Formal Methods (Tools)

Scribe: Sergey Berezin

J. Wing

These tools are supposed to help proving that a system is secure. How can we use a model checker or a theorem prover to prove the security of the system, and what does it mean formally for the system to be secure? We will address this issue, along with strengths and limitations of the approaches.

1 Historical Prospectives.

Since NSA (NCSC) was a major source of funding for verification systems in the US in the mid-70's to mid-80's, a lot of theorem proving came from there. NSA was concerned with the security of their operating systems, specifically with the security of kernels. The security of kernels relied on use of a *reference monitor* which is supposed to satisfy three properties:

- It must mediate all accesses to objects; that is, check all user interactions with the kernel and make sure they are "secure";
- It must be tamperproof;
- It must be small enough to be subjected to analysis and test.

And one such form of the analysis is formal verification. But to use formal methods we first need to define formally what it means for a system to be secure. One of the most obvious models was the *Bell-LaPadula model.* This model interprets a system as a collection of *subjects* (e.g. users), *objects* (e.g. files) and *access modes* (e.g. read, write) with the state machine semantics. The main security property is a so-called $*$-property: *writes up and reads down in the security hierarchy are only allowed.* That is, Jeannette (unclassified) can write to her (confidential) files, and Jim (confidential) has to be able to read Jeannette's files, e.g. to figure out her salary. But Jeannette can not read confidential files (because Doug's salary can also be there), and Jim can't write to unclassified files, since he might leak the confidential information of Doug's salary to Jeannette.

The precise security levels were described in *The Orange Book.* It has the hierarchy of 4 security levels:

D: *Minimal*; reserved for systems that have been evaluated but failed to meet the requirements of higher evaluation levels.

C: *Discretionary* (3 classes); provides "need to know" protection and accountability of subjects and their actions. Looks similar to UNIX access control, where users have the control.

B: *Mandatory* (2 classes); here system (not a user) labels objects according to a fixed policy. This level requires that a system developer state the policy model, provide a specification of TCB, and *the evidence* to demonstrate that the reference monitor has been implemented correctly.

A: *Verified*; this is similar to B, except that it requires the use of "formal security verification methods" for the *evidence.*

Examples of the "A" systems are

- MITRE SCOMP: 8-bit Honeywell XTS-200 processor;

- Ford Aerospace: Multinet Gateway System.

These are the only (known) "A" systems, which means that getting the A-level certification is really tough.

1.1 Four Theorem Proving Systems (1984).

To prove that a kernel is secure one has to show that it satisfies the A1-security requirements. The idea then is to write such a kernel that can be proved secure using formal methods tools. And since someone has to write these tools, NSA were giving a lot of grants to support the development of different theorem provers.

- **FDM** (built at SDC). The theorem prover is called InaJo. It used the method of proof by contradiction. The model is a nondeterministic state machine.
- **HDM** (built at SRI). Has a very rich specification language. Incorporates lambda calculus, second order predicate calculus, Hoare sentences and some kind of Pascal code. The theorem prover has decision procedures for the first order logic and for a subset of integer arithmetic.
- **Gypsy** (University of Texas). It is Pascal-like program verification system. It generates *verification conditions* that you need to prove in order to show that the program is correct.
- **Affirm** (USC/ISI). It uses algebraic specifications and the rewrite rule engine of Knuth and Bendix.

1.2 What Does It Mean for a System to be Secure?

To be able to use formal tools, one has to formulate what the security means formally. At first, the *Basic Security Theorem* (BST) was proposed, that states that

- A state is *secure* if it satisfies the $*$-property (and two other related properties).
- A system is *secure* if and only if the initial state is secure and the state before and after each action is secure.

What Does It Mean to *Prove* a System to be Secure?
As McLean showed in the IPL'85 paper, the BST does not guarantee any security. He proved that an obviously insecure system (with the *dagger property*, the reverse of the $*$-property) can still satisfy the BST. In the draft of this paper McLean constructed a *system Z* that has only one action that downgrades all subjects and objects to the lowest possible level. This system still satisfies the BST.

Conclusion: "We have shown that the BST does nothing to establish that a system is really secure." To prove that a system is really secure one has to show that during every action the state remains secure.

After that theorem proving quieted down at NSA, but other independent researchers picked up the ideas and have been developing them separately. In 1989 on one of the conferences there was a circulating document saying that "if you are going to build a secure system you need to generate a proof checkable by an independent proof checker." Another document provided a list of approved tools and methods for reasoning about security. This shows that even after the failure of the BST the formal approaches were still alive.

