<'message',C>
```

```
Known Variables:
{
'message'
C.1
}
```

```
Trace:
Create(),
Consume(),
Delete(X.1),
Create(),
Create(),
Create(),
Create()...
```

A Minimal Example: Restriction

```
restriction create_once:
    "All #i #j.
        Create() @ #i & Create() @ #j
        ==>
        #i = #j
    "
```

Sorts

- !F denotes that F is a persistent fact.
- ~x denotes that x is fresh.
- #i denotes that i is temporal variable (within a lemma)
- m denotes that m is a message
- ‘c’ denotes that c is a public constant.

Agenda

- Discussion of the problem
- Introduction to Tamarin Prover
  - Overview
  - Discussion of Formal Verification
  - Model Specification
- Introduction to key exchange through Diffie-Hellman
- Applying Tamarin to QUIC

Settings in Cryptography

- Symmetric
  - Single shared key
  - Not computationally expensive
- Asymmetric
  - Key pair - public and private
  - Computationally expensive

Why Key Exchange?

Key exchange allows parties to securely agree on a shared key to be used in the symmetric setting.

Diffie-Hellman Key Exchange

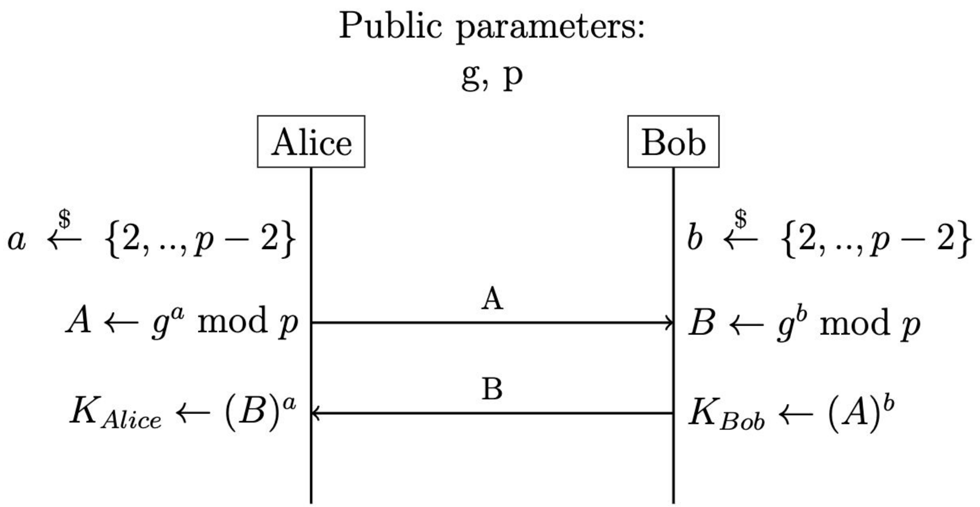


