Robbing the Bank with a Theorem Prover
(Transcript of Discussion)

Jolyon Clulow

Cambridge University

So it's a fairly provocative title, how did we get to that? Well automated tools have been successfully applied to modelling security protocols and finding attacks, and some good examples here are Gavin Lowe's work, using FDR to model the Needham-Shroeder protocols, and Larry Paulson's work using Isabella to prove the SET protocol secure. Now we come to the observation that security protocols, and security application programming interfaces are very closely related. So just to define what we mean by a security API here. We're talking devices that offer security services, that will obviously have some interface, typically the application programming interface, and unlike a normal API it also has to enforce policy onto the user, it has to make sure that keys remain secret, that PINs aren't revealed, and that users can't generally do things that would violate the security policy.

Now the security group at Cambridge has been looking at security APIs for a couple of years now, and we know of quite a few attacks on various security devices, and some examples here are attacks on the IBM common cryptographic architecture. This is a device that is typically used in a banking environment, it performs the backend PIN verification and key management tasks within the bank, and this is how we get to the title of Robbing the Bank, with the Theorem Prover being an automated reasoning tool. The question we were trying to answer is, could we take one of these tools and rediscover all the attacks we knew of already, and then could we go one step further, could we find any new attacks.

The tool we chose is OTTER, OTTER is a first order predicate calculus theorem prover, so it's a system of deduction that extends propositional logic. The basic idea is, you start with a set of initial knowledge, you have a whole load of inference rules, you will apply those inference rules to that set of knowledge and generate new pieces of knowledge which if they are new and unique you add back to your set, and while it's doing this OTTER is trying to look for any inconsistencies, where an inconsistency here is something that's claimed to be true and not true at the same time.

So how do we go about using OTTER to model an API? There are a couple of things we have to do: we have to have a way of representing initial knowledge, we have to have a way of representing the commands and the API themselves. We also have to have a way of representing the attackers offline abilities, so for example, if I know two values I can XOR them together, or I can encrypt one with the other, and I don't actually need to use the security device to do that, I can do that on my own PC. The other crucial thing we need is a way to represent

B. Christianson et al. (Eds.): Security Protocols 2007, LNCS 5964, pp. 172–177, 2010.

the security properties that we tried to model. The reason for this is without security properties we can't find violations which would represent the attacks. So let's just go through those.

Initial knowledge: the fundamental unit of our model is this user knowledge predicate, so if we want to say that the adversary knows the value x, we just write U of x, so for example, here, U of m means the adversary knows the plaintext message m, we would also expect the adversary to know the specified control vectors, such as the data control vector, and the adversary will also be able to see wrapped keys, it won't know what the clear value of the keys are, but it will certainly see the encrypted token. So then we go to offline computations. These represent using the inference rules, and the example here is of XOR, and the way you should read this is, the user does not know x, or the user does not know y, or the user knows the XOR of x and y, and that's just another way of writing the statement that says, if I know x and y, then I know the XOR of x and y. So now to the actual commands in the API, again they get represented by inference rules, and if we go to our example of encrypting with the data key, this would be the message, that would be the wrapped key coming in, and if I have both of those then the device will output the message encrypted under the key. And just to make the point, if we don't know the clear value of the key k, then this is the only way we can obtain messages encrypted under that key, and again, encryption is assumed perfect here, and everything is consistent with a Dolev and Yao style model.

The security properties: well by example, one security property is that the master key remains secret, at no stage should the adversary ever learn the value of km, so we want to be able to identify when the term, user knows km, is derived. Since OTTER looks for inconsistencies, we put the negative of this, we specify as initial knowledge that the user does not know km, so now if OTTER ever does derive km this will be an inconsistency which OTTER will have proved, and the steps in that proof will actually represent the attack. So, so far so good. We should be able to plug this model into the tool and run it, but when we do this we find it doesn't terminate, or at least after a few days, it was still running happily crunching the equations.

So then we had to look for some optimisations. When we analysed what the tool was doing, it was wasting a lot of time with nested encryptions, so for example, it would take two terms, encrypt one with the other, then encrypt it again, and again, and one could imagine maybe one or two levels of encryption being useful, but excessive nesting would seem to be of little value. Another thing is it would use things like these constant control vectors as keys, and again, it's hard to imagine how that might be useful in an attack. So we removed the tool's ability to perform nested encryptions by adding in two new predicates, that's Un and Ue. The idea with Un is it means the user knows this piece of knowledge, but it's not the result of an encryption operation, and Ue is the user knows this piece of knowledge but it is the result of an encryption operation. So this modified offline encryption inference rules says, if I know x and I know k and neither is the product of an encryption operation, then I'm allowed to use

them in an offline encryption and generate this term, and this term is marked as having been generated by this process, so it could never be fed back as an input. So we've pruned the search tree here, which means potentially we're going to miss an attack.

