Encoding ZRTP in CryptoVerif

Di Long Li (Darren) u5490127
Australian National University

October 27, 2017

Abstract

Encryption between two parties require shared key materials in order for the receiving party to decrypt and render the data to be useful. Ideally the communicating parties would share the keys used for encryption in an already secure environment. However, this is not always practical - said environment may be expensive to establish. Thus in practice, keys are often exchanged in a public and untrusted environment, and can be done via the use of key exchange algorithms such as Diffie-Hellman. While key exchange algorithms protects the parties from eavesdropping agents which are not attacking, the key exchange process is vulnerable to Man-in-The-Middle(MiTM) attacks. As a result, key exchange algorithms are normally used inside key exchange protocols, which defend against said attacks via use of authentication.

However, traditional key exchange protocols still require some form of shared and existing key materials, such as public-private keys, certificates, to provide such authentication. ZRTP key exchange protocol does not rely on long-term, static key materials to provide authentication, and still achieves high level of protection against MiTM attacks via use of hash commitment and Short Authentication String(SAS).

This project explored the possibility of encoding ZRTP in CryptoVerif, though unsuccessful, the project process would hopefully serve as a meaningful case study of use of protocol provers.

Acknowledgement

First and foremost, I would like to thank professor **Rajeev Gore**, leader of the ANU Logic and Computation Group for supervising me in this project. He has given me a lot of helpful advice in regards to scoping and directions of this project, and also feedback which helped me to improve my writings.

I would also like to thank the following people for their comments and suggestions for this writing : **Ming-Dao Chia** (from ANU), **Yin Hei Lee** (from Hong Kong University of Science and Technology), **Tsz Fung Kwong** (from Chinese University of Hong Kong), and **Ho Yin Or** (from ANU).

Contents

Glossary

Alice, Bob	-	legitimate communicating agents
Eve	-	eavesdropping agent/attacker
MiTM	-	Man-in-The-Middle

Notes

Interchangeable terms

In the following sections, the following terms may be used interchangably

Key agreement protocol	-	Key exchange protocol		
Key agreement algorithm	-	Key exchange algorithm		
Key materials	-	Cryptographic materials	-	Secrets
Plaintext	-	Clear text	-	Unencrypted data
Ciphertext	-	Encrypted data		

ZRTP name

ZRTP specification does not state the unabbreviated name of ZRTP, if any exists, thus the author of this report assumes ZRTP is the full name of the protocol.

The author of this report deduces Z may stand for Zimmermann, the last name of the main author of ZRTP, and RTP may stand for Real-time Transport Protocol(which will be mentioned below).

This is only to address potential confusions, and the supposed "full" name is not used in any significant way in this report, if at all. Thus the reader should not be concerned and may treat ZRTP as a purely nominative term.

1 Background

1.1 Key Agreement Protocol

1.1.1 Motivation

Encryption by itself does not necessarily require key exchanges if there is only one party involved, e.g. a user does an encrypted backup that it only intends to share with itself.

However, when encryption is used between two parties, e.g. HTTPS between browser and web server, encrypted voice chat, exchange of cryptographic key materials is required, as otherwise there is no way for the opposite side to decrypt the content.

The exchange of key materials or "secrets" can be in many forms, which we will illustrate using a concrete and (hopefully) approachable example.

Suppose Alice wants to send a file securely to Bob, but also wants to allow future exchanges of files with Bob, so Alice intends to establish a long term method. Alice first meets up with Bob, and both agree on a lengthy passphrase to be used in 7-zip(or similar archiving programs). Then later that day, Alice sends the file using the previously agreed passphrase to encrypt and archive the file, and send the archive over to Bob via email, or file sharing services(e.g. Dropbox, Google Drive). Bob downloads the archive, and decrypt using the previously agreed passphrase. Bob now has the file Alice wanted to share, and this completes the secure file exchange process.

In the above example, the meet up and agreement on a passphrase is the exchanging of key materials. It should also be obvious that without this stage, Bob has no way to decrypt Alice's archive, short of cracking the passphrase.

Similarly, for the encryption we use on a daily basis, in the form of HTTPS, SSL, SSH and so on, the encrypted traffic on one side can only be decrypted by the other side if a key exchange process has occured already.

However, one important thing we should recognise at this point is that Alice cannot always meet up with Bob - physically meeting up is not always possible, and can be very expensive even if possible.

Thus, the most practical way of exchanging key materials would be to do it via public and untrusted channel, i.e. across the Internet or across adversaries' network. The difficulty should be obvious now, but we will state it explicitly in the following : In our previous example, Alice and Bob could rely on trust of the physical environment(e.g. no one could steal the key without them noticing). However, in the public environment, it is impossible to guarantee the lack of eavesdropping. The specific difficulty, as illustrated above, would be : How does Alice exchange encryption keys with Bob such that (1) both of them get the final key they can use for encryption, and (2) eavesdropping agent(no attacking), even with knowledge of all their communication, cannot derive the key they are going to use?

The above two difficulties are what key exchange **algorithms** intend to address. Note that Alice and Bob are still vulnerable to MiTM(Man-in-The-Middle) attacks when Eve, outside of purely eavesdropping, also engages actively in sabotaging the communication. MiTM attacks are handled by key exchange **protocols**.

The vulnerability to MiTM attacks is analogous to a real life situation of meeting up with a person one has never seen before. Suppose Alice wishes to meet up with Bob, similar to the above situation, but with the added condition that they have never seen each other before, and has zero information about what each other looks like. Then it is entirely possible(and trivial) for Eve to pretend to be either of them - neither Alice nor Bob can tell. Furthermore, now it is also possible that Eve agrees on a passphrase with Alice, then also agrees on a passphrase with Bob. If done so, Eve then can decrypt files sent between Alice and Bob in

either direction.

Also importantly, Eve can still eavesdrop silently. When Alice sends a file to Bob, Eve can decrypt it first using Alice's passphrase, and encrypt it using Bob's passphrase before passing onto Bob, and vice versa. This way, neither Alice nor Bob can notice their files are compromised(no longer secretive).

1.1.2 Principle

We start by discussing key agreement algorithms in general terms first, then we will reiterate the motivation of key agreement protocols.

Key agreement algorithms rely on mathematical properties of certain calculations, for example, the well-known Diffie-Hellman(DH), or finite-field DH, relies on modular arithmetic.

While DH is commonly used as reference for behaviour of key agreement algorithms, the process can be put in more general terms.

Below is a modified version of the *General Overview* section of Wikipedia page on *Diffie–Hellman key exchange*, specifically the paint analogy displayed[1]. The author of this report would like to note that, to the best of his knowledge, the section aligns with lecture notes such as Kirby A. Baker's[2].

The important steps put in general terms, are as follows (the order is not necessarily strict)

- A common parameter (not private) is shared and agreed on between Alice and Bob, denoted as c
- Alice generates a secret that ONLY Alice knows, denoted as sA
- Bob generates a secret that ONLY Bob knows, denoted as sB
- There exist irreversible (computationally infeasible to invert/reverse) functions, $mix1, mix2$, such that $mix2(mix1(c, sA), sB) = mix2(mix1(c, sB), sA)$ holds
- Alice computes $rA = mix1(c, sA)$, and sends rA to Bob
- Bob computes $rB = mix1(c, sB)$, and sends rB to Alice
- Alice computes $r = mix2(rB, sA)$
- Bob computes $r = mix2(rA, sB)$
- Note that due to the properties of $mix1, mix2$ as stated above, Alice's r = Bob's r, thus the key agreement process is complete, and Alice, Bob now share a common secret r

There are some important properties to be observed in the above process

- $mix1, mix2$ do not need to be different, as long as the above property holds - DH uses the same function
- sA, sB are never exposed in public, and also due to the irreversible property of the two functions, Eve cannot find out sA, sB from rA, rB
- Since sA, sB are never exposed, Eve has no way to compute r

However, the one last important property is what motivates key agreement protocol

- Neither Alice nor Bob can verify each other's identity in above steps - there is lack of authentication

The lack of authentication gives rise to Man-in-The-Middle(MiTM) attacks, thus key agreement algorithms are normally not used alone, but used within key agreement protocols which contain some form of authentication.