Figure 3.1: Diffie-Hellman Key Exchange. Notice that $K_{Alice} = K_{Bob} = g^{ab}$ and we are operating in the fixed group $\mathbb{Z}_p^*$. We write $x \xleftarrow{\$} X$ to denote that x is chosen uniformly from set X.

Agenda

- Discussion of the problem
- Introduction to Tamarin Prover
  - Overview
  - Discussion of Formal Verification
  - Model Specification
- Introduction to key exchange through Diffie-Hellman
- Applying Tamarin to QUIC

QUIC

- Transport layer
- Built on UDP, not TCP
- 0-RTT connection establishment in some cases

SQUIC 0-RTT

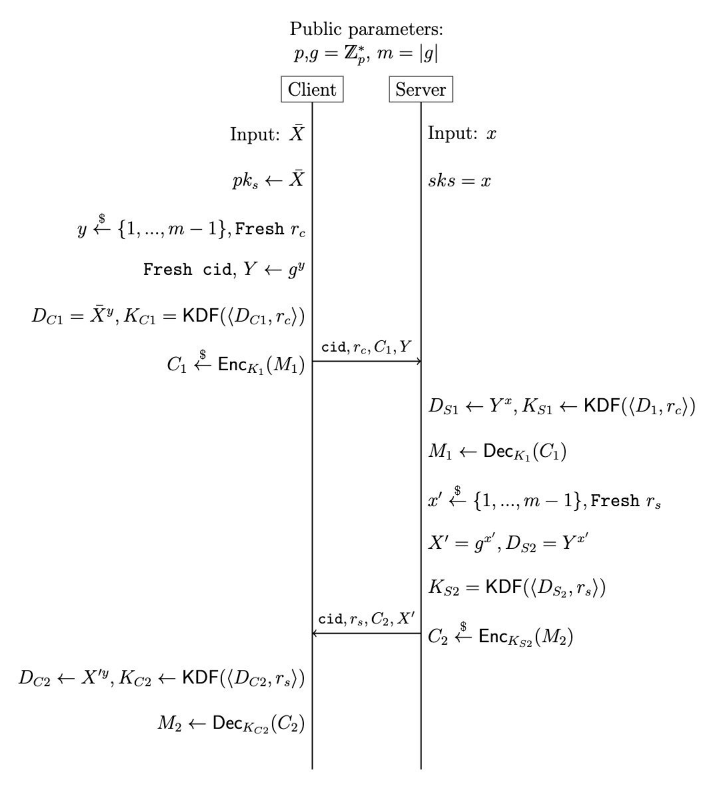
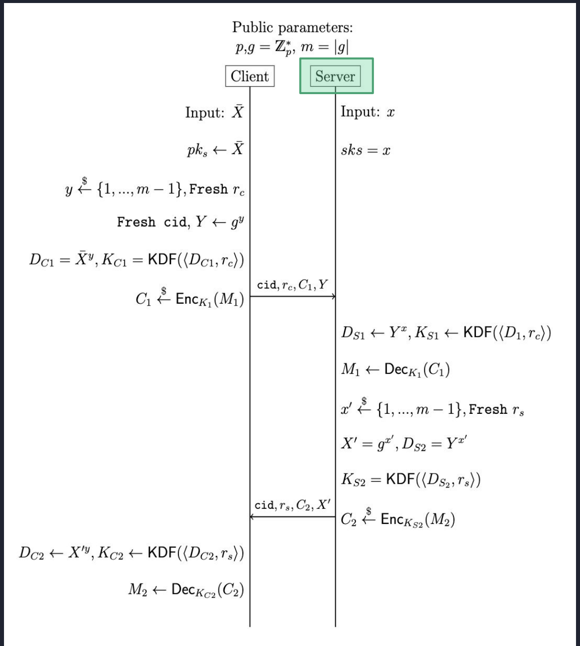


Figure 4.1: SQUIC 0-RTT connection resumption protocol. Here, KDF is a key derivation function. Note that $\bar{X} = g^x$. Fresh x denotes that x is being chosen uniformly from some domain. In reality, this domain is given by the technical specification of the protocol, and we expect secret values to be sufficiently long so as to be secure. In this model the values are assumed to be hard to guess, but not represented in a manner specific to their length.

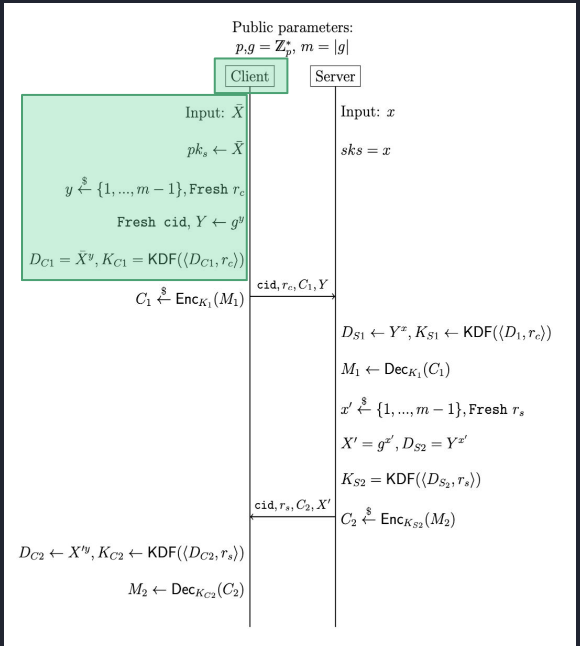
QUIC Model: Rules

