Programming Satan's Computer

Ross Anderson and Roger Needham

Cambridge University Computer Laboratory
Pembroke Street, Cambridge, England CB2 3QG

Abstract. Cryptographic protocols are used in distributed systems to identify users and authenticate transactions. They may involve the exchange of about 2–5 messages, and one might think that a program of this size would be fairly easy to get right. However, this is absolutely not the case: bugs are routinely found in well known protocols, and years after they were first published. The problem is the presence of a hostile opponent, who can alter messages at will. In effect, our task is to program a computer which gives answers which are subtly and maliciously wrong at the most inconvenient possible moment. This is a fascinating problem; and we hope that the lessons learned from programming Satan's computer may be helpful in tackling the more common problem of programming Murphy's.

1 Introduction

Cryptography is widely used in embedded distributed systems such as automatic teller machines, pay-per-view TV, prepayment utility meters and the GSM telephone network. Its primary purpose is to prevent frauds being carried out by people forging payment tokens or manipulating network messages; and as distributed client-server systems replace mainframes, it is also being introduced to general systems via products such as Kerberos which let a server identify remote clients and authenticate requests for resources.

As an increasing proportion of gross world product is accounted for by transactions protected by cryptography, it is important to understand what goes wrong with the systems which use it. Here, the common misconception is that a clever opponent will 'break' the cryptographic algorithm. It is indeed true that algorithms were broken during the second world war, and that this had an effect on military outcomes [Welc82]. However, even though many fielded systems use algorithms which could be broken by a wealthy opponent, it is rare for an actual attack to involve a head-on assault on the algorithm.

Surveys conducted of failure modes of banking systems [Ande94] and utility meters [AB95] showed that, in these fields at least, the great majority of actual security failures resulted from the opportunistic exploitation of various design and management blunders.

It is quite common for designers to protect the wrong things. For example, modern prepayment electricity meter systems allow the customer to buy a token from a shop and convey units of electricity to the meter in his home; they replace coin operated meters which were vulnerable to theft. Clearly one ought to prevent tokens — which may be magnetic tickets or suitably packaged EEPROMS — from being altered or duplicated in such a way that the customer can get free electricity. Yet one system protected the value encoded in the token, without protecting the tariff code; the result was that tokens could be produced with a tariff of a fraction of a cent per kilowatt hour, and they would keep a meter running almost for ever.

A lot of the recorded frauds were the result of this kind of blunder, or from management negligence pure and simple. However, there have been a significant number of cases where the designers protected the right things, used cryptographic algorithms which were not broken, and yet found that their systems were still successfully attacked. This brings us to the fascinating subject of cryptographic protocol failure.

2 Some Simple Protocol Failures

In this section, we will look at three simple protocol failures which highlight the improper use of shared key encryption. For a tutorial on cryptographic algorithms, one may consult [Schn94]; in this section, we will simply assume that if Alice and Bob share a string called a *key*, then they can use a shared-key encryption algorithm to transform a *plaintext* message into *ciphertext*. We will assume that the algorithm is strong, in that the opponent (conventionally called Charlie) cannot deduce the plaintext from the ciphertext (or vice versa) without knowing the key. We will write encryption symbolically as

$$C = \{M\}_K$$

Our problem is how to use this mechanism safely in a real system.

2.1 A simple bank fraud

A simple protocol failure allowed criminals to attack the automatic teller machine systems of one of Britain's largest banks. This bank wished to offer an offline service, so that customers could still get a limited amount of money when its mainframes were doing overnight batch processing. It therefore had the customer's personal identification number (PIN) encrypted and written to the magnetic strip on the card. The teller machines had a copy of the key, and so could check the PINs which customers entered.

However, a villain who had access to a card encoding machine discovered that he could alter the account number of his own card to someone else's, and then use his own PIN to withdraw money from the victim's account. He taught other villains how to do this trick; in due course the fraud led to at least two criminal trials, and to bad publicity which forced the bank to move to fully online processing [Lewi93].

Here, the protocol failure was the lack of linkage between the PIN and the account number. In fact, interbank standards call for PIN encryption to include the account number, but the bank's system antedated these standards.

This illustrates the fact that encryption is not as straightforward as it looks. It can be used for a number of purposes, including keeping data secret, guaranteeing the authenticity of a person or transaction, producing numbers which appear to be random, or binding the parts of a transaction together. However, we have to be very clear about what we are trying to do in each case: encryption is not synonymous with security, and its improper use can lead to errors.