1.2 Authentication

Usually, the authentication is done by using existing, shared cryptographic materials, such as keys from a Public-Key-Infrastructure(PKI) setting, certificates issued by a Certificate Authority(CA), or shared secrets.

In the above cases, the verification of identity can be done for example in form of cryptographic challenges. Either side can issue a "cryptographic challenge", which is a computational problem that is only solvable if one has the private key, or shared secrets. This means, you only trust the other side if they have sucessfully solved your problem. A proof of identity test, so to speak.

The challenge in the simplest form may just be asking the other side to decrypt an encrypted message which you sent, and send the plaintext(or clear text, i.e. unencrypted) back to you.

In a PKI setting, the simplest challenge to be issued by Alice may just be Alice encrypt a randomly generated text using Bob's public key. The encrypted message can only be decrypted by Bob's private key(the cryptographc explanation is beyond the scope of this report). Thus if the other side can decrypt successfully and send the original text back to Alice, then Alice has evidence to believe the other side is indeed Bob. Certificates are dealt with similarly.

In a symmetric setting, that is, some form of shared secrets are used symmetrically - encrypt using the same key, decrypt using the same key, then the challenge can be as follows. Alice encrypts a randomly generated text using the symmetric key, if the other side can decrypt and send the text back to Alice, then Alice has evidence to believe the other side shares that particular key with Alice, and that implies the other side is Bob.

In the above settings, by relying on existing cryptographic materials, the authentication process is fairly straightforward - you test the knowledge of the other person, and you gain resistance to attacks by having longer and higher quality keys etc.

However, ZRTP does not rely on existing cryptographic materials and yet still exhibits strong resistance to MiTM attacks in the authentication phase by usage of Short Authentication String(SAS) and hash commitment, which will be covered in the following sections.

1.3 ZRTP

1.3.1 Introduction

ZRTP is a "key agreement protocol that performs a Diffie-Hellman key exchange during call setup". A call refers to a voice/video communication session in the Voice over IP(VoIP), or IP telephony, sense. Examples of VoIP applications would be Skype, and other mobile applications(or apps) where one calls someone via the Internet(the call does not rely on the mobile network, but on the data network or WiFi instead).

As mentioned in the **Key Agreement Protocol** section, key agreement protocols address the need of cryptographic materials exchange/agreement before the actual encryption takes place. In this context, the encryption would be the Secure Real-time transport Protocol(SRTP) traffic.

SRTP provides security features on top of Real-time transport Protocol(RTP), such as "confidentiality, message authentication, and replay protection to the RTP traffic and to the control traffic" [3, pg. 3].

RTP is the core part of delivery of content, it "provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video" [4, pg. 1].

ZRTP "generates a shared secret, which is then used to generate keys and salt for a Secure RTP (SRTP) [RFC3711] session"[5, pg. 4].

To summarise, **SRTP**, which is based on **RTP**, provides the encryption of traffic (and also some other security features) between two communicating parties, and **ZRTP** handles the key exchange process before the encryption takes place.

Both SRTP and RTP are beyond the scope of this report, but they are mentioned in above paragraphs to hopefully provide sufficient context for the reader to understand the role of ZRTP during a call.

1.3.2 Overview

There are roughly four stages to ZRTP before the key exchange is complete and SRTP session is established. The stages are : **Discovery**, **Commitment**, **Key Generation**, and **Confirmation**.

There are three modes of ZRTP, which differ at the **Commitment** stage, and the **Key Generation** stage. The modes are : **Diffie-Hellman mode**, **Preshared mode**, **Multi-stream mode**. Since in the latter two modes, Diffie-Hellman is not used, the standard also refers them as **non-Diffie-Hellman modes**.

1.4 Stages

1.4.1 Discovery

This stage is similar to what usually occurs in other protocols, the two parties attempt to establish whether the other supports ZRTP. And if so, then begin to negotiate the parameters used in remainder of the ZRTP process.

The parameters include the protocol version, and key exchange algorithms.

Note that while only two types of Diffie-Hellman algorithms are used : finite-field Diffie-Hellman(DH) and Elliptic Curve Diffie-Hellman(ECDH), there are multiple variants in each type which differ by sizes and/or elliptic curve parameters.

1.4.2 Commitment

Commit messages mainly act to confirm parameters such as hash, cipher algorithm choices. The messages also serve important purpose in DH mode, which will be explained in detail in **Core security components of ZRTP** section below.

1.4.3 Key Generation

All three modes apply key derivation function(KDF) on the final shared secret(whether derived via DH or not) to generate **ZRTP session key**(which is used for either Multi-stream mode to generate additional shared secrets, or to generate PBX secrets), **Short Authentication String(SAS)**, **SRTP keys and salts**, **message authentication code keys(MAC keys)**, and **ZRTP keys**(which are used to encrypt confirmation messages described below).

"A Key derivation function (**KDF**) is a basic and essential component of cryptographic systems: Its goal is to take a source of initial keying material, usually containing some good amount of randomness, but not distributed uniformly or for which an attacker has some partial knowledge, and derive from it one or more cryptographically strong secret keys. We associate the notion of "cryptographically strong" keys with that of pseudorandom keys, namely, indistinguishable by feasible computation from a random uniform string of the same length. In particular, knowledge of part of the bits, or keys, output by the KDF should not leak information on the other generated bits. " [6].

In the context of this report, it suffices to keep in mind that KDF tranforms some already random materials to actual cryptographic materials to be used by encryption algorithms. This is necessary because outside of the algorithmic choice, the strength of encryption also depends on "quality"(uniformly random, unpredictable, etc) of the key materials used, and KDF standardises and raises the quality of the key materials one provides, so to speak informally.

PBX refers to the private branch exchange, and is part of a business phone system(a possibly internal phone network) [7]. A PBX may serve as a information relay or a proxy in VOIP communication. However, PBX is beyond the scope of this report, and it sufficies to know that some part of ZRTP messages are dedicated to negotiating PBX parameters, and PBX is used for underlying systems that facilitate use of ZRTP, and is not a core component of ZRTP.

SAS simply refers to a short piece of alphanumerical text that is displayed to both communicating users and needs to be verified to be match at the start of a call [5, p. 77]. This will be further explained in later sections.

SRTP keys and salts are used for the actual encryption that takes place after key exchange, and the actual usage procedures are beyond the scope of this report [5, p. 33-34].

MAC addresses the need of defending against message forgery by providing assurance about the source and integrity of an object [8]. In other words, to authenticate a message. MAC are attached in some ZRTP messages [5, p. 55-63], but is not a core part of the key exchange protocol, thus any further explanation is beyond the scope of this report. It suffices to know it allows either party to detect forgery of messages, and while MAC depends on the content of message, MAC itself is not confidential - adversary cannot forge messages with legitimate MAC even with knowledge of previous MACs [9].

1.4.4 Confirmation

Encrypted confirmation messages are exchanged as the final part of the key exchange process. The encryption uses the **ZRTP keys** derived in the **Key Generation** stage [5, p. 33-34].

1.5 Modes

The three modes differ by behaviour in **Commit** and **Key Generation** stage.

1.5.1 Diffie-Hellman mode

This mode is the most critical mode as it establishes the initial key materials from scratch, and the other two modes rely on the key materials derived in this mode. Thus DH mode will be the focus of this report.