```
rule CreateServer:
    [ Fr(~x) ]
  --[ CreateServer() ]->
    [ Server($S,~x) ]
```



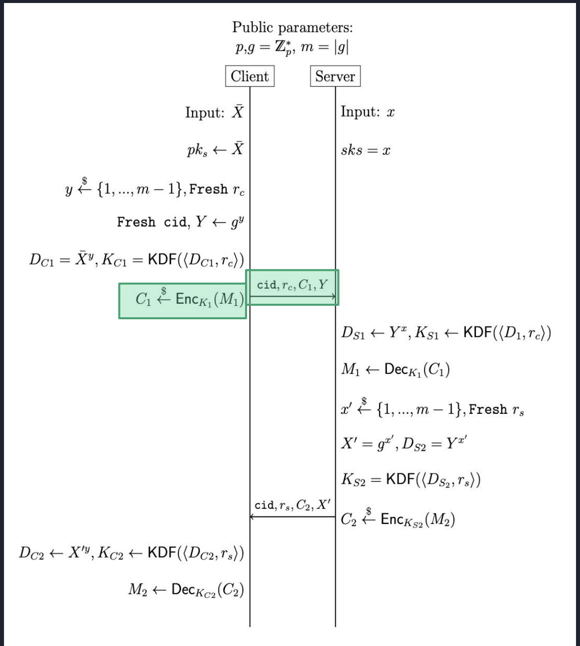
QUIC Model: Rules

```
rule CreateClient:
    let
        X = 'g'^~x
        Y = 'g'^~y
        D1 = X^~y
        K1c = kdf(<D1,~rc>)
    in
    [ Fr(~y),Fr(~cid),Fr(~rc),Server($S,~x) ]
    --[ CreateClient() ]->
    [ Server($S,~x),Client($C,~y,Y,X,K1c,~rc,~cid), Out(Y),Out(X) ]
```



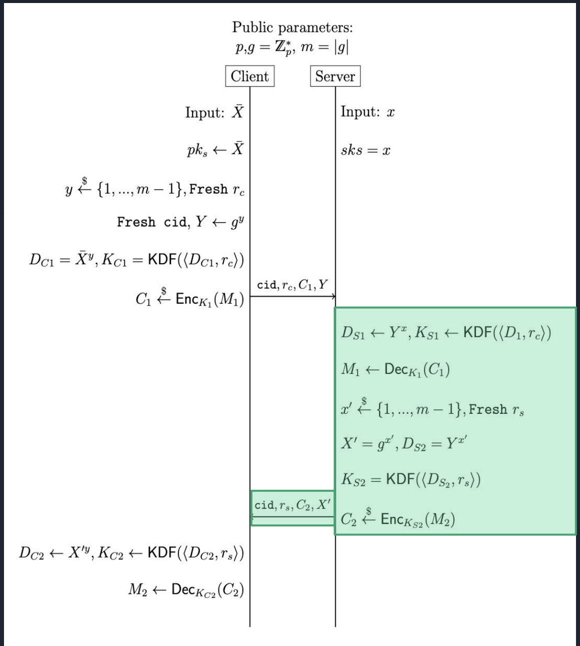
QUIC Model: Rules

```
rule ClientHello:
    let
        c1 = senc(K1c,<'message'>)
    in
    [ Client($C,~y,Y,X,K1c,~rc,~cid) ]
    --[ ClientHello() ]->
    [ Out(<~cid,~rc,c1,Y >) ]
```

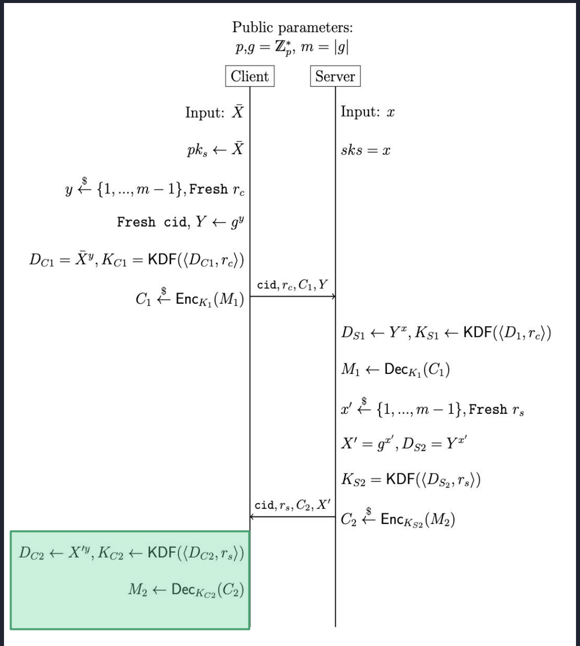


QUIC Model: Rules

