# (In-)secure messaging with SCIMP and OMEMO

Sebastian Verschoor[1]

University of Waterloo / Eindhoven University of Technology

ShmooCon 2017
January 15th



[1] joint work with Tanja Lange

# Outline

# Secure Messaging protocols


TO THE TECHNOLOGY COMMUNITY:

## Your threat model just changed.

**Incoming President Donald Trump made campaign promises that, if carried out, threaten the free web and the rights of millions of people.** He has praised attempts to undermine digital security, supported mass surveillance, and threatened net neutrality. He promised to identify and deport millions of your friends and neighbors, track people based on their religious beliefs, and suppress freedom of the press.

And he wants to use your servers to do it.

EFF ad in Wired magazine (source)

# History of online secure messaging (1/2)

- 1991: Phil Zimmermann creates PGP
- 2004: Nikita Borisov, Ian Goldberg and Eric Brewer create OTR
  - Secure, but requires synchronous environment
- 2011: Gary Belvin introduces SecureSMS (master's thesis)
- 2012: SCIMP (Silent Circle instant messaging protocol)
  - By Vinnie Moscaritolo, Gary Belvin and Phil Zimmermann
  - SecureSMS for XMPP
  - Even copies variable names and equation numbering from Belvin's thesis (despite creating internal inconsistencies)
- February 2014: Open Whisper Systems releases TextSecure v2
  - Allows offline initial user message
  - Later renamed to Signal

# History of online secure messaging (2/2)

UNIVERSITY OF WATERLOO

- May 2014: SC updates to SCIMP v2
  - Allows offline initial user message
- August 2015: SC releases code for SCIMP v2
  - Adds more inconsistencies between code and documentation
- September 2015: SC discontinues SCIMP, switches to Signal based protocol
- October 2015: Andreas Straub proposes OMEMO
  - Multi-device Signal for XMPP
- Oct-Nov 2016: Trevor Perrin and Moxie Marlinspike release official specification for the Signal protocol
- Dec 7th 2016: OMEMO gets standardized by the XMPP Standard Foundation: XEP-0384 (experimental)

# My involvement

- December 2015: My Master's thesis (at TU/e) on SCIMP
  - SCIMP v1 is formally verified by ProVerif to be secure
  - SCIMP v2 contains cryptographic flaws
  - the implementation contains many security bugs
- June 2016: My cryptographic report on OMEMO
  - Minor bug found in multidevice setting
    - Developer patches it the same day as reported
- July 2016: Tanja Lange and I release SCIMP preprint paper
  - Some of Silent Circle's code (copied from SCIMP implementation) still contains bugs that were reported in my thesis
    - Bugs got patched a few days later
    - Initial bug report: **September 2015**

# Formal verification

UNIVERSITY OF WATERLOO

- Create a mathematical model of a program
- Ask the computer to prove several theorems about the program. For example:
  - correctness
  - confidentiality of data/keys
  - authenticity of messages

# Formal verification

A successful formal verification is *not* a guarantee that the implementation is secure:

- The model might not accurately describe the implementation
  - Model often needs to be a simplification of the code
  - Code evolves
- The attacker model might not be strong enough
  - For example, the attacker might implement a side-channel attack
- The context in which the model is deployed might break model assumptions

# Formal verification

So why do we still go through the trouble?

- Unsuccesful formal verification reveals bugs!
- Building the model itself exposes many bugs and omissions
- The model exposes assumptions that would remain implicit in specs/code
- Successful verification adds trust in protocol by eliminating possibility of a large class of vulnerabilities

# ProVerif

- Formal language for modelling interactive protocols
  - Requires a model for attacker capabilities
    - ProVerif uses Dolev-Yao: attacker has full control over the network
- Limited computing power: have to break up the protocol in pieces
  - We have to manually prove that their composition is still secure
- No memory model
  - But there are some tricks to model key erasure/future secrecy
  - Requires manual "proof" that sequential composition is still secure

# ProVerif

- Formal language for modelling interactive protocols
  - Requires a model for attacker capabilities
    - ProVerif uses Dolev-Yao: attacker has full control over the network
- Limited computing power: have to break up the protocol in pieces
  - We have to manually prove that their composition is still secure
- No memory model
  - But there are some tricks to model key erasure/future secrecy
  - Requires manual "proof" that sequential composition is still secure

