Pacific Northwest
NATIONAL LABORATORY

PNNL-33541

DRA Adaptation

Double-Ratchet Algorithm Modifications for Authentication

September 2022

PNNL
S. Cullen Tollbom
Jacob M. Benz
Bill Nickless
Marie V. Whyatt

OSU
Camille J. Palmer
Mike Rosulek
Jaspal Singh
Gayathri Garimella

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights**. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from
the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov

PNNL-33541

DRA Adaptation

Double-Ratchet Algorithm Modifications for Authentication

September 2022

PNNL
S. Cullen Tollbom
Jacob M. Benz
Bill Nickless
Marie V. Whyatt

OSU
Camille J. Palmer
Mike Rosulek
Jaspal Singh
Gayathri Garimella

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Acronyms and Abbreviations

AES	Advanced Encryption Standard
CSIDH	Commutative Supersingular Isogeny Diffie-Hellman
DHKEX	Diffie Hellman Key Exchange
DRA	Double Ratchet Algorithm
ECC	Elliptic-Curve Cryptography
ECDH	Elliptic Curve Diffie Helman
EK	Ephemeral Root Key
ESK	ECDH Salting Key
HKDF	Host Key Derivation Function
HMAC	Hashed Message Authentication Code
IK	Long-Term Identity Key (per party)
IV	Initialization Vector (used for AES)
KDF	Key Derivation Function
KEY	secret encryption key (not an acronym)
OPK	One-Time Pre-Key
SK	Secret Key (a.k.a. root Secret Key)
SPK	Signing Pre-Key
SHA-512	Secure Hash Algorithm 512 (a.k.a. Keckak, SHA-3)
SIDH	Supersingular Isogeny-based Diffie-Hellman
X3DH	Extended Triple Diffie Hellman Key Exchange

Acknowledgments

This work is supported by the U.S. Department of Energy, National Nuclear Security Administration, Office of Defense Nuclear Nonproliferation Research and Development.

Contents

Acronyms and Abbreviations ii
Acknowledgments iii
1.0 Introduction 1
2.0 Adapted Double-Ratchet Algorithm Overview 2
3.0 Adapted Double-Ratchet Algorithm Detail 4
3.1 HostPi 4
3.2 InspectorPi 4
3.3 Host Review 4
3.4 Inspector Review 5
3.5 Signed Video Frame Format 5
3.6 Signing Process 5
4.0 Camera-HostPi-InspectorPi Message Exchange and Image Signing 6
4.1 Operations 6
4.1.1 Initialization Mode 6
4.1.2 Start Mode 6
4.1.3 Generation Mode 7
5.0 Implementation 9
6.0 References 10
Appendix A – Key Generation Performance A.1
Appendix B – Resistance to Cracking Keys by a Quantum Computer B.1
Appendix C – Original Double Ratchet Algorithm C.1
Appendix D – Start Mode Details D.1
Appendix E – SYMMRATCHET Detail E.1

Figures

Figure 1: Single-Camera Setup 3
Figure 2: Initialization step of the Ratchet [7] uses traditional Diffie-Hellman (DH) key exchange C.1
Figure 3: Process for a Single Message Exchange [7] C.2
Figure 4: Process for Receipt of Message and Generation of New Keys for Subsequent Message [7] C.3

Tables

Table 1: Double-Ratchet Algorithm Functions 4
Table 2: Key Message Exchange Functions 6

Table 3: Modes of Operation A.1
Table 4: Performance Evaluation of Quantum Cryptography for Signal Protocol A.1

1.0 Introduction

The goal of this project is to demonstrate the capability to perform real-time authentication of video images. This is a capability which does not currently exist in the arms control toolbox, and one that can provide significant benefits to future monitoring and verification regimes. There are different ways to tackle this problem, each with their own benefits and challenges. This project is using a concept sometimes termed “self-healing encryption”, meaning this approach can recover from compromised encryption keys. The Double Ratchet Algorithm (DRA) is the encryption protocol making this possible. The DRA is utilized in several secure messaging applications including Facebook Messenger, WhatsApp, and Signal the mobile devices app for encrypted messaging.

DRA is what is called forward secret and backward secure. Forward secrecy means that compromising one encryption key will only allow a single encrypted message/image to be decrypted and viewed, but not messages/images that follow since each ensuing message/image is encrypted with a different encryption key. Backward security means someone with an encryption key cannot go back and decrypt and view - or modify - previous messages/images.

