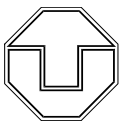
Modeling and verification of security protocols

Part I:
Basics of cryptography and introduction to security protocols

Advanced seminar paper



Dresden University of Technology
Martin Pitt
email: mp809054@inf.tu-dresden.de

November 13, 2002

Abstract

This advanced seminar is about the design, modeling and formal validation of security protocols. As the first presentation, this paper introduces the notions, explains the principles of cryptographic definitions, algorithms, and pitfalls and deals with some example protocols together with an overview of different validation techniques used today.

Contents

Preface

Security protocols are a critical element of the infrastructures needed for secure communication and processing information.

Most security protocols are extremely simple if only their length is considered. However, the properties they are supposed to ensure are extremely subtle, and therefore it is hard to get protocols correct just by informal reasoning and "eyeballing". The history of security protocols (and also cryptography) is full of examples, where weaknesses of supposedly correct protocols or algorithms were discovered even years later. In Roger Needham's remark 'they are three line programs that people still manage to get wrong'.

Thus, security protocols are excellent candidates for rigorous formal analysis. They are critical components of distributed security, are very easy to express and very difficult to evaluate by hand.

The seminar deals with formal analysis and validation of such protocols. This paper—being the first in the seminar—introduces the matter by dealing with some well known standard protocols and pointing out several formal analysis methods used today.

To design and/or verify security protocols, one needs to have at least a basic understanding of cryptography since encryption and authentication algorithms are the basic blocks of protocols. So this paper starts with a thorough introduction to the principles of cryptography, without going into the mathematical details.

Martin Pitt
November 2002

Acknowledgements

First of all, I want to thank Prof. Horst Reichel, who is the coordinator of this seminar, for his various interesting lectures I took until now. Due to him I learned that theoretical computer science is by far not that dry it is generally supposed to be.

I also want to express my appreciation to Prof. Andreas Pfitzmann for his lectures "Basics of data security" and "Cryptography". His subtly provoking and never boring style of presentation and his habit to let the students think and discuss first caused my interest in the matter and sharpened my sense for dealing with both the mathematics and the physical realization of computer and network security.

Chapter 1

Aspects of security

1.1 Security properties

When talking about "security", "secure systems", "authenticity", the meanings of these words are frequently taken as obvious and widely understood. But if you try to explain them in detail, you may find it remarkably difficult to be precise in every desired aspect. Different people even have different interpretations of some terms.

Thus, the first question somebody should ask when constructing a "security system" is: What exactly do I want to protect? So we start with informal definitions of different aspects of security, such that specifying required properties of a system (and also properties it need *not* have) can be done using these new terms.

1.1.1 Secrecy

Secrecy—also called concealment or confidentiality—has a number of different possible meanings; the designer of an application must decide which one is appropriate.

The strongest interpretation would be: an intruder is not able to learn *anything* about *any* communication between two participants of a system by observing or even tampering the communication lines. That is, he cannot deduce the contents of messages, sender and receivers, the message length, the time they were sent, and not even the fact that a message was sent in the first place.

In theory, this "perfect" secrecy can be approximated quite close by exploiting today's cryptographic tools: encryption and digital signatures, a lot of dummy traffic to keep a constant network load, anonymizing proxies to "shuffle" packets across the nodes to make routing analysis infeasible, and a huge calculation overhead on the message recipients, since decryption and signature checking must be performed with each possible key (since there is no sender or receiver name attached). So for most practical applications this would neither be efficient nor necessary.

A very weak interpretation, which is used in today's encrypted email communication, confines secrecy to the actual content of messages. This may be the most important part, but especially for very small messages it must be taken into consideration that an attacker learns something about the exact (if a stream cipher is used) or approximate (with a block cipher) length of the message. Additionally, depending on the controlled network area attackers can log a partial or complete traffic analysis. But this approach can be performed with no useless overhead.

These two extreme situations point out that the designer of a distributed system must analyze exactly which attributes of communication have to be concealed and which can go unprotected to allow a more efficient implementation.

1.1.2 Authentication

A system provides *strong* authentication if the following property is satisfied: if a recipient R receives a message claiming to be from a specific sender S then S has sent exactly this message to R.

For most applications this formulation must be weakened, since in most cases communication channels are subject to both technical errors and tampering by attackers. A system provides *weak* authentication if the following property is satisfied: if a recipient R receives a message claiming to be from a specific sender S then *either* S has sent exactly this message to R *or* R unconditionally notices this is not the case.

Some authors make a distinction to the two aspects of authentication: a validated sender name is referred to as "authentication of origin" and the fact that the message has not been altered in any way is called "integrity". But neither property alone increases the confidence to a message, so both must always be present.

An additional property authentication systems can have is *non-repudiation*. This property states that a recipient is not only confident that the message is authentic (sent by S and unmodified) but can also prove this fact to a third party. In analogy to handwriting signatures these systems are called *digital signature systems*.

