Design Tools for the Development of Reliable Secure Communication Software

Johann Schumann, RIACS / NASA Ames
Microsoft Posts Critical Patches

Four security bulletins cover many fixes in Windows, IE, Outlook.

... among the most critical holes Microsoft is warning about is a buffer overrun vulnerability in the Local Security Authority Subsystem Service (LSASS), ...

The second critical vulnerability is a buffer overrun hole in the Private Communications Transport (PCT) protocol...
Introduction

Software for Secure Communications can become insecure due to *flaws* during any phase of the software life cycle

- **Design**
  - wrong algorithm
  - wrong requirements

- **Implementation**
  - buffer overrun, uninit’d variable, . . .
  - sleeper codes

- **Verification and Validation**
  - wrong tests
  - insufficient test coverage

- **Deployment**
  - wrong code (e.g., disabled crypto)
  - code tampering
Our Goal

Cost-effective development for *reliable, secure communications software*

- unified, tool-supported framework from specification to deployment
- support iterative process
- automatic certification support
Overview
The Individual Tools: Overview

Requirements:

- iterative process
- fast turnaround
- security analysis
- reliable, secure code
- certification

- High-level Specification
  - UML for specification
- Automatic Protocol Analysis
  - Simulation, MC, ...
- Correct Protocol Optimization
- Automatic generation of code
  - Design document gen.
  - multi target platforms
- Automatic test case generation
- Automatic certification support
Protocol Specification

- specification of protocol as UML sequence diagrams and Use Cases
- distributed representation
- OCL annotations regarding reliability and security properties
- UMLSec (?)

Explicit specification of protocol with safety/security properties
Generation of Design

- automatic Synthesis of UML-designs from sequence diagrams
- generation of readable, highly structured statecharts from sequence diagrams and OCL constraints
- provably correct designs
- automatic detection of inconsistencies and loops

Automatic synthesis of Statecharts facilitate rapid, consistent turn-around
Statechart Synthesis: The naive Approach

Diagram showing statechart interactions involving Agent 1, Agent 2, and Beach, with transitions including encounter, requestCoord, grantCoord, sendGoal(park), tm(5), endCoord, naive merging, and naive translation.
State Vector

- formed by OCL specifications on SD Messages and global invariants
- we use a state vector (set of variables):
  \(<\text{coordWith, hasSessionKey, msgCounter}>\)
- Constraints (domain theory + safety policy + security policy)

  context A::getKey()
  pre: self.hasSessionKey = false ;  \(<?,F,?>\)
  post: self.hasSessionKey = true ;    \(<?,T,?>\)

- undetermined values are marked by a '?'
- not all messages need to be annotated with pre/postconditions
1. Annotation with constraints for justified merge of SDs
2. Conflict Detection
3. Generation of a flat statechart for each SD
4. Merge of the SCs
5. Introduction of Hierarchy
Set variable assignment in SD

- **Unification:** assign values to '?'
- **Frame axiom:** if variable not set by a message, carry it over to the next message
- **Loop detection**

State vector: `<coordWith, hasSessionKey, msgCounter>`
Conflict Detection

Extend variable assignment in SD

- **Unification:** assign values to ‘?’
- **Frame axiom:** if variable not set by a message, carry it over to the next message
- loop detection

State vector: <coordWith, hasSessionKey, msgCounter>
Merging SDs

- Translate SD → flat SC
Merging SDs

- multiple flat SCs $\Rightarrow$ single flat SC
  
  *two nodes are similar if they have identical state vectors and have $\geq 1$ incoming transition in common*
Introduction of Structure/Hierarchy

Synthesized statecharts must be readable/understandable for

- manual refinement and modifications of the initial design
- system architecture (high-level states) often known

What is a “well-designed”, readable statechart?

- consolidation of related behavior
- separation of unrelated behavior
- introduction of meaningful abstractions
- heuristics: number of nested levels, nodes in one supernode, and inter-level transitions should have reasonable values
Protocol Analysis

- integration of OTS analysis and verification tools
  - theorem proving
  - model checking
  - simulation

- integration of results/feedback into our framework
Protocol Optimization

- specific communication requirements require specific protocol variants
  - low bandwidth
  - low computational resources (memory, CPU)
  - high latency (e.g., 20’ to Mars)
- avoid repeated “wrapping” of data
- avoid unnecessary messages and synchronisation
- without loosing correctness

Logic-based protocol optimization (collaboration with Cornell) will be integrated into our framework

Formal methods based protocol optimization enables the designer to specifically tailor the communication protocol without compromising reliability and security
Certification Support

We are developing a product-oriented approach to certification

- don’t verify the generator, but focus on produced artifacts
- common techniques:
  - testing and simulation
  - code review
  - program analysis
  - model checking
  - program verification
- important requirements
  - automatic processing
  - tamper-proof certificates
  - no annotations to be provided by the user
Our Approach: Synthesizing Certifiable Code
Our Approach: Synthesizing Certifiable Code

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)
Our Approach: Synthesizing Certifiable Code

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)

Basic Idea II:

Certify generated programs, not the generator
Our Approach: Synthesizing Certifiable Code

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)

Basic Idea II:

Certify generated programs, not the generator

Basic Idea III:

Introduce code certificates
Our Approach: Synthesizing Certifiable Code

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)

Basic Idea II:

Certify generated programs, not the generator

Basic Idea III:

Introduce code certificates

Basic Idea IV:

Use Floyd-Hoare program verification techniques
Certifiable Properties

Most software errors are violations of safety properties
⇒ see Introduction
⇒ on the Aerospace top-ten code review list

- language-specific properties
  - array bounds (memory safety) (pack/unpack)
  - variable initialization-before-use
  - underflow/overflow (e.g., message counter)

- domain-specific properties
  - module input-use
  - volatile memory access limitations
  - security properties

- effectiveness properties
  - Worst Execution Time analysis
  - Data rates
Generation and Processing of Safety Obligations

- annotation of program with pre-/post-conditions
- generation of logic formulas
- formal proof of these formulas

Example: (variable initialization)

```plaintext
for( i=0; i < 10; i++) {
    x[i] = 5;
}
```

- post-condition: \( x(0 : 9) \) is initialized
- loop invariant: if \( x(0 : i - 1) \) is initialized, then \( x(i) \) is initialized
- proof obligation has to demonstrate that from code and invariants, the post-condition can be inferred.
• only few and small trusted components
• easy combination with Proof Carrying Code (PCC) to produce tamper-proof code
Conclusions

tool-supported framework for the reliable development and deployment of secure communication software

- supports entire life-cycle from specification and analysis to tamperproof code
- basic technologies already developed
  - Protocol analysis/verification
  - Scenario to Statechart
  - Automatic Code Certification

- areas for collaboration
  - generation of testcases
  - software process
  - protocol optimization

- two project phases:
  - Requirements analysis, initial architecture and case study
  - tool maturation