In other words, given two parties with no prior communication, Diffie-Hellman mode is the only mode that can establish shared key materials, as key exchange only happens in this mode.

This mode uses either finite-field Diffie-Hellman(DH) or Elliptic Curve Diffie-Hellman(ECDH) as the core algorithm. Both algorithms are based on the same principle described in the **Key Agreement Protocol** section. They differ by the mathematics used and the cryptographic explanations are beyond the scope of this report.

In this mode, the initiator also performs **hash commitment** (which will be explained in detail in later section) to reduce probability of a successful attack.

Note that if this mode is used when there are cached shared secrets, the cached secrets may be used as part of the computation of the final new shared secret.

1.5.2 Preshared mode

This mode relies on cached secrets from previous session, and does not rely on DH to compute the shared secret.

This mode is useful for quick re-establishment of communication, as recomputing via DH is expensive and can take a long time. This also benefits low-power devices by reducing number of expensive computations required.

1.5.3 Multistream mode

This mode is mainly used to establish additional media streams between two parties which already have an active SRTP session.

The ZRTP session keys generated previously can be used to derive the keys and salts for the new media streams without performing DH computations.

1.6 Core security components of ZRTP

1.6.1 Hash commitment and SAS

This section is dedicated to DH mode, the other two modes are not considered as key exchanges do not happen in the non-DH modes. Furthermore, this section is concerned largely with **Commitment** stage

The following text is based on a paragraph from the standard as shown below, which contains a short description of security properties related to hash commitment and SAS.

"The use of hash commitment in the DH exchange constrains the attacker to only one guess to generate the correct Short Authentication String (SAS) (Section 7) in his attack, which means the SAS can be quite short. A 16-bit SAS, for example, provides the attacker only one chance out of 65536 of not being detected. Without this hash commitment feature, a MiTM attacker would acquire both the pvi and pvr public values from the two parties before having to choose his own two DH public values for his MiTM attack. He could then use that information to quickly perform a bunch of trial DH calculations for both sides until he finds two with a matching SAS. To raise the cost of this birthday attack, the SAS would have to be much longer. The Short Authentication String would have to become a Long Authentication String, which would be unacceptable to the user. A hash commitment precludes this attack by forcing the MiTM to choose his own two DH public values before learning the public values of either of the two parties." [5, pg. 21].

The following text is an attempt of a clearer and more explicit explanation of the above statements by the author of this report.

Note that MiTM attacks of key exchange process without authentication happens easily if the attacker(Eve) can intercept the traffic between the two parties(Alice, Bob), as then Eve can negotiate a separate key with Alice, and a separate key with Bob, allowing itself to decrypt and reencrypt later traffic sent in either direction. This concept has already been explained in **Key Agreement Protocol** section. The following cases argue for how a half-complete authentication(that is, only using SAS, but no hash commitment) still fails.

We first start by clarifying the terms used in following paragraphs, then we will follow with a description of the sequential ordering of events.

- We use the word **"settle"** in the following sections to refer to the act of making some value constant in the process(neither party can change the value later on in the process)
- Commit message is a message that makes one settles on a public value(which in turn implies settling on the corresponding secret as well). This is sent by the Initiator, and contains the hash of the DHPart2 message
- Initiator is whichever party that sends the commit message
- Responder is whichever party that receives the commit message
- A DH message is a message that contains hashes of cached shared secrets of the sending party, and the result of mixing(application of mix functions mentioned in section 1.1) between sending party's secret and public common parameter
    - The cached shared secrets are ignored in the following paragraphs to simplify explanations, as they do not affect the hash commitment and SAS mechanism
    - The standard refers the **i**nitiator's secret as **svi**, and the result of mixing as public value **pvi**
    - The standard refers the **r**esponder's secret as **svr**, and the result of mixing as public value **pvr**

- The private and private values above map to the names in **Principle** section as follows, with Alice being the Initiator(using **svi**, **pvi**) and Bob being the Responder(using **svr**, **pvr**). The common parameter c is agreed on via Hello messages that are exchanged before everything in this section.
  svi – sA = secret that ONLY Alice knows
  pvi – $rA = mix1(c, sA)$
  svr – sB = secret that ONLY Bob knows
  pvr – $rB = mix1(c, sB)$

- DHPart1 is the DH message sent from **responder** to **initiator**, which contains **pvr**
- DHPart2 is the DH message sent from **initiator** to **responder**, which contains **pvi**
- Hash commitment refers to the action of sending the hash of entire DHPart2 in the commit message. This is done by the initiator, and forces the initiator to settle on its public value **pvi**. Since if the initiator decides to change the **pvi** later, responder can detect it due to the hash of the new DHPart2 message being different from the one contained in the commit message
- Short Authentication String(SAS) refers to a short piece of text that the two communicating parties need to confirm (verbally) to be matching at the start of SRTP session(a voice call). The text is displayed on whatever device the user may be using. SAS can also be confirmed via use of cryptographic signatures, but overall is still simply an action to make sure they match, thus we will just consider the case where users check them manually

The strict sequential ordering of events are as follows

1. Initiator settles on its public value **pvi**, and generates the DHPart2 message. Then embed the hash of the DHPart2 message into the commit message
2. Initiator sends the commit message, completing hash commitment in process of doing so
3. Responder receives the commit message
4. Responder sends DHPart1
5. Initiator receives DHPart1
6. Initiator sends DHPart2
7. Responder receives DHPart2
8. Responder makes sure the hash of the received DHPart2 message matches the hash embedded in the commit message

Above process contains the hash commitment mechanism in ZRTP, which is part of the core defense against MiTM attacks, and the following paragraphs will only focus on the above steps.

An additional detail reader should keep in mind is that the final secret produced from the above key exchange process will be used to generate the SAS, thus if the users notice their SASs are different, they know that they no longer share the same final secret, which entails that someone has interfered with the key exchange process. However, this is a one way implication : it is possible that they no longer share the same final secret, but still have the same SAS - this is exactly what the attacker wants to achieve.

Thus, we define a successful MiTM attack as follows : the SAS displayed to both users are same, and as a result neither user knows MiTM attack has occured. But the attacker shares the same final secret with them separately, allowing silent eavesdropping by decryption and re-encryption(similar to the example given in **Principle** section.

We now explain the importance of above process by going through all potential scenarios(similar to proof by exhaustion, albeit informal), and how MiTM attacks may be carried out successfully in each of them. Note that we ignore timeouts(we simply assume nothing fails due to timeout), and network specifics etc.

We also introduce a new function to further abstract away the technical details :

$$genSAS(\text{finalsecret}) = \text{SAS to be displayed to user}$$

We also introduce a few notations :

$\underline{\text{var}}$	-	the variable is still under control by someone, not settled yet
~~var~~	-	the actual value of the variable is not known by a certain person

The exact person the notation refers to will be mentioned along with the usage of it.