1.1.3 Anonymity

A system that is anonymous over a set of users has the following property: when an user sends a message, then any observer will be unable to identify the user, although he might be able to detect the fact that a message was sent. Examples are an open discussion board or the prevention of traffic analysis on web servers.

Sometimes it is not intended that the messages can be sent completely independent from each other. In an electronic voting protocol, the property shall be constrained: every user can send exactly one message anonymously, but the protocol must ensure that double votes can not occur unnoticed. Either it is constructed in a way that it is impossible to vote twice or the offending user's identity is revealed in such a case.

1.1.4 Fairness

An application where fairness properties get important is electronic contract signing. A protocol that achieves this must secure that the process of applying all contractor signatures is transactional. That means, that e. g. the third contractor must not be able to halt the protocol in a way that the first two signatures are already valid.

There are several protocols available which have to trade off the need of a trusted third party against a lot of communications for incremental commitment.

1.1.5 Availability

This property may seem rather technical, but becomes vital in applications like distress signals, emergency telephones or remote surgery. If a certain service is needed, it must actually be available.

Of course, cryptography can do only little to achieve this. Neither mathematics nor protocols can help if an attacker just "cuts the wire". Solutions (which can never be perfect, though) comprise highly redundant networks, redundant nodes, or, in the case of alarms, absence triggers (the alarm is caused if a periodic "no alarm" sensor signal is not sent any more).

1.2 Attacker models

Now, that we can talk about *what* we want to protect, we need to specify *who* to protect it from.

Clearly it is impossible to protect something from an almighty attacker. Such an attacker could:

- gather all data he is interested in right at their origin, thus preventing confidentiality (secrecy)
- break integrity by unperceivedly modifying data, because if the user would always know which value they should have, he would not need them at all
- foil availability by destroying the whole system.

Thus, a system can not plainly claim to be "secure", but must instead specify against which type of attacker.

Since nobody can know what technology will be available in ten or fifty years, specifying the capabilities of attackers is by far not easy and always an estimation. Taking this into consideration it is only helpful to overly exaggerate at this point: an allegation to be "secure against an attacker who is bound by the currently known physical laws and has at most 10,000 times of today's globally available calculation capacity" is concrete and explicit and can—at least theoretically—be proved.

Another important point is the amount of control an attacker can potentially have on a system. Again, it is impossible to defend to an omnipotent attacker. Every kind of security needs a physical support; it is not possible to "create" security from scratch. According to the author's knowledge, it is currently not possible to read or influence higher functions of a human brain, so at least we can trust our own mind.

But in practical applications one would not want to perform all calculations and protocols just with our mind; usually machines are used for this purpose. But this extends the ultimate trust (the physical support) to all machines a participant uses to act in a system, thus confining the attacker's allowed area of influence. This statement may sound trivial, but this is the point where most systems are broken, since it is usually their weakest part. A popular example may be online banking: for an attacker it would be incomparably difficult to try to invest vast amounts of technology and time to break a sophisticated algorithm; if one examines the vast majority of today's home computers, it would certainly be enough to send to the victim an email that silently installs a Trojan horse that logs all passwords, transaction numbers and anything else the attacker is interested in.

An usual goal at this level would be: every participant has its own ultimately trusted device he uses to participate in a distributed system, and the complete network is subject to observation and modification by attackers. Some protocols may also need further constraints, but the important part is, that all assumptions and constrains are *explicitly* defined and stated.

1.3 Limits of cryptography and security protocols

Cryptographic science developed algorithms providing encryption and authentication. Some of these algorithms can formally be proved, others have been used for decades without revealing severe weaknesses. Out of these building blocks, security protocols can achieve goals like secret key exchange and fairness whose security can also be proved using the correctness of the cryptographic algorithms.

But all these considerations do apply to *mathematical level only*. When a security system is built, the algorithms get instantiated and refined in soft- and hardware which introduces many new aspects and side effects not covered by the abstract mathematical model.

As an example, chip cards are widely thought of being secure due to the inaccessibility of the hardware. But in fact, there were very successful attacks on these chip cards by measuring their power consumption that directly depends on its calculations. Similarly successful approaches include the measuring of execution times and the remote recording of electromagnetic emissions, which allow to reconstruct a monitor's display and keyboard activities from a distance of several meters. A third example are covert channels which can leak confident information to attackers.

Chapter 2

Principles of cryptographic algorithms

2.1 Keys and why they are needed

In every distributed system there must be something that distinguishes the legitimate recipient from all other participants. In cryptographic systems, this feature is the knowledge of a specific secret.

In earlier times it was common to negotiate a non–disclosed algorithm between the persons wanting to communicate securely. Even today it occurs from time to time. But this approach has several deficiencies:

- Most of these algorithms were cracked quite easily since conceiving a really secure one is very difficult
- If a third party (e. g. a cryptologist) was engaged to do this job, he was a potential risk afterwards. Due to this, many capable cryptologists "accidentally" disappeared from this world...
- There is no chance to analyze the algorithm thoroughly since every person that could do this would certainly suffer the same fate as in the last point.
- As the number of participants grow, it gets very hard to conceive enough algorithms.

