Application of Formal Verification to Software Security

Reynald Affeldt David Nowak

AIST-RCIS, Japan

TWISC Summer School 2006

Verification of Security Properties of Software

Generally speaking,

- Software security is difficult to define
  - Many unclear notions (e.g., "privacy")
  - Often many details (e.g., technical details)
- Pencil-and-paper verifications/proofs are difficult to check
  - Many abbreviations (e.g., "We see that...")
  - Often many cases (e.g., lengthy enumerations)

There is a need for:

1. Mathematical definitions of what to verify
2. Computer means to do (or at least check) verifications

Formal Verification

- Appropriate in the case of critical systems
- Formal verification consists of:
  1. A mathematical model $\mathcal{M}$ of the system
  2. A property φ expressed in a formal logic
  3. Techniques to prove and check that $\mathcal{M}$ satisfies φ
- There are mainly two approaches:
  - Proof-assistants
    - \+ Very expressive (infinite models handled by induction)
    - − Requires human interaction
  - Model-checking
    - \+ Automatic proof
    - − Finite models only (unless safe abstractions are made)

Proof-assistants

- A proof-assistant consists of:
  - A language for writing mathematical models $\mathcal{M}$, statements φ, and proofs that $\mathcal{M}$ satisfies φ
  - An automatic way to <u>check</u> proofs
  - An interactive way to <u>build</u> proofs

    Automatic discovery of proofs for simple statements only

- Worthwhile if the cost of mistakes is extremely high

  E.g., critical parts of microprocessor design

The Coq Proof-assistant [INRIA, 1984–2006]

- A programming language with powerful types...
  - Inductive/coinductive types for finite/infinite data structures
    Lists, trees, streams, etc.
  - Dependent types
    The output-type of a function can vary according to its argument
- ...for writing models, properties, and proofs:
  - Properties are types
  - Proofs are programs (Heyting semantics)
    In particular, proof-checking = type-checking
- Remarkable achievements:
  - Verification of virtual machines for smartcards
    [Trusted Logic, 2003]
  - The four color theorem [Gonthier and Werner, 2004]

The Four Color Theorem

Four colors are enough to color any geographical map in such a way that no neighboring two countries are of the same color.

- The proof requires the verification of many cases
- Long history:

  1853 first statement
  1976 first proof, using a computer
  2004 certified proof in Coq

- Practical application:

  reduce the number of used broadcasting frequencies for mobile phones

Verification of Functional Programs in Coq

General approach:

- Mathematical model $\mathcal{M}$: a function in the Coq language
- Property φ: a statement in the Coq language
- Verification that $\mathcal{M}$ satisfies φ: by interactive proof

Demo

Verification of Imperative Programs in Coq

- Problem: the Coq language is not imperative

  Imperative programs cannot be represented directly

- Solution: use the Coq language to model imperative programs

  This amounts to formalization of their semantics

- General approach:
  - Mathematical model $\mathcal{M}$:
    the formal model of an imperative program
  - Property φ: a statement in the Coq language
  - Verification that $\mathcal{M}$ satisfies φ: by interactive proof

Verification of Imperative Programs

Hoare Logic (1/2)

Empty statement axiom

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}}$$

Assignment axiom schema

$$\frac{}{\{P[E/x]\}\ x{:=}E\ \{P\}}$$

Example: $\{x + 5 < 20\}\ x{:=}x + 5\ \{x < 20\}$

Sequence rule

$$\frac{\{P\}\ C\ \{Q\} \quad \{Q\}\ D\ \{R\}}{\{P\}\ C;D\ \{R\}}$$

Verification of Imperative Programs

Hoare Logic (2/2)

Conditional rule

$$\frac{\{E \wedge P\}\ C\ \{Q\} \quad \{\neg E \wedge P\}\ D\ \{Q\}}{\{P\}\ \text{if } E \text{ then } C \text{ else } D \text{ endif } \{Q\}}$$

While rule

$$\frac{\{E \wedge \boxed{Inv}\}\ C\ \{\boxed{Inv}\}}{\{\boxed{Inv}\}\ \text{while } E \text{ do } C \text{ done } \{\neg E \wedge \boxed{Inv}\}}$$