Without the hash commitment (no commit messages)

case 1 :

Alice	⟷	Eve	⟷	Bob
Initiator		Responder | Initiator		Responder

The events may unfold in the following way which results in Eve's success

1. Eve picks its pvr and svr, denoted as **eapvr** and **easvr**, where the **eapvr** is shared with Alice
2. Eve sends **eapvr** to Alice via DHPart1 message
3. Alice picks its pvi and svi, denote as **apvi** and **asvi**, where **apvi** is shared with Eve
4. Alice sends **apvi** to Eve via DHPart2 message
5. Bob picks its pvr and svr, denoted as **bpvr** and **bsvr**, where **bpvr** is shared with Eve
6. Bob sends **bpvr** to Eve via DHPart1 message
7. Eve now still has **ebpvi** and **ebsvi** under its control, so it can bruteforce the two values such that the following holds, where aSAS = Alice's SAS and bSAS = Bob's SAS

$$\begin{aligned} bSAS &= genSAS(mix2(\underline{\mathbf{ebpvi}}, \mathbf{bsvr})) && \text{(Bob's calculation)} \\ &= genSAS(mix2(\mathbf{bpvr}, \underline{\mathbf{ebsvi}})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\mathbf{apvi}, \mathbf{easvr})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\mathbf{eapvr}, \mathbf{asvi})) && \text{(Alice's calculation)} \\ &= aSAS \end{aligned}$$

8. Eve sends **ebpvi** to Bob, and this results in Alice and Bob seeing the same SAS

Above concludes Eve's successful MiTM attack, and now Eve can eavesdrop or modify Alice's and Bob's traffic silently.

case 2 :

Alice	⟷	Eve	⟷	Bob
Responder		Initiator | Reciever		Initiator

Notice this case is symmetrically identical to our first case, with Alice and Bob swapped, so in a fashion similar to our first case, we argue that Eve can successfully accomplish a MiTM attack.

case 3 :

Alice	⟷	Eve	⟷	Bob
Initiator		Responder | Responder		Initiator

Eve has no way to succeed in this case, which we will show below

1. Eve picks its first pvr and svr, denoted as **eapvr** and **easvr**, where the **eapvr** is shared with Alice
2. Eve sends **eapvr** to Alice via DHPart1 message
3. Eve picks its second pvr and svr, denoted as **ebpvr** and **ebsvr**, where the **ebpvr** is shared with Bob
4. Eve sends **ebpvr** to Bob via DHPart1 message
5. Eve now has no control over any of the remaining variables, thus no attacks are possible

Above concludes Eve's failed MiTM attack

case 4 :

Alice	$\longleftrightarrow$	Eve	$\longleftrightarrow$	Bob
Responder		Initiator | Initiator		Responder

1. Alice picks its pvr and svr, denoted as **apvr** and **asvr**, where **apvr** is shared with Eve
2. Alice sends **apvr** to Eve via DHPart1 message
3. Bob picks its pvr and svr, denoted as **bpvr** and **bsvr**, where the **bpvr** is shared with Eve
4. Bob sends **bpvr** to Eve via DHPart1 message
5. Eve now has control over **eapvi**, **easvi**, **ebpvi**, and **ebsvi**, so it can bruteforce the four values such that the following holds, where aSAS = Alice's SAS and bSAS = Bob's SAS

$$\begin{aligned} bSAS &= genSAS(mix2(\mathbf{ebpvi}, \mathbf{bsvr})) && \text{(Bob's calculation)} \\ &= genSAS(mix2(\mathbf{bpvr}, \mathbf{ebsvi})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\mathbf{apvr}, \mathbf{easvi})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\mathbf{eapvi}, \mathbf{asvr})) && \text{(Alice's calculation)} \\ &= aSAS \end{aligned}$$

6. Eve sends **eapvi** to Alice
7. Eve sends **ebpvi** to Bob
8. This results in Alice and Bob seeing the same SAS

Above concludes Eve's successful MiTM attack, and now Eve can eavesdrop or modify Alice's and Bob's traffic silently.

We see that Eve cannot attack successfully in the 3rd case, but Eve can just force renegotiation whenever it encounters the 3rd case(the exact messages to be sent are beyond the scope of this report), thus Eve always has a chance of successfully attacking Alice and Bob.

Obviously we are assuming Eve can bruteforce the values within timeout period, thus we can potentially void the assumption by making SAS very lengthy. However, this invalidates the original intention of having SAS reasonably short and yet still effective.

With hash commitment

case 1 :

Alice	⟷	Eve	⟷	Bob
Initiator		Responder | Initiator		Responder

1. Alice settles on its pvi and svi, denote as **apvi** and **asvi**, where **apvi** is shared with Eve
2. Alice sends hash of **apvi** to Eve via commit message
3. Eve settles on its first pvi and svi, denoted as **ebpvi** and **ebsvi**, where **ebpvi** is shared with Bob
4. Eve sends hash of **ebpvi** to Bob via commit message
5. Bob picks its pvr and svr, denoted as **bpvr** and **bsvr**, where **bpvr** is shared with Eve
6. Bob sends **bpvr** to Eve via DHPart1 message
7. Eve now still has **eapvr** and **easvr** under its control. but it does not have the actual value of **apvi**, only the hash of it. We denote the variables which Eve lacks knowledge of by crossing them out, and we reiterate the equality Eve needs to uphold

$$\begin{aligned} bSAS &= genSAS(mix2(\mathbf{ebpvi}, \mathbf{bsvr})) && \text{(Bob's calculation)} \\ &= genSAS(mix2(\mathbf{bpvr}, \mathbf{ebsvi})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\cancel{\mathbf{apvi}}, \underline{\mathbf{easvr}})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\underline{\mathbf{eapvr}}, \mathbf{asvi})) && \text{(Alice's calculation)} \\ &= aSAS \end{aligned}$$

8. Now the only way Eve can obtain **apvi** is by sending **eapvr** to Alice via DHPart1 message, but this means Eve has only one chance of picking the correct **eapvr** such that the above equality holds

The standard uses 16-bit SAS as an example, which we will use as well. For a 16-bit SAS, Eve then only has one chance out of 65536 of not being detected, which translates to probability of 0.00153% (rounded to 3 most significant digit).
We consider this to be sufficiently low chance of success, and above concludes Eve's failed attack.

case 2 :

Alice	⟷	Eve	⟷	Bob
Responder		Initiator | Responder		Initiator

Notice this case is symmetrically identical to our first case, with Alice and Bob swapped, so in a fashion similar to our first case, we argue that Eve failed to accomplish a MiTM attack.

case 3 :

Alice	⟷	Eve	⟷	Bob
Initiator		Responder | Responder		Initiator

Since we have concluded Eve has no way to succeed in this case even with no hash commitments, we simply state that Eve has no way to succeed in this case either.

case 4 :

Alice	⟷	Eve	⟷	Bob
Responder		Initiator | Initiator		Responder

1. Eve settles on its first pvi and svi, denoted as **eapvi** and **easvi**, where **eapvi** is shared with Alice
2. Eve sends hash of **eapvi** to Alice via commit message
3. Alice picks its pvr and svr, denoted as **apvr** and **asvr**, where **apvr** is shared with Eve
4. Alice sends **apvr** to Eve via DHPart1 message
5. Eve settles on its second pvi and svi, denoted as **ebpvi** and **ebsvi**, where **ebpvi** is shared with Bob
6. Eve sends hash of **ebpvi** to Bob via commit message
7. Bob picks its pvr and svr, denoted as **bpvr** and **bsvr**, where the **bpvr** is shared with Eve
8. Bob sends **bpvr** to Eve via DHPart1 message
9. Eve now has no control over any variables as it has already settled down on **eapvi**, **easvi**, **ebpvi**, and **ebsvi** via hash commitment. Reiterating the equality that needs to be upheld

$$\begin{aligned} bSAS &= genSAS(mix2(\mathbf{ebpvi}, \mathbf{bsvr})) && \text{(Bob's calculation)} \\ &= genSAS(mix2(\mathbf{bpvr}, \mathbf{ebsvi})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\mathbf{apvr}, \mathbf{easvi})) && \text{(Eve's calculation)} \\ &= genSAS(mix2(\mathbf{eapvi}, \mathbf{asvr})) && \text{(Alice's calculation)} \\ &= aSAS \end{aligned}$$

   We see that Eve now can only bet on **apvr** and **bpvr** to be of certain values such that the above equality holds. Similar to the first case, we know that the chance is extremely slim.