We see that it is best to use just one algorithm that is publicly documented and available, and can thus be studied thoroughly, normed, and implemented in masses.

But then, such an algorithm needs to have a certain parameter (called "key") to provide the unique feature for the distinction of participants. These keys must have some important properties:

- Their creation must be based on a truly random number.
- The number of all possible different keys must be many magnitudes higher than the number of participants (a factor of 2^{100} is by no way too much). This prevents that two keys are accidentally identical and minimizes the possibility of a right guess.
- The relationship of a key to his owner cannot be secured by any technique, but must be verified by first-hand knowledge. With open algorithms, every person is able to create keys with any name attached to them!
- A basic maxim is that the whole system can be at most as good (or bad) as this initial key generation!

2.2 Symmetric systems

Up to the advent of asymmetric systems in 1976, all cryptography was symmetric. Their history is already several thousand years old; one of the earliest famous ciphers is the Caesar chiffre which works by rotating the letters of the plaintext by a previously agreed number of characters.

The most common systems used today are the one-time-pad concealment system (also known as Vernam Chiffre) and their counterpart, the authentication codes, the ubiquitous DES (Data Encryption Standard) and its designated successor AES (Advanced Encryption Standard). Detailed descriptions can be found in [Pf00].

2.2.1 Notational conventions and symbols

$\mathcal{X}$	set of all plaintext messages
$\mathcal{C}$	set of all ciphertexts
$\mathcal{S}$	set of all signatures
$\mathcal{K}$	set of all symmetric keys
$k_{AB} \in \mathcal{K}$	a specific key belonging to A and B

2.2.2 Working principle

Symmetric systems are characterized by the fact that encryption and decryption, or signing and testing, respectively, are performed with the same key.

Concealment

Algorithms:
$encrypt : \mathcal{X} \times \mathcal{K} \to \mathcal{C}$
$decrypt : \mathcal{C} \times \mathcal{K} \to \mathcal{X}$

Symmetry and injection condition:
$\forall k \in \mathcal{K}, x \in \mathcal{X}.\ decrypt\big(encrypt(x,k),k\big) = x$

Sending an encrypted message from A to B:

- encryption: A chooses a message $x \in \mathcal{X}$ and calculates $c = crypt(x, k_{AB})$
- transfer: c is now sent to the recipient (and possibly to observers and attackers)
- decryption: since—apart from A—only B knows k_{AB}, only he is able to calculate $x = decrypt(c, k_{AB})$

Authentication

Algorithm:
$sign : \mathcal{X} \times \mathcal{K} \to \mathcal{S}$

Sending a signed message from A to B:

- signing: A chooses a message $x \in \mathcal{X}$ and calculates $s = sign(x, k_{AB})$
- transfer: $x; s$ is now sent to the recipient (and possibly to attackers)
- receiving: B receives a message $x'; s'$ (either the original or modified by attackers)
- test: B now calculates $s'' = sign(x', k_{AB})$; if $s'' = s'$, the message is valid. Since—apart from A—only B knows k_{AB}, nobody can alter the message to x' and fake a correct signature s' for it.

2.2.3 Key distribution

Before two participants can use these algorithms, they must agree to a common symmetric key to use between them. If the participants are two humans and have the chance to meet each other to exchange a floppy disk or CD-ROM, then this process is uncritical.

Problems arise if the participants cannot meet. They may not know each other personally, live too far away from each other, or the participants are just machines. Then the support of a third party is needed which both participants trust. The exchange must be:

- secret: only the two participants (and possibly the trusted third party) must get to know the key values

- authentic: the protocol must ensure that the keys cannot be modified underway, or after receiving the keys, the participants must verify the equality afterwards.

A protocol that achieves this is the "Needham-Schroeder-Secret-Key" protocol (NSSK) which is described in section 3.2.2. This also solves the problem of key explosion: in a system with n participants, at most $n(n-1)$ keys are needed in the worst case (if everybody wants to communicate with everyone else and an one time pad is used), so the number of keys grows quadratically.

2.3 Asymmetric systems

The first paper that introduced this class of systems was published in 1976 by Diffie and Hellman. They described a protocol that enabled two participants to derive a common secret key solely from publicly available information; it is described in more detail in section 2.5. In 1978 another paper was published by Ronald Rivest, Adi Shamir and Leonard Adleman who described the famous RSA algorithm.

All asymmetric systems are based on a mathematical "one-way" function, i. e. an operation that can easily be done in one direction, but whose calculation of the reverse is (assumed to be) virtually intractable. Practically there are only two that are used and well-studied:

factorization: it is easy to take two different large prime numbers p and q and calculate their product $n = p \cdot q$. But in spite of several centuries of research, no efficient algorithm is known to calculate the two prime factors of n.

discrete logarithm: taking an exponent is easy in finite fields, i. e., given two numbers a, x, and a prime p it is easy to calculate $y = a^x \pmod p$. But on the other hand, there is no known algorithm that, given y, a, and p, can efficiently determine x.