Rule of consequence

$$\frac{P \Rightarrow P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}$$

Verification of Imperative Programs

Example

$\{a > 0 \ \wedge \ b > 0\}$
$x{:=}a;\ y{:=}b;$
$\{\boxed{x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)}\}$
while $x \neq y$ do
  $\{x \neq y \ \wedge \ x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)\}$
  if $x < y$ then
    $\{x < y \ \wedge \ x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)\}$
    $y{:=}y - x$
    $\{x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)\}$
  else
    $\{x > y \ \wedge \ x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)\}$
    $x{:=}x - y$
    $\{x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)\}$
  endif
  $\{x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)\}$
done
$\{x = y \ \wedge \ \boxed{x > 0 \ \wedge \ y > 0 \ \wedge \ gcd(x, y) = gcd(a, b)}\}$

The conclusion implies that $x = gcd(a, b)$

Application to Software Security

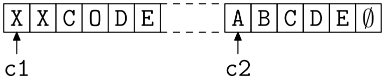
- Memory management in C
  - Buffer overflows
  - Security issues on multi-users systems
- Implementation of cryptographic devices (smartcards)
  - Efficient arithmetic on large integers

Buffer Overflow

- A dangerous program:

  ```
  for (c1=buf, c2=str; (*c1++ = *c2++)!=0; );
  ```

  - The buffer may be smaller than the string:

  - How can we prevent such bugs using formal verification?

Verification of Memory Management

Separation Logic (1/2)

- Hoare logic with a notion of mutable memory [Reynolds, 2002]
  - Singleton heap:

$$h \models (E \mapsto E') \text{ iff } \ dom(h) = E \ \wedge \ h(E) = E'$$

  - Memory accesses:

Mutation

$$\frac{}{\{E \mapsto ?\}\ [E] := E'\ \{E \mapsto E'\}}$$

Example:

$$\left\{\begin{array}{c}\boxed{?}\\ \uparrow\\ x\end{array}\right\} [x] := 4 \left\{\begin{array}{c}\boxed{4}\\ \uparrow\\ x\end{array}\right\}$$

is written $\{(x \mapsto ?)\}\ [x] := 4\ \{(x \mapsto 4)\}$

Lookup

$$\frac{}{\{E \mapsto E'\}\ x := [E]\ \{E \mapsto E' \ \wedge \ x = E'\}}$$

Verification of Memory Management

Separation Logic (2/2)

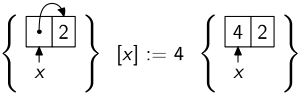
- Compositional reasoning using a logic extension
  - Compound heap:

$$h \models P \star Q \text{ iff}$$
$$\exists h_1, h_2 \text{ s.t. } h_1 \perp h_2 \;\wedge\; h_1 \uplus h_2 = h \;\wedge\; h_1 \models P \;\wedge\; h_2 \models Q$$

  - Frame Rule

$$\frac{\{P\}C\{Q\} \wedge \mathit{modified}(C) \cap \mathit{free}(R) = \emptyset}{\{P \star R\}C\{Q \star R\}}$$

Example:



is written

$\{(x \mapsto p) \star (p \mapsto 2)\}$ $[x] := 4$ $\{(x \mapsto 4) \star (p \mapsto 2)\}$

Verification of Memory Management

Example: Buffer Overflow

$\{buf \Mapsto B_0 \cdots B_{n-1} \star str \Mapsto S_0 \cdots S_{m-1}\}$

$c1 := buf; c2 := str; tmp := [c2];$

$$\left\{ \boxed{\begin{array}{l} buf \Mapsto S_0 \cdots S_{i-1} B_i \cdots B_{n-1} \star str \Mapsto S_0 \cdots S_{m-1} \wedge \\ c1 = buf + i \wedge c2 = str + i \wedge tmp = S_i \end{array}} \right\}$$

while $tmp \neq 0$ do

$\quad [c1] := tmp;$

$\quad c1 := c1 + 1;$

$\quad c2 := c2 + 1;$

$\quad tmp := [c2]$

done;

$$\left\{ tmp = 0 \wedge \boxed{\begin{array}{l} buf \Mapsto S_0 \cdots S_{i-1} B_i \cdots B_{n-1} \star str \Mapsto S_0 \cdots S_{m-1} \wedge \\ c1 = buf + i \wedge c2 = str + i \wedge tmp = S_i \end{array}} \right\}$$

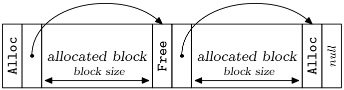
$[c1] := tmp$

$\{buf \Mapsto S_0 \cdots S_{m-1} \star \mathrm{T} \star str \Mapsto S_0 \cdots S_{m-1}\}$