Above concludes Eve's failed MiTM attack.

Conclusion

From above, we see that hash commitment forces Eve to have exactly only one chance to pick the correct public values to accomplish a successful MiTM attack. Thus even picking a relatively short SAS length(16-bit in above case) still allows Alice and Bob to be pracitcally immune to MiTM attacks, given that they do verify the SAS.

1.6.2 Hash images

The following text is based on section 9 of the standard[5, pg. 95-96].

Hash image chain is used to enforce the linkage of messages, and also to allow rejection of false messages as explained further below.

The hash images H0, H1, H2, H3 are generated as follows

- H0 is generated randomly by the sending party
- H1 = hash (H0)
- H2 = hash (H1)
- H3 = hash (H2)

Note that above forms a one-way hash chain.

The above hash images are used in reverse order, that is, Hello message contains H3, Commit message contain H2, DH message contains H1, and the final confirm message contains H0 in encrypted form.

Since H0 is unpredictable due to being randomly generated, there is no way to pre-determine H1-H3 by the attacker.

Note that any missing hash image in the chain can be inferred given the previous hash image is known. For example, in non-DH modes, DH messages are never sent, thus H1 is never actually received, but since H0 is received in the confirm messages, H1 can be generated by hashing H0, and then used to verify H2, completing the chain.

Hash images overall enforce "linkage" of messages, as hash image of a message is used as HMAC key of the previous message, e.g. H2 is used as HMAC key for Hello message, H1 is used as HMAC key for Commit message. This overall enforces the condition that the same two parties must participate in all the message exchanges. Note that this alone does not prevent MiTM attacks, but it means from the perspective of one of the legitimate users, it must be either the attacker or another legitimate user that is actively participating in the session, not both.

Furthermore, hash images allow early detection of injected tampered messages in an active sessions, as there is no way to predict the next hash image value, it is impossible to forge the next message if a previous legitimate message has already been processed. This allows rejection of false messages before the expensive Diffie-Hellman computation, which in turn reduces chance of a successful Denial-of-Service(DoS) attack.

1.7 ProVerif and CryptoVerif

1.7.1 Overview

ProVerif and CryptoVerif are protocol provers used in this project. Unlike interactive theorem provers such as Coq, which provides immediate feedback with users' input, ProVerif and CryptoVerif normal usage consists of using high level specification (or encoding) of the protocol as input and finally outputting proof results. This is similar to a compiler workflow, except instead of providing source code of a program as input, the high level specification(or encoding) of the protocol is provided, and instead of a program being outputted, the proof result is outputted instead.

1.7.2 Input and output

ProVerif

ProVerif, as of time of writing, supports several input formats : untyped pi-caluclus, untyped Horn clauses, typed Horn clauses, and typed pi-calculus. However, it is recommended by the author to use typed pi-calculus as other formats are no longer actively being developed [10].

The output of ProVerif consists of results of the queries(detailed in next section).

CryptoVerif

CryptoVerif takes in code files formatted in process calculus, which is similar to the typed pi-calculus used in ProVerif, but with slightly different semantics.

The output of CryptoVerif consists of results of the queries and are detailed in next section.

1.7.3 Query system

Both uses a query system to specify "requests" to the provers : properties that the user would like the prover to verify.

The two types of properties are secrecy and authentication(via event correspondence).

Queries are specified along with the specification of protocol in the input code files.

The following are some examples of the queries :
ProVerif

```
query attacker(secretA).      (* challenge attacker to compute secretA *)
query event(B) ==> event(A).  (* ask ProVerif to show occurance of B   *)
                              (* implies occurance of A                *)
```

CryptoVerif

```
query secret secretA. (* ask CryptoVerif to prove secrecy of secretA    *)
query event B ==> A.  (* ask CryptoVerif to show occurance of B implies *)
                      (* occurance of A                                 *)
```

The semantics of the queries are explained in Notion of secrecy and Authentication sections.

1.7.4 Modelling of cryptographic primitives

ProVerif

ProVerif is based on the symbolic model. Cryptographic primitives are defined as constructors and destructors.

Constructors are similar to function signatures, in that only the domain and co-domain of the function is specified. But unlike functions, the "structure" of the result is of concern rather than the result. This is similar to the idea of type defintions and subsequently data constructors in programming languages that support algebraic data type(ADT).

Below are how hash and encryption can be defined in ProVerif :

```
fun hash(bitstring) : bitstring.

fun encrypt(bitstring, bitstring) : bitstring.
reduc
  forall x : bitstring, y : bitstring;
  decrypt(encrypt(x,y),y) = x.
```

For hash, only the constructor is provided. The lack of destructors models the property that hash is a one-way function.

Encryption is defined with a constructor, similar to hash. Decryption is defined in form of destructor, which gives a for all statement and the property(in form of equation) the function needs to uphold. The equation `decrypt(encrypt(x,y),y) = x` reads "if I decrypt an encrypted message(`encrypt(x,y)`) using the key used for encryption(`y`), then I should get the original plaintext message back(`x`)".

An almost similar notion can be captured in Haskell using ADT

```
data Hashed = Hash a
```

But the problems are that data constructors(`Hash` in above example) are always reversible(cannot express one way function in ADT) and cannot express injective property in data constructors, i.e. we can always destruct the data by pattern matching with the data constructor, and we cannot assert that each construction using `Hash` gives a unique result. While in ProVerif, without the destructor, we cannot destruct the data at all.

So the parallel lies in that the structure of the constructed data is of interest, but the difference is that the exact result of the function call or the definition of function are not of concern in ProVerif modelling.

Furthermore, the symbolic model cannot specify cryptographic primitives of various strength, e.g. any encryption defined is as good as any other encryption defined.

CryptoVerif

CryptoVerif is based on the computational model. The basic cryptographic primitives are defined in the "standard library"(`crytpoverif1.2.7/default.cvl`), and how they are defined specifically is beyond the author's knowledge as the author of this report lacks the required knowledge in cryptography.

CryptoVerif provides more accurate encoding of cryptographic primitives since it is possible to encode computational assumptions and probability of breakage, etc.

Examples shown below illustrate how hash and encryption can be defined in CryptoVerif :

```
type block [fixed,large].
...(other type definitions)

expand ROM_hash(hashkey, block, block, hash).
expand IND_CPA_sym_enc(keyseed, block, block, block, encseed, keygen,
                       encrypt, decrypt, injbot, zeros, Penc).
```

```
fun hash_symbolic (hashkey, block) : block [compos].
```

`ROM_hash` refers to hash under the Random Oracle Model, which overall means a perfect hash.

`IND_CPA_sym_enc` refers to indistinguishable under chosen plaintext attacks probabilistic symmetric encryption(probabilistic means entropy/randomness is used in encryption, namely via use of encryption seed).

The above two definitions demonstrate the usage of the standard library, `expand` is used to apply the two macros `ROM_hash` and `IND_CPA_sym_enc`, which then expand into the full definitions of the two cryptographic primitives when the file is processed in CryptoVerif.

`hash_symbolic` above is defined to draw parallel to the symbolic model.

The macro system is simplistic, which can be viewed either as a syntactic(e.g. Racket, Lisp) or textual(e.g. C preprocessor) macro system. However, it can be viewed superior to simplistic textual macro systems in that macros are preparsed(syntactic checks occur before complete textual replacements, unlike C preprocessor), but it is equal to syntactic macros in that boundendess of identifiers are not checked until macro expansion is complete.

1.7.5 Notion of secrecy

Symbolic model (ProVerif)

Weak secrecy / Syntactic secrecy
: Weak secrecy /Syntactic secrecy refers to the condition that the attacker cannot compute exactly the secret i.e. weak secrecy property holds even if attacker is able to derive partial information.