# ProVerif

- ProVerif has limited available primitives
- We can define the required primitives using standard "tricks"
- For example, to create an authenticated channel:
  - Create a private channel (confidential and authentic)
  - Create a process that runs in parallel with the main process. Publish everything that is communicated on this channel in a public channel
- Pro: we can model the protocol very accurately
- Con: requires expertise to implement the correct primitive
- Con: the added complexity makes it very computationally intensive to prove security

# ProVerif example

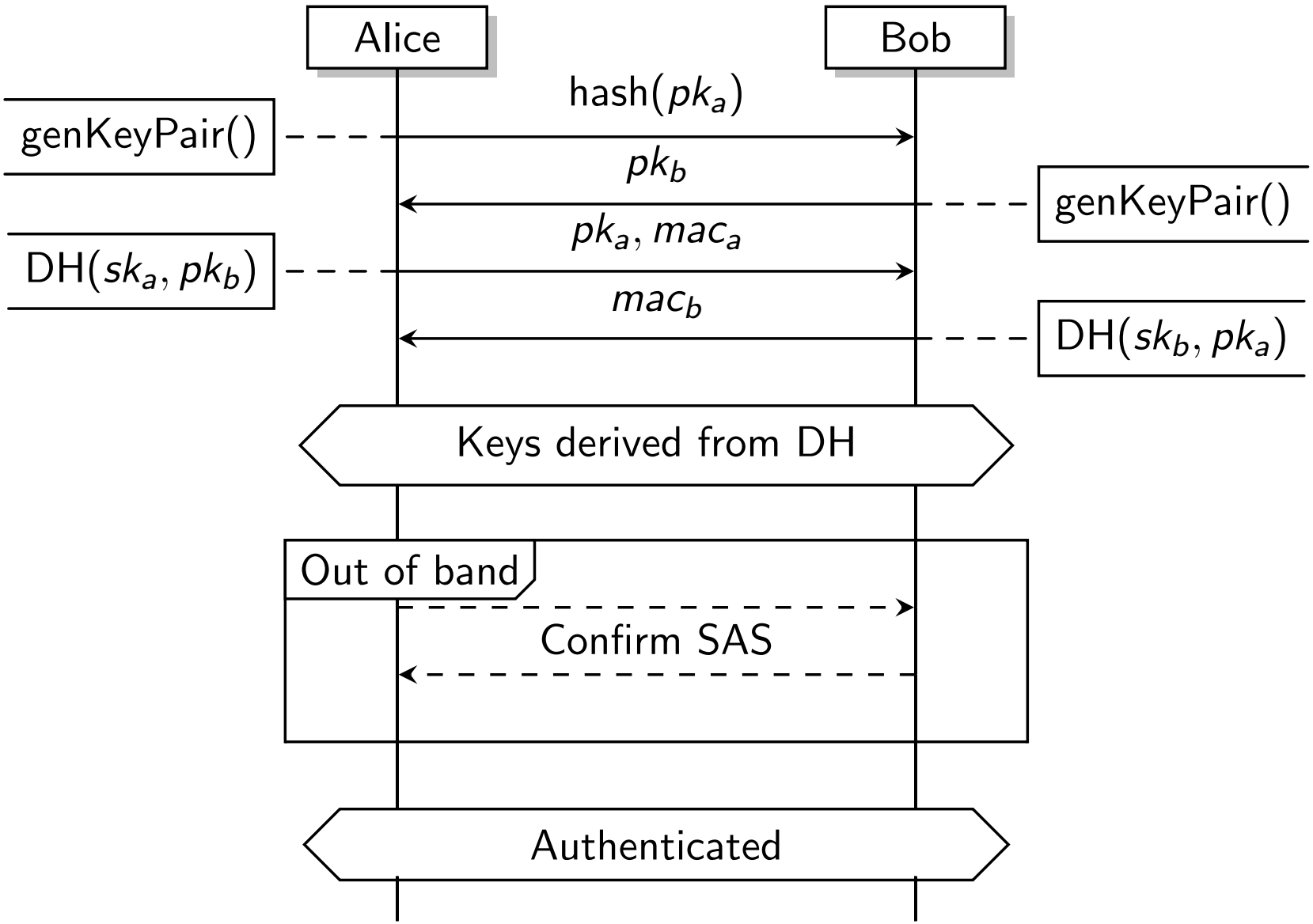
```
(*** Main ***)

process
  (* Allow arbitrary many protocol runs *)
  !
  (* Let the adversary decide who will engage in key negotation *)
  in(ch, (init:identity, resp:identity));
  (* Create a new phone channel *)
  new phone : channel;
  (* Allow eavesdropping on the phone channel *)
  (! in(phone, x:bitstring); out(ch, x)) |

  if init = Compromised then (
    out(ch, phone);
    processResponder(init, resp, phone)
  ) else if resp = Compromised then (
    out(ch, phone);
    processInitiator(init, resp, phone)
  ) else (
    processInitiator(init, resp, phone) |
    processResponder(init, resp, phone)
  )
```

