Public-Key Crypto-Systems using Symmetric-Key Crypto-Algorithms

Bruce Christianson
Bruno Crispo
James A. Malcolm

The prospect of quantum computing makes it timely to consider the future of public-key crypto-systems. Both factorization and discrete logarithm correspond to a single quantum measurement upon a superposition of candidate keys transformed into the fourier domain. Accordingly, both these problems can be solved by a quantum computer in a time essentially proportional to the bit-length of the modulus, a speed-up of exponential order.

At first sight, the resulting collapse of asymmetric-key crypto-algorithms seems to herald the doom of public-key crypto-systems. However, for most security services, asymmetric-key crypto-algorithms actually offer relatively little practical advantage over symmetric-key algorithms. Most of the differences popularly attributed to the choice of crypto-algorithm actually result from subtle changes in assumptions about hardware or domain management.

In fact it is straightforward to see that symmetric-key algorithms can be embodied into tamper-proof hardware in such a way as to provide equivalent function to a public-key crypto-system, but the assumption that physical tampering never occurs is too strong for practical purposes. Our aim here is to build a system which relies merely upon tamper-evident hardware, but which maintains the property that users who abuse their cryptographic modules through malice or stupidity harm only themselves, and those others who have explicitly trusted them.

We mention in passing that quantum computing holds out the prospect of devices which can provide unequivocal evidence of tampering even at a distance.

This talk addresses three issues. First, we review the differences between symmetric and asymmetric key crypto-algorithms. Advantages traditionally ascribed to asymmetric-key algorithms are that they:

- scale better than symmetric-key, and solve the key-management problem
- allow local generation and verification of credentials, avoiding the need for servers to be on-line
- prevent servers from masquerading as their clients
- provide locality of trust and allow the consequences of key or hardware compromise to be confined
- provide attributability, preventing clients from masquerading as one another or from repudiating commitment.

We explain in detail why these advantages actually result from system-level assumptions about the hardware or application architecture, rather than from the choice of crypto-algorithm.

Next, we consider the problem of constructing security protocols based upon symmetric-key algorithms embodied in tamper-evident hardware, under corresponding system-level assumptions. The protocol building-blocks which we use are all well known:

- private key certificates for on-line introductions
- hash chaining for forward secrecy
- key-mapping for unforgeable attribution
- capability conversion across domain boundaries to prevent theft
- explicit delegation certificates to avoid key-movement
- secrets hashed with identities to provide public keys

and so on. However by combining these elements together with tamper-evident hardware, we obtain a novel system infrastructure which allows the properties desired of a public-key crypto-system to be achieved when the underlying crypto-algorithm is symmetric. In particular, our proposals allow explicit *lack* of trust to be exploited systematically as a lever so as to make the system more secure.

Finally, we turn our previous arguments inside out, and observe that existing public-key crypto-systems – where the underying crypto-algorithm is asymmetric – can also be made more robust by deploying the new infrastructure and the associated tamper-evident hardware.

Public-Key Crypto-Systems using Symmetric-Key Crypto-Algorithms

Bruce Christianson

University of Hertfordshire: Hatfield

Mike has told me that I must finish on time, so I'll try not to say anything uncontroversial[1].

Many of you will have seen the recent announcement that the number 15 has been factorized at Bletchley Park by a quantum computer. The timing of the announcement[2] is significant, but given all the public money that's being pumped into quantum computers, it's worth considering the implications for us if this effort succeeds.

Discrete logarithm and factorization both succumb to an attack by a quantum computer in a time that's essentially linear in the length of the modulus. The reason for this is that, in the Fourier domain, you're looking for the period of a particular sub-group, and you can do that by a single measurement on a superposition of states. You transform the states into the Fourier domain, which takes $N \log N$ multiply operations, and by superposing you can do N of those in parallel. So it's $\log N$ time, and then you do a single measurement.

You don't get exponential speedup on every problem, only on those where you are essentially making a single measurement in the Fourier domain. On general sorting or matching or searching problems you get quadratic speedup, but that's enough to destroy Merkle-puzzle type key algorithms.

On the other hand, symmetric ciphers are alive and well. You need to double the block length and double the key length, but apart from that you can carry on as before. And it's actually easier to produce tamper-evident devices, in fact now you can even tell remotely whether they've been tampered with, which you couldn't before.