But one must be aware of the fact, that these two are really just well-studied conjectures and cannot be proved or refuted by now.

These conjectures can be used to construct public key cryptographic systems. There, every new participant generates two keys: a secret one which he must never tell *anyone* else, and a public key that can and should be spread as widely as possible. The public key can be used to encrypt messages or test their authenticity, whereas decryption or signing can only be done with the secret key.

The advent of such systems can be regarded as the breakthrough of "cryptography for the masses", since almost all problems of symmetric key exchange (but the validation of the relation owner – key) fell away.

The first publicly available open source program for asymmetric cryptography was Phil Zimmermann's "Pretty good privacy" (PGP), whose newer versions unfortunately went commercial today. The open source counterpart is the "GNU privacy guard" (GPG) which is mostly compatible with PGP and open source, so its correctness can be verified by everyone.

2.3.1 Notational conventions and symbols

$\mathcal{X}$	set of all plaintext messages
$\mathcal{C}$	set of all ciphertexts
$\mathcal{S}$	set of all signatures
$\mathcal{PUB}$	set of all public keys
$\mathcal{SEC}$	set of all secret keys
$pub_A \in \mathcal{PUB}$	specific public key of participant A
$sec_A \in \mathcal{SEC}$	specific secret key of participant A

2.3.2 Working principle

Concealment

Algorithms:
$encrypt : \mathcal{X} \times \mathcal{PUB} \to \mathcal{C}$
$decrypt : \mathcal{C} \times \mathcal{SEC} \to \mathcal{X}$

Injection condition:
$\forall x \in \mathcal{X}.\ decrypt\big(encrypt(x, pub_A), sec_A\big) = x$

Sending an encrypted message from A to B:

- encryption: A chooses a message $x \in \mathcal{X}$ and calculates $c = encrypt(x, pub_B)$
- transfer: c is now sent to the recipient (and possibly to observers and attackers)
- decryption: since only B knows sec_B, only he is able to calculate $x = decrypt(c, sec_B)$

Authentication

Algorithm:
$sign : \mathcal{X} \times \mathcal{SEC} \to \mathcal{S}$
$test : \mathcal{X} \times \mathcal{S} \times \mathcal{PUB} \to \{\text{correct}, \text{wrong}\}$

Creating a signed message by A:

- signing: A chooses a message $x \in \mathcal{X}$ and calculates $s = sign(x, sec_A)$
- transfer: $x; s$ is now sent to all desired recipients (and possibly to attackers)
- receiving: a participant B receives a message $x'; s'$ (either the original or modified by attackers)
- test: B now checks if $test(x', s', pub_A) =$ correct. Since only A knows sec_A, nobody can alter the message to x' and fake a correct signature s' for it.

Note that now not only one receiver can check signatures, but *everyone* knowing A's public key (which anyone can get, though). Thus, asymmetric authentication provides a digital signature system, since it has the property of non-repudiation.

2.3.3 Disadvantages

If public key cryptography has so many advantages, then why not forget about those "old" symmetric systems? Unfortunately asymmetric systems also have some shortcomings which still (and will forever) justify the existence of the symmetric ones:

- They require extensive mathematical calculations, about 10^3 to 10^5 more than symmetric systems. So they are inapplicable on small embedded systems that have very limited memory, calculation speed and/or power supply.
- Given enough calculation capacity they can be cracked easily even without intercepting a single "real" message: since an attacker knows a public key, he can encrypt as many messages he wants and search the private key that decrypts it back. A similar approach works for authentication.
- The truth of the used conjectures cannot be proved by now. To the contrary, the development of new and faster algorithms went much faster than the development of computation power.

2.4 Cryptographic hash functions

Generally, a hash function maps an input of arbitrary length to an output of fixed length. In cryptography, one application is the reduction of calculation needed for digital signatures: instead of signing the whole message (which can be several MB or even GB long, if binary files such as videos or programs are to be signed), the message's hash code is calculated first and this hash code gets signed then.

Widely known and easy hash functions are parity bits and CRC sums. They are well-suited if they only need to detect accidental corruption due to technical flaws or noisy communication

channels. But since it is very easy to modify a message in a way that it has the same parity/CRC value as the original, they cannot be used in cryptography.

Hash functions that have the property that it is intractably difficult to find two messages that produce the same hash code, and thus it is even more difficult to find another message to a given hash value, are called "collision resistant". Detailed examples and proofs can be found in [Pf00].

2.5 Diffie-Hellman key exchange

Since the understanding of this method is necessary for dealing with some protocols, the idea is described in brief. As already explained earlier, it enables two participants to calculate a common symmetric secret key solely from public information, and of course, their own knowledge. The algorithm exploits the discrete logarithm assumption and the commutativity of exponentiation.