Our perhaps biggest problem was dealing with XOR, Now XOR is very common in many cryptographic protocols, and certainly for the attacks we were trying to model it's a requirement. The problem with XOR is it induces term equivalence classes: there's two ways to write A XOR B, you can write it A XOR B or B XOR A, and this gets more complicated as you have more terms, and you include brackets. It's not just XOR that has this problem, it's any equivalence inducing term, anything that is associative, commutative, has identity elements, is nilpotent, or has this problem that different syntactic descriptions have the same semantic meaning. So how do we deal with this? For example, if we have user knows A XOR B, and we claim that the user does not know B XOR A, we would want OTTER to spot that as an inconsistency.

So what can we do? One thing we could do is write all the properties of XOR down as inference rules, so if you have A XOR B then you also can derive B XOR A, and we could essentially rewrite every piece of knowledge we have in every other possible way, and this would probably solve the problem, but it's a very expensive approach: it adds extra inference rules that you have to keep applying, and it adds a whole lot of extra data that you have to store and work with, so it's very inefficient. So we opted instead to force everything into a single canonical form, so if we did have B XOR A we forced it into a form A XOR B, and we had rules for defining what was our canonical form. And we could do this using OTTER by a fairly common rewrite rule call demodulation, which is there to simplify clauses within the theorem prover. But this in itself was not enough, because there were some situations where we couldn't force demodulation. So if you have a look at this example, we're saying that if we have x and we have y, and we have the XOR of x and y, then we achieve some goal. And then we have these three statements, and you see that we would like B XOR C to unify with x, D XOR E to unify with y, and B XOR C XOR D XOR E to unify with the XOR of x and y. But this will fail because it will be looking for that expression in this form, which is different to that. So how did we deal with this? Well we introduced something we called intermediate clauses, and we would take that inference rule and split it into two new rules, as you can see here we've created an intermediate predicate. So what happens now is when B XOR C unifies with x, and D XOR E unifies with y, we generate this term, and once we've generated this term, demodulation can happen, and that will then rewrite it in our canonical form, at which stage it can now unify with that other piece of knowledge we had on the previous slide. And so everything will unify as we would like, and we can progress.

So what were our results like? We analysed two models. We analysed a minimal model, which was just the commands we needed to recreate these attacks. Now this presupposes some expert knowledge, you effectively know what you are looking for, and in some sense this is, of course, unrealistic, particularly if you're going to try and find new attacks. So we then had a complete model in which we added a whole lot of additional commands which we thought somebody

trying to model this without initial knowledge would have done. And with our optimisations OTTER was able to derive unknown attacks. Now depending on how we specified the security goals, we could find different variations on attacks, so if we said, could you find a key encrypting key, OTTER would find that, and if we said, could you find a PIN encryption key, OTTER would find that. OTTER also, which we didn't expect, found a novel variant of these attacks where it first recovered a key of a certain type called a PIN derivation key, and then it went on to derive a user's PIN via the appropriate algorithm. And just to emphasise, these are non-trivial attacks, it takes OTTER 16 logical steps to get to the result.

So, to look at the numbers. Without optimisations, and using the smallest possible model, we failed to terminate, OTTER was still busy thinking after a couple of days, but as soon as we include our optimisations, we very quickly find there are attacks. So if you're thinking of applying one of these tools to your particular problem, whether it's a security protocol in an anonymity system, or whatever it happens to be, the lesson here is, a naïve first pass attempt might well fail because of these complexity issues, and you'll have to think long and hard about what optimisations you would need in order to get the result that you desire. It's not necessarily that the result is impossible, but you'll have to be a bit clever about it.

So conclusions: we have demonstrated that we can use an automated reasoning tool to find attacks in an API, and we've developed a couple of techniques to aid us in this. And the techniques are actually very general, although we've used OTTER, all we actually need is any automated reasoning tool that supports demodulation, or the ability to rewrite terms into a canonical form, our techniques should move across to other tools. And we're hopeful that these techniques will be able to find some new attacks, we can't claim that we have yet, but hopefully some time in the near future.

Bruce Christianson: You were almost starting to put semantic information in as tags to the bit patterns that you're getting, you're distinguishing between those which come about as a result of a cryptographic operation, and those which don't.

Reply: Yes, we have, but we've done it for a very specific reason.

Bruce Christianson: I'm not saying it's a bad thing to do, in fact I'm wondering whether you might take that further.

Reply: There are other tools that use more type information, and that's not something we've looked at. But in protocol analysis, often attacks are type-casting attacks, where you think something is of this type, and then the attacker goes and uses it in an unexpected way, and the danger with a tool that does use typing is that you might miss those kinds of attacks.