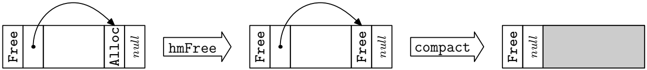
Possible only if $n \geq m$

Memory Management and Multi-users Systems

- Security issue: privacy of the data of users
- Example: memory management in O.S. [Marti et al., 2006]
  - Dynamically memory uses linked lists:



  - Separation property:
    "Newly allocated blocks do not override old ones"
  - Related problem found during verification of existing code:
    - Memory exhaustion:



Verification of the Implementation of Cryptosystems

- Algorithms <u>and</u> their implementation must be certified
- Cryptographic devices require low-level programming
- In low-level languages, properties depend on physical data:
  - Counter-intuitive arithmetic properties
    - Machine integers wrap around (integer overflow)
    - Confusing conversions:

      ```
      unsigned int u;
      ...
      if (u > -1) ... /* always false! */
      ```

    - The sign of the remainder of an integer depends on its size
  - Unsafe casts
    - Ariane 5 bug:
      Conversion from 64-bit floating-point to 16-bit signed integer

Formalization of Machine Integers in Coq (1/2)

- A machine integer is a list of bits
  - Examples:

    `i::i::i::i::nil` stands for (1111)

    `o::o::o::i::nil` stands for (0001)

- Hardware circuitry is a set of recursive functions
  - Example: “strictly less than”

```
Fixpoint listbit_lt (a b:list bit) {struct a} : bool :=
  match a with
    o::tla => match b with
                o::tlb => listbit_lt tla tlb
              | i::_ => true
              | _ => false
              end
  | i::tla => match b with
                o::_ => false
              | i::tlb => listbit_lt tla tlb
              | _ => false
      end
  | _ => false
  end.
```

Formalization of Machine Integers in Coq (2/2)

- Signed integers in two's complement notation:
  - Definitions:

$$
\begin{aligned}
(a_n \dots a_0)_u &= a_n 2^n + \cdots + a_0 \\
(a_n \dots a_0)_s &= -a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_0
\end{aligned}
$$

  - Examples:
    - $(0001)_u = (0001)_s$ but $(1111)_u \neq (1111)_s$
    - In Coq:

```
[[ o::o::o::i::nil ]]u = 1  | [[ i::i::i::i::nil ]]u = 15
[[ o::o::o::i::nil ]]s = 1  | [[ i::i::i::i::nil ]]s = -1
```

- We retrieve the "expected" properties:
  - $-1 \not< 1$
  - In Coq:

```
listbist_lt (i::i::i::i::nil) (o::o::o::i::nil) = false
```

Verification of Efficient Arithmetic on Large Integers

Formalization of machine integers is necessary because of:

- Target functions in assembly
  - Resource constraints
  - Application-specific extensions (e.g., SmartMIPS)
- Specifications at the bit-level
  - Carries and flags

Formal Verification of the Modular Multiplication in Coq

- Specification of the Montgomery algorithm:

$$\left\{\begin{array}{lll} X, Y, M & \text{such that} & |X|, |Y|, |M| = k \text{ and } X, Y < M \\ Z & \text{such that} & |Z| = k+1 \text{ and } Z = 0 \\ \alpha & \text{such that} & \alpha * M_0 \equiv -1[\beta] \end{array}\right\}$$

$$\texttt{montgomery}\ X\ Y\ M\ Z\ \alpha$$

$$\{\ \beta^k * Z \equiv X * Y[M] \text{ and } Z < 2 * M\ \}$$

- Example: $10^5 * 39796 \equiv 5792 * 1229\ [72639]$
- Basic idea: zero the least significant word of partial products

0	0	0	0	0	0	5	8	3	5	7	0
0	0	0	0	0	6	5	0	5	3	0	0
0	0	0	0	5	0	9	4	9	0	0	0
0	0	0	3	4	7	6	5	0	0	0	0
0	0	3	9	7	9	6	0	0	0	0	0

- Verification of a SmartMIPS implementation in Coq using machine integers and Hoare logic [Affeldt and Marti, 2006]

Other Applications of Proof-assistants to Software Security

- Proof-carrying code [Hamid et al., 2002]
  - Mobile code sent <u>with</u> its safety proof
- Security protocols [Paulson, 1998]
  - Inductive proofs in the Isabelle proof assistant
- Internet applications
  - Mail server using a Coq implementation of the π-calculus and temporal logic [Affeldt et al., 2005]