Strong secrecy
: Strong secrecy refers to the condition that the attacker "cannot detect a change in the value of the secret" i.e. the attacker has zero knowledge regarding the secret.

Computational model (CryptoVerif)

Secrecy in computational model refers to the condition that a probabilistic polynomial-time adversary has a negligible probability of distinguishing the secret from a random number.

The Adversary model section below details the modelling of adversary.

1.7.6 Authentication

"Authentication means that, if a participant (participant A) runs the protocol apparently with another participant (participant B), then B runs the protocol apparently with A, and conversely. In general, one also requires that A and B share the same values of the parameters of the protocol." [11]

In other words, authentication means that during a session of communication, if A participates in the session with B, then B participates in the session with A, and vice versa.

In the symbolic model, authentication is modelled by event correspondence : the condition that occurance of an event corresponds to a prior occurance of another event. Using the query used in Query system `query event(B) ==> event(A)`, it translates to showing that if event B has occured(or has been executed), then event A must have occured prior as well.

In the computational model, the notion is captured similarly.

1.7.7 Adversary model

Network capability

In both ProVerif and CryptoVerif, the modelled adversary is capable of intercepting any messages sent out from any process, and is also capable of manipulating the traffic i.e. the adversary may modify/corrupt the messages, send traffic out of order, send the traffic intact, and may even drop the messages entirely.

The implication under this premise is that encoding of protocol processes cannot rely on any network properties concretely, and thus any properties proven by the prover implicitly hold under all forms of network attacks i.e. all the attacks can be made but the adversary still cannot violate the secrecy properies or authentication properties if proven.

It should be intuitive that this is a close approximation to real life scenarios, in which one cannot make any assumptions regarding the network in which the data is sent out.

Computational capability

In the symbolic model, the adversary is capable of all computations that the clients are capable of, but is limited to the ones defined in the code files. In other words, if hash is defined, then both the clients and the adversary are capable of hashing any information. However, if encryption is not defined, then neither clients nor adversary may attempt to encrypt or decrypt any information.

In the computational model, the adversary is a probabilistic polynomial-time Turing machine. Probabilistic refers to the condition that adversary is non-deterministic - it may use random numbers as part of its computation. Polynomial-time refers to the condition that the adversary is limited to polynomial time computations, this is to model the fact that we are only interested in "efficient" adversary. Adversaries which are able to break a security property in 100 years(or whatever time lengthier than the protocol's lifespan) are not useful entities to be taken into account in the model.

1.7.8 Examples

Below are code snippets of some small CryptoVerif code to show how different simple scenarios may be encoded roughly.

Authentication via signature using preshared secrets

```
query msg : block, s : signature; event B1(msg, s) ==> A1(msg, s).
query                             event B2 ==> A2.
query s : signature;              event B3(s) ==> A3(s).
query msg : block;                event B4(msg) ==> A4(msg).

let processA =
  in (cA1, ());
  new r1 : seed;
  let s : signature = sign(msg, skA, r1) in
  event A1(msg, s);
  event A2;
  event A3(s);
  event A4(msg);
  out (cA2, (msg, s)).

let processB =
  in (cB1, (msg_AtoB : block, s : signature));
```

```
  if check(msg_AtoB, pkA, s) then
  if msg_AtoB = msg1 then (
    event B1(msg_AtoB, s);
    event B2;
    event B3(s);
    event B4(msg_AtoB)
  ).

process
  in(start, ());
  new rkA : keyseed;
  let pkA = pkgen(rkA) in
  let skA = skgen(rkA) in
  new msg : block;
  out(end, ());
  (processA | processB)
```

The full code is stored as `artefacts/ZRTP_in_CryptoVerif/examples/sig.cv`

Authentication via HMAC using preshared secrets

```
query event B1 ==> A1.
query msg:block; event B2(msg) ==> A2(msg).

let processA =
  in (cA1, ());
  new msg_A : block;
  let hmac_msg = hmac(msg_A, shared_key) in
  event A1;
  event A2(msg_A);
  out (cA2, (msg_A, hmac_msg)).

let processB =
  in (cB1, (msg_A : block, hmac_msg : block));
  if checkhmac(msg_A, shared_key, hmac_msg) then (
    event B1;
    event B2(msg_A)
  ).

process
  in(start, ());
  new keyseed : mkeyseed;
  let shared_key = mkgen(keyseed) in
  new msg : block;
  out(end, ());
  (processA | processB)
```

The full code is stored as `artefacts/ZRTP_in_CryptoVerif/examples/hmac.cv`

Above examples are of original work, more examples from official CryptoVerif project can be see in the folder
`artefacts/ZRTP_in_CryptoVerif/official_examples`

2 Project

2.1 Studying of ZRTP specification

This is the first stage of the project. The purpose of this stage is to provide background information of the ZRTP protocol. The result of this stage is the writing of the Background section above.

2.2 Review and cleanup of ProVerif encoding of ZRTP

The ProVerif encoding of ZRTP done by Bresciani, Butterfield [12] is reviewed and translated to typed pi-calculus.

The original untyped ProVerif encoding of ZRTP as attached in the appendix of the above paper is stored as `artefacts/ZRTP_in_ProVerif/zrtp_orig.pv` (with reformatting for better readability).

As process calculus of CryptoVerif is typed, the untyped pi-calculus encoding is translated to typed pi-calculus for easier comparison in later stages. The translated version is stored as `artefacts/ZRTP_in_ProVerif/zrtp.pv`.

2.3 First encoding of ZRTP in CryptoVerif

The ZRTP encoding in typed pi-calculus and the ZRTP specification [5] are used as basis for the CryptoVerif encoding.

The typed pi-calculus encoding is reviewed thoroughly in this stage with respect to the official specification. Some minor errors of the encoding were discovered(as listed below), the corrected version is stored as `artefacts/ZRTP_in_ProVerif/zrtp_corrected.pv`

Line(s) in zrtp.pv	Line(s) of fixes in zrtp_corrected.pv	Description	Page(s) in ZRTP specification
141 - 142	140 - 143	HMAC should take encrypted part of the message as input rather than unencrypted	63 - 64
205	206	HMAC should take encrypted part of the message as input rather than unencrypted	63 - 64

The corrected version produces the same result, this can be verified by executing
`proverif artefacts/ZRTP_in_ProVerif/zrtp.pv` and
`proverif artefacts/ZRTP_in_ProVerif/zrtp_corrected.pv`

The partial output are shown below :

```
proverif proverif artefacts/ZRTP_in_ProVerif/zrtp.pv
      -- Query not attacker(SECR[])
      Completing...
      200 rules inserted. The rule base contains 114 rules. 14 rules in the queue.
      400 rules inserted. The rule base contains 148 rules. 62 rules in the queue.
      600 rules inserted. The rule base contains 177 rules. 72 rules in the queue.
      800 rules inserted. The rule base contains 213 rules. 90 rules in the queue.
      1000 rules inserted. The rule base contains 317 rules. 4 rules in the queue.
      Starting query not attacker(SECR[])
      RESULT not attacker(SECR[]) is true.
      -- Query not attacker(SECI[])
      Completing...
      200 rules inserted. The rule base contains 114 rules. 14 rules in the queue.
      400 rules inserted. The rule base contains 148 rules. 62 rules in the queue.
      600 rules inserted. The rule base contains 177 rules. 72 rules in the queue.
      800 rules inserted. The rule base contains 213 rules. 90 rules in the queue.
      1000 rules inserted. The rule base contains 317 rules. 4 rules in the queue.
      Starting query not attacker(SECI[])
      RESULT not attacker(SECI[]) is true.
```

```
proverif proverif artefacts/ZRTP_in_ProVerif/zrtp_corrected.pv
      -- Query not attacker(SECR[])
      Completing...
      200 rules inserted. The rule base contains 114 rules. 14 rules in the queue.
      400 rules inserted. The rule base contains 148 rules. 62 rules in the queue.
      600 rules inserted. The rule base contains 177 rules. 72 rules in the queue.
      800 rules inserted. The rule base contains 213 rules. 90 rules in the queue.
      1000 rules inserted. The rule base contains 317 rules. 4 rules in the queue.
      Starting query not attacker(SECR[])
      RESULT not attacker(SECR[]) is true.
      -- Query not attacker(SECI[])
      Completing...
      200 rules inserted. The rule base contains 114 rules. 14 rules in the queue.
      400 rules inserted. The rule base contains 148 rules. 62 rules in the queue.
      600 rules inserted. The rule base contains 177 rules. 72 rules in the queue.
      800 rules inserted. The rule base contains 213 rules. 90 rules in the queue.
      1000 rules inserted. The rule base contains 317 rules. 4 rules in the queue.
      Starting query not attacker(SECI[])
      RESULT not attacker(SECI[]) is true.
```