This report documents the approach taken to modify the existing DRA implementation as used in Signal to meet future arms control needs. Whereas Signal’s DRA implementation is focused on encryption for secure confidential communications, DRA for arms control instead stamps each video frame with a unique uncounterfeitable key for the purpose of authentication. The modified DRA doesn’t encrypt video frames and so each image can be reviewed by the host and if needed redacted. For future arms control needs, this adapted DRA with focus on authenticity and integrity enables heightened confidence.

2.0 Adapted Double-Ratchet Algorithm Overview

This project begins with the Signal Protocol, which uses the Double-Ratchet Algorithm (DRA), because the DRA is open source and available to use and adapt. At a high level the Signal Protocol implements DRA in the following way: Two people, traditionally named Alice and Bob, who wish to communicate securely each create (manually) their own public-private encryption key pairs. Bob initiates the conversation by sending his public key to Alice. Alice uses her private key and Bob's public key to create a unique secret key then uses it to encrypt a message which she sends to Bob along with her public key. The nature of the initial key exchange process lets Bob use Alice's public key with his private key to recreate Alice's secret key and decrypt Alice's message. This process where a new secret key is created and used to encrypt is repeated for every message exchange; the public-private key pairs create a new secret key to encrypt each new message.

With real-time video authentication, there is no benefit from encrypting images. Rather the critical function is to ensure the integrity of the image until it reaches the inspector for review. This project is researching solutions to demonstrate feasibility for both a single- and multi-camera system. The first step in this research is to modify the DRA from its current focus on encryption to that of authentication, of image integrity.

The project team has created a construct (Figure 1) made up of equipment and players. Equipment is *HostPi*, *InspectorPi*, *Camera*, *HostSD*, *InspectorSD*. *HostReview*, and *InspectorReview*. The players are people performing specific job roles who need to interact with the process to ensure image integrity from creation to final inspector review, the *Host* and the *Inspector*. The setup of equipment and players for the current focus single-camera system are shown in Figure 1. The purple box represents a joint enclosure where neither *Host* nor *Inspector* has sole access. The blue boxes represent areas where the *Inspector* has sole control, and the pink areas are where the *Host* has sole control.

An overview of the process highlighted in Figure 1 is as follows:

1. initialization occurs between *HostPi* and *InspectorPI* inside *Joint Authentication & Certification Boundary* to establish a communication pathway and ability to generate the initial secret key (all ensuing secret keys are similarly generated over this communication pathway)

2. an image is acquired by *Camera*, digitally signed using an HMAC generated using a unique secret key and "stamped" into the image

3. the signed (stamped) image is stored on *HostSD* and the secret key used to stamp the image is also stored on *InspectorSD*

4. the "ratchet" then turns and the next secret key for HMAC is generated to be used for the next image acquired

Steps 2 - 4 repeat until physical access is required to retrieve the SD cards whereupon the *Host* and *Inspector* take their respective SD cards. The *Host* reviews each signed image, redacting any as necessary, and upon completing the review the *Host* provides all the remaining

unredacted images to the *Inspector*. The *Inspector* then confirms the integrity of each received image by using the ordered list of secret keys on *InspectorSD* to recompute each image's HMAC and compare it to the HMAC stamped on each image. A more detailed description of each player is provided below.