So what are the implications of this for crypto-systems or crypto-protocols, rather than just for the algorithms people? It's been known for a long time that you can do public key cryptography using symmetric key algorithms and tamper-proof boxes. For example, Desmedt and Quisquater have a system[3] where essentially you get the public key K^+ from the private key K^- by applying a symmetric algorithm to K^- using a master key M that is known only to the hardware, so that $K^+ = \{K^-\}_M$. The problem with this is that drilling into one verification box blows the whole crypto-system, because the master key is a global secret. And it doesn't just blow the entire system, it does it retrospec-

[1] Because that would waste time.

[2] 10:59 GMT, Saturday 1 April 2000AD.

[3] Y Desmedt and J-J Quisquater, 1987, Public Key Systems Based on the Difficulty of Tampering: Is there a Difference between DES and RSA? Advances in Cryptology – Crypto86.

tively: it's always been the same secret so you can go back and tamper with event logs and things like that.

So the first thing we want to see is how far we can get with merely *tamper-evident* hardware. This is hardware which has the property that you can tell whether or not somebody has drilled into it: you can't get the secret out without leaving some physical evidence. Of course, the physical matrix within which tamper-evident hardware is deployed must be irreproducibly unique: someone who gets the secret out of one box mustn't be able to blow the secret into another box and pass the second box off as the first. And so it also has to be possible to verify that the physical matrix matches the secret inside, as well as being physically intact[4]. But all of this is still a weaker requirement than tamper-proof, which says you can't get the secret out at all.

We want to retain all the good things that we associate with public key crypto-systems, like confinement of trust, localization of operation, containment of partial compromise, retrospective repair; this general instinct that we have about public key, which says that malice or stupidity harms only the person who misuses their crypto-module, or people who explicitly trust the misuser's honesty or competence.

Let's start by reviewing the well-known advantages that asymmetric algorithms allegedly have over symmetric key. I want to convince you that these advantages are to a large extent mythical. The first claim is that asymmetric key supposedly scales better than symmetric key and solves the key management problem. Well no, not really. Asymmetric key does use N keys rather than N^2 keys in total for N people, but each individual user still needs to manage N different keys. They need one key for everybody they're actually ever going to talk to.

The only way to reduce the total number of keys to N is by forcing everyone to use a single public key. Each public key is used by N other people. This is exactly what makes the revocation problem hard in the asymmetric case.

Revocation with symmetric key is relatively straightforward: you tell the person with whom you share the secret that the secret is blown. There are protocol implications where Bob doesn't want to hear what Alice is telling him, for example, because Bob in the middle of trying to rip Alice off, or where Alice wants to pretend to have told Bob because she's in the process of ripping Bob off, but it's an end-to-end, one-to-one problem, it's not one of these dreadful things requiring revocation authorities to do fanout.

The initial key distribution requires the same effort to bootstrap whichever algorithm you use, because you've got N people who have to register with some central authority. To make the cost scale linearly with the system size, you need some kind of third party. Everybody has got to register with the third party, and registration requires some sort of real world artifact, no matter how you do it. You can't register by electronic means to do the initial bootstrap, the initial bootstrap requires some sort of tamper-evident token. This token might be a letter on company letterhead signed in blood by various people, or it might be

[4] See LNCS 1550, pp157, 166-167.

a tamper-evident device. Asymmetric key doesn't require confidentiality of the real world artifact, just integrity, but if you've got a tamper-evident token then that's not an issue anyway. So there's not a lot of difference there.

But we do need some sort of third party, a trusted third party, or better, several mutually mistrusting parties, which I'll call either a CA or an AS depending on whether the algorithm is asymmetric key or a symmetric key. Of course what these actually are, whether they're really certification authorities or authentication servers or not, depends on the application. When I say CA I just mean a third party in an asymmetric key world, I don't necessarily mean an actual certification authority.

The second myth is that asymmetric key allows local generation and verification of credentials. This is a big advantage, it means you don't have to support on-line checking of credentials. So that means the CA can be off-line, which means it's more secure, and it also means the protocol is more efficient and more convenient. But the way we use asymmetric key cryptography at the moment, the revocation authority has to be on-line, and misbehaviour of the revocation authority is a show-stopper. So the advantage claimed for asymmetric key becomes illusory as soon as you have revocation.