# ProVerif for SCIMP

Security properties we expect of SCIMP and how to prove them with ProVerif:

- **confidentiality**: can an adversary can learn private keys/messages?
- **authenticity/integrity**: can an adversary trigger the "user-accepts" event?
- **forward secrecy**: leak an updated key, then check confidentiality of old messages/keys
- **future secrecy**: publish all key material, then run a key negotiation over a private channel and check confidentiality
- **deniability**: run the protocol with the honest user and with the adversary, then check if the transcripts are indistinguishable

# SCIMP v1: Key negotiation

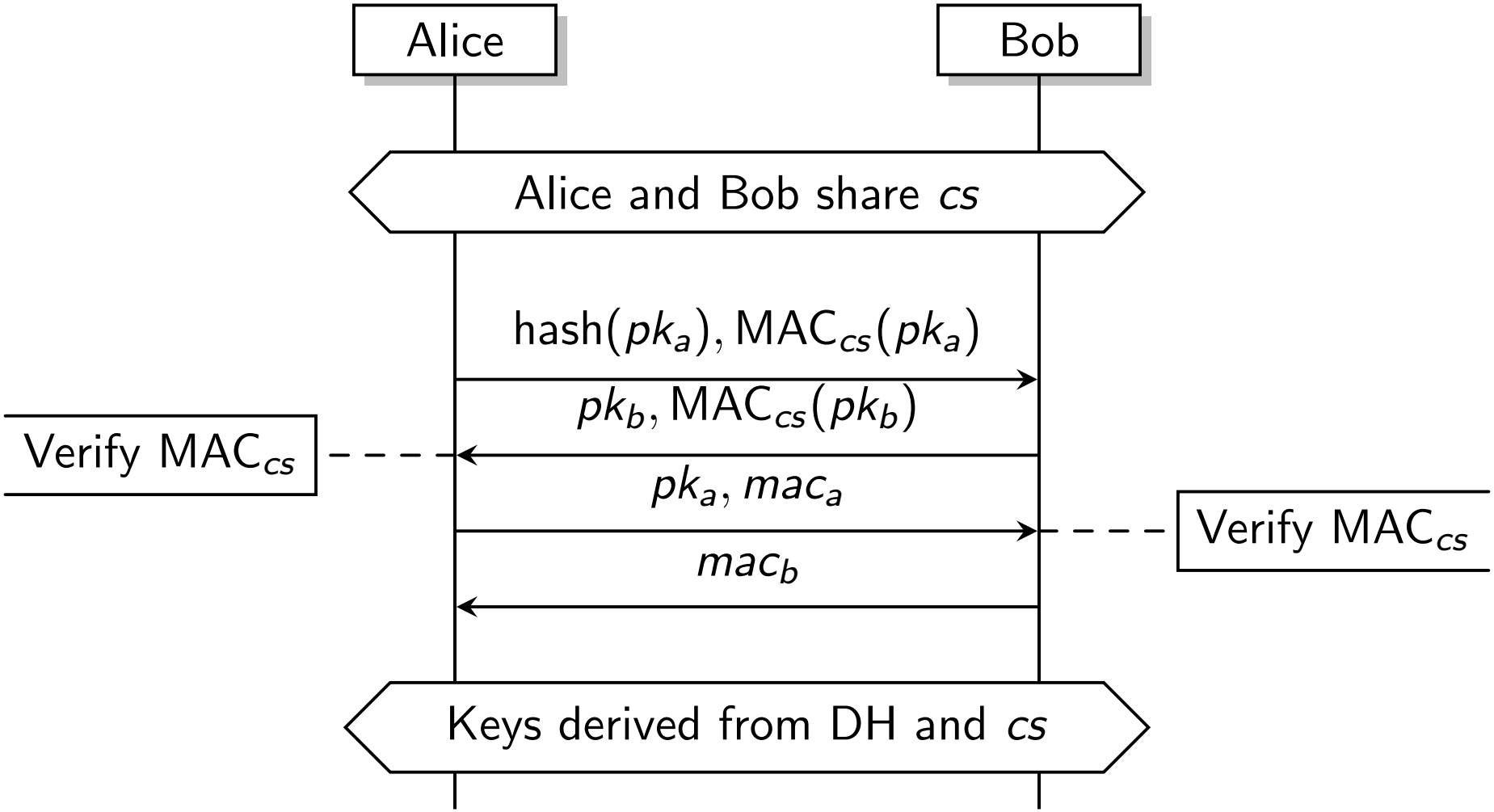



SAS: short authentication string

# SCIMP v1: Key negotiation

- ECDHE gives shared secret $Z$, from which are derived:
  - $k_{snd,0}$, $k_{rcv,0}$, $i_{snd,0}$, $i_{rcv,0}$; for message encryption and authentication
  - $mac_a$, $mac_b$; to confirm knowledge of $Z$
  - SAS; for authentication of identity
  - $cs$; for rekeying
- User messages can be sent after four key exchange messages
- SAS confirms identity all previous communication
  - Requires commitment to $pk_a$ to prevent collision attack

# SCIMP v1: Rekeying



# SCIMP v1: Rekeying

- First: store old decryption key (messages might arrive out of order)
- Optional: SAS comparison only after several rekeyings
- Rekeying ensures *future secrecy*
- It is not specified when to rekey
- Protocol aborts on error