2.2 Hacking pay-per-view TV

We will now turn from offline to online systems, and we will assume that our opponent Charlie controls the network and can modify messages at will. This may seem a bit extreme, but is borne out by the hard experience of the satellite TV industry.

Satellite TV signals are often encrypted to make customers pay a subscription, and the decoders are usually personalised with a secure token such as a smartcard. However, this card does not have the processing power to decrypt data at video rates, so we will usually find at least one cryptoprocessor in the decoder itself. Many decoders also have a microcontroller which passes messages between the cryptoprocessor and the card, and these have been replaced by attackers. Even where this is hard, hackers can easily interpose a PC between the decoder and the card and can thus manipulate the traffic.

- If a customer stops paying his subscription, the system typically sends a message over the air which instructs the decoder to disable the card. In the 'Kentucky Fried Chip' hack, the microcontroller was replaced with one which blocked this particular message. This was possible because the message was not encrypted [Mcco93].
- More recently a crypto key was obtained in clear, possibly by a microprobing attack on a single card; as the card serial numbers were not protected, this key could be used to revive other cards which had been disabled.
- In another system, the communications between the decoder and the card were synchronous across all users; so people could video record an encrypted programme and then decrypt it later using a protocol log posted to the Internet by someone with a valid card [Kuhn95].

These attacks are not just an academic matter; they have cost the industry hundreds of millions of pounds.

So in the following sections, our model will be that Alice wishes to communicate with Bob with the help of Sam, who is trusted, over a network owned by Charlie, who is not. In satellite TV, Alice would be the user's smartcard, Bob the decoder, Charlie the compromised microcontroller (or a PC sitting between the set-top box and the smartcard) and Sam the broadcaster; in a distributed system, Alice could be a client, Bob a server, Charlie a hacker and Sam the authentication service.

2.3 Freshening the breath of the wide mouthed frog

Time plays a role in many cryptographic protocols. Often we want to limit our exposure to staff disloyalty and equipment capture, and if system components are given keys which give them access to valuable resources and which have a long lifetime, then revocation can become complicated and expensive.

One common solution is to have each user share a key with an authentication server, Sam. We will write the key which Alice shares with him as K_{AS}, Bob's as K_{BS}, and so on. Now when Alice wishes to set up a secure session with Bob, she gets Sam to help them share a session key (say K_{AB}) which will have a strictly limited lifetime. So if Charlie is caught selling company secrets to the competition, his access can be revoked completely by having Sam delete K_{CS}.

However, many authentication protocols designed for use in this context have turned out to be wrong. One of the simplest is the so-called 'Wide Mouthed Frog' protocol [BAN89], in which Alice chooses a session key to communicate with Bob and gets Sam to translate it from K_{AS} to K_{BS}. Symbolically

$$\begin{aligned} &A \longrightarrow S : \{T_A, B, K_{AB}\}_{K_{AS}} \\ &S \longrightarrow B : \{T_S, A, K_{AB}\}_{K_{BS}} \end{aligned}$$

Here, T_A is a timestamp supplied by Alice, and T_S is one supplied by Sam. It is understood that there is a time window within which Bob will be prepared to accept the key K_{AB} as fresh.

This protocol fails. The flaw is that Sam updates the timestamp from Alice's time T_A to his time T_S. The effect is that unless Sam keeps a list of all recent working keys and timestamps, Charlie can keep a key alive by using Sam as an oracle.

For example, after observing the above exchange, Charlie could pretend to be Bob wanting to share a key with Alice; he would send Sam $\{T_S, A, K_{AB}\}_{K_{BS}}$ and get back $\{T'_S, B, K_{AB}\}_{K_{AS}}$ where T'_S is a new timestamp. He could then pretend to be Alice and get $\{T''_S, A, K_{AB}\}_{K_{BS}}$, and so on.

The practical effect of this flaw would depend on the application. If the users ran the above protocol in smartcards which then passed the session key in clear to a software bulk encryption routine, they could be open to the following attack. Charlie observes that Alice and Bob are setting up a session, so he keeps the session key alive until he can steal one of their smartcards. Indeed, Alice and Bob might become careless with their cards, having thought the message irrecoverable once its plaintext and K_{AB} had been destroyed, and the validity period of the timestamps had expired.