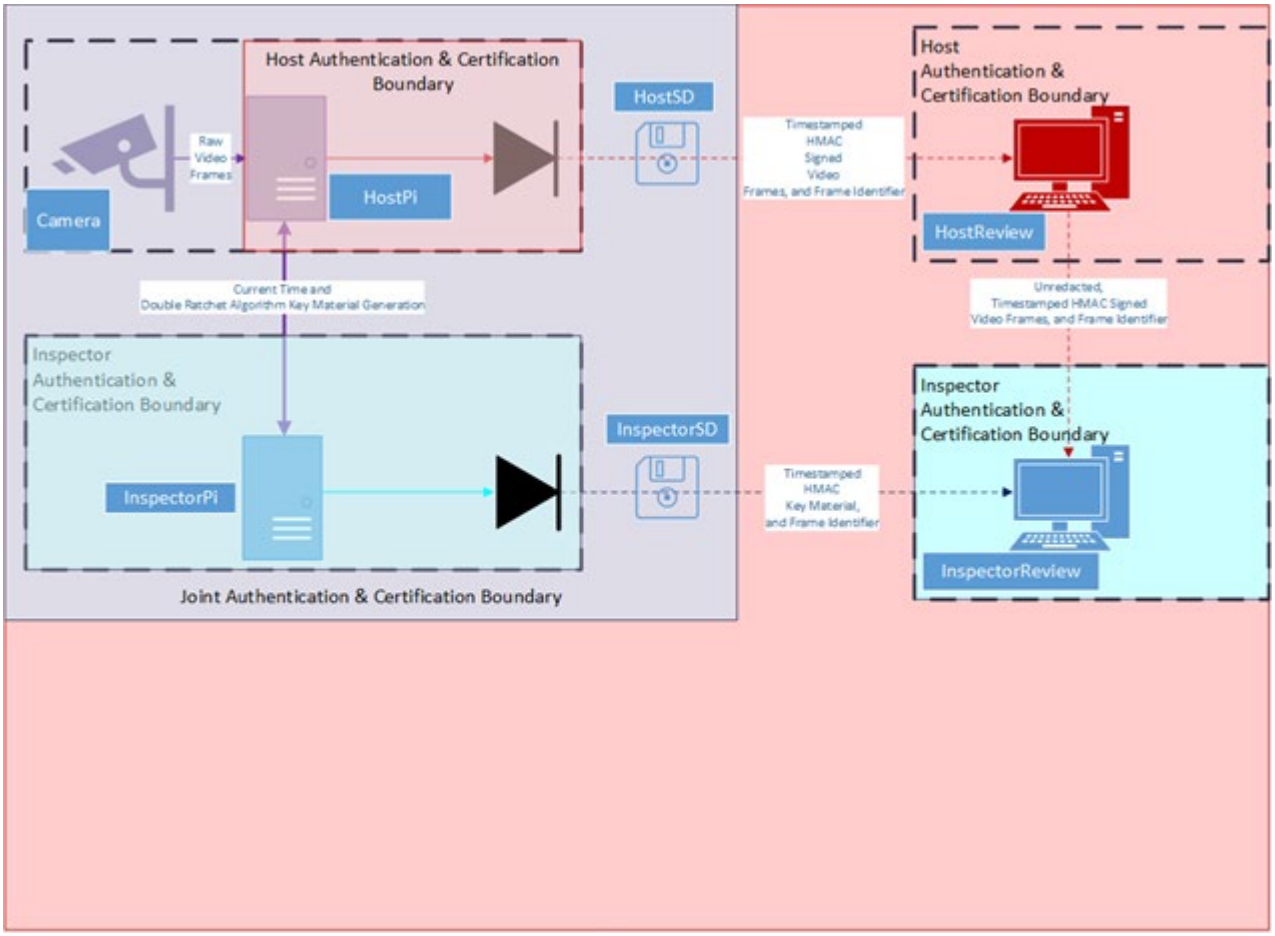


Figure 1: Single-Camera Setup

3.0 Adapted Double-Ratchet Algorithm Detail

Table 1: Double-Ratchet Algorithm Functions

Label	Description
ECDH	ECDH Diffie Hellman Key Elliptic Curve Key Exchange
KEYGEN	ECDH Key Generator
HKDF	Host Key Derivation Function
HMAC	Hash-based Message Authentication Code
SHA-512	NIST Secure Hash Algorithm 3 Also known as SHA-3 and Keccak
SHA-256	NIST Secure Hash Algorithm 3 Also known as SHA-2
SYMMRATCHET	Double Ratchet Symmetric Ratchet Key Chain Iterator
AES-CBC-256	256-bit AES Cipher Block Chain Encryption and Decryption

3.1 HostPi

The *HostPi* receives video frames from the camera, digitally signing each with a hashed message authentication code (HMAC), embedding the result in the frame image and storing each image on removable media. The HMAC is computed using the SHA-512 (aka SHA-3 or Keccak) algorithm along with a 32-octet key (KEY) known to the *HostPi* and the *InspectorPi*, generated by a modified Double Ratchet Algorithm (DRA). The HMAC provides integrity and verifiable authentication of each video frame.

The KEY is updated by ratchet for each video frame. This occurs through two-way communication with *InspectorPi* using DRA. After each video frame is signed, *HostPi* will create a new public-private key pair, and using the previous public key received from *InspectorPi* and *HostPi's* new private, will create KEY. *HostPi* will then send its new public key to *InspectorPi*.

3.2 InspectorPi

InspectorPi receives public key from *HostPi* and uses *InspectorPi* private key to compute KEY. *InspectorPi* then creates and new public-private key pair and sends the new public key to *HostPi*.