2 Approaches to Protocol Analysis (Today's View).

Derek R. Dreyer & Darrell Kindred

Most formal approaches to reasoning about security are combinations of of the following:

- **Algebraic:** a collection of rules for transforming and reducing algebraic expressions representing messages. Example is the Dolev-Yao algebra.
- **State-transition:** sending and receiving of messages are modeled as state transitions. The problem reduces to finding if a certain (bad) state is reachable.
- **Logical:** set of axioms and inference rules describe a system of knowledge and beliefs. This is, e.g. the BAN logic.

2.1 The TMN Protocol.

In the Tatebayeshi-Matsuzaki-Newmann (TMN) protocol A and B want to obtain a common session key though the mediation of a trusted server S. The protocol uses public key cryptography for communication with the server, and symmetric key otherwise. A typical run is on figure 1.

r_1 is the (symmetric) key of A, r_2 is the (symmetric) shared key between A and B established during the protovol run, and e is the Server's public key.

Obvious flaws in this protocol are

- No secure authentication;
- Intruder can impersonate B.

The last attack is not possible if the intruder can not intercept messages. But even with this assumption it is still vulnerable to the attack.

Simmons' flaw.

Assumptions:

- two collaborating intruders,
- public key encryption is a homomorphism, i.e. $\{a\}_e \cdot \{b\}_e = \{a \cdot b\}_e$ (this is true for RSA).
- symmetric-key encryption is "commutative", i.e. $\{r_2\}_{r_1} = \{r_1\}_{r_2}$, which is a strange assumption as it is not true of most algorithms including DES (but TMN was originally for use with one-time pad).

The flaw is described in the chart below (fig. 2).

C and P conspire. A starts a protocol session by sending $B, \{r_1\}_e$ to S. Then C starts another session with P sending $\{r_c\}_e \cdot \{r_1\}_e$ to the server, which is the same as $\{r_e \cdot r_e\}_e$. Then the protocol proceeds normally and at the end C receives $\{r_p\}_{r_c \cdot r_1} = \{r_c \cdot r_1\}_{r_p}$. Since C knows r_p and r_c, he can get r_1.

2.2 Ina Jo.

This theorem prover has a formal specification language from the Formal Development Methodology. It is a general-purpose tool for software development and correctness proofs. It acts as a theorem-proving assistant, that is a proof checker. In our context it is used to prove that security properties hold throughout the run of the protocol, through *each transition.*