The lesson here is that one must be careful about how we ensure temporal succession and association.

2.4 The perils of challenge-response

Many systems use a challenge-response technique in which one party sends the other a random number, which is then subjected to some cryptographic transformation and sent back. The purpose of this may be to identify someone wishing to log on to a system, or it may play a role similar to that of a timestamp by ensuring freshness in a protocol. In what follows, we will write N_A for a random number challenge issued by Alice.

One widespread application is in password calculators. To log on to a system which uses these, the user first enters her name; the system displays a seven digit challenge on the screen; the user enters this into her calculator, together with a secret PIN; the calculator concatenates the challenge and the PIN, encrypts them with a stored key, and displays the first seven digits of the result; and the user finally types this in as her logon password.

For users logging on to a single system, this is a simple and robust way to avoid many of the problems common with passwords. However, the picture changes when a number of systems are involved, and attempts are made to concentrate the calculator keys in a single security server. For example, consider the following protocol of Woo and Lam, which attempts to let Alice prove her presence to Bob's machine despite the fact that she does not share her key with Bob but with Sam:

$$
\begin{aligned}
&A \longrightarrow B : A \\
&B \longrightarrow A : N_B \\
&A \longrightarrow B : \{N_B\}_{K_{AS}} \\
&B \longrightarrow S : \{A, \{N_B\}_{K_{AS}}\}_{K_{BS}} \\
&S \longrightarrow B : \{N_B\}_{K_{BS}}
\end{aligned}
$$

This protocol is wrong. The only connection between Bob's query to Sam, and Sam's reply, is the coincidental fact that the latter comes shortly after the former; this is insufficient against an opponent who can manipulate messages. Charlie can impersonate Alice by trying to log on to Bob's machine at about the same time as her, and then swapping the translations which Sam gives of Alice's and Charlie's replies.

One of us therefore proposed in [AN94] that the last message should be:

$$S \longrightarrow B : \{A, N_B\}_{K_{BS}}$$

However, this is not always enough, as Sam goes not know the name of the host to which Alice is attempting to log on. So Charlie might entice Alice to log on to him, start a logon in Alice's name to Bob, and gets her to answer the challenge sent to him. So we ought to put Bob's name explicitly in the protocol as well:

$$
\begin{aligned}
&A \longrightarrow B : A \\
&B \longrightarrow A : N_B \\
&A \longrightarrow B : \{B, N_B\}_{K_{AS}} \\
&B \longrightarrow S : A, \{B, N_B\}_{K_{AS}} \\
&S \longrightarrow B : \{N_B, A\}_{K_{BS}}
\end{aligned}
$$

All this might be overkill in some applications, as someone who could manipulate the network could take over an already established session. Whether there is actually a risk will depend closely on the circumstances.

Anyway, the two lessons which we can learn from this are firstly, that we should be very careful in stating our security goals and assumptions; and secondly, that where the identity of a principal is essential to the meaning of a message, it should be mentioned explicitly in that message.

For more attacks on shared key protocols, see [BAN89] and [AN94].

3 Problems with Public Key Protocols

The shared-key encryption algorithms used in the above protocols are not the only tools we have; there are also public-key algorithms which use different keys for encrypting and decrypting data. These are called the *public key*, K, and the *private key*, K^{-1}, respectively. A public key algorithm should have the additional property that Charlie cannot deduce K^{-1} from K.

The best known public-key algorithm is the RSA scheme [RSA78]. This is straightforward in the sense that the encryption and decryption operations are mutual inverses; that is, $C = \{M\}_K$ holds if and only if $M = \{C\}_{K^{-1}}$. So we can take a message and subject it to 'decryption' $S = \{M\}_{K^{-1}}$, which can be reversed using K to recover the original message, $M = \{S\}_K$.

In this context, the 'decryption' of a message M is usually referred to as its *digital signature*, since it can only be computed by the person possessing K^{-1}, while anybody who knows K can check it. Actually, decryption and signature are not quite the same thing, but we shall postpone discussion of this.