All calculations are made in a finite field. A modulus p and a primitive Element a are publicly known and can be the same for all participants. (A primitive element is an element that generates the whole field by successive exponentiation.)

The participants X and Y randomly choose their own secret keys x, and y respectively. Because of the discrete logarithm assumption, the values a^x and a^y can be published as public keys.

Now the participants can calculate the shared secret key $k = (a^x)^y = (a^y)^x = a^{xy}$.

The security of the scheme can be enhanced by having each participant choose his own p and a (which then get part of their public keys). This defeats the most powerful attacks to this method, which involve pre-calculated large lookup tables for speeding up logarithm calculations.

Chapter 3

Security protocols

Now it is time to use the building blocks established in the previous chapters to construct protocols using them.

The meaning of "protocol" is taken as usual: a prescribed sequence of interactions between entities designed to achieve a certain goal and end. Security protocols in particular shall provide security properties of distributed systems, that were already described in section 1.1.

3.1 Notation of protocols

A pretty compact and easy-to-write form that is widely used is the list notation. A protocol is formulated as a sequence of messages, together with the names of the sender and receivers:

Message n $\quad a \rightarrow b : data$

The message content *data* can be composed of:

atoms: This may be names, variables and literal constants.

nonces: A nonce, usually notated like n_A, is an unpredictable, freshly generated unique number. The subscript indicates which participant created it, but that is just a notational convenience. In the real protocol there is no attached name tag or something similar that indicates its creator.

encryption: The term $\{data\}_k$ denotes the encryption of *data* with the key k.

authentication: $Sign_k(data)$ denotes the signature of *data* using the key k.

concatenation: $a.b$ denotes the concatenation of a and b, i. e. the two terms are sent consecutively.

3.2 Example protocols

In the following, three protocols are described which are widely used in practical systems today. But note, that no protocol is completely secure against all types of attacks presented in section 3.3. Finding such attacks is the purpose of formal validation, which the subsequent presentations are designated for.

3.2.1 Challenge-Response

This protocol has the purpose of verifying that two parties A and B share a common secret key k without revealing it. It is commonly after a key exchange to assure that the keys were not modified either accidentally or by an attacker:

1. A→B: n_A
2. B→A: $\{n_A\}_k.n_B$
3. A→B: $\{n_B\}_k$

After receiving the first message, B encrypts the nonce with his version of k and sends it back. Now A can decrypt it again and compare the result to the number he originally sent. If they match, then under the assumption that k is not known to any attacker, A can be sure that B has the same k as he has. The challenging is then performed the other way round for convincing B of the fact.

3.2.2 Needham-Schroeder Secret Key

The NSSK protocol is one of the earliest protocols that enables two participants to establish a common secret key using only symmetric cryptography and a trusted third party. It is the basis of the well-known Kerberos authentication and—apart from a subtle vulnerability that is discussed in the succeeding presentations—stood the test of time.

As a preliminary, all participants X share pairwise distinct secret keys SX with a central trusted party S (server). Apart from this, no other keys need to be stored permanently. This scenario stems the key explosion problem and makes adding and deleting participants easy.

If two participants A and B want to communicate securely with each other, they must first establish a common secret session key k_{AB} between them using the following protocol:

1. A→S: $A.B.n_A$
2. S→A: $\{n_A.B.k_{AB}.\{k_{AB}.A\}_{SB}\}_{SA}$
3. A→B: $\{k_{AB}.A\}_{SB}$
4. B→A: $\{n_B\}_{k_{AB}}$
5. A→B: $\{n_B - 1\}_{k_{AB}}$

A step-by-step walk-through follows:

1. A tells S that it wants to talk with B and supplies a nonce. Note that this information is not concealed, so it is both subject to observation and modification.
2. S now generates a fresh session key k_{AB} for A and B and answers with protocol step 2.
3. A can decrypt this answer using its server key, obtaining $n_A.B.k_{AB}.\{k_{AB}.A\}_{SB}$. A should verify that n_A and B match the values from step 1 to preclude modifications. The last part that A cannot decrypt is forwarded to B.
4. B can decrypt the data using his server key, obtaining the session key and the partner's name.
5. Step 4 and 5 form a simplified challenge response authentication to verify the integrity and equality of k_{AB}.

NB! This protocol assumes that encrypting a concatenation of elements together has the effect of binding them together, i. e. that encrypted data can not deliberately modified in a way that only a specific part (e. h. a participant name) is affected. This is an integrity property which encryption algorithms are not designed for in the first place! It works if a block cipher like DES is used in conjunction with an appropriate operating mode (like cipher block chaining). But a counterexample which fails spectacularly is the one time pad: although its secrecy is absolute, every single bit can be modified independently.