The full output are stored as
`artefacts/ZRTP_in_ProVerif/zrtp_output` and
`artefacts/ZRTP_in_ProVerif/zrtp_corrected_output`

The first version of ZRTP encoding in CryptoVerif is stored as
`artefacts/ZRTP_in_CryptoVerif/zrtp_1st_try.cv`

The encoding overall is similar to the ProVerif encoding, but a different direction was chosen. ProVerif encoding reasons with the Diffie-Hellman mode of ZRTP with cached secrets(SAS is not used, and both parties do not start with zero knowledge). CryptoVerif encoding attempts to reason with Diffie-Hellman mode of ZRTP during its first run(when both parties have zero knowledge, and SAS is required to complete the authentication as explained in Background section).

This direction is chosen because the author of this report deems the ability of authenticate in the first run to be a unique trait of ZRTP, since many protocols are already capable of providing authentication with prior shared secrets, and lack of reliance on such premise is not often seen.

The verbal SAS verification(where the two users read out the SAS vocally and check if they match up) is modelled by sending the encrypted and signed SAS to both sides after the completion of the other stages. The key materials used for signing and encryption of the SAS are only used for this part of modelling.

The above model is a simplistic and "perfect" model - since the key materials are kept secretive and shared internally only, there are no attack surfaces. A perfect model is used since in the specification, it is indicated that the SAS verification should be unforgeable and is assumed to be correct(the human users are able to check correctly whether the string matches). This turned out to be problematic as shown in the Results and discussion section.

Below is a code snippet of the SAS verification modelling of the Initiator side, the Responder side is modelled similarly :

```
      (* Encrypt then sign sas value *)
      new seed2 : encseed;
      let encrypted_sasvali = encrypt(sasvali, SAS_enc_key, seed2) in
      new r1 : seed;
      let encrypted_sasvali_sig = sign (encrypted_sasvali, SAS_sign_skI, r1) in
```

```
(* I -> R : Encrypted and signed SAS *)
out (cI13, (encrypted_sasvali, encrypted_sasvali_sig));

(* R -> I : Encrypted and signed SAS *)
in (cI14, (encrypted_sasvalr     : block,
           encrypted_sasvalr_sig : signature));

(* confirm SAS signature *)
if check(encrypted_sasvalr, SAS_sign_pkR, encrypted_sasvalr_sig) then (
  (* pattern match to successful decryption *)
  let injbot(sasvalr_RtoI) = decrypt(encrypted_sasvalr, SAS_enc_key) in (
    (* check that the received SAS matches *)
    if sasvalr_RtoI = sasvali then (
      event endI
    )
  )
)
```

The first version of encoding fails to terminate when executed with CryptoVerif successfully(exits with stack overflow error/exception), several individual of ZRTP are implemented separately in CryptoVerif to identify the issue, this is detailed in Taking ZRTP apart section.

Partial output is shown below (with manual text wrapping):

```
cryptoverif zrtp_1st_try.cv
      Proof of (one-session) secrecy of s0_R failed:
        s0_R is not defined only by restrictions or assignments.
      Proof of (one-session) secrecy of s0_I failed:
        s0_I is not defined only by restrictions or assignments.
      Proof of (one-session) secrecy of sasvali failed:
        sasvali is not defined only by restrictions or assignments.
      Proof of (one-session) secrecy of sasvalr failed:
        sasvalr is not defined only by restrictions or assignments.
      Trying equivalence uf_cma(sign) ... Failed.
      Trying equivalence rom(hash) ... Transf. OK Transf. done SA renaming...
      Simplifying...
      Eliminated collisions between ZIDr_273 and ZIDi_246 Probability: 1. / |block|
      Eliminated collisions between H0I and H0R Probability: 1. / |block|
      Expanding and simplifying... Internal error: Stack overflow
      Please report bug to Bruno.Blanchet@inria.fr, including input file and output
```

The full output is stored as `artefacts/ZRTP_in_CryptoVerif/zrtp_1st_try_output`

2.4 Taking ZRTP apart

Several basic components used in the CryptoVerif encoding are implemented in simplest form for testing in this stage. This includes HMAC, symmetric encryption, and hash images. And the issue was identified to be related to hash images.

The problematic encoding of hash images is stored as
`artefacts/ZRTP_in_CryptoVerif/hash_image_stuck.cv`.
Executing `cryptoverif hash_image_stuck.cv` did not terminate for an unreasonably long time and thus was concluded this is abnormal(ProVerif/CryptoVerif processing should not take more than a few minutes).

Full captured output is stored as `artefacts/ZRTP_in_CryptoVerif/hash_image_stuck_output`

The fix applied was to change hash from using the `ROM_hash` to define hash to simply

defining hash symbolically, similar to defintion in ProVerif.

Specifically, the change was

from	`expand ROM_hash(hashkey, block, block, hash).`	(line 14 of hash_image_stuck.cv)
to	`fun hash (hashkey, block) : block [compos].`	(line 14 of hash_image_okay.cv)

Executing `cryptoverif hash_image_okay.cv` promptly terminates, but since it does not contain useful output, the output is not attached here.

The full output is stored as `artefacts/ZRTP_in_CryptoVerif/hash_image_okay_output`.

In summary, CryptoVerif takes a very long time to terminate when chained usage of hash(defined using ROM_hash) occurs.

2.5 Second encoding of ZRTP in CryptoVerif

Using the result as described in above section, we change the hash functions used for hash images from one defined using ROM_hash to a symbolic version, the specific changes are shown below.

Line number in `zrtp_2nd_try.cv`	Description
41	Added `hash_symoblic` definition
147 - 149	Replaced `hash` with `hash_symbolic`
303 - 305	Replaced `hash` with `hash_symbolic`

The changes allowed the command `cryptoverif zrtp_2nd_try.cv` to terminate promptly.

The partial output is shown below (with manual text wrapping):

```
RESULT Proved event endR ==> startI up to probability Psigncoll
+ Psign(time(context for game 7) + time + time(check), 1.)
... (ignored)
RESULT Proved event endI ==> startR up to probability
Psign(time(context for game 11) + time, 1.) + Psigncoll
+ Psign(time(context for game 7) + time + time(check), 1.)
... (ignored)
RESULT Could not prove secrecy of sasvalr,
secrecy of sasvali, secrecy of s0_I, secrecy of s0_R.
```

The full output is stored as `artefacts/ZRTP_in_CryptoVerif/zrtp_2nd_try_output`

Summary
After the changes, `cryptoverif zrtp_2nd_try.cv` terminates promptly, and the proof result shows that CryptoVerif succeeded in proving authentication, but failed to show secrecy of key materials. However, the successful part of the proof is not useful, this is explained in the Results and discussion section.