3.3 Host Review

The host examines video frames stored on *HostSD*, choosing which if any frames to erase (redact) from the removeable media. The host can see the HMAC stamped into the image but has no knowledge of the KEY that was used to create the HMAC. Once complete, the host passes the unredacted frames to the inspector for review.

A new KEY and HMAC is created via another "turn of the ratchet" every time an image is captured which in this single-camera system will be once every second. Regarding cryptographic resilience to brute-forcing, although no KEY is absolutely secure this scheme is practically secure, for example if a large number of NVIDIA Titan chips deployed in parallel were used to try to brute force a KEY, with the strategy to minimize the search space, it would still

require 2.89×10^{144} years to brute force one single KEY[1]. Much longer than the timeframe available in expected use cases.

3.4 Inspector Review

The inspector authenticates and validates video frames received after redaction. To accomplish this, the Inspector computes a SHA-3 of each the video frame along with the KEY to produce the HMAC, the same as *HostPi*. The resulting HMAC is then compared with the HMAC embedded in the video frame. A match authenticates the frame.

3.5 Signed Video Frame Format

PNG format with the last 88 octets of frame pixels replaced by a watermark as follows:

- 64-bit timestamp (seconds since January 1, 1970)
- 64-bit nanosecond timestamp
- 64-bit frame sequence number
- 512-bits of HMAC. The HMAC is computed over the raw video frame after replacement minus the last 512 bits with the result embedded here. Option: embed the first 256 or 384 bits lowering the number of bytes required for the watermark.

3.6 Signing Process

Signing proceeds as follows, given a frame of video, along with a timestamp, a sequence number, and a KEY:

signaturesize = hmac-sha-512.size + sizeof(timestamp) + sizeof(sequence)
watermark = length(frame.bitmap) – signaturesize

Next, compute the signature hash. The timestamp and sequence number are folded in to detect possible tamper.

hash = hmac(SHA-512,frame.pixmap[:watermark-1],timestamp,sequence,KEY)

Then, replace the last pixels in the video frame with the constructed watermark.

frame.bitmap[watermark:] = pack(sequence,timestamp,hash)

And finally, write the completed video frame to storage in PNG format.

write(png.encode(frame) , “PIC-%016x-%s.png” % (sequence,
strftime(“%Y-%m-%dT%H:%M:%S-%m-%s.%fZ”,timestamp)))

The last line writes the file to storage, for example, with the name “PIC-000000001234ABCD-2023-01-15T23:59:10.999484Z.png” The timestamp here is in ISO format, universal time. 0…01234ABCD is the sequence number, expressed in hexadecimal as it is somewhat more compact than decimal, while maintaining sort ability.

[1] This estimate assumes use of 1024 NVIDIA Titan and a 15% chance of success when brute-forcing

4.0 Camera-HostPi-InspectorPi Message Exchange and Image Signing

4.1 Operations

The following sections explain in greater detail the stages of the entire process (DRA for key update + signing modification) described at a high-level above. Pseudocode is given to help explain the code being developed. The pseudocode is being converted into Python and will be implemented on and demonstrated as part of the single camera system under development.

Table 2: Key Message Exchange Functions

Format	Sender	Use
Status, Count, Text, ESK.pub	Both	Send optional text and mandatory ECDH Ephemeral Salting Key, used by both *HostPI* and *InspectorPi*
Status, Count, Text, EK.pub	*InspectorPi*	Send optional text and mandatory ECDH ephemeral root key, used only during root key negotiation
Status, Count, Text, SPK.pub, OPK.pub	*HostPi*	Send optional text and mandatory ECDH Signing Key and one-time pre-keys, used only during root key negotiation

4.1.1 Initialization Mode

1. Since this system has no trusted root authority this “prerequisite” Initialization Mode step must be done by humans, it cannot be automated (unless *HostPi* and *InspectorPi* could be co-located, meaning the two programs execute on the same computer, which isn’t the case in this scenario). Since this step must be done manually no code development is possible, and thus there is no pseudocode.

HostPi and *InspectorPI* each manually generate a long-term ECDH identity key (IK). Then they share with each other their public ECDH identity key. After that has been done Start Mode can begin, meaning the Double-Ratchet programs can be launched and begin executing on the *HostPi* and *InspectorPi* computers.