Ina Jo uses the *state-transition approach:* the user specifies

- a set of *state variables* that change values as protocol proceeds, and

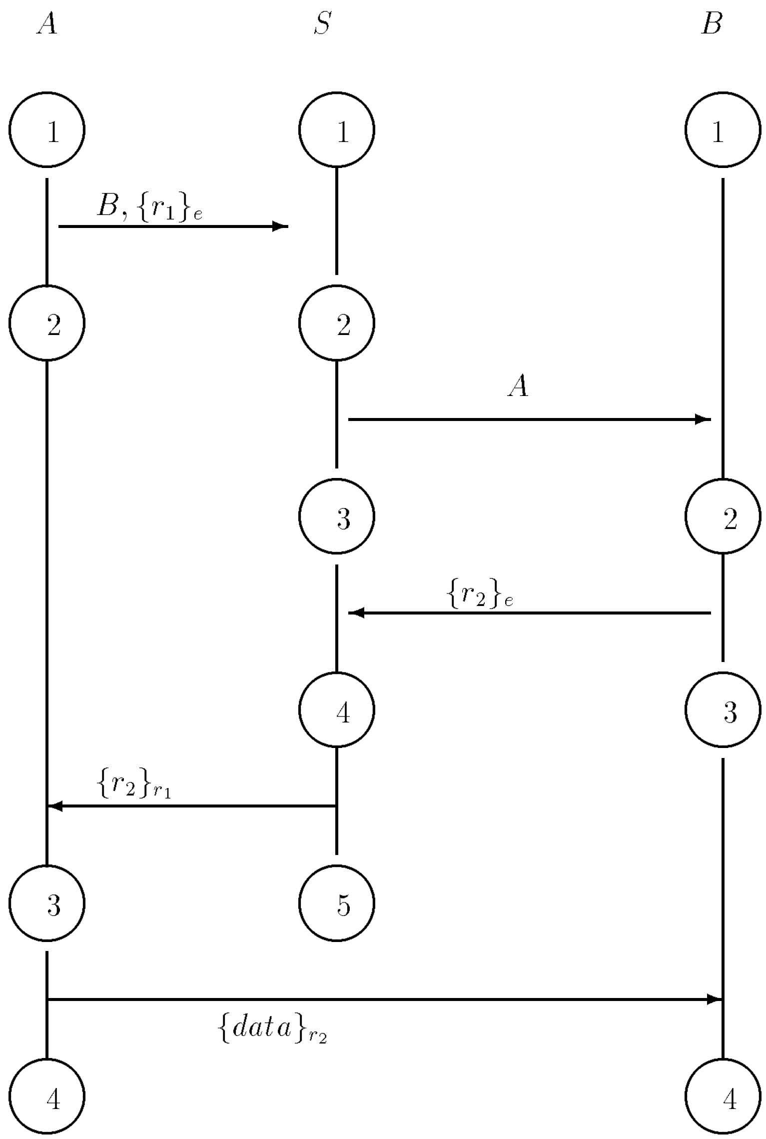

Figure 1: TMN Protocol Run.

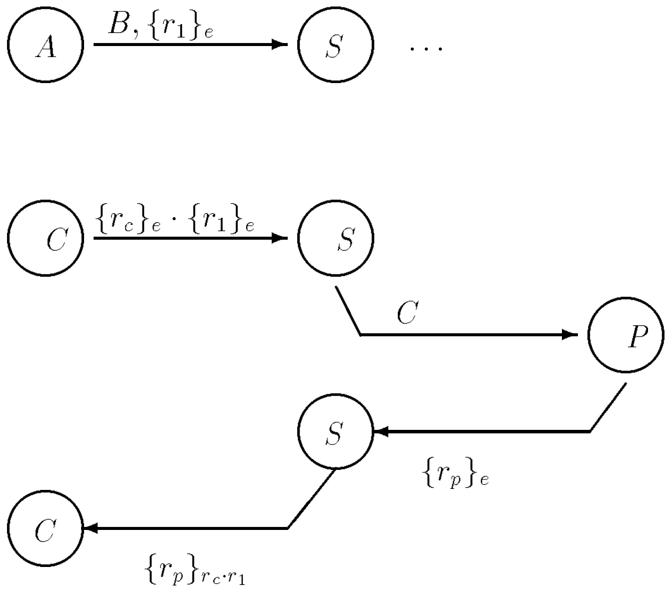

Figure 2: Simmons attack on the TMN Protocol.

- *transforms* that model how variables get modified during state transitions.

Axioms describe assumptions about system and properties of cryptographic primitives (e.g. commutativity, homomorphism, etc.).

Criterion describes what invariant properties of the protocol are desired (e.g. "an intruder doesn't know what he shouldn't know").

The prover provides a list of proof obligations, one of the form

$$\text{INIT} \rightarrow \text{CR}$$

(Given initial conditions, prove that CR holds), and the others of the form

$$\text{CR \& R \& E} \rightarrow \text{N}'' \text{ CR}$$

(if CR holds in the current step, then it holds at the next step).

Then it can aid you in verification of the listed theorems by acting as a proof checker and doing bookkeeping.

Failed Proofs.

In the TMN example, an attempt at proving the theorems that Ina Jo generates would fail. And hopefully, failed proofs hint at flaws (this is connected with an idea of "proof-directed debugging"). The *Inatest* will help you to "step through" a suspected flaw scenario.

Kemmerer's Inatest

It is a tool for symbolic execution of a protocol run. Given an Ina Jo specification, a start state, a sequence of transforms and an optional resultant state, Inatest interactively allows you to:

- create temporary variables;
- save states;
- fire transforms;
- do algebraic manipulations, etc.

It can also be used to try out the protocol specification on different test cases (one of which might be an attack).

Evaluation of Ina Jo/Inatest

The tool is:

- useful for working through formal proofs of correctness (Ina Jo) or incorrectness (Inatest);
- flexible; can be applied to different protocol settings;
- not too useful for finding previously unknown flaws;
- Derek's major criticism: "You have to feed it the flaw!"

2.3 Millen's Interrogator.

The tool is implemented in Prolog and uses Prolog's search engine to achieve higher automation. It also uses state-transition approach: the user specifies a set of communicating state transition machines.

