The Eighth International Conference INCOSE_IL 2015

- כלים ובדיקות
- **Formal Methods Security Tools in the Service of Cyber Security**

- **Dr. Michael J. May**
- **Kinneret College on the Sea of Galilee**

האיגוד הישראלי להנדסת מערכות

About me

Dr. Michael J. May
Lecturer
Software and Information Systems Department
Achi Racov Engineering School
Kinneret College on the Sea of Galilee

What interests me

Security modeling of protocols

Formal methods analysis of communication protocols and security software

Modeling privacy policies

Murphy's Law of computer programming:

Every non-trivial program contains bugs

Corollary:

Any program without bugs is trivial

The Bug Business

For fame (and business):

- Common Vulnerabilities and Exposures (CVE)
  - 14 related to SSL (so far) in 2015
  - Hundreds related to SSL in 2014

For fortune (bug bounties):

- Google, Facebook, GitHub, Microsoft, Mozilla, etc.

For intelligence:

- "According to an NSA document, the agency intended to crack 10 million intercepted https connections a day by late 2012." (Der Spiegel December 28, 2014)
- Adi Shamir: NSA is the world's premier code avoidance organization… (Technion Cyber Day 2014)

Conclusion: Full employment theorem for penetration testers and security analysts!

Can we improve things?

Formal methods security tools are

Mature

Ready to use

Worth your time to learn

Spherical Cows

Milk production at a dairy farm was low, so the farmer wrote to the local university, asking for help from academia. A team of professors was assembled, headed by a theoretical physicist, and two weeks of intensive on-site investigation took place.

The scholars returned to the farmer saying, “We have the solution, but it only works in the case of spherical cows in a vacuum".

(via Wikipedia)

Formal Methods Security

Best practices in software development

- Test driven development
- Agile methods
- Pair programming
- Code review

improve code quality.

Formal tools to describe

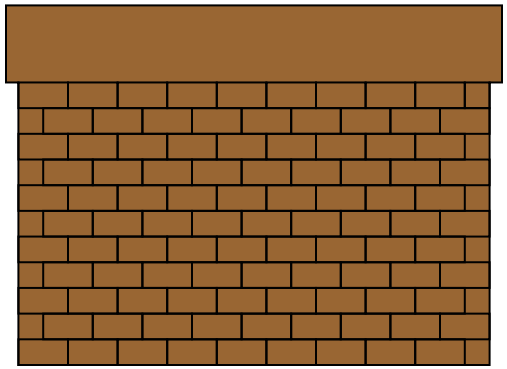
- Security assumptions
- Requirements
- Use of cryptographic tools

can do the same for software security.

Security Metrics

Penetration Tester: "I hit it with a 10kg sledge for 1 hour and the wall didn't break"

Design Analyst: "We used ABC brand bricks and XYZ brand mortar when building it."



Standards Compliance: "The bricks and mortar came from an ISO 9000 certified factory and the bricklayer is certified by the bricklayers guild."

Formal Analysis: "The wall is designed to withstand 1000kg of force the whole face or 200kg on a point location. The wall's foundation is…"

Formal Security Tools

ProVerif

CryptoVerif

Spin is popular open-source software verification tool, used by thousands of people worldwide. The tool can be used for the formal verification of multi-threaded software applications.

The tool was developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments. In April 2002 the tool was awarded the ACM System Software Award.

Source: spinroot.com

About

How to use it:

Write a model of processes in Promela and queries about the model (i.e. what's reachable, what's possible). Run.

What it does:

Returns answers to the queries, creates state machine figure.