Anyway, if Alice and Bob publish encryption keys KA, KB respectively while each keeps the corresponding decryption key KA^{-1} or KB^{-1} secret, then they can ensure both the integrity and privacy of their messages in the following elegant way [DH76]. To send a message M to Bob, Alice first signs it with her private key KA^{-1} and then encrypts it with Bob's public key KB:

$$C = \{\{M\}_{KA^{-1}}\}_{KB}$$

When Bob receives this message, he can first strip off the outer encryption using his private key KB^{-1}, and then recover the message M by applying Alice's public key KA.

It might be hoped that this technology could make the design of cryptographic protocols a lot easier. Of course, there are a lot of problems which we must solve in a real implementation. For example, how does Bob know that he has received a real message? If Charlie had replaced the ciphertext C with some completely random value C', then Bob would calculate $M' = \{\{C'\}_{KB^{-1}}\}_{KA}$. For this reason, we would usually want to insert redundancy into the message.

Some systems do indeed suffer from a lack of redundancy [AN95]. However, this is not the only thing which can go wrong with a public key protocol.

3.1 The Denning-Sacco disaster

One obvious problem with public key cryptography is that Charlie might generate a keypair K_X, K_X^{-1} and place K_X in the directory with the legend "This is Alice's public key: K_X". So there is still a security requirement on key management, but we have reduced it from confidentiality to authenticity.

One common solution is to use our trusted third party, Sam, as a certification authority (the public key equivalent of an authentication server). Sam would issue each user with a certificate containing their name, public key, access rights (whether credentials or capabilities) and expiry date, all signed with Sam's own secret key (whose corresponding public key we will assume to be well known). We can write

$$CA = \{A, K_A, R_A, E_A\}_{K_S^{-1}}$$

However, public key certificates are not the whole answer, as it is still easy to design faulty protocols using them. For example, one of the first public key protocols was proposed by Denning and Sacco in 1982; it provides a mechanism for distributing conventional shared encryption keys, and runs as follows:

$$\begin{aligned}
&A \longrightarrow S : A, B \\
&S \longrightarrow A : CA, CB \\
&A \longrightarrow B : CA, CB, \{\{T_A, K_{AB}\}_{K_A^{-1}}\}_{K_B}
\end{aligned}$$

It was not until 1994 that a disastrous flaw was noticed by Abadi. Bob, on receiving Alice's message, can masquerade as her for as long as her timestamp T_A remains valid!

To see how, suppose that Bob wants to masquerade as Alice to Charlie. He goes to Sam and gets a fresh certificate CC for Charlie, and then strips off the outer encryption $\{...\}_{K_B}$ fromthe third message in the above protocol. He now re-encrypts the signed key $\{T_A, K_{AB}\}_{K_A^{-1}}$ with Charlie's public key — which he gets from CC — and makes up a bogus third message:

$$B \longrightarrow C : CB, CC, \{\{T_A, K_{AB}\}_{K_A^{-1}}\}_{K_C}$$

This failure can also be seen as a violation of the principle that names should be mentioned explicitly within messages.

3.2 The middleperson attack

John Conway pointed out that it is easy to beat a grandmaster at postal chess. Just play two grandmasters, one as white and the other as black, and act as a message relay between them.

The same idea can often be used to attack crypto protocols. For example, one of us proposed in [NS78] that an authenticated key exchange could be carried out as follows. Here we will assume for the sake of brevity that both Alice and Bob already have a copy of the other's key certificate.

$$
\begin{aligned}
&A \longrightarrow B : \{N_A, A\}_{K_B}\\
&B \longrightarrow A : \{N_A, N_B\}_{K_A}\\
&A \longrightarrow B : \{N_B\}_{K_B}
\end{aligned}
$$

This was already known to be vulnerable to a replay attack [BAN89]. However, Lowe has recently pointed out a middleperson attack [Lowe95]. Here, Charlie sits between Alice and Bob. He appears as Charlie to Alice, but pretends to be Alice to Bob:

$$
\begin{aligned}
&A \longrightarrow C : \{N_A, A\}_{K_C}\\
&C \longrightarrow B : \{N_A, A\}_{K_B}\\
&B \longrightarrow C : \{N_A, N_B\}_{K_A}\\
&C \longrightarrow A : \{N_A, N_B\}_{K_A}\\
&A \longrightarrow C : \{N_B\}_{K_C}\\
&C \longrightarrow B : \{N_B\}_{K_B}
\end{aligned}
$$

The fix here is also straightforward: just put the principals' names explicitly in the messages.

