Designing distributed interactions

Ernesto Damiani

Design recipe

- Two simple steps:
  - Build a mathematical model of the object - let’s call it the design
  - Use maths to check that the design, in the context where it is deployed, has the properties we need
- Three basic caveats
  - The design may not faithfully represent the real thing,
  - The expected context may differ from the real one
  - Calculations may contain errors

Applying the recipe to computations

- Maths = formal logics
- Designs = formal descriptions of computations in some logics
  - programs represented as logic formulas
- Property check = deduction:
  - automated theorem proving, model checking, static analysis, etc.
  - Soundness & completeness
- Problem: deduction of properties is computationally hard or even undecidable

Handling complexity

- Restrict/weaken properties to be checked
- Give up deduction soundness and/or completeness
- Represent the computation only partially, via approximate designs
- Use human guidance

Comparison with test and simulation

- Simulation considers a model of the computation, but it's **not** a design
  - Sim model are conceived for execution rather than analysis
- Testing considers the real software implementing the computation
  - Model can be used to generate test cases
- Sim and Test examine only some of the possible behaviors
  - Can't extrapolate from partial tests/executions: only statistical projections

Example

- Lamport’s Bakery Algorithm
  - http://en.wikipedia.org/wiki/Lamport's_bakery_algorithm
- In a waiting room, a machine dispenses tickets printed with numbers that increase monotonically
- People enter the waiting room; when entering, each person takes a ticket from the machine and starts waiting
- When the service becomes available, the waiting person with the lowest numbered ticket is served, and leaves the waiting room

Example

- Which properties are we interested in?
  - Safety: at most one person is being served at a time
  - Liveness: each person is eventually served
- Looks straightforward: the ticket dispenser never prints the same number twice and service time is finite
- Can we preserve these properties without a ticket dispenser?

Example

- Each process has a public register, initially zero
- When it wants to access the service, a process sets its register at a value greater than the one of any other waiting process
- Then it waits until its register is smaller than that of any other process
- At which point it access the service as soon as it is available
- After the service, the register goes back to 0
- EXERCISE: prove safety!

How to do it

- Build a mathematical model (design) of the protocol
- Analyze it for the desired property (safety)
  - Must choose
    - a modeling style that supports the analysis
    - how much detail to include in the design
- The protocol uses shared memory and is sensitive to:
  - memory faults (what if a public register contains a wrong value?)
  - atomicity and ordering of concurrent reads and writes (what if two processes enter the room at the same time?)
- Need the “right” assumptions

Modeling datatypes in logic

- Registers must be modeled as natural numbers
  - Natural numbers : Peano axioms
  - Constructors:
    - 0, succ (i.e. nats are 0, succ(0), succ(succ(0)),..
    - Corresponds to the induction axiom / scheme
  - Freeness axioms:
    - forall x exists : nat 0 =/ succ(x)
    - forall x,y : succ(x) = succ(y) implies x = y

Example

- **Assume: faultless memory, totally ordered atomic read/writes, two processes only**
- Process can be in 3 states: outside_room, in_room_waiting, being_serviced
  - Local memory is represented by my_reg (a natural number)
- Initial state: outside_room (my_reg=0)
  - Transition 1: start: outside_room(0) next: in_room_waiting(otherproc:succ(my_reg))
  - Transition 2: start: in_room_waiting(my_reg), condition(my_reg<otherproc:myreg), next: being_serviced(my_reg)
  - Transition 3: start: being_serviced(my_reg), next: outside_room(0)
- LOOKS SAFE (BUT NOT LIVE)..

Example

- safety: NOT (pr1 = being_serviced AND pr2 = being_serviced);
- We have to show that the space of states of our (two-automata) example is a model for the above formula, i.e. that the formula is true for any reachable point in the state space.
- Can do it by enumeration..