After you've introduced yourself to somebody for the first time, once you've established a relationship with somebody and you're doing business with them, there's very little difference between asymmetric and symmetric. Because if you make your server stateless, which you probably want to do anyway, then you can just push out the credentials that the client needs in order to reintroduce themselves. The client is told, this is part of your state, you've got to present these credentials in the next message that you send.

You almost certainly need to log transaction order whichever world you're in, because there are disputes about whether a transaction occurred before or after a particular key had been revoked, or a particular certificate had been revoked. And there's the old push versus pull debate, the transmitter needs to log that they've revoked a key, the receiver needs to log that they have done a check to see whether a key has been revoked or not. There's a slight tendency to use push technology with asymmetric key and pull technology with symmetric key. But if you say, look I want to design a push protocol, and you sit down and you do it, it's really spooky that the difference between an asymmetric key version and a symmetric key version is so very slight. You are not only writing down the same messages but you're actually putting the same fields in them, it's quite macabre.

At introduction time – now I've never met you before but I want you to store my files for me – there still isn't a vast amount of difference at the system level. In the asymmetric case, if we were being sensible, the first time I did business with Mike I would get the fact that my public key certificate was being issued to him registered somewhere. Because that way the revocation problem becomes straightforward. Either there's a single authority that Mike knows he has to go to to check whether the certificate has been revoked, or the revocation authority knows that when that certificate is repealed it should tell Mike about it. In fact, you can make a good argument that there should be a separate instance of the

certificate for each user to whom it's issued, and that the capability should be bound to those instances, harping back to Virgil Gligor's talk earlier[5].

Alternatively, we can use introduction certificates with symmetric key, tokens like $\{A, B, S, Kab\}_{Kas}; \{A, B, S, Kab\}_{Kbs}$ etc. You don't have to do the introduction while the AS is on-line, you just at some previous point have to have indicated that you will wish to engage in transactions with the other person. This is a requirement which you usually have to satisfy anyway at some point well in advance, regardless of which algorithm you're using, it's a systems issue.

So we get onto the third reason that asymmetric key is alleged to be better: it prevents authorities from masquerading as their clients. This is where asymmetric key appears to have a genuine advantage over symmetric key. Well of course a CA *can* masquerade as a client, but the point is that it will get caught. You can break into a CA and issue false key certificates, but when you're challenged you can't produce the real world artifact that goes with the electronic certificate, and then people can go back and they can work out which transactions were committed with those false credentials, and then they can undo those transactions. Or you can have some other protocol that says what to do when you discover this. Provided it didn't involve confidentiality because then the undo involves shooting people. And provided the provenance of the real world artifact is secured.

Under all these conditions it's clear that a CA with asymmetric key doesn't allow *retrospective* breaks. You can't go back and alter the history of which certificates things were signed with, provided people kept careful logs – which is a point I'll come back to later – and didn't just verify the credentials and then throw them away. You can go back and you can work out which transactions are suspect. You say, oh look, there's some sawdust and a little drill-hole in the CA, let's find out which transactions we need to roll back.

But you can also do forward secrecy using symmetric crypto-algorithms. Alice has a secret s_0 that she shares with the AS authority S, and at each step she uses the key $k_i = h(0|s_i)$ which she gets by hashing that secret with a zero, and then she replaces the shared secret by $s_{i+1} = h(1|s_i)$ which she gets by hashing the shared secret with a one, and then she carries on. Now if someone drills into the authority, then the authority is compromised from that point on, but it doesn't get the attacker access back into the past. Everything up to the last point at which the authority was physically secure, is still secure.

And if we always put protocols in these outer wrappers, just as was being suggested in the previous talk by Tuomas Aura[6], then we have gradually increasing authentication, gradually increasing certification. If you receive credentials and the outer wrapper is broken, that's the point at which you say, I want an integrity check now please. This box is using the secret one ahead from me, somebody has done an extra transaction, let's have a look and see whether this is just a failure of the outer protocol or whether there's sawdust on the floor.

[5] xref.
[6] xref.