3.3 CCITT X.509

Another problem was found by Burrows, Abadi and Needham in the CCITT X.509 protocol [CCITT88]. This is described in [BAN89]; briefly, the idea is that Alice signs a message of the form $\{T_A, N_A, B, X, \{Y\}_{K_B}\}$ and sends it to Bob, where T_A is a timestamp, N_A is a serial number, and X and Y are user data. The problem here is that since the order of encryption and signature are reversed — Y is first encrypted, then signed — it is possible for Charlie to strip off Alice's signature and add one of his own.

We have recently found an even sneakier attack — with many public key encryption algorithms it is possible, given a message M and ciphertext C, to find some key K with $C = \{M\}_K$. The mechanics of this key spoofing attack are described in [AN95]; they depend quite closely on the mathematics of the underlying public key encryption algorithm, and we will not go into them in much detail here[1].

[1] Consider for example RSA with a 512 bit modulus. If Alice sends Bob M, and if the modulus, public exponent and private exponent of party α are n_α, e_α and d_α, and if we ignore hashing (which makes no difference to our argument), the signed encrypted message would be $\{M^{e_B} \pmod{n_B}\}^{d_A} \pmod{n_A}$. But since Bob can factor n_B and its factors are only about 256 bits long, he can work out discrete logarithms with respect to them and then use the Chinese Remainder Theorem to get discrete logs modulo n_B. So he can get Alice's 'signature' on M' by finding x such that $[M']^x = M \pmod{n_B}$ and then registering (xe_B, n_B) as a public key.

The effect is that if we encrypt data before signing it, we lay ourselves open to an opponent swapping the message that we thought we signed for another one. In the above example, if Bob wants to convince a judge that Alice actually sent him the message Z rather than Y, he can find a key K' such that $\{Z\}_{K'} = \{Y\}_{K_B}$, and register this key K' with a certification authority.

This provides a direct attack on CCITT X.509, in which Alice signs a message of the form $\{T_A, N_A, B, X, \{Y\}^{e_B} \pmod{n_B}\}$ and sends it to Bob. Here T_A is a timestamp, N_A is a serial number, and X and Y are user data. It also breaks the draft ISO CD 11770; there, Y consists of A's name concatenated with either a random challenge or a session key.

Whether there is an actual attack on any given system will, as usual, depend on the application detail (in this case, the interaction between timestamps, serial numbers and any other mechanisms used to establish temporal succession and association). However, it is clearly a bad idea to bring more of the application code within the security perimeter than absolutely necessary.

In any case, the lesson to be learned is that if a signature is affixed to encrypted data, then one cannot assume that the signer has any knowledge of the data. A third party certainly cannot assume that the signature is authentic, so nonrepudiation is lost.

3.4 Simmons' attack on TMN

False key attacks are not the only protocol failures which exploit the mathematical properties of the underlying algorithms. These failures can sometimes be quite subtle, and an interesting example is the attack found by Simmons on the TMN (Tatebayashi-Matsuzaki-Newmann) scheme [TMN89].

Here, two users want to set up a session key, but with a trusted server doing most of the work (the users might be smartcards). If Alice and Bob are the users, and the trusted server Sam can factor N, then the protocol goes as follows:

$$
\begin{aligned}
&A \longrightarrow S : r_A^3 \pmod{N}\\
&B \longrightarrow S : r_B^3 \pmod{N}\\
&S \longrightarrow A : r_A \oplus r_B
\end{aligned}
$$

Each party chooses a random number, cubes it, and sends in to Sam. As he can factor N, he can extract cube roots, xor the two random numbers together, and send the result to Alice. The idea is that Alice and Bob can now use r_B as a shared secret key. However, Simmons pointed out that if Charlie and David conspire, or even if David just generates a predictable random number r_D, then Charlie can get hold of r_B in the following way [Simm94]:

$$
\begin{aligned}
&C \longrightarrow S : r_B^3 r_C^3 \pmod{n}\\
&D \longrightarrow S : r_D^3 \pmod{n}\\
&S \longrightarrow C : r_B r_C \oplus r_D
\end{aligned}
$$

The lessons to be learned here are that we should never trust in the secrecy of other people's secrets, and that we must always be careful when signing or decrypting data that we never let ourselves be used as an oracle by the opponent.