# SCIMP v1: Sending user messages

- Encrypt
  - ciphertext $= \text{AES}_{k_j}(i_j, \text{plaintext})$
- Update keys (ratchet)
  - $k_{j+1} = \text{MAC}_{k_j}(i_j)$
  - $i_{j+1} = i_j + 1$
- Send message:
  - $i_j$
  - ciphertext

- No message signatures: deniable
- Ratchet enables key erasure, but:
  - Out of order messages require you to store old keys
  - Old keys compromise future keys

# SCIMP v1: Sending user messages

- Encrypt
  - ciphertext $= \text{AES}_{k_j}(i_j, \text{plaintext})$
- Update keys (ratchet)
  - $k_{j+1} = \text{MAC}_{k_j}(i_j)$
  - $i_{j+1} = i_j + 1$
- Send message:
  - $i_j$
  - ciphertext
- No message signatures: deniable
- Ratchet enables key erasure, but:
  - Out of order messages require you to store old keys
  - Old keys compromise future keys

# SCIMP v1: Sending user messages

- Encrypt
  - ciphertext $= \text{AES}_{k_j}(i_j, \text{plaintext})$
- Update keys (ratchet)
  - $k_{j+1} = \text{MAC}_{k_j}(i_j)$
  - $i_{j+1} = i_j + 1$
- Send message:
  - $i_j$
  - ciphertext

- No message signatures: deniable
- Ratchet enables key erasure, but:
  - Out of order messages require you to store old keys
  - Old keys compromise future keys

# SCIMP v1: Sending user messages

UNIVERSITY OF WATERLOO

- Encrypt
  - ciphertext $= \mathrm{AES}_{k_j}(i_j, \text{plaintext})$
- Update keys (ratchet)
  - $k_{j+1} = \mathrm{MAC}_{k_j}(i_j)$
  - $i_{j+1} = i_j + 1$
- Send message:
  - $i_j$
  - ciphertext