If you've issued false key certificates, you've got to prevent the person whose real key certificate you've replaced from seeing the false one. And that means that following a CA compromise, you need an active wiretap in the asymmetric case. In the symmetric case you can force the situation to be that you still need an active wiretap following AS compromise, and you've got the same time window in which to do it[7].

The fourth myth about asymmetric key is that it gives locality of trust. Except that if you're crossing multiple security domains, which is the only case in which locality of trust is any big deal, you've got to have a separate authentication channel to begin with, because the global hierarchy of trust is never going to happen, see previous talks. And when you think about it, the integrity of your verification key requires tamper-evident hardware local to the verifier anyway. Otherwise how do you know what your hardware is doing, how do you know it's got the right verification key in it, and how do you know it's actually applying that key to the credentials and not just saying "OK". You've got to have some guarantee that what's in the hardware is what you thought it was when you put it in there, and since you've got to have that mechanism anyway, you might as well use it.

So all this third party stuff is really a red herring. The advantage claimed for asymmetric key reduces ultimately to the assertion that asymmetric key makes signatures unforgeable. The people who share a key can't masquerade as each other, because they don't have the same secret. But you need to log transactions to establish time order relative to events like revocation of a key, or of a certificate or a capability or a role or something like that. You need to keep a log at each end, and anything that keeps a log can do key mapping.

Matt Blaze: Are there any events like revocation other than revocation? If you design systems that don't have revocation, then is it important to allow revocation?

Reply: A lot of people say, I don't have revocation in my system. When I probe a little deeper, it usually turns into a push vs pull debate, a debate about whose responsibility it is to get a freshness certificate. I can say, you have to send me a freshness certificate or I don't believe your key, or I can say, I have to get a freshness certificate from someone or I don't believe your key, or I can say, I will conditionally commit your transaction, and then I will go and get a freshness certificate that was issued after the commit point and then I'll believe it was your key. Now if you want to say that those systems don't have revocation then I claim there are things like revocation that aren't revocation, but for which my argument applies.

John Ioannidis: It's a fine point between needing a freshness certificate or having a short term certificate.

Reply: Sure, and I'm not trying to say that revocation is an issue in every system. There are shades of grey, and eventually you get to the point where you

[7] For details see Bruno Crispo, 1999, Delegation of Responsibility, PhD thesis, Computer Laboratory, University of Cambridge.

can say, I don't have revocation, or at least I'm happy for it to take a month to revoke things.

Sometimes the thing you're trotting off to isn't a revocation authority or a freshness certificate server, sometimes it's a time authority, or sometimes it's actually a logging authority: part of the credentials you have to present to commit this transaction is evidence that your intentions have been logged for the purpose of recovery or whatever. The point is that you're going off somewhere, and something is happening in an environment which facilitates key mapping. But I agree that while you can stretch this argument out in various directions, it doesn't apply to everything.

So the point about key mapping is, that Alice sends her token, encrypted with the key that she shares with Bob, but also with the key that she shares with the port S that the token is going through.

$$A \longrightarrow S : \{X\}_{Kab}; \{X\}_{Kas}$$

The port does the key-mapping and appends the token encrypted under the key that Bob shares with the port.

$$S \longrightarrow B : \{X\}_{Kab}; \{X\}_{Kas}; \{X\}_{Kbs}$$

Bob can't forge all the tokens that are logged, because he doesn't know the key that Alice shares with the port. But Alice can't forge the credentials that are presented to Bob either. Neither can the port.

More generally, if you're communicating between domains, security policy is typically enforced by firewalls, which may be internal to the domain. It's not the case that every message has to go through the firewall, but typically setting up an association requires some sort of policy approval, and firewalls do protocol mapping anyway, firewalls keep logs anyway, and firewalls apply security policies anyway, and the two firewalls at either end of a pipe tend to be mutually mistrusting, and therefore they're good pieces from which to build a system like this.

There are considerable merits in forcing tokens to have different bit representations in different security domains. So the idea is to have border controls. You have to change your money as you go through the firewall, and if you present money that you've received from another domain by a covert channel, it won't work. If someone's rung you up and dictated the bit pattern of a capability down the phone, it won't work, because it didn't go through the appropriate security policy.