What it's good at:

Verifying parallel computations

Verifying complicated state machine descriptions

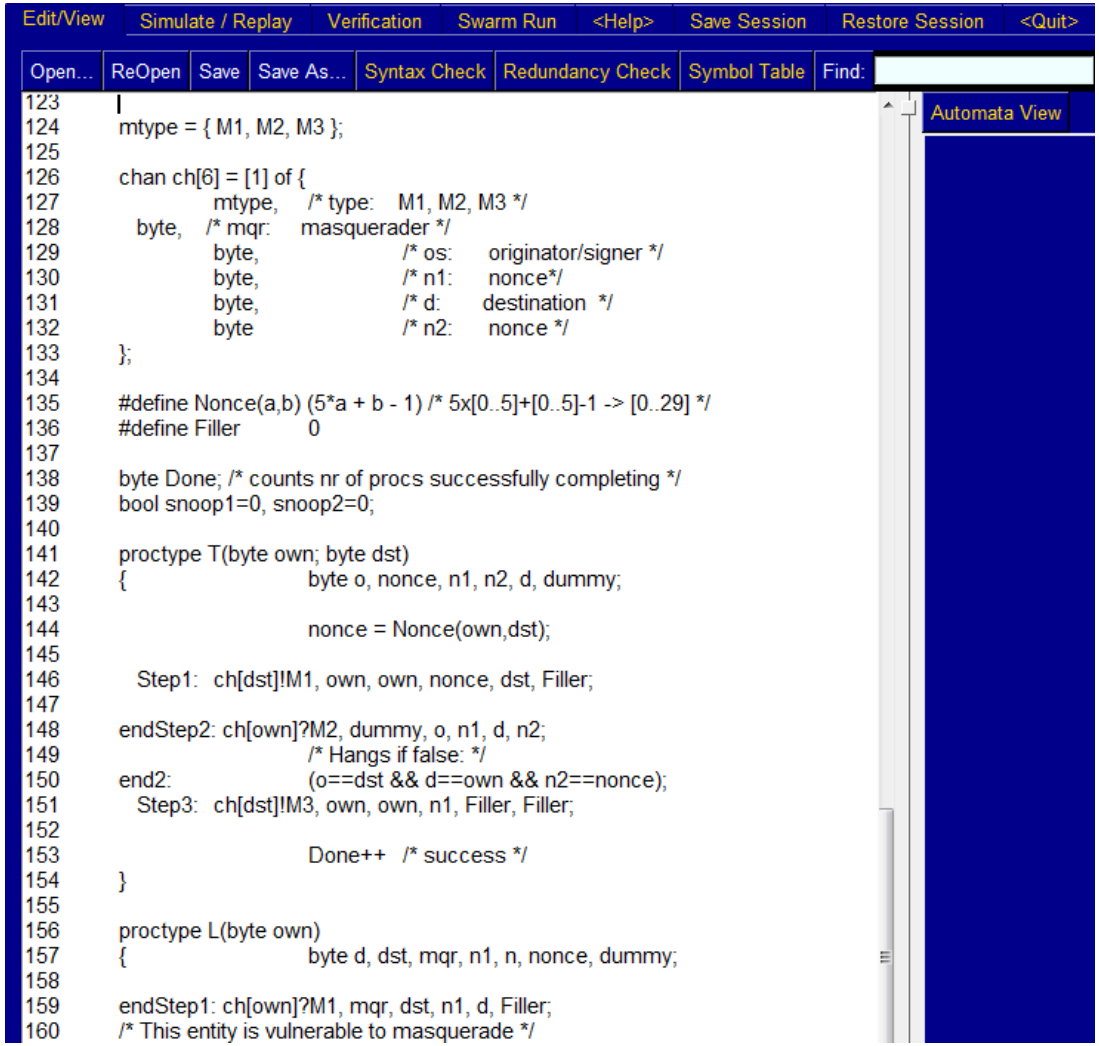
Extras

Model extraction from C

Extensive documentation and support (spinroot.com)

Annual workshop

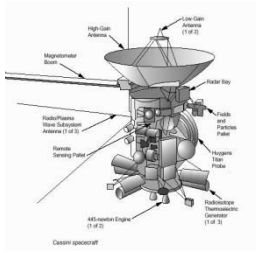
What it looks like



Who's used it

1. Verifying flood control systems for Netherlands (link) (1996)
   - Formally specified the design to avoid ambiguities and incompleteness
   - Validated parts of the design, interfaces with the outside

2. Algorithms for: Mars Science Laboratory, Deep Space 1, Cassini, Mars Exploration Rovers, Deep Impact



3. Toyota Camry '05 unintended acceleration investigation (2011) (link)

4. My experience: Modeling HIPAA privacy policy requirements (2006) (link)

Photos from spinroot.com

F* is a new higher order, effectful programming language (like ML) designed with program verification in mind. …

The F* type-checker aims to prove that programs meet their specifications using an automated theorem prover (usually Z3) behind the scenes to discharge proof obligations.