3.2.3 Station-To-Station

This protocol also provides the establishment of a common secret key between two participants, but without the need of a trusted third party. It is based on Diffie-Hellman key exchange (see section 2.5 on p. 10), but uses a (previously established) signature system for authenticating the public keys:

1. A→B: a^x
2. B→A: $a^y.\{Sign_B(a^y.a^x)\}_k$
3. B→A: $\{Sign_A(a^x.a^y)\}_k$

A short explanation follows:

1. A chooses a random x and computes $a^x \pmod p$, which is sent to B.
2. B chooses his own random y and can compute the common secret key $k = (a^x)^y = a^{xy}$ now. The data according to protocol step 2 is now sent back to A.
3. A has received a^y and thus can calculate $k = (a^y)^x = a^{xy}$ himself. But until now there is no guarantee that the messages were authentic.

   Using the freshly generated k A can decrypt the second part of the message to obtain B's signature of (his version of!) the public keys. This signature can now be checked against A's version of a^x and a^y.

   Now A sends back the signature of the public keys to B, so that B can verify their integrity on his own.

3.3 Examples of vulnerabilities and attack types

3.3.1 Man in the middle

This style of attack involves the intruder I imposing himself between the communications between two participants A and B. This can always happen if the messages, or even worse, the keys are not properly authenticated. The attack enables I to masquerade as B to A (or vice versa).

As a first example, lets consider a rather naive protocol for "secure" asymmetric communication between A and B where no participant would even need to know the other's public key. This works for asymmetric ciphers that are commutative (like RSA):

1. $A \to B$: $\{X\}_{p_A}$ (message X encrypted with A's public key)
2. $B \to A$: $\{\{X\}_{p_A}\}_{p_B}$ (B cannot decrypt the message, but further encrypt it with his own public key)
3. Due to the commutativity $\{\{X\}_{p_A}\}_{p_B} = \{\{X\}_{p_B}\}_{p_A}$, A can strip off his own encryption and send back
   $A \to B$: $\{X\}_{p_B}$ (B is now able to decrypt it)

All encrypted messages look pretty much alike: equally distributed random numbers, so the problem is that A would never know whether it was really B who received and further encrypted the message. If an I intercepts the first message, he can equally well apply his own public key and send back $\{\{X\}_{p_A}\}_{p_I}$ to A. A will duly strip off his own encryption and sends back $\{X\}_{p_I}$ to I which he can decrypt with ease.

If I wants to, he can perform the same protocol with B now to let B also receive the message. In this case, neither A nor B recognize the eavesdropper in between (apart from a slight time discrepancy that they probably will ignore).

This rather stupid protocol is really just an academic example to demonstrate the principle. In practical applications the initial key exchange is the most susceptible part to this attack. Their authenticity must always base on first-hand knowledge and cannot be gained with any clever protocol.

The initial key exchange plays the role of the "physical support" every security must be based on. From time to time there are people exchanging public keys via email. Every attacker is able to generate keys with any name tags attached to them and send them to anyone, claiming to be a friend. This is the point where man-in-the-middle attacks get successful.

3.3.2 Mirror

This attack, which is also known as "reflection", has its name from the trick to let a participant answer his own questions. The challenge-response protocol described above has such a weakness.

Suppose, there is a server S (e. g. a mainframe) which several clients (terminals) can log on. Authentication is done with challenge-response to avoid sending passwords through the possibly observed connection and also provide the authentication of the server.

If the system permits several parallel login procedures (which may not be unusual on big servers), an attacker A can fool the server with a second "dummy" session A':

1.	$A \to S$:	n_A	(first login request)
2.	$S \to A$:	$\{n_A\}_k.n_S$	(attacker cannot encrypt n_S by now)
3.	$A' \to S$:	n_S	(second "dummy" login request)
4.	$S \to A'$:	$\{n_S\}_k.n'_S$	(attacker got encrypted n_S and abandons this session)
5.	$A \to S$:	$\{n_S\}_k$	(taken from the last answer)

Solutions would be to make the login protocol atomic (no interleaved parallel logins) or to refuse encrypting nonces that was just sent out for client authentication. It does *not* help to let the client authenticate itself first, since then an attacker could authenticate itself as the server using the same trick.

3.3.3 Interleave

Here the attacker uses several parallel runs of a protocol to exploit their interactions. An example is the Needham-Schroeder public key protocol, which was believed to provide mutual authentication and secret exchange of two nonces (which could be used as symmetric session keys):

1.	$A \to B$:	$\{A.n_A\}_{p_B}$
2.	$B \to A$:	$\{n_A.n_B\}_{p_A}$
3.	$A \to B$:	$\{n_B\}_{p_B}$

The protocol was even analyzed with BAN logic (see next chapter) and thus has actually been believed to be secure for many years. But then the following interleave attack has been discovered (after which it was found that this attack fell outside the assumptions of BAN logic):

a.1.	A→I:	$\{A.n_A\}_{p_I}$	(A innocently starts the run with I)
b.1.	I(A)→B:	$\{A.n_A\}_{p_B}$	(I decrypts the message, encrypts it again for B and forwards it to him)
b.2.	B→I(A):	$\{n_A.n_B\}_{p_A}$	(Thinking that he talks with A, B answers properly)
a.2.	I→A:	$\{n_A.n_B\}_{p_A}$	(I cannot decrypt this, but forwards it to A; it is exactly what A expects, since A cannot determine who actually created n_B!)
a.3.	A→I:	$\{n_B\}_{p_I}$	(A answers duly, so I gets to know n_B)
b.3.	I(A)→B:	$\{n_B\}_{p_B}$	(now also B gets the expected response)

a and b indicate the two instances of the protocol; $I(A)$ means that I sends the message, but making it appear to come from A.