4.1.2 Start Mode

When the adapted DRA software begin executing on the *InspectorPi* and *HostPi* computers the programs begin in Start Mode. This is the point in time when *InspectorPi* and *HostPi* identify themselves to one another and secure communications begin. The startup process uses the standard X3DH algorithm to authenticate *HostPi* and *InspectorPi* to one another and to exchange keys (a.k.a. key entropy data). No pseudocode for X3DH is shown here because the algorithm is quite involved, it’s clearest simply to describe it. The Python code the project is developing will use a well-tested and trusted cryptographic code library which implements the X3DH functionality.

4.1.2.1 *HostPi* (Host-controlled computer connected to camera)

- generate SPK (Signing Pre-Key)
- generate ESK (ECDH Salting Key, input to Extended Triple Diffie-Hellman)
- generate OPK ECDH (One-time Pre-Key, input to Elliptic Curve Diffie-Hellman ECDH)

4.1.2.2 *InspectorPi (Inspector-controlled computer)*

- generates SPK
- generates ESK
- generates EK ECDH keys.

The *HostPi* and *InspectorPi* exchange messages containing public key material to negotiate secret keys used for authentication and encryption. All messages report status along with optional text and at least one ECDH public key. Both HostPi and InspectorPi use the first message type to exchange ephemeral salting key material that is used as input to SYMMRATCHET. The remaining two messages are used at the start to mutually authenticate the HostPi and InspectorPi and negotiate an initial root secret (SK) using Extended Triple Diffie Hellman Key Exchange X3DH.

The HostPi starts the conversation by sending a message containing the HostPi's public SPK and OPK keys. On receipt, the InspectorPi reads the HostPi's two public keys (SPK.pub and OPK.pub) supplies its own ephemeral root public key (EK.pub), and then sends this key back to the HostPi. Both sides then initiate a X3DH key exchange to compute the initial root secret (SK) used in the DRA.

After X3DH, the system is ready to begin generation mode where two-way communications are used to sign every generated video frame until Inspector and Host arrive to retrieve InspectorSD and HostSD for review.

4.1.3 Generation Mode

The following steps are performed to update and exchange keys via ratchets every time a video frame is received by HostPi.

1) *HostPi* receives a video frame from the Camera. *HostPi* advances its send chain one more time to generate the authenticating HMAC KEY.

   ```
   KEY = SYMMRATCHET.next(shared_send_key)
   ```

2) *HostPi* takes KEY, timestamp, sequence number, and video frame and executes the Signing Process described above. Once complete, the signed video frame and sequence number will be stored on *HostSD.*

3) *HostPi* generates new ESK public key. *HostPi* also increments the sequence number. *HostPi* sends the ESK and sequence number to the *InspectorPi* and then waits for the InspectorPi to respond with its next ESK public key.

4) The *InspectorPi* takes the *HostPi's* ESK public key, performs an ECDH key exchange to obtain the ephemeral salt and then uses it to replace the *InspectorPi* receive chain as follows:

   ```
   dh_salt = DHKEX(Inspector.ESK.priv, Camera.ESK.pub).outkey
   shared_recv_key = Inspector.root ratchet.next(dh_salt)
   Inspector.receive chain = SYMMRATCHET(shared_recv_key)
   ```

5) *InspectorPi* advances its receive chain one more time to generate the authenticating HMAC KEY. *InspectorPi* sends KEY and sequence number to the *InspectorSD*.

```
KEY = SYMMRATCHET.next(shared_recv_key)
```

6) *InspectorPi* then generates a new ESK, performs another ECDH key exchange using the *HostPi's* ESK public key replaces its send chain ratchet and then sends new ESK to *HostPi*:

```
Inspector.ESK = KEYGEN()
dh_salt = DHKEX(Inspector.ESK.priv, Camera.ESK.pub)
shared_send_key = Inspector.root ratchet.next(dh_salt)
Inspector.send chain = SYMMRATCHET(shared_send_key)
```

7) The *HostPi* received new ESK from *InspectorPi* and updates the receive chain, generates a new ESK for the next salt, and updates the send chain as follows:

```
dh_salt = DHKEX(Camera.ESK.priv, Inspector.ESK.pub)
shared_recv_key = Camera.root ratchet.next(dh_salt)
Camera.receive ratchet = SYMMRATCHET(shared_recv_key)
Camera.ESK = KEYGEN()

dh_salt = DHKEX(Camera.ESK.priv,Inspector.ESK.pub)
shared_send_key = self.root ratchet.next(dh_salt)
Camera.send chain = SYMMRATCHET(shared_send_key)
```

5.0 Implementation

Although this document is meant as a high-level discussion of the double-ratchet algorithm and the intended adaptation for authentication rather than encryption, we briefly discuss the intended implementation in this section. The application is expected to be implemented in the Python programming language and use the cryptography functions in the Cryptography[2] and PyCryptoDome (forked from now-defunct PyCrypto library which has been abandoned due to security vulnerabilities) libraries. These are well-supported libraries widely used by the Python programming community in software applications where encryption is needed, thus they are well-tested and under continual scrutiny by professional programmers including cryptographers.