3.5 The difference between decryption and signature

Nonrepudiation is complicated by the fact that signature and decryption are the same operation in RSA, which many people use as their mental model of public key cryptography. They are actually quite different in their semantics: decryption can be simulated, while signature cannot. By this we mean that an opponent can exhibit a ciphertext and its decryption into a meaningful message, while he cannot exhibit a meaningful message and its signature (unless it is one he has seen previously).

Consider for example another protocol suggested by Woo and Lam [WL92]:

$$
\begin{aligned}
&A \longrightarrow S : A, B \\
&S \longrightarrow A : CB \\
&A \longrightarrow B : \{A, N_A\}_{KB} \\
&B \longrightarrow S : A, B, \{N_A\}_{KS} \\
&S \longrightarrow B : CA, \{\{N_A, K_{AB}, A, B\}_{KS^{-1}}\}_{KB} \\
&B \longrightarrow A : \{\{N_A, K_{AB}, A, B\}_{KS^{-1}}\}_{KA} \\
&A \longrightarrow B : \{N_A\}_{K_{AB}}
\end{aligned}
$$

There are a number of problems with this protocol, including the obvious one that Bob has no assurance of freshness (he does not check a nonce or see a timestamp). However, a subtler and more serious problem is that Alice never signs anything; the only use made of her secret is to decrypt a message sent to her by Bob. The consequence is that Bob can only prove Alice's presence to himself — he cannot prove anything to an outsider, as he could easily have simulated the entire protocol run. The effect that such details can have on the beliefs of third parties is one of the interesting (and difficult) features of public key protocols: few of the standards provide a robust nonrepudiation mechanism, and yet there is a real risk that many of them may be used as if they did.

So we must be careful how entities are distinguished. In particular, we have to be careful what we mean by 'Bob'. This may be 'whoever controls Bob's signing key', or it may be 'whoever controls Bob's decryption key'. Both keys are written as KB^{-1} in the standard notation, despite being subtly different in effect. So we should avoid using the same key for two different purposes, and be careful to distinguish different runs of the same protocol from each other.

For more examples of attacks on public key protocols, see [AN95].

4 How Can We Be Saved?

We have described the crypto protocols design problem as 'programming Satan's computer' because a network under the control of an adversary is possibly the most obstructive computer which one could build. It may give answers which are subtly and maliciously wrong at the most inconvenient possible moment.

Seen in this light, it is less surprising that so many protocols turned out to contain serious errors, and that these errors often took a long time to discover — twelve years for the bug in Denning-Sacco, and seventeen years for the middleperson attack on Needham-Schroeder. It is hard to simulate the behaviour of the devil; one can always check that a protocol does not commit the old familiar sins, but every so often someone comes up with a new and pernicious twist.

It is therefore natural to ask what we must do to be saved. Under what circumstances can we say positive things about a crypto protocol? Might it ever be possible to prove that a protocol is correct?

4.1 Protocol verification logics

There were some attempts to reduce security claims of specific protocols to the intractability of some problem such as factoring on which the strength of the underlying encryption algorithm was predicated. However, the first systematic approach involved the modal logic of Burrows, Abadi and Needham. Here, we have a series of rules such as

> If P believes that he shares a key K with Q, and sees the message M encrypted under K, then he will believe that Q once said M

and

> If P believes that the message M is fresh, and also believes that Q once said M, then he will believe that Q believes M

These rules are applied to protocols, and essentially one tries to follow back the chains of belief in freshness and in what key is shared with whom, until one either finds a flaw or concludes that it is proper under the assumptions to believe that the authentication goal has been met. For full details, seen [BAN89].

A more recent logic by Kailar [Kail95] looks at what can be done without making any assumptions about freshness. Its application is to systems (such as in electronic banking) where the question is not whether Q recently said M but whether Q ever said M — as for example in whether Q ever did the electronic equivalent of endorsing a bill of exchange.

Curiously enough, although public key algorithms are based more on mathematics than shared key algorithms, public key protocols have proved much harder to deal with by formal methods. A good example is encryption before signature. This is easy enough as a principle — we normally sign a letter and then put it in an envelope, rather than putting an unsigned letter in an envelope and then signing the flap. It is intuitively clear that the latter practice deprives the recipient of much of the evidential force that we expect a normal letter to possess.