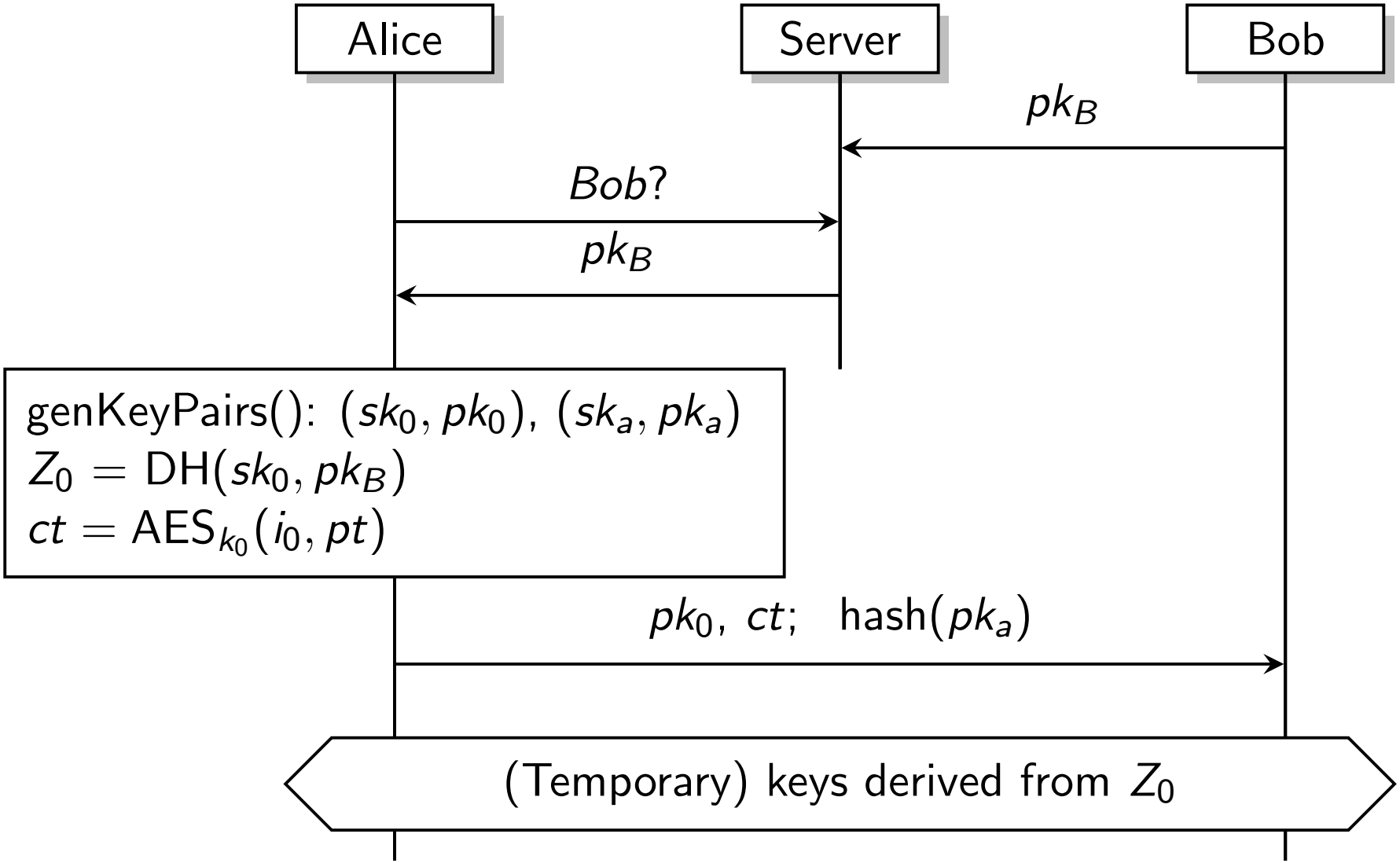
- No message signatures: deniable
- Ratchet enables key erasure, but:
  - Out of order messages require you to store old keys
  - Old keys compromise future keys

# Proverif results for SCIMP v1

UNIVERSITY OF WATERLOO

- First key negotiation (**if** SAS confirmed over authenticated channel)
  - ✓ Confidentiality of keys
  - ✓ Authenticity of keys and other party identity
- Rekeying
  - ✓ Confidentiality of keys
  - ✓ Authenticity of keys and other party identity
  - Future secrecy
    - ✓ When attacker misses first rekeying after compromise
    - ✓ When users reconfirm the SAS
- Sending user message
  - ✓ Confidentiality of keys
  - ✓ Strong secrecy of messages
  - ✓ Authenticity of messages and keys
  - ✓ Forward secrecy (**if** keys can be erased)
  - ✓ Deniability

# SCIMP v2: Progressive encryption



# Proverif results for SCIMP v2

- Progressive encryption
  - ✗ Confidentiality/authenticity of first message
  - ✓ Confidentiality/authenticity of all messages and keys (**after** SAS)

# SCIMP

- ProVerif reports that the initial message of SCIMP v2 is not confidential
  - This does not impact SCIMP v1
- But, ProVerif also verifies that users can detect this when confirming the SAS later
- To determine the impact of this vulnerability, we had to look at the source code


MY EYES
quickmeme.com

A short example to give a flavor of the code

```
unsigned long ctxStrLen = 0;
size_t kdkLen;
int keyLen = scSCimpCipherBits(ctx->cipherSuite);
```