George Danezis: Surely though a good thing to do is to take this analysis and say, typing information is good because it allows us to reason about this thing formally, and then modify the CCA protocols to actually enforce the typing information, so that you cannot use the result of an encryption, then re-encrypt, and then use it as a key, and such things.

Reply: Yes, I would agree with that, I mean, in one of the attacks, we effectively turn a key encrypting key into a data key, and so we can export a key under this key, and then we can use the resulting wrapped token, as data into a data decrypt core, and extract the key, and we generally feel that that's a failing in the key token and the key typing system, but that's a separate paper.

George Danezis: I guess my comment may be slightly more general, but it would be a good idea when people design protocols to run them through a theorem checker rather than design, implement, deploy, and then ten years down the line having other people running them through theorem provers, and finding out then.

Reply: There was some discussion saying, it would be nice to have a tool that would take your protocol specification, analyse it for correctness and for security properties, and then at the same time produce the actual C or Java code at the end of the day. That would be a really nice tool to have, but it might mean a lot of people start designing protocols, which would be potentially a downside.

Bruce Christianson: You use the word prover, but really it's much more like a debugging tester, it will show you that there's an error, but it won't show you that there isn't one[1].

George Danezis: In this particular case you can exhaustively search the space.

Michael Roe: No, they can only make the search space finite by ignoring some things that potentially might be attacks, so the fact it didn't find it doesn't mean there isn't an attack.

Bruce Christianson: It's worse than that, the danger is that you will find the attack and put in a fix, and the fix has the effect of moving the problem to a part of the space that isn't searched.

George Danezis: But if you enforce the type system at the same time, because you're just designing the thing you have the flexibility to change it, and then you can guarantee that there is no possibility of having other attacks.

Bruce Christianson: That's quite a major change in the way protocols are designed, yes, we should think about that.

Michael Roe: The hard part of this is that using XOR in the way CCA uses it is a bad idea, it causes problems in the reasoning using a theorem prover, because these attacks rely on XOR having properties that you didn't need for the protocol to work, but which the attacker can exploit to make the attack go further.

Reply: Yes, you get the same problem when you start having public key protocols, and where you have g to the x, and you multiply it by g to the y, and you have addition and multiplication, and that's an equivalence inducing operation as well. You have the same issues there, so I agree the attacks we were modelling here are XOR specific, but the problem of dealing with equivalence classes is more general.

[1] Editors' note: Formal methods can show the presence of errors but never their absence.

Michael Roe: For example, you can often replace XOR with a hash function, so instead of x XOR y you have hash of x concatenated with y, and that both makes your theorem proving task much easier, and makes quite a lot of the attacks go away.

Bruce Christianson: Show me your hash function?

Michael Roe: Well, assuming your hash has the right properties.

Bruce Christianson: It lets you replace one set of assumptions with another set that you can't test[2].

Shishir Nagaraja: But still, Jolyon what's the amount of text that OTTER generates with the optimisation?

Reply: It's about three hundred thousand pieces of knowledge, the clauses it generates. The minimal model has about three commands in it, and the complete model has about seven or eight, so you can see the explosion in the number of terms.

Shishir Nagaraja: So I'm just wondering about the amount of horse power that you need to analyse the OTTER output.

Reply: Well when it finds the attack it's very simple, you get your printout of the 16 steps, and you can go through and verify that everything is correct. When it's not been successful then we were just looking at random samples and saying, look at all these obviously useless terms, but that is hardly a comprehensive analysis. There might be some more elegant and efficient optimisations that we could have applied, maybe if we'd looked for longer we would have seen them.

Alf Zugenmaier: Is it easy to figure out whether what you have specified in OTTER is in fact the protocol that you're trying to verify if you don't find an attack? If you find an attack you have one trace, you can run through it and verify that this attack actually works, but if you just get three hundred clauses then ...

Reply: Well, yes, if we've done something wrong and modelled the command incorrectly, then that would be undetected, we were relying on our own expert knowledge to verify that we were doing the right thing.

Michael Roe: Of course there's quite often additional properties that the real implementation has that you're not modelling, so there might be computations the attacker can do that you haven't modelled as deductions.

Reply: Yes, this is, I guess, a typical statement: we only have searched within the model that we have created, and we make no claim about what lies outside the model that should have been in there.

Michael Roe: But that does mean that when you don't find anything it doesn't give you any kind of warm fuzzy feeling the system is OK.

Reply: Yes, but we carefully don't make that claim.

Bruce Christianson: You're not out to make the banks feel warm and fuzzy.

[2] Hash Functions with Mystic Properties, LNCS 3364 p 207.