Security properties

- Defining security properties and context
- Context: Network model, adversarial power
- The notion of secure computations

Heuristic Approach to Security

1. Build a protocol
2. Try to break the protocol
3. Fix the break
4. Return to (2)

Heuristic Approach – Drawbacks

- You can never be really sure that the protocol is secure
- Hackers will do anything to exploit a weakness – if one exists, it may well be found
  - Security cannot be checked empirically (see later)

Another Heuristic approach

- Design a protocol
- Provide a list of attacks that (provably) cannot be carried out on the protocol
- Claim that the list is complete

- Problem: often, the list is **not** complete…

A Rigorous Approach

- Provide an exact problem definition
  - Adversarial power
  - Network model
  - Meaning of security
- Prove that the protocol is secure
  - Often by reduction to an assumed hard problem, like factoring large composites
- The history of computer security shows that the heuristic approach is likely to fail
  - Security is very tricky and often anti-intuitive

Sample properties

- Confidentiality
  - Sensitive information is only available to authorized persons
  - No unauthorized participant (user) can discover content of locations and/or messages.
- Integrity
  - Sensitive information is only composed by authorized persons
  - No unauthorized participant (user) can manipulate data
- Availability
  - Sensitive activities are available (in tim) to authorized persons

Specific problems

- Which parts should we choose for modeling ?
  - Security/safety critical parts have a precise semantics
- What is the appropriate level of abstraction ?
  - Completeness vs. complexity, critical aspects of security
- Properties in the model are also properties in our system (critical for security !)

Distributed processes..

- Research is moving from isolated, single-user programs to distributed computations (e.g., processes on service oriented architectures)
- Security mechanisms always chase emerging program paradigms !
- Some issues of distributed processes
- Communication between different systems
  - Secure channels
  - Security protocols
- No static border between „in‟ and „out‟
- Evolving programs („service composition‟)
  - Security checks on the fly?

Basic notions

- A distributed protocol consists of a set of rules (conventions) which determine the exchange of messages between two or more participants.
  - participants: users, processes machines, ...
  - often called “principals”
- Protocol steps
  - n: A→B: M – “A sends M to B according to the n-th protocol step.”
  - Messages may be structured: M = M1, ... , Mn

Example: security protocols

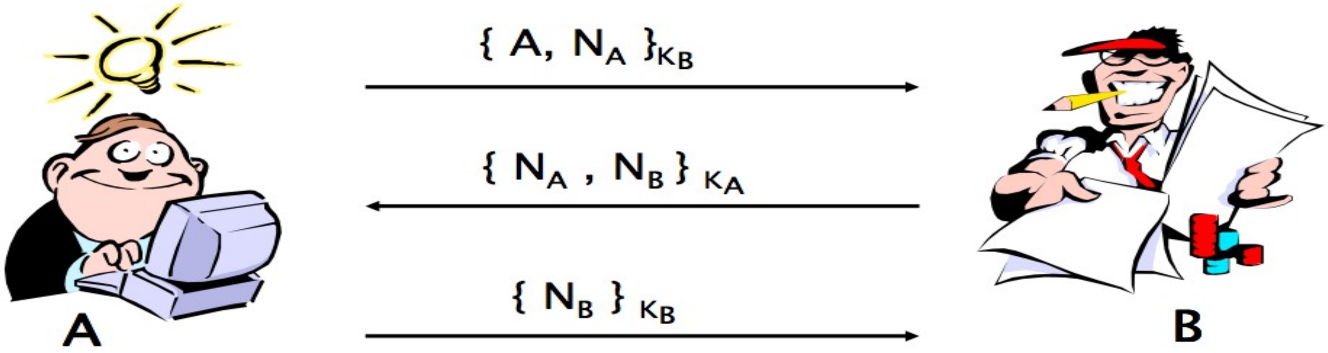
- Security protocols are used to establish a secure channel
- More technically:
  - exchange a shared key
  - authenticate each other

Encryption aspects

- encryption of messages: $n : A \rightarrow B : \{M\}K$ “M is encrypted using key K.”
- for each K exists an “inverse” K^{-1} : $K=(K^{-1})^{-1}$
- keys indexed by participants:
  - K_A public key of A; $K_{A,B}$ symmetric key shared between A and B
- for symmetric encryption : $K^{-1} = K$
- for asymmetric systems (recall asymmetric schemes!) – K^{-1} private key,
- signatures – K public key: (asymmetric) encryption

Example: the Needham-Schroeder protocol

- K_B: B's public key
- K_A: A's public key
- Nonces: N_B N_A



A

$\{ A, N_A \}_{K_B}$

$\{ N_A , N_B \}_{K_A}$

$\{ N_B \}_{K_B}$

B

Is this protocol *secure* ?