Li Gong's ICAP mechanism[8] can readily be made to be like this. ICAP actually had a central server and application of authentication to individual users using shared keys and hash functions. But if you put this server in the firewall, in the gateways instead of in the middle of the domain, and you put domain-based secrets into the authentication code, then you can build capabilities which have

[8] Li Gong, 1990, Cryptographic Protocols for Distributed Systems, PhD thesis, Computer Laboratory, University of Cambridge.

a different form in the different domains, but which can be checked by the policy servers in the host domain. The authenticator is

$$h(\text{Object} \mid \text{Subject} \mid \text{Rights} \mid \text{Domain Secret}).$$

So modifying Li Gong's ICAP gives you another valuable building block.

The next building block is the use of hash chaining, which you can use to take the mutually mistrusting parties off-line. The MMPs have to be involved when you first set up the association, but the firewall typically has to be involved then in any case. After that the MMPs can go off-line until there's a dispute. The general hash-chaining technique here is a very standard one. We authenticate a shared secret X_0 at the beginning, verify that this really is the shared secret, then I choose a successor secret X_1, hash it, hash that together with X_0 and the signature S_1 of the thing I'm about to commit to.

$$A \longrightarrow B : h(X_0|h(X_1)|S_1)$$

I then get a commitment from you, confirmation that this intention has been logged at the far end. Then I reveal the signature, the current secret and the hash of the next secret, and on we go.

$$A \longrightarrow B : X_0, h(X_1), S_1, h(X_1|h(X_2)|S_2)$$

This is one brick, it's not the wall. What we do is to take this and put it into a Desmedt and Quisquater type box, but one where there isn't a global secret. Instead, the secret depends on the different certificates are that are involved in the transaction. These are symmetric-key certificates, not asymmetric-key certificates, so parts of the certificates are secret. But the key you end up with can be made public. In the event of a dispute you say to everybody, OK produce your tamper-evident token, together with a log book that includes the hash chains and the signatures and so forth[9].

Although you have control of the token, it may actually be the property of the other domain. The other domain has given you this token and said, keep this token safe because otherwise disputes are likely to be resolved against you.

The log book you can keep on a piece of paper or on a write-only optical disk, but it's your responsibility to keep it. If you can't produce it, or if the transaction done by the verifier doesn't tie the current value of the hash in the token with the end value of the log, the last entry on the log, then the dispute is likely to be resolved against you. So you have the same incentive to keep this stuff secure that you have to not blurt out your private key in the asymmetric case.

Now we come to the first set of conclusions. Users shouldn't know the bit-values of the secrets that they control. And it's a very bad idea to share all of the secret information that you need to commit a transaction with any one other party, it's a particularly bad idea to share it all with an arbiter. You should

[9] For details see chapter 8 of Bruno's thesis, cited earlier.

divide pieces between enough mutually mistrusting parties to ensure compliance with the protocol.

In other words, lack of trust makes the system more secure. Isolate the mutually mistrusting parties, or isolate the parts of them that do this bit of the protocol. Putting them in the middle of a firewall is a very good way to isolate them from things you don't want them to see. Isolate them in such a way that it is very hard for them to agree upon anything other than what the protocol says should be the agreed story.

We're not talking about what actually happened here, we're talking about institutional truth, what we're all going to claim happened given the state of the logs. You impose a heavy penalty on being the odd one out and you make it very difficult to agree on anything else, the prisoners' dilemma type of thing.

The tokens which a user "controls" may be part of another domain, because the controller doesn't have access to their internals. The idea is that you bind things that look like capabilities – I don't want to get into the theological dispute about what they're called, I can't keep up with recent fashion – you bind things that look like capabilities to things that look like certificates, and then you revoke the certificates or the capabilities forward to the point of use, rather than back to the issuer.

In fact if you think about what a key certificate should really look like in the asymmetric key world, it should say, "I, the certification authority for domain A, assert that this key for Alice can be used by Bob in domain B, provided that Alice wants me to certify this, and provided that the domain B is willing to receive this certificate, and provided Bob said he actually wants to use it to bind something to."

$$T = Ka^{+}, Kas^{+}, X, Kb^{+}, Kbs^{+}; \qquad S = \{T\}_{Ka^{-}}, \{T\}_{Kas^{-}}, \{T\}_{Kbs^{-}}, \{T\}_{Kb^{-}}$$

In fact this looks just like a delegation token, a point that Mike Roe made a couple of years ago[10].