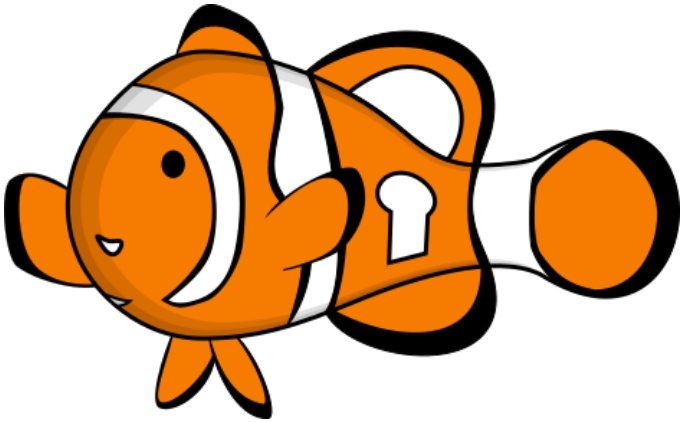
- **ctxStrLen** length in bytes (computing function returns size_t)
- **kdkLen** length in bytes
- **keyLen** function name suggests bit-length, but $k_{snd}$ is $2 * \textbf{keyLen}$ bits long

- Each message has a plaintext tag identifying the type:
  - keying message; or
  - user message
- The adversary can block the key negotiation using just this tag, thereby having set up a succesful MitM
- The vigilant user might detect this
- But the code contains another bug: the receiver overreacts when a keying message is received out of order
  - only the message tag is inspected
  - the receiver deletes all local key material
  - thereby annulling any security set up in the past
- The adversary can desynchronize any secure session with a single out-of-order key message and set up a MitM undetected

- Each message has a plaintext tag identifying the type:
  - keying message; or
  - user message
- The adversary can block the key negotiation using just this tag, thereby having set up a succesful MitM
- The vigilant user might detect this
- But the code contains another bug: the receiver overreacts when a keying message is received out of order
  - only the message tag is inspected
  - the receiver deletes all local key material
  - thereby annulling any security set up in the past
- The adversary can desynchronize any secure session with a single out-of-order key message and set up a MitM undetected

- Each message has a plaintext tag identifying the type:
  - keying message; or
  - user message
- The adversary can block the key negotiation using just this tag, thereby having set up a succesful MitM
- The vigilant user might detect this
- But the code contains another bug: the receiver overreacts when a keying message is received out of order
  - only the message tag is inspected
  - the receiver deletes all local key material
  - thereby annulling any security set up in the past
- The adversary can desynchronize any secure session with a single out-of-order key message and set up a MitM undetected

# OMEMO

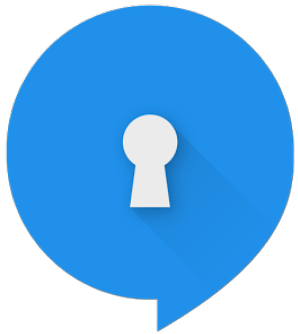


https://conversations.im/omemo/


https://radicallyopensecurity.com/

https://pacificresearchalliance.com/

# Multi-device Signal



https://whispersystems.org/

Shares the private key between multiple devices[2]

1. Generate ephemeral Signal key-pair on desktop
2. Scan public key with phone (using QR-code)
3. Set up Signal session between phone and desktop
4. Send phone's private-key to desktop
5. Replace desktop ephemeral key with the received private key

[2] Limited to one smartphone and a desktop app

# Multi-device OMEMO (flawed)

UNIVERSITY OF WATERLOO

- Each device has its own private key
- Each pair of devices sets up a Signal session
- A user message gets authenticated-encrypted with a random key
- The ciphertext and tag are sent to each receiving device
- The random key is sent to each user inside the Signal session

# Multi-device OMEMO (flawed)

UNIVERSITY OF WATERLOO

One malicious device breaks authenticity of all messages in the conversation

- Assume Eve convinces Alice that her device belongs to Bob
- Eve could intercept any message by Alice...
- ...get the random key that Alice sent her...
- ...encrypt her own message using the same key...
- ...and send it to Bob, who thinks it came from Alice.