The tool tries to satisfy the goal `pKnows(D,H,B,S)`, where `D` is data that parties want to communicate secretly, `H` is message history (the attack), and `B` and `S` represent insecure goal state. The program searches backward from insecure goal to find a penetration scenario `H`. It is very advisable to specify `H`, otherwise the program may not terminate.

Structure of the Interrogator.

The core Interrogator comprises four main sections, first two are built-in, the last two are user-specified.

1. **Knowledge**. It contains `pKnows`, rules for data extraction from messages and rules for representing and interpreting encrypted data.

2. **Reachability** searches for message histories that take the network from an initial state to the goal state. It is implemented by moving backward one state transition and recurring (valid state transitions are specified in the Protocol section).

   The way it works could be describe as follows: If an intruder knows the data, then someone must have sent it over the network. Hence, this someone was in a certain state, and now we check if this state is reachable or not. So, we look at the current state, find previous states and check if they are reachable.

   These two sections interact when the Interrogator modifies the messages of the penetrator.

3. **Protocol** is the heart of the specification. `transmit` and `receive` rules specify transitions between states. Only one party changes its state at a time. Any received or sent message appears in the network buffer. The specifications are fairly easy to read.

4. **Scenario** describes assumptions about the penetrator's initial knowledge, "hypotheses" (i.e. suspicions) about the final state of the network after penetration. This is not so much a scenario, but rather a description of what is the state after the attack. Naturally, the more information is given, the more likely the system terminates and finds a flaw.

The Encryption Model.

The baseline Interrogator has a fairly simple representation of encryption:

- **Public Key crypto:** if `inverseKey(pkey,skey)` is in the knowledge, then the knowledge of `[pkey,data]` and `skey` implies the knowledge of `data`.
- **Symmetric-key crypto** is simply when `pkey=skey`.

- **Algebra Section** supports algebraic approach to protocol analysis and separates public key and symmetric key crypto as well (e.g. `dec(k,enc(k,x))`$\rightarrow$`x`).

Protocol Specifications

Consist of **transmit** and **receive** rules, each of which has three arguments: the prior state, the message transmitted/received, and the next state. A state is a list with the party's name, state number and all data he has learned up to that state. Messages are lists with source, destination, data fields and format label.

Specifying the penetration scenario

`pKnowsInitially` clause states what the penetrator knows at the beginning. It's almost necessary to specify a goal state in order to have any hope of the Interrogator terminating. Also the user can cut certain useless search paths by designating some data `hopeless`.

Once the specification is written, Interrogator runs free of human interaction. In the case of TMN, Millen writes: "After about 12 seconds on a Macintosh IIcx, the program prints 'Found data!' and displays the penetration history `H`..."

Drawbacks

Again, quoting Millen: "The principal drawback of the baseline version of the Interrogator is that the time it takes to find a penetration can vary wildly depending on details of the protocol specification and the assumptions made for the scenario... The program takes more time if less information is specified about the goal state. For example, if either the key used by a or the final state of b is left unspecified, the program runs for so long it has never seemed worth the time to find out exactly how long: at least 10 minutes."

"The running time is also sensitive to the order in which constants are declared, especially keys, since there are many situations where it tries the possible keys in the given order, and it finds the solution more quickly if the one it needs occurs early in the list."

And the last drawback is an artifact of the system's being implemented in Prolog.

2.4 NRL Protocol Analyzer.

This tool has been developed by Catherine Meadows in about 1991. it is intended to verify security properties and detect flaws in protocols, and is similar in flavor to Interrogator (e.g., similar backwards reachability analysis), also implemented in Prolog. Some 'feathers in its cap':

- Simmons Selective Broadcast Protocol,
- Burns-Mitchell Resource Sharing Protocol.

NRL Model

is Dolev-Yao-style term rewriting. Participants accumulate beliefs and keep message counters and id's of the protocol runs. The protocol is a machine the intruder manipulates to produce words, beliefs (in itself and other parties) and events.

NRL Specifications.