The Cyptography library is sort of a Swiss army knife of cryptographic functions. It has X3DH and DH (Diffie-Hellman) support as well as HMAC-SHA-512 (a.k.a. SHA-3 or Keccak) and other needed functionality. Regarding developing software application that need cryptography, it is always strongly discouraged to “roll your own” from primitives because that is easy to botch. As to whether to use the Cryptography library or not comes down to several factors. One of them is that Cryptography was one of the first to support the Python 3 language (end-of-life for the long-running Python 2 was January 1, 2020). Another reason has to do with how the primitives are implemented. Some other cryptography libraries are known to be vulnerable to side channel attacks which Cryptography works to avoid. Under the hood Cryptography calls OpenSSL using the Rust language, and so is likely to avoid some of the side channel pitfalls. Yet another reason for Cryptography is it supports RSA keygen, again making it easy to implement in this demonstration software.

[2] The **`cryptography`** open-source project https://cryptography.io

6.0 References

[1] Markus Alvila, “A Performance Evaluation of Post-Quantum Cryptography in the Signal Protocol”, 2019, Master’s thesis, Linköping University | Department of Electrical Engineering, https://liu.diva-portal.org/smash/get/diva2:1331520/FULLTEXT02.pdf

Appendix A – Key Generation Performance

We should conduct timing tests on selected camera and inspector. The computation runtime bottleneck for the DRA algorithm will occur during ephemeral key generation (Mode 3 below.)

Should new ephemeral salting keys be computed and exchanged every time the ratchet is updated? If any give KEY is discovered, an attacker could, given sufficient resources, predict the next KEY from the ratchet. The original implantation of Signal recognized this and chose to negotiate new salts using ephemeral ECDH key exchange if both parties continue to communicate bidirectionally. However, in the case when only one party is sending, that party cannot negotiate a new ephemeral salting key and is forced to either use the same send key or the next unsalted ratchet key.

If the *HostPi* and *InspectorPi* do not negotiate a new salting ephemeral key with every ratchet advance, then should a given KEY be discovered, that video frame and subsequent video frame KEYS could be more quickly discovered, allowing that and subsequent video frames to be undetectably forged and replaced up to the point where the next ephemeral salt exchange occurs. To minimize this possibility, a new salt could be negotiated every time the ratchet is advanced, however this may be computationally infeasible given the computational cost of generating ECDH keys.

A.1 Conduct of Operation - Modes of Operation

Table 3: Modes of Operation

Mode	Description	Network (Camera and Inspector)
1	Initialization (Camera and Inspector)	Disconnected
2	Root Key Negotiation (Camera and Inspector)	Connected
3	Key Sequence Generation (Camera and Inspector)	Connected
4	Video Capture (Camera)	Disconnected
5	Video Export (Camera to Redactor)	Disconnected
6	Video Import (Redactor to Inspector)	Disconnected
7	Video Inspection	Disconnected

A.2 Timing tests conducted on a Samsung Galaxy Note 8

Alvila, Markus, “A Performance Evaluation of Post-Quantum Cryptography in the Signal Protocol”, 2019, Linkoping University, Sweden, a master’s thesis[3].

Table 4: Performance Evaluation of Quantum Cryptography for Signal Protocol

ECDH Curve	Performance	Key Size	Post-Quantum Resistance
25519	352 ms	128 bits	No
CSIDH-512	4110 ms, about 11 times worse	512 bits	Yes
CSIDH-1024	22181 ms, about 58 times worse	1024 bits	Yes

[3] https://liu.diva-portal.org/smash/get/diva2:1331520/FULLTEXT02.pdf

Appendix B – Resistance to Cracking Keys by a Quantum Computer

To improve resistance, CSIDH keys can be generated and used during Initialization and Start Modes. However, during Generation Mode CSIDH keys may be too computationally expensive. For example, if N = 1,000,000, and generating a CSIDH-1024 key takes 10 seconds per, this would require about 120 days to compute 1,000,000 ephemeral salting keys. To contrast, 25519 keys relatively performant but are also thought to not be quantum resistant. Even so, using 25519 keys for salt negotiation still require about 1/3 of a second each depending on the speed of the *HostPi* and *InspectorPi* hardware, meaning generating 1,000,000 KEYs could take about 4 days of compute time alone, leaving enough compute resources to capture about three images per second. This too may be unacceptable.