```
rule ServerRespond:
    let
        D1 = Y^~x
        K1s = kdf(<D1,~rc>)
        Xprime = 'g'^~xprime
        D2 = Y^~xprime
        K2s = kdf(<D2,~rs>)
        c2 = senc(K2s,<'message2'>)
    in
    [ In(<~cid,~rc,c1,Y>),Fr(~xprime),Fr(~rs),Server($S,~x) ]
    --[ServResp(K2s)]->
    [!Server_2($S,~x,Xprime,Y,K2s,~rs,~cid) , Out(<~cid, ~rs, c2, Xprime>)]
```



QUIC Model: Rules




```
rule ClientReceiveResponse:
    let
        D2=Xprime^~y
        K2c=kdf(D2,~rs)
    in
    [Client($C,~y,Y,X,K1,~rc,~cid),In(<~cid, ~rs, c2, Xprime>)]
    --[ClientReceive(K2c)]->
    [!Client_2($C,~y,Y,X,K2c,~rc,~cid)]
```

QUIC Model: Rules

```
rule ClientSendData:
    let
        clientmessage = 'clientmessage'
        ciphertext = aenc(K2c,clientmessage)
    in
    [!Client_2($C,~y,Y,X,K2c,~rc,~cid)]
    --[CSD(ciphertext,clientmessage)]->
    [Out(ciphertext)]

rule ServerSendData:
    let
        servermessage = 'servermessage'
        ciphertext = aenc(K2s,message)
    in
    [!Server_2($S,~x,Xprime,Y,K2s,~rs,~cid)]
    --[SSD(ciphertext,servermessage)]->
    [Out(ciphertext)]
```

```
rule ClientRecData:
    let
        plaintext_c = adec(K2c,ciphertext_c)
    in
    [!Client_2($C,~y,Y,X,K2c,~rc,~cid),In(ciphertext_c)]
    --[CRD(plaintext_c)]->
    []

rule ServerRecData:
    let
        plaintext_s = adec(K2s,ciphertext_s)
    in
    [!Server_2($S,~x,Xprime,Y,K2s,~rs,~cid),In(ciphertext_s)]
    --[SRD(plaintext_s)]->
    []
```

QUIC Model: Lemmas

```
lemma message_correctness:
    "All #i #j c p0 p1.
        CSD(c,p0) @ #i & SRD(p1) @ #j
        ==>
        p0 = p1"

lemma shared_key:
    "All K2c K2s #i #j.
        ServResp(K2s) @ #i & ClientReceive(K2c) @ #j
        ==>
        K2c = K2s & #i < #j
    "
```

QUIC Model: Lemmas

```
lemma secrecy:
"
  All K2c K2s #i #j.
  (
    ServResp(K2s) @ #i &
    ClientReceive(K2c) @ #j &
    #i < #j
  )
  ==> not(Ex #i #j . K(K2s) @ #i & K(K2c) @ #j)
"
```

Takeaways

- Exploration of key exchange and formal verification of cryptographic protocols
- Introduction of Tamarin Prover
- Minimal example of protocol specification in Tamarin
- Cryptography concepts to build up to QUIC analysis
- Model Diffie-Hellman with Tamarin Prover
- Modeled SQUIC, a simplified version of QUIC
- Proved in Tamarin that SQUIC has desirable liveness and security properties
- Demonstrated the practical utility of Tamarin for key exchange