2.6 Results and discussion

Below is the partial output from the above section but with number markers on the right:

```
RESULT Proved event endR ==> startI up to probability Psigncoll  (1)
+ Psign(time(context for game 7) + time + time(check), 1.)
... (ignored)
RESULT Proved event endI ==> startR up to probability            (2)
Psign(time(context for game 11) + time, 1.) + Psigncoll
+ Psign(time(context for game 7) + time + time(check), 1.)
... (ignored)
RESULT Could not prove secrecy of sasvalr,                       (3)
secrecy of sasvali, secrecy of s0_I, secrecy of s0_R.
```

From `(3)`, we can see CryptoVerif failed to show secrecy of the shared secrets(s0_I, s0_R), and secrecy of the SAS values(sasvali, sasvalr) which are derived from s0_I and s0_R.

From `(1)` and `(2)`, we can see CryptoVerif successfully showed that the authentication property holds, however, it is bounded by probability of breakage related to the "perfect" modelling of SAS verification, and completely failed to reason with probability of attack in relation to length of SAS.

Overall, the CryptoVerif code failed to address the original intention entirely, which is to show that hash commitment along SAS provides high level of security even with very short SAS length.

2.7 Conclusion and reflection

Overall the author of this report underestimated the difficulty of modelling the first run of ZRTP, and computed an unintended proof result due to an error in modelling. The progress was also hindered by the fact that the author's understanding of secrecy lies much closer to syntactic secrecy than secrecy defined in the computational model. Personally this serves as a useful reminder that what would seem intuitive may very well be completely incorrect when put under the scrutiny of mechanical provers. The experience also illustrates that cryptographic proofs require much higher level of care and expertise compared to formal verification of software in general.

The very original scope of this project was to specify ZRTP in Coq(an interactive theorem prover), and prove ZRTP's useful properties in Coq and then generate a proven implementation in Haskell/OCaml via the code extraction mechanism in Coq.

However, since this is a single semester project(6 units), my supervisor deemed it to be too demanding in terms of workload and with limited usefulness, and concluded that familiarising oneself with existing specialised tools would prove to be more beneficial than struggling with building up a much larger foundation.

Facing this circumstance, the direction of the project was subsequently changed to focus on ProVerif and Tamarin, both of which are provers based on the symbolic model, with the premise that I was to construct the ProVerif and Tamarin encoding purely from ZRTP specification, and then compare the two encodings.

But as the project progressed, with the discovery that ProVerif encoding of ZRTP were already done by others, and also with the stall of progress of studying the ZRTP specification, as I initially had difficulty navigating the specification documents, my supervisor advised to further reduce the scope by using the existing ProVerif encoding, and only construct Tamarin encoding of ZRTP as the original work.

However, I later discovered that Tamarin encoding of ZRTP was also completed by another group already[13], with which I seeked approval from my supervisor to use the existing Tamarin encoding, and change the original component to being the CryptoVerif encoding of ZRTP.

The choice of using CryptoVerif as a replacement turned out to be somewhat a mistake, as it required much deeper knowledge in cryptography, and this in turn has made setbacks of my progress especially in later stages of the project. I subsequently had to notify my supervisor of my intention of dropping my analyis of Tamarin both due to time limitation as well as the langauge complexity of Tamarin.

As a result, I overall had to do more readings than planned, but also produced much less original work than originally anticipated and hoped. But since I had no prior experience with cryptography related work, this was still an extremely meaningful exploration of the area.

2.8 Limitations

This project did not explore the manual mode of CryptoVerif, which might have been able to provide more useful results.

Models are abstraction of the actual scenarios. Neither ProVerif nor CryptoVerif take side channel attacks into account, such as physical attacks which steal keys from mobile or computers [14][15][16].

Furthermore, being an abstraction, certain assumptions regarding security properties of the underlying system need to be made, e.g. operating system and language runtime would be functioning correctly. Implementation extraction in CryptoVerif also assumes the cryptographic library the generated implementation relies on, cryptokit, is to be trusted.

Lastly, CryptoVerif are implemented in uncertified OCaml code(need to trust 42000 lines of OCaml code) [17], this may be a shortcoming compared to EasyCrypt, which is implemented in Coq(and thus certified).

2.9 Future directions

Immediate extensions to this project would be to explore possibility of proving SAS verification is sufficient in other provers such as Coq, which in turn may require encoding of ZRTP in frameworks like EasyCrypt.

On the other hand, for direct comparison to ProVerif code, the CryptoVerif code can be modified to reason with second run and onward of ZRTP only. However, this seems to have limited usefulness as the secrecy property of first run has already been proven successfully in the ProVerif code, this direction only leads to somewhat duplicate work.

References

[1] *Diffie–hellman key exchange*, in. [Online]. Available: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange (visited on 08/31/2017).

[2] K. A. Baker, *Diffie-hellman key exchange*, 1999. [Online]. Available: http://www.math.ucla.edu/~baker/40/handouts/rev_DH/node1.html.

[3] M. Baugher, D. A. McGrew, M. Naslund, E. Carrara, and K. Norrman, *The secure real-time transport protocol (SRTP)*, Mar. 2004. [Online]. Available: https://tools.ietf.org/html/rfc3711.

[4] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, *RTP: A transport protocol for real-time applications*, Jul. 2003. [Online]. Available: https://tools.ietf.org/html/rfc3550.

[5] P. Zimmermann, A. Johnston, and J. Callas, *ZRTP: Media path key agreement for unicast secure RTP*, Apr. 2011. [Online]. Available: https://tools.ietf.org/html/rfc6189.

[6] H. Krawczyk, *Cryptographic extraction and key derivation: The HKDF scheme*, 2010. [Online]. Available: http://eprint.iacr.org/2010/264.

[7] K. Bartley, *What is a PBX phone system?*, Jul. 31, 2017. [Online]. Available: https://www.onsip.com/blog/what-is-pbx-phone-system.

[8] F. B. Schneider and T. Roeder, *Hashes and message digests*, 2005. [Online]. Available: http://www.cs.cornell.edu/courses/cs513/2005fa/NL20.hashing.html.

[9] I. Koren, *Introduction to cryptography ECE 597xx/697xx part 12 message authentication codes (MACs)*. [Online]. Available: http://euler.ecs.umass.edu/ece597/pdf/Crypto-Part12-MAC.pdf.

[10] B. Blanchet, *ProVerif automatic cryptographic protocol verifier user manual for untyped inputs*, Jul. 10, 2017.

[11] ——, "Security protocol verification: Symbolic and computational models", in *Principles of Security and Trust: First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, P. Degano and J. D. Guttman, Eds., DOI: 10.1007/978-3-642-28641-4_2, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–29, ISBN: 978-3-642-28641-4. [Online]. Available: https://doi.org/10.1007/978-3-642-28641-4_2.

[12] R. Bresciani, "The ZRTP protocol analysis on the diffie-hellman mode", Jun. 12, 2009.

[13] S. Lu, W. Wang, and Q. Cheng, "Formal analysis of the real-time multimedia network protocol ZRTP", *Recent Advances in Electrical & Electronic Engineering*, vol. 8, no. 1, pp. 12–17, May 2015, DOI: http://dx.doi.org/10.2174/2352096508666150317234447. DOI: 10.2174/2352096508666150317234447. (visited on 08/20/2017).

[14] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs", Feb. 9, 2016.

[15] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels", Mar. 1, 2016.

[16] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation", Feb. 27, 2015.

[17] B. Blanchet, "CryptoVerif: State of the art, perspectives, and relations to other tools", Apr. 2017.