Appendix C – Original Double Ratchet Algorithm

This section summarizes the double-ratchet algorithm as it exists in the Signal implementation. Also, a concise summary of the DRA algorithm is contained in the Alvila paper cited in the References section above.

Begin with two people who want to share secure messages, Alice and Bob. First, Bob generates a public-private key pair and shares his public key with Alice. Alice performs the same process to generate her own public-private key pair using Bob's public key. Once both parties have performed these actions, they have the capability to generate a joint secret key that only they know and is determined through the DH output step of the process illustrated graphically in Figure 2. This process is called the Diffie-Hellman (DH) key-exchange and remains a widely used method to exchange secret keys over untrusted networks.

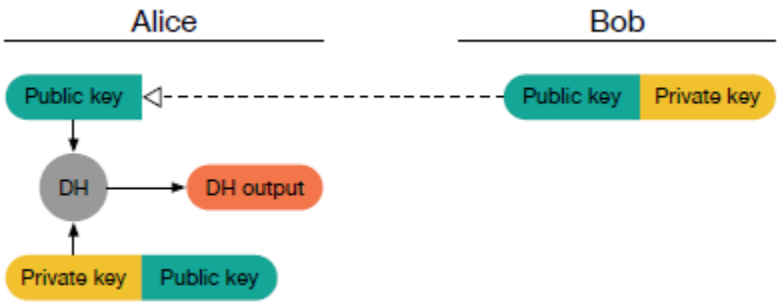


Figure 2: Initialization step of the Ratchet [7] uses traditional Diffie-Hellman (DH) key exchange

Next, when Alice sends Bob a message, she sends along her public key, and Bob uses the same DH protocol to calculate a new public-private key pair based on Alice's public key. This is illustrated as the steps to the right of the blue line in Figure 3. Using Alice's public key and his private key, the mathematics of DH create a secret key that is shared between both parties. During this process Bob calculates two DH outputs. The first allows him to read the encrypted message sent by Alice which is encrypted with the shared secret key, and the second is a new secret key to encrypt the next message sent to Alice, thus continuing the ratchet process to update keys with each message sent.

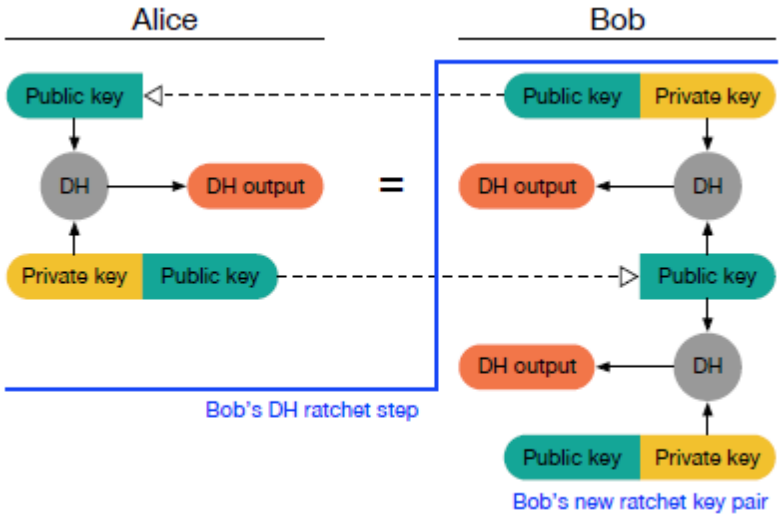


Figure 3: Process for a Single Message Exchange [7]

Now that Alice has received a message from Bob, she mimics the process Bob just did. This is illustrated to the left and below the blue line in Figure 4. Generally, each ratchet step after initialization is simply a repeat of the following steps:

1. Create private/public key pair
2. Calculate shared secret key using the DH key exchange protocol
3. Send message encrypted with shared secret key and sender's public key
4. Use sender's public key and recipient's private key to calculate shared secret key and decrypt message