Programs written in F* can be translated to OCaml, F#, or JavaScript for execution.

Source: https://fstar-lang.org/

Developed by Microsoft Research (US & UK), INRIA (France), and IMDEA (Spain)

About

How to use it:

Write a functional specification of libraries, processes, procedures, cryptographic ops, etc. in F* (looks like ML). If it compiles, it's verified.

What it does:

Compiler verifies correctness using typing rules.

What it's good at:

Verifying security policy enforcement (access control, authentication)

Specifying correctness and termination of programs

Extras

Can link with concrete libraries to run the model as a program

What it looks like

```
module ACLs
open String
open List
type file = string

let canWrite = function
  | "C:/temp/tempfile" -> true
  | _ -> false

let publicFile = function
  | "C:/public/README" -> true
  | _ -> false

let canRead (f:file) =
  canWrite f            (* 1. writeable files are also readable *)
  || publicFile f       (* 2. public files are readable *)
  || f="C:/acls2.fst"   (* and so is this file *)

(* two dangerous primitives *)
assume val read:   file:string{canRead file} -> string
assume val delete: file:string{canWrite file} -> unit

(* some sample files, one of them writable *)
let pwd    = "C:/etc/password"
let readme = "C:/public/README"
let tmp    = "C:/temp/tempfile"

let test () =
  delete tmp; (* ok *)
//delete pwd; (* type error *)
  let v1 = read tmp in     (* ok, rule 1. *)
  let v2 = read readme in (* ok, rule 2. *)
  ()

(* some higher-order code *)
val rc: file -> ML (unit -> string)
let rc file =
  if canRead file
  then (fun () -> read file)
  else failwith "Can't read"
```

It's been used for:

1. Verifying implementation of cryptographic constructions and protocols
   - 

(using F7 and F#)

2. Verifying browser extensions (JavaScript)
   - Magnify web pages (HoverMagnifier)
   - Grab contact details from Facebook friends (Facepalm)
   - Format text entered into form (Typograf)

3. Semantics of JavaScript
   - Compiling F* to JavaScript

4. Certifying the F* type checker

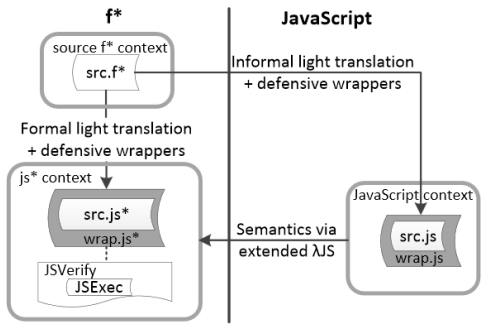


Figure 1. Architecture

Figure from research.microsoft.com

miTLS

Reference implementation of TLS

Used to find a bunch of recent TLS attacks:

- FREAK
- SMACK (Skip-TLS attack)
- Triple Handshake

(link)

The Java Modeling Language (JML) is a behavioral interface specification language that can be used to specify the behavior of Java modules.

It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus.

Source: www.jmlspecs.org/

Open source language developed by a group of academics in the USA

About

How to use it:

Take an existing Java program and annotate the methods with comments describing the input, output, and behavior of the methods. JML engine verifies assertions and annotations.

What it does:

Examines annotations and checks for correctness.

What it's good at:

Basis for special use verification tools
Extensions for Java code analysis, specification checking