However, encryption before signature can cause serious problems for formal verification. Neither the BAN logic nor Kailar's logic is fooled by it, but more complex tools that try to deal with algorithm properties (such as those discussed in [KMM94]) do not seem able to deal with key spoofing attacks at all.

Another curious thing about formal methods has been that most of the gains come early. The BAN logic is very simple, and there are a number of protocols which it cannot analyse. However, attempts to build more complex logics have met with mixed success, with problems ranging from inconsistent axiom schemes to the sheer difficulty of computation if one has a hundred rules to choose from rather than ten. The BAN logic still has by far the most 'scalps' at its belt.

It is also our experience of using logic that most of work lies in formalising the protocol. Once this has been done, it is usually pretty obvious if there is a bug. So perhaps the benefit which it brings is as much from forcing us to think clearly about what is going on than from any intrinsic mathematical leverage.

4.2 Robustness principles

Another approach is to try to encapsulate our experience of good and bad practice into rules of thumb; these can help designers avoid many of the pitfalls, and, equally, help attackers find exploitable errors. We have given a number of examples in the above text:

- be very clear about the security goals and assumptions;
- be clear about the purpose of encryption — secrecy, authenticity, binding, or producing pseudorandom numbers. Do not assume that its use is synonymous with security;
- be careful about how you ensure temporal succession and association;
- where the identity of a principal is essential to the meaning of a message, it should be mentioned explicitly in the message;
- make sure you include enough redundancy;
- be careful that your protocol does not make some unexamined assumption about the properties of the underlying cryptographic algorithm;
- if a signature is affixed to encrypted data, then one cannot assume that the signer has any knowledge of the data. A third party certainly cannot assume that the signature is authentic, so nonrepudiation is lost;
- do not trust the secrecy of other people's secrets;
- be careful, especially when signing or decrypting data, not to let yourself be used as an oracle by the opponent;
- do not confuse decryption with signature;
- be sure to distinguish different protocol runs from each other.

This is by no means a complete list; more comprehensive analyses of desirable protocol properties can be found in [AN94] and [AN95]. However, experience shows that the two approaches — formal proofs and structured design rules — are complementary, we are led to wonder whether there is some overarching principle, which underlies the success of formal methods in the crypto protocol context and of which the above points are instances. We propose the following:

> **The Explicitness Principle:** Robust security is about explicitness. A cryptographic protocol should make any necessary naming, typing and freshness information explicit in its messages; designers must also be explicit about their starting assumptions and goals, as well as any algorithm properties which could be used in an attack.

This is discussed in greater detail in [Ande94] [AN94] [AN95]. However, there is more going on here than a sloppy expression of truths which are either self evident, or liable one day to be tidied up and proved as theorems. There is also the matter of educating what we call 'commonsense'.

Commonsense can be misleading and even contradictory. For example, we might consider it commonsensical to adhere to the KISS principle ('Keep It Simple Stupid'). However, much erroneous protocol design appears to be a consequence of trying to minimise the amount of cryptographic computation which has to be done (e.g., by omitting the names of principals in order to shorten the encrypted parts of our messages). So one might rather cleave to the description of optimisation as 'the act of replacing something that works with something that almost works, but is cheaper'. Resolving such conflicts of intuition is one of the goals of our research; and both robustness principles and formal methods seem to be most useful when they provide that small amount of support which we need in order for commonsense to take us the rest of the way.

5 Conclusions

We have tried to give an accessible introduction to the complex and fascinating world of cryptographic protocols. Trying to program a computer which is under the control of an intelligent and malicious opponent is one of the most challenging tasks in computer science, and even programs of a few lines have turned out to contain errors which were not discovered for over a decade.

There are basically two approaches to the problem. The first uses formal methods, and typically involves the manipulation of statements such as 'A believes that B believes X about key K'. These techniques are helpful, and have led to the discovery of a large number of flaws, but they cannot tackle all the protocols which we would like to either verify or break.

The complementary approach is to develop a series of rules of thumb which guide us towards good practice and away from bad. We do not claim that these robustness principles are either necessary or sufficient, just that they are useful; together with formal methods, they can help to educate our intuition. Either way, the most important princple appears to be explicitness: this means that, for example, a smartcard in a satellite TV set-top box should not say 'here is a key with which to decode the signal' but 'I have received your random challenge to which the response is X, and Y is a key with which Mrs Smith is authorised to decode the 8 o'clock news using decoder number Z on the 15th June 1995'.