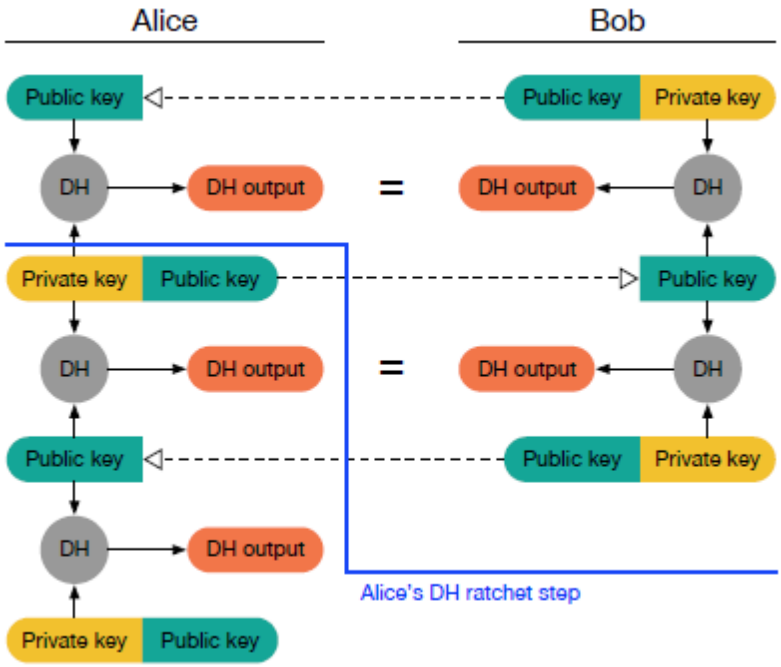


Figure 4: Process for Receipt of Message and Generation of New Keys for Subsequent Message [7]

Appendix D – Start Mode Details

D.1.1 X3DH Root Key Negotiation Detail

The Camera and Inspector are connected over a network and mutually authenticated using their pre-shared public keys.

The Camera sends SPK.pub and OPK.pub to the Inspector using a MSG2.

The Inspector sends EK.pub to the Camera using a MSG1.

Using the X3DH algorithm the Camera derives a root secret key (SK) as follows using the Diffie Hellman Key Exchange algorithm (DHKEX) with the Host Key Derivation Function (HKDF).

Compute DH1 = DHKEX(SPK,IK.pub)

Compute DH2 = DHKEX(IK,EK.pub)

Compute DH3 = DHKEX(SPK,EK.pub)

Compute DH4 = DHKEX(OPK,EK.pub)

SK = HKDF(DH1 + DH2 + DH3 + DH4, 32)

In X3DH, DH1 and DH2 provide mutual authentication. DH3 and DH4 provide perfect forward secrecy. OPK is a one-time public-private key and should be deleted after use.

Similarly, the Inspector also computes SK as follows:

Compute DH1 = DHKEX(IK,SPK.pub)

Compute DH2 = DHKEX(EK,IK.pub)

Compute DH3 = DHKEX(EK,SPK.pub)

Compute DH4 = DHKEX(EK,OPK.pub)

SK = HKDF(DH1 + DH2 + DH3 + DH4, 32)

The Camera then computes the root ratchet for its DRA send key chain using the SK as input.

```
root ratchet = SYMMRATCHET(SK)
```

send chain = root ratchet.next().outkey

receive chain root ratchet.next().outkey

Appendix E – SYMMRATCHET Detail

SYMMRATCHET is a simple iterator that either initializes itself using an input root key as its initial state or advances (turns the ratchet) when its next() method is called. The next() method can be optionally salted.

```
SYMMRATCHET.init(key):
   self.state = key
SYMMRATCHET.next(saltkey):
   output = HKDF(self.state + saltkey, 80)
   self.state = output[:32]
   outkey = output[32:64]
   iv = output[64:]
   return (outkey,iv)
```

The Inspector then computes the root ratchet for its DRA receive key chain:

1. root ratchet = SYMMRATCHET(SK)
2. receive chain = root ratchet.next().outkey
3. send chain = root ratchet.next().outkey

To maintain forward secrecy, the ratchet is resalted by negotiating new ephemeral salts using ECDH key exchange. With each rotation, a new ephemeral ECDH key is generated, exchanged using DHKEY and the result is then passed as the SALTKEY argument to SYMMRATCHET. This keeps each key and ratchet material unknown to an observer. It also requires an attacker that expends the resources needed to discover a KEY must allocate equivalent resources to discover the next one, maintaining forward secrecy.

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354
1-888-375-PNNL (7665)

www.pnnl.gov