# Multi-device OMEMO (fixed)

UNIVERSITY OF WATERLOO

- Each device has its own private key
- Each pair of devices sets up a Signal session
- A user message gets authenticated-encrypted with a random key
- The ciphertext and tag are sent to each receiving device
- The random key **and tag** are sent to each user inside the Signal session

# Conclusions

- Implementing crypto protocols is hard
  - Even experienced cryptographers get it wrong sometimes
- Despite all the shortcomings of ProVerif, the tool did help me analyze the protocol and expose flaws
  - I would love to see more powerful and easier to use tools
  - if possible: a tool that extracts the model from the code
- For those who just want painfree secure messaging: use Signal
  - with OMEMO, that no longer means that you are tied to the Signal *application* and *server*
    - set up your own compatible XMPP server (or create an account at an existing one)

# Conclusions

- Implementing crypto protocols is hard
  - Even experienced cryptographers get it wrong sometimes
- Despite all the shortcomings of ProVerif, the tool did help me analyze the protocol and expose flaws
  - I would love to see more powerful and easier to use tools
  - if possible: a tool that extracts the model from the code
- For those who just want painfree secure messaging: use Signal
  - with OMEMO, that no longer means that you are tied to the Signal *application* and *server*
    - set up your own compatible XMPP server (or create an account at an existing one)

# Conclusions

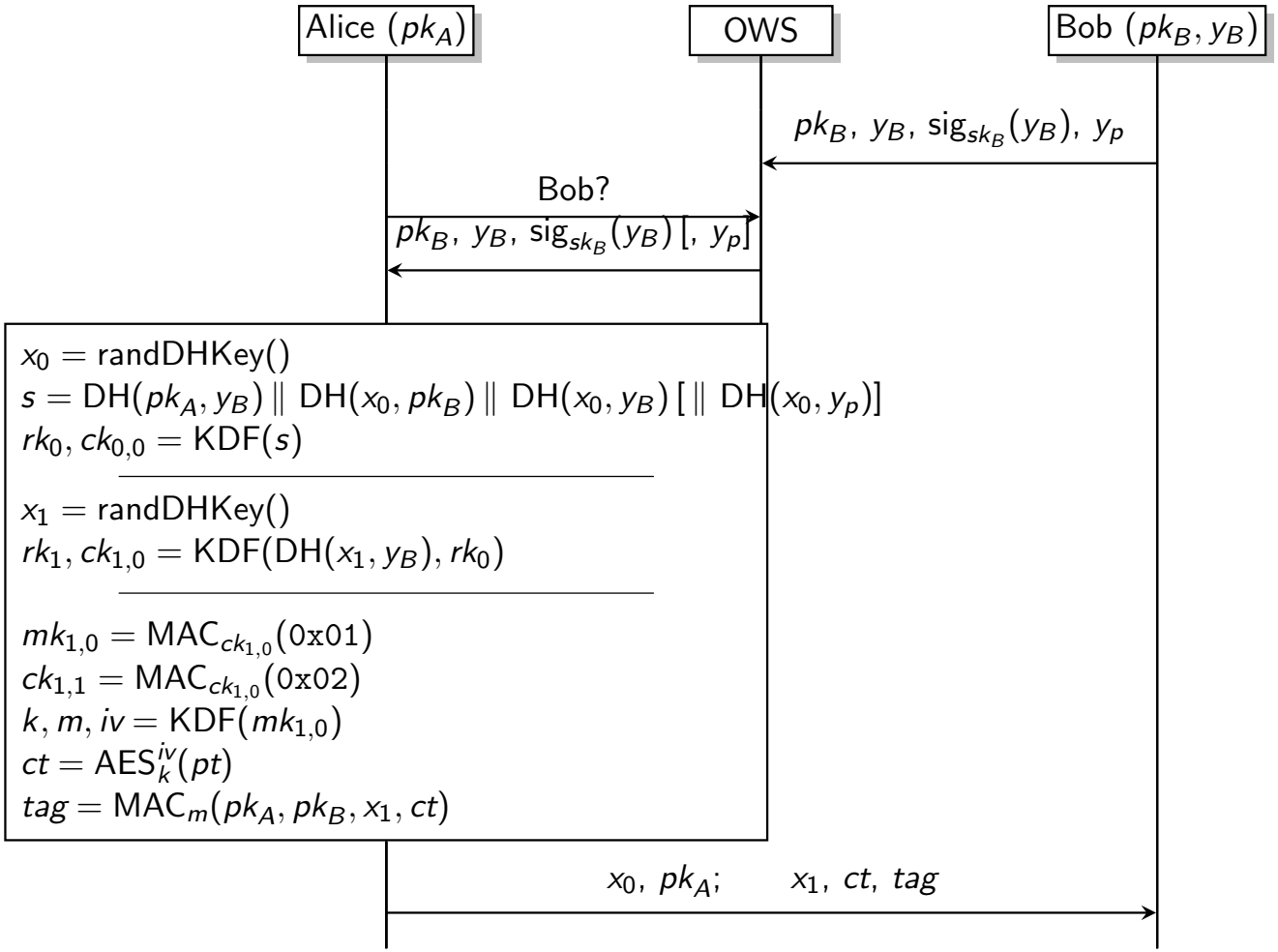
- Implementing crypto protocols is hard
  - Even experienced cryptographers get it wrong sometimes
- Despite all the shortcomings of ProVerif, the tool did help me analyze the protocol and expose flaws
  - I would love to see more powerful and easier to use tools
  - if possible: a tool that extracts the model from the code
- For those who just want painfree secure messaging: use Signal
  - with OMEMO, that no longer means that you are tied to the Signal *application* and *server*
    - set up your own compatible XMPP server (or create an account at an existing one)