Specifications have four sections:

1. **Transition rules** for honest principals: pre-conditions, post-conditions, event statements. One can reason here about, say, exposure of a key, and can model "expected failures".
2. **Operations** available or unavailable to the intruder (e.g. encryption/decryption, etc.). For each intruder ability special transitions are introduced. There are some built-in operations, like `concat, head, tail`, but *not* encryption. For example:

   ```
   fsd1:e(X,Y):length(X)=1:length(X)=1:length(e(X,Y))=length(Y):pen
   ```

   A sample transition rule:

```
rule(3)
If:
count(server)=N,
intruderknows([user(A,Y),user(B,X),R]),
then:
count(server)=s(N),
lfact(server,N,key,s(N))=[user(A,Y),user(B,X),key(server,N)],
intruderlearns([e(key(user(A,Y),(user(B,X),key(server,N),R)))]),
EVENT:
event(server,N,sentkey,s(N))=[user(A,Y),user(B,X),R,key(server,N)].
```

3. **Atoms** used in the protocol, such as names, keys, nonces, etc. They may or may not be known to the intruder. E.g.

   ```
   atom8:pubkey(server):1:known
   ```

4. **Rewrite rules** reduce words to simpler words, e.g.

   ```
   rr1:pke(privkey(X),pke(pubkey(X),Y))=>Y
   ```

NRL Assumptions.

- Principals are honest or cooperate with the intruder;
- Intruder only learns words or affects a state by application of transitions and rewrites. Because of that, e.g. compromised session keys have to be modeled specially.
- Intruder understands significance of any word it sees;
- words can not be identical unless they unify.

NRL Analysis Goals.

The goal is to prove that undesirable states are unreachable. The user has to provide a "bad state" to the analyzer, and then it tries to prove that no initial state can lead to it. The process requires significant user guidance to narrow the search. There is also an automatic mode that performs BFS, but this is not much practical.

TMN Analysis.

NRL is able to find the 'impersonate-B' attack, but does not find the Simmons' flaw, since it requires algebraic properties which are hard to model by rewrite rules.

Summary

NRL has a sophisticated state model. It's failed proofs produce sample attacks, and there are some success stories. The drawbacks are that the user has to make sure that the "bad states" he proved unreachable are the only bad states. The tool also requires substantial human interaction.

2.5 Inductive Method: Isabelle.

This method is developed by Paulson in 1996 and it uses Isabelle/HOL — a general-purpose higher-order logic theorem prover. Sets of events are inductively defined, and properties are proved by induction over event traces.

Messages can consist of agent names, nonces, keys, etc. and are composed by concatenation and encryption.

Assumptions.

- Encrypted messages cannot be altered or read without a key.

- Structure of messages is explicit (that is, no "type errors"). ("Attempting to model careless implementations would be futile.")

Operations.
There are three main operations on sets of messages:

- **parts H**: messages God could extract from H;
- **analz H**: messages anyone could extract from H, if the key appears in H for encrypted parts of it;
- **synth H**: messages anyone could construct from H using encryption/decryption, concatenation, etc. on messages from H.

Protocol.
Events have the form **Says A B X**. Protocol specification consists of inductive definition of possible traces. An intruder (**Spy**) can produce an event

Says Spy B X

for any X in synth(analz(H)), where H contains messages sent in the past, plus intruder's initial knowledge.

There is also an explicit "oops" rule that represents compromise of a session key.
Guarantees.

- Possibility properties (like successful runs);
- Forwarding lemmas ("forwarding encrypted item reveals nothing new");
- Regularity lemmas ("a spy can never get X" — stated in terms of **parts**). These are special cases of secrecy theorems.
- Authenticity/Uniqueness theorems (provide guarantees about origin of encrypted messages).
- Secrecy theorems. These are most difficult to prove. Here we make use of the **analz** operator.

Summary.
The inductive method was used to analyze Otway-Rees protocol, Kerberos. (Kerberos suggested new "temporal properties".) It has relatively simple principles and exploits a general-purpose theorem prover. The proofs require a fair amount of human guidance.

2.6 PVS (Bryans & Schneider, 1997).

This tool uses Schneider's framework for expressing security properties in CSP. CSP is embedded in PVS — a general-purpose verification system. They check authentication properties like "receipt of a message in a set T guarantees previous transmission of a message in a set R".

The method was applied to "recursive authentication protocol" that has a number of parties as a parameter, and a known flaw was revealed.

2.7 Other "Deductive" Tools.

- **AUTOLOG** is an automatic tool in Prolog by Kessler and Wedel. It is basically a prolog automation of a BAN-like logic.
- **Revere** provides automatic reasoning with BAN-like beliefs logics. It uses "theory generation" and checks for "honesty" and "responsibility" properties. The work is still in progress.
- **CryptoLog** by De Decker and Piessens. It is a belief-based checker, and it produces GNY-like specifications.