What is the wider relevance? In most system engineering work, we assume that we have a computer which is more or less good and a program which is probably fairly bad. However, it may also be helpful to consider the case where the computer is thoroughly wicked, particularly when developing fault tolerant systems and when trying to find robust ways to structure programs and encapsulate code. In other words, the black art of programming Satan's computer may give insights into the more commonplace task of trying to program Murphy's.

References

[Ande92] RJ Anderson, "UEPS - A Second Generation Electronic Wallet", *Computer Security — ESORICS 92*, Springer LNCS v 648 in 411–418

[Ande94] RJ Anderson, "Why Cryptosystems Fail", in *Communications of the ACM* v 37 no 11 (November 1994) pp 32–40

[AB95] RJ Anderson, SJ Bezuidenhout, "Cryptographic Credit Control in Pre-Payment Metering Systems", in *1995 IEEE Symposium on Security and Privacy* pp 15–23

[AN94] M Abadi, RM Needham, *'Prudent Engineering Practice for Cryptographic Protocols'*, DEC SRC Research Report no 125 (June 1 1994)

[AN95] RJ Anderson, RM Needham, "Robustness principles for public key protocols", *Crypto 95, to appear*

[BAN89] M Burrows, M Abadi, RM Needham, "A Logic of Authentication", in *Proceedings of the Royal Society of London A* v 426 (1989) pp 233–271; earlier version published as DEC SRC Research Report no 39

[CCITT88] CCITT X.509 and ISO 9594-8, "The Directory — Authentication Framework", CCITT Blue Book, Geneva, March 1988

[DH76] W Diffie, ME Hellman, "New Directions in Cryptography", in *IEEE Transactions on Information Theory*, IT-22 no 6 (November 1976) p 644–654

[Kail95] R Kailar, "Reasoning about Accountability in Protocols for Electronic Commerce", in *1995 IEEE Symposium on Security and Privacy* pp 236–250

[Kuhn95] M Kuhn, *private communication, 1995*

[KMM94] R Kemmerer, C Meadows, J Millen, "Three Systems for Cryptographic Protocol Verification", in *Journal of Cryptology* v 7 no 2 (Spring 1994) pp 79–130

[Lewi93] B Lewis, "How to rob a bank the cashcard way", in *Sunday Telegraph* 25th April 1993 p 5

[Lowe95] G Lowe, "An Attack on the Needham-Schroeder Public-Key Authentication Protocol", *preprint, May 1995*

[Mcco93] J McCormac, *'The Black Book'*, Waterford University Press, 1993

[NS78] RM Needham, M Schroeder, "Using encryption for authentication in large networks of computers", in *Communications of the ACM* v 21 no 12 (Dec 1978) pp 993–999

[RSA78] RL Rivest, A Shamir, L Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", in *Communications of the ACM* v 21 no 2 (Feb 1978) pp 120–126

[Schn94] B Schneier, *'Applied Cryptography'*, John Wiley 1994.

[Simm94] GJ Simmons, "Cryptanalysis and Protocol Failures", in *Communications of the ACM* v **37** no 11 (November 1994) pp 56–65

[TMN89] M Tatebayashi, N Matsuzaki, DB Newman, "Key distribution protocol for digital mobile communication systems", in *Advance in Cryptology — CRYPTO '89*, Springer LNCS 435 pp 324–333

[WL92] TYC Woo, SS Lam, "Authentication for Distributed Systems", in *IEEE Computer* (January 1992) pp 39–52

[Welc82] G Welchman, *'The Hut Six Story — Breaking the Enigma Codes'*, McGraw Hill, 1982

Ross Anderson learned to program in 1972 on an IBM 1401. He has worked in computer security for about ten years, starting with commercial applications and more recently as a Senior Research Associate at Cambridge University Computer Laboratory, where he has been since 1992. His interests centre on the performance and reliability of security systems.

Roger Needham learned to program in 1956 on the EDSAC. He is Head of the Cambridge University Computer Laboratory, and is interested in distributed systems. He has made a number of contributions to computer security over the last twenty five years: most recently, he has been working on cryptographic protocols. He was one of the inventors of the Burrows-Abadi-Needham logic and has been busy elucidating the nature of robustness in the context of security protocols.