# Thank you

# Further reading

- These slides will be available on my website:
  https://www.zeroknowledge.me/
- ProVerif models are available:
  https://github.com/sebastianv89/scimp-proverif
- My thesis about SCIMP:
  http://repository.tue.nl/844313
- Preprint about SCIMP (with Tanja Lange):
  https://eprint.iacr.org/2016/703
- Get Signal (Android/iPhone):
  https://whispersystems.org/
- Try OMEMO (on Android):
  https://conversations.im/
- OMEMO audit report:
  https://conversations.im/omemo/audit.pdf

# Signal


Alice ($pk_A$)
OWS
Bob ($pk_B, y_B$)
$pk_B,\ y_B,\ \mathrm{sig}_{sk_B}(y_B),\ y_p$
Bob?
$pk_B,\ y_B,\ \mathrm{sig}_{sk_B}(y_B)\ [,\ y_p]$
$x_0 = \mathrm{randDHKey}()$
$s = \mathrm{DH}(pk_A, y_B) \parallel \mathrm{DH}(x_0, pk_B) \parallel \mathrm{DH}(x_0, y_B)\ [\parallel \mathrm{DH}(x_0, y_p)]$
$rk_0, ck_{0,0} = \mathrm{KDF}(s)$
$x_1 = \mathrm{randDHKey}()$
$rk_1, ck_{1,0} = \mathrm{KDF}(\mathrm{DH}(x_1, y_B), rk_0)$
$mk_{1,0} = \mathrm{MAC}_{ck_{1,0}}(\mathtt{0x01})$
$ck_{1,1} = \mathrm{MAC}_{ck_{1,0}}(\mathtt{0x02})$
$k, m, iv = \mathrm{KDF}(mk_{1,0})$
$ct = \mathrm{AES}^{iv}_{k}(pt)$
$tag = \mathrm{MAC}_{m}(pk_A, pk_B, x_1, ct)$
$x_0,\ pk_A;$
$x_1,\ ct,\ tag$

# More on SCIMP

Other discrepancies between the model and the implementation

- Group messages have a single symmetric key
  - Relies on trust in the SC server
  - Subject to a trivial MitM attack
- CCM-mode implementation did not validate authentication tags
  - Problem in LibTomCrypt (fixed)
- Code contains many timing side-channel vulnerabilities
- The message parsing queue has a race condition
- Unchecked function error codes
  - Including memory allocations
- State machine based design: good coding style
  - and helps in making a model of the code
  - in case of SCIMP: helps find where specs and code differ

# More on SCIMP file transfer

- Convergent encryption
  - key = hash(file)
    - send as SCIMP message
  - ciphertext = $\text{AES\_CCM}_{key}$(file)
    - upload to cloud
- Known vulnerabilities of CE:
  - confirmation of a file
  - learn the remaining information
- SC: receiver does not check hash(file) = key
  - file injection attack
  - This attack remained in the code until July, when we looked at the updated code again

# More on SCIMP file transfer

- Convergent encryption
  - key = hash(file)
    - send as SCIMP message
  - ciphertext = $\text{AES\_CCM}_{key}$(file)
    - upload to cloud
- Known vulnerabilities of CE:
  - confirmation of a file
  - learn the remaining information
- SC: receiver does not check hash(file) = key
  - file injection attack
  - This attack remained in the code until July, when we looked at the updated code again

# More on SCIMP file transfer

UNIVERSITY OF WATERLOO

- Convergent encryption
  - key = hash(file)
    - send as SCIMP message
  - ciphertext = $\text{AES\_CCM}_{key}$(file)
    - upload to cloud
- Known vulnerabilities of CE:
  - confirmation of a file
  - learn the remaining information
- SC: receiver does not check hash(file) = key
  - file injection attack
  - This attack remained in the code until July, when we looked at the updated code again