Note, that in this attack the intruder plays an active role, i. e. at the start A really *intends* to communicate with I, but I does not obey to the protocol.

At the end of the attack, I knows both nonces and caused the following mismatch in A's and B's perception: A assumes that he exclusively shares the knowledge of n_A and n_B with I, whereas B assumes that he ran the protocol with A and exclusively shares the knowledge only with him.

3.3.4 Replay

Here the attacker monitors a (possibly partial) run of the protocol and later replays some messages. This can happen if the protocol does not have any mechanism for distinguishing between separate runs or cannot determine the freshness of messages.

As an example, suppose a military ship which gets its commands from a base using an encrypted protocol. An attacker must not be able to crack any cipher, he must not even now what protocol is used. If he observes the communication and the ship for a while, then after some time he will have a fairly rich table mapping a certain dialog to the ship's reaction (turns, speed corrections and the like). This enables him to control the ship just be repeating a desired dialog.

This style of attack can be foiled with devices like nonces, run identifiers, timestamps, and indeterministic encryption.

3.3.5 Algebraic

Aside from purely "logical" attacks described above which exploit a weakness in the protocol itself it must not be forgotten that it may also be possible to break the underlying cryptographic algorithms. Almost every known algorithm has one or more algebraic identities (which may be already known or—worse—not yet discovered) which are not necessary for the algorithm's purpose but are a consequence of the mathematical structure. There are known cases of "unfortunate" combinations of such an identity with an otherwise secure protocol that lead to a completely flawed system.

Chapter 4

Formal approaches

The problem to establish secure session keys between two participants of a distributed system is so fundamental that it has led to a great deal of research. This, in turn, has led to a greater and even more sophisticated problem: the formal analysis of security protocols, although most of the work is done on authentication and key exchange.

People have found flaws in seemingly secure protocols even years after they were proposed. Section 3.3 showed some examples. This situation is highly unsatisfying; protocol designers wanted tools that could prove a protocol's security from the start.

Today, after roughly 20 years of research, there are four basic approaches to cryptographic protocol analysis (see [Schn96], [Pa02]), which are described here in brief. But some general considerations must be done before.

4.1 What formal verification can and cannot do

Formal methods can help to

- specify the system's boundary, i. e. the interface between the system and its environment
- characterize a system's behavior precisely; some of the current systems are even able to reason about real-time behavior
- precisely define the system's desired properties
- prove that a system meets its specification; some methods can even provide counterexamples if this is not the case
- force the designer to think about the protocol in a proper and thorough way: he must have a clear idea of what exactly he wants to achieve and must make assumptions explicit and non-ambiguous.

It should be emphasized that any proof of correctness is always relative to the formal specification of the system and the required properties and also to the assumptions of the formal system itself. An attack to the BAN–certified Needham-Schroeder-Public-Key protocol (see subsection 3.3.3) spectacularly demonstrated the importance of this principle. There will always be a gap between what a designer has in mind and his first codification, which no formalization can eliminate.

Another problem is the difference between an abstract mathematical model and a real-world instantiation of a system. Systems do not run isolated; they operate in some environment. The formal specification of a system must always include the assumptions made about an environment. As soon as one of these assumptions does not hold (maybe the environment changed or the designer forgot about a particular aspect), the conclusion (security) is invalid and all bets are off. So, a clever intruder analyzes all assumptions and tries to break the weakest one.

4.2 Using general-purpose verification tools

The main idea of this approach is to treat a cryptographic protocol as any other distributed program and use a model checking tool to prove its correctness.

Some ideas involve general techniques like state machines or petri nets, other researchers developed more specific tools like Varadhrajan's "LOTOS" or Kemmerer's "Ina Jo". The CSP approach this seminar deals with also falls in this category. The correctness proof is conducted automatically by exhaustive search in a model. Thus, this approach is capable of delivering a counterexample if a system does not meet its specification.

The problem of this approach is that such systems are not exactly suitable for this job since they were designed to prove *correctness*, but correctness is not the same as security; subtle pitfalls that are peculiar to security are either not considered or are extremely difficult to specify. Furthermore, including the intruder in the model with full generality quickly leads to the state explosion problem and makes analysis infeasible.

In [HPS01], a recent work from Heisel, Pfitzmann, and Santen, this approach is enriched by proof techniques. The authors use a slight extension of CSP to develop a necessary and sufficient condition under which a concrete (refined) system preserves confidentiality to an abstract specification. They demonstrated the applicability on a toy example, but a lot of research in this direction is necessary to actually make it applicable to real-world problems. Automating the proof with tool support may be the biggest open problem by now.