Example

- A single instance is secure.. but if multiple instances are run in parallel, things change
- How to win a chess game against a grand-master
  - Challenge two grand-masters at once
  - Reproduce the moves of the first grand-master on the checkboard of the second..

The attack

A man-in-the-middle attack:

- alice ——— { alice, Nalice }Kchar ——→ charlie
- charlie ——— {alice, Nalice }Kbob ——→ bob
- (bob ——— {Nalice, Nbob }Kalice ——→ alice)
- charlie ——— {Nalice, Nbob }Kalice ——→ alice
- alice ——— { Nbob }Kchar ——→ charlie
- charlie ——— {Nbob }Kbob ——→ bob

What's wrong?

- What’s wrong with the protocol?
- Bob wrongly believes that he is communicating with Alice.
- Problem is in the second message specification:
  - 2: B→A: {NA ,NB}KA
- instantiation in the failed run:
  - bob (charlie) —— {Nalice, Nbob }Kalice —→ alice
- Repair: specification 2: B→A: {B,NA ,NB}KA
  - bob —— {bob, Nalice, Nbob }Kalice —→ alice

The problem is solved

- Trying the same attack:
- alice —— { alice, Nalice }Kchar — charlie
- charlie —— { alice, Nalice }Kbob — bob
- bob —— {bob, Nalice, Nbob }Kalice — alice
- charlie —— {bob, Nalice, Nbob }Kalice — alice

BUT: Alice expects an answer from Charlie (and not from Bob).

But this is an ad-hoc solution

- General solution:
  - Encode problem of a security protocol analysis as a problem in a logic
  - Apply a theorem prover for the logic to the problem
- Challenge: develop specialized logics,programs and/or (meta-)theories for the security analysis of distributed protocols

Challenge in detail

- Formal methods can do the analysis of a finite state problem (as we saw at the beginning)
- However, distributed protocols have infinitely many states:
  - arbitrary number of principals
  - arbitrary number of protocol runs
  - arbitrary size of messages (generated by the attacker)
- How to handle it
  - restrict number of principals
  - restrict number of protocol runs
  - combine different states into a single state by some criterion

Relevant research: OFMC

- Lazy and intelligent enumeration of the search space
  - Organize the search space as a tree.
  - Each node is a trace of the protocol and continues the trace of the predecessor node.
- Based on D.Basins‵s work on Lazy Infinite-State Analysis of Security Protocols (1999)
- Part of the AVISPA-toolset (www.avispa-project.org)

Modeling the protocol

- Enumeration of all possible traces (shortest first) using protocol rules and checking the results wrt. to insecure states
- Attacker is the network: all messages are sent or received via the attacker
- Rules of the form:
  - msg(m1) AND state(m2) AND N1 -> state(m3) AND msg(m4) AND P2
- representing positive (P1, P2) and negative (N1) facts concerning the attacker
  - Examples: „intruder knows NA", „M is secret and only known to A" , „A has not seen the message NB"
- and actual states of principals (state(m))
  - Examples: state(roleA, step0, A, B), state(roleB, step2, A, B, NA, NB),
- Application of rules is checked via matching of messages and facts

Modeling the success

- Definition of attack-condition:
- condition under which an attack is successful
- Syntactically, has the form of the left hand side of a rule:
- ar = msg(m1).state(m2).P1 .N1 ...
  - Example: secret(M, {A, B}), i_knows(M), : secret(M, i)
- State S is a successful attack iff ar is „applicable" in S.
- Protocol is secure iff for all reachable states S and all attack conditions ar: ar is not „applicable" in S.

Other approaches

- Strand objects
  - Framework on security protocols
    - exploring the structure of a protocol,
    - exploring the possible combination of local runs (at the principles) of a protocol to a common protocol
  - Based on the Dolev-Yao model
  - Developed by: Joshua Guttman, Jonathan C. Herzog, F. Javier Thayer (1998)
  - Implemented (partly) in the Athena – system
- Inductive theorem proving
  - Modeling security protocols in an expressive, universal logic (HO-logic)
  - Messages and protocol traces as abstract data types
  - Modeling the knowledge of principals and attacker as functions on message lists (that the principal has seen before)
  - Pioneered by L. Paulson using Isabelle (later: other proof tools like Coq, VSE, etc)