So we come to the final conclusions. I started in a somewhat fanciful way by saying, let's see what happens if asymmetric key doesn't work anymore. What would we do if we had to rely on tamper-evident hardware and symmetric key algorithms? Thinking about this suggests a number of things which it would be sensible to do differently, and these changes to how we currently do things still appear remarkably sensible even when the underlying algorithm remains asymmetric. There are differences between symmetric and asymmetric algorithms from the point of view of the resulting crypto-system, but they are very subtle, and they are not at all the differences you find described in books suitable for children.

Any questions or heckling?

John Ioannidis: I suppose if an AS is compromised then whether a signature is subsequent to the compromise cannot be resolved.

Reply: You can't use signatures ad infinitum even now, unless they've been logged or archived or put into something that gives you assurance that they

[10] LNCS 1550, pp168–176.

existed prior to the key being revoked or compromised or the certificate expiring, or some other thing that would have violated your policy if you had checked it.

Otherwise you can't know which happened first, even in the asymmetric case.

Ross Anderson: You assume that publication of a secret key compromises part of the system, but the existence of an append-only file makes it trivial to use one-time signatures.

Reply: That's true. The question that needs to be addressed there is the issue that Matt Blaze raised last year[11], which is the need to layer confidentiality servers on top of the event logger, so that people can't see bits of the log that they mustn't see, whilst still having an assurance that the chain is intact for the bits that they are allowed to see. But again, that's a problem now, it's not a problem that comes about from the fact that asymmetric key stops working.

Mark Lomas: How can I make backups of my private key?

Reply: You make backups of your actual private key? Shame on you.

Mark Lomas: The reason I make backups of my private key is because my machine might be unreliable.

Matt Blaze: But if you had a good public key infrastructure you'd simply create new keys.

Reply: Exactly. What you actually have is a requirement to be able to transfer your rights to a different key in a controlled way under certain circumstances, or equivalently here to a different physical token, without having a period of a month while you can't access your system.

There are two arguments for never moving private keys around. The first is that if it's physically possible to move a private key from one place to another, then someone is going to do it when you don't want them to, or make a copy while it's in transit. It's usually when keys are moving around that they get filched.

I agree that you can make the process of moving them safer, but what I'm saying is that it's possible to give an excellent solution to the wrong problem and very often moving a private key is the wrong problem, it doesn't need to be solved.

The second reason is that very often when we run a protocol, one of the things we want to authenticate is that a particular specific piece of hardware, say a particular hand-held device, was involved in the protocol. Then it's very important that we satisfy ourselves that it wasn't some other piece of hardware that had the same noughts and ones in it. The easy way to do this is if there *is* no other piece of hardware with the same noughts and ones in it. And in those cases you have to be able to guarantee not simply that you didn't move the keys around, but that it wasn't possible for you to move the key, you could not have done it.

How can we meet this requirement? We tend to think that binding and unbinding capabilities from certificates is really hard, and revoking certificates is a real pain, and certifying things is really difficult, and it's a rare event anyway, so let's try and avoid doing this as much as possible. What I'm suggesting is

[11] LNCS 1796, p46.

a world view where these things actually happen all the time. Whenever you want someone to use a public key that *they* haven't used before, that's a certification transaction. When they say "I've finished using it", that's a revocation transaction. What we need is an infrastructure that deals efficiently with these events, and then you can just put keys in one physical place and bind them there for life. Then all your bindings between bit patterns and real world entities are happening at the edge of cyberspace instead of in the middle. The last thing you want to do is to bind two bit patterns to each other via a real world entity because then you can't verify the binding remotely[12].

Matt Blaze: I guess the major reason for not revoking secret keys is that it's psychologically a bit embarrassing to have to revoke one. Revoking a key is a very public event.

Reply: You're absolutely right, we have to take the social stigma out of revoking your key. We need a sort of revocation pride march or something. Reclaim the night, revoke the right.

John Ioannidis: It's not too wise to say why you revoked your key, because then everyone knows it's lost and that you don't want other people to send you anything.

Reply: I don't have to say why I revoked a key. Maybe the session ended, or maybe I'm just bored with it. All I'm going to say is, my name is Bruce and I just revoked this key.

[12] For more on this theme see LNCS 1361, pp105–113.