Extras

Plugin for Eclipse, many extensions

What it looks like

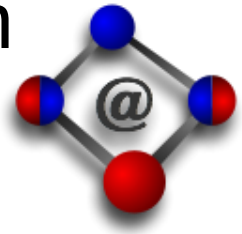
generated_java_SocialEventPlanner_multi_threaded ▸ src ▸ SocialEventPlanner_multi_threaded.events ▸ delete_list ▸

```
1  package SocialEventPlanner_multi_threaded.events;
2
3  import SocialEventPlanner_multi_threaded.*;
6
7  public class delete_list extends Thread{
8      private ref8_socialpermissions machine; // reference to the machine
9      private int eventId; // event identifier
10
11     /*@ requires true;
12         assignable \everything;
13         ensures this.machine == m && this.eventId == i; */
14     public delete_list(ref8_socialpermissions m,int i) {
15         this.machine = m;
16         this.eventId = i;
17     }
18
19     /*@ requires true;
20         assignable \nothing;
21         ensures \result <==> (machine.listow.domain().has(l) && machine.persons.has(ow) && machine.listow.apply(l) == ow); @*/
22     private boolean guard_delete_list( Integer l, Integer ow) {
23         return (machine.listow.domain().has(l) && machine.persons.has(ow) && machine.listow.apply(l) == ow);
24     }
25
26     /*@ requires guard_delete_list(l,ow);
27         assignable machine.listow, machine.listpe, machine.policies, machine.disjointness;
28         ensures \old((machine.listow.domain().has(l) && machine.persons.has(ow) && machine.listow.apply(l) == ow)) &&  machine.listow.equals(\old(mac
29      also
30         requires !guard_delete_list(l,ow);
31         assignable \nothing;
32         ensures true; @*/
33     private void run_delete_list( Integer l, Integer ow){
34         if(guard_delete_list(l,ow)) {
35             BRelation<Integer,Integer> listow_tmp = machine.listow;
36             BRelation<Integer,Integer> listpe_tmp = machine.listpe;
37             BRelation<Integer,Integer> policies_tmp = machine.policies;
38             BRelation<Integer,Integer> disjointness_tmp = machine.disjointness;
39
40             machine.listow = listow_tmp.domainSubtraction((new BSet<Integer>(l)));
41             machine.listpe = listpe_tmp.domainSubtraction((new BSet<Integer>(l)));
42             machine.policies = policies_tmp.domainSubtraction((new BSet<Integer>(l))).rangeSubtraction((new BSet<Integer>(l)));
43             machine.disjointness = disjointness_tmp.domainSubtraction((new BSet<Integer>(l))).rangeSubtraction((new BSet<Integer>(l)));
44
45             System.out.println("delete_list executed l: " + l + " ow: " + ow + " ");
46         }
47     }
```

Who's used it

Academic projects:

- MultiJava: extension of Java with open classes and multiple dispatch (code run by a method call depends on the run time types of the parameters)



- KeY: deductive verification of software

- Bytecode Modelling Language (BML)

…and many others

ProVerif

An automatic cryptographic protocol verifier in the formal model.

Developed by INRIA

How to use it:

Write process specifications in the applied π calculus. Write assertions about the protocols and events. Run the verifier

What it does:

Checks for reachability of the assertions using a state space exploration (ala Spin). Shows a proof if there is one.

What it's good at:

Proper security verification tool

Ideal for protocol verification, but in the formal model (black box cryptography)

CryptoVerif

CryptoVerif is an automatic protocol prover sound in the computational model. It can prove:

secrecy and correspondences, including authentication.

Developed by INRIA

How to use it:

Write process and cryptographic specifications in the applied π calculus. Write assertions about the protocols, secrets, and events. Run the verifier

What it does:

Checks for reachability of the assertions by converting the processes into games and seeking a path to reduce games until a proof or counterexample is found. Shows a proof if there is one.

What it's good at:

Proper security verification tool

Ideal for protocol verification in the computational model

Summary

Name	Write in	Developed By	Paradigm	Notes
Spin	Promela / C	Various, includes Gerard Holzmann and Moti Ben-Ari (Weizmann)	Concurrent code model checking via state space exploration	Can do model extraction to C
F*	F*	MS Research, INRIA, IMDEA	Type-based program correctness	Descendent of F# and F7, compiles to JavaScript, F#
JML	Java annotations	Various, includes Gary Leavens	Contract based - annotated Java methods with formal summaries	Has Eclipse plugin, many extensions, attached to runnable code

Summary

Name	Write in	Developed By	Paradigm	Notes
ProVerif	Applied π calculus	INRIA	Process calculi with security assertions	Formal model
CryptoVerif	Applied π calculus	INRIA	Process calculi with security assertions	Computational model