4.3 Developing expert systems

The idea of this approach is to develop an expert system that the protocol designer can use to generate and investigate various scenarios. The system starts with a given undesirable state (e. g. the intruder knows the content of a secret message) and attempts to discover if this state is reachable from an initial state.

While this approach better identifies known flaws, it neither guarantees security nor provides techniques for developing attacks. It is good at determining whether a protocol contains a given flaw, but is unlikely to discover new ones.

One of the earliest systems is the "Interrogator" by Millen, Clark and Freedmann which models protocol participants as communicating state machines whose messages are subject to interception and alteration by attackers.

The "NRL Protocol Analyzer" by C. Meadows operates similarly. But unlike the Interrogator, an unlimited number of—possibly interleaved—protocol rounds are allowed.

4.4 Modal logic

Modal logics consist of a language to describe various statements of a protocol such as what participants know or believe, and some inference rules which are used to derive new statements from the current ones. The goal of the analysis is to derive a statement that represents the correctness condition of the protocol. The designer's inability to derive it indicates that the protocol may not be correct.

This approach is the most popular by now and was pioneered by Michael Burrows, Martin Abadi, and Roger Needham. They developed a formal logic for the analysis of knowledge and belief, called "BAN logic". It does not provide a proof of security, it merely deals with authentication.

BAN logic does not attempt to model a protocol in its full richness, but uses a form called "idealized protocol". The statements itself are quite simple and straightforward; they include things like "Alice believes X", "Alice said X", or "X is fresh". By using inference rules the participant's beliefs in the protocol can be discovered.

BAN logic has been successfully used to find flaws in several protocols including Needham-Schroeder and an early draft of the widely used X.509. However, it also has some severe disadvantages that confine its utility:

- BAN logic uses many universal and quite strong assumptions; e. g. that all participants are trustworthy and do not release secrets and that each encrypted message contains sufficient redundancy to allow a participant who decrypts it to decide whether he has used the right key. In many real-world systems it is very hard to ensure these properties.
- The mapping from a real protocol to its idealized form abstracts away many important aspects. BAN logic makes no difference between seeing a message and understanding it, and does not model the revision of belief, trust (or the lack of it), or knowledge.

Thus, BAN logic is a great tool for finding attacks, but cannot give a trustworthy proof of security. Other logic systems have been published that extend BAN's abilities and try to reduce its shortcomings.

4.5 Algebraic approach

Another approach to apply formal methods to cryptographic protocol analysis is to model the protocol as an algebraic system. Research in this area has not been as active as research in developing logics of belief and knowledge. However, algebraic models were successful to represent very subtle kinds of knowledge in cryptographic protocols.

The first work in this direction is by Dolev and Yao. In their model, they assume that the network is under the control of the intruder who can read all traffic, later and destroy messages, and perform any operation, such as encryption, that is available to legitimate users of the system. However, it is assumed that initially the intruder does not know any information that is to be kept secret, such as encryption keys belonging to legitimate users of the system. Since the intruder can prevent any message from reaching its destination, and since he/she can also create messages of her own, Dolev and Yao treat any message sent by a legitimate user as a message sent to the intruder and any message received by a legitimate user as a message received from the intruder. Thus, the system becomes a machine used by the intruder to generate words. These words obey certain rewrite rules, such as the fact that encryption and decryption with the same key cancel each other out. Thus, finally, the intruder manipulates a term rewriting system. If the goal of the intruder is to find out a word that is meant to be secret, then the problem of proving a protocol secure is equivalent to the problem of proving that a certain word cannot be generated in a term rewriting system.

A more recent work from M. Abadi and A. Gordon uses an extension of the pi calculus, called "spi calculus". It permits an explicit representation of the use of cryptography in protocols. The intruder is not explicitly modeled, and this is a main advantage of the approach. Modeling the intruder can be tedious and can lead to errors (e.g., it is very difficult to model that the intruder can invent random numbers but is not lucky enough to guess the random secrets on which the protocol depends). Instead, the intruder is represented as an arbitrary spi calculus process.

Bibliography

[RySchn01] Peter Ryan, Steve Schneider. Modeling and analyzing security protocols: the CSP approach. Addison-Wesley, 2001.

[Pf00] Andreas Pfitzmann. Sicherheit in Rechnernetzen: Mehrseitige Sicherheit in verteilten und durch verteilte Systeme. Lecture script, 2000.

[Schn96] Bruce Schneier. Applied Cryptography. Second Edition. John Wiley & Sons, Inc., pages 65–68, 1996.

[Pa02] Abhinay Pandya. Formal specification and verification of security protocols. 2002. Online available at `http://www.it.iitb.ac.in/~abhinay/seminar/`

[HPS01] Maritta Heisel, Andreas Pfitzmann, Thomas Santen. Confidentiality-preserving refinement. 2001.