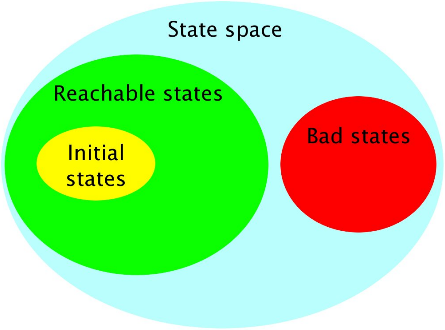
Model-checking

- The system is represented by a transition system, i.e., a directed graph where:
  - Nodes represent states
  - Edges represent changes of states
- Verification is done by exploring the transition system
  - The transition system should be finite
    (not necessarily the model)
  - Execution paths can be infinite (cycles)
- Mainly two families of specifications:
  1. State properties: reachability of a particular state
  2. Path properties: feasible of particular executions

Verification of State Properties

Example of state properties:

- Deadlocks (absence of successors)
- Satisfaction/violation of assertions

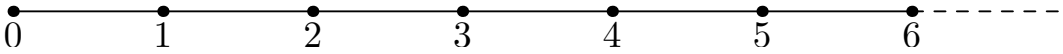


$$\mathit{Reachable}(\mathit{Init}) \cap \mathit{Bad} \; = \; \emptyset$$

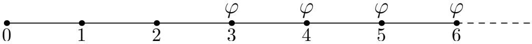
Specification of Path Properties

Path properties are better expressed with temporal logics

- A path is a sequence of states:



- Sample path properties
  - Stability: "There will be a state from which φ is always true."



    Linear Temporal Logic (LTL) notation: $\Diamond\Box\varphi$

  - Response:
    "Always, whenever there is a request, there will be eventually a reply."

    LTL notation: $\Box(\mathit{Request} \rightarrow \Diamond \mathit{Reply})$

Application to Software Security

Example

A simple client-server application:

- The server serves up-to-date files
- The client wants the latest version

We want to verify that:

- After a session, the client has an up-to-date file
- LTL notation: $\Box\Diamond(\mathit{client\_version} = \mathit{server\_version})$

For concreteness, we will use the Spin model-checker

Overview of the Basic Model

In Spin, transition systems are written
using concurrent processes, communicating via channels

```
/********************
  global definitions
 ********************/
typedef Message {
 int file_version;
 mtype signature
}

mtype = { client_key, server_key }

chan server_chan =
   [0] of { Message, chan };

int client_version = 100;
int server_version = 102;
```

```
/*********************
  processes skeletons
 *********************/
proctype client () {
 /* next slides */
}

proctype server (int version_number) {
 /* next slides */
}

init {
 run client ();
 run server (server_version)
}
```

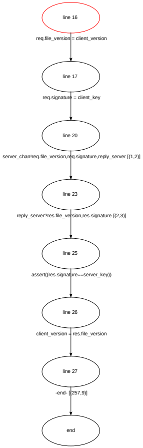
```
/********************
  property to verify
 ********************/
[] (<> (client_version == server_version))
```

Model of the Client

Promela code:

```
proctype client () {

 /* request construction */
  Message req;
  req.file_version = client_version;
  req.signature = client_key;

 /* request to the server */
  chan reply_server = [0] of { Message };
  server_chan ! req, reply_server;

 /* response from the server */
  Message res;
  reply_server ? res;

 /* signature and version checks */
  assert (res.signature == server_key);
  assert (res.file_version >= client_version);
  client_version = res.file_version
}
```

Transition system:

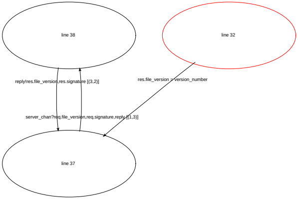


Model of the Server

Promela code:

```
proctype server (int version_number) {

 /* response construction */
  Message res;
  res.file_version = version_number;
  res.signature = server_key;

 /* repeatedly answers response */
  Message req;
  chan reply;
  do
  :: server_chan ? req, reply; reply ! res
  od
}
```

Transition system:



Verification of the Property for the Basic Model

- The property also can be represented as a transition system:

```
/*******************
  property to verify
 *******************/
[] (<> (client_version == server_version))
```

$\longrightarrow$

```
never {    /* !([] <> p) */
T0_init:
 if
 :: (! ((p))) -> goto accept_S4
 :: (1) -> goto T0_init
 fi;
accept_S4:
 if
 :: (! ((p))) -> goto accept_S4
 fi;
}
```

  - The resulting transition system loops as long as p is false
- Transition systems can be composed into a global one (product of *automata*)
- Verification amounts to look for a cycle in the global system

Model of the DNS

Usually, internet connections rely on a DNS:

```
/*****************
  model of the DNS
 *****************/
chan server_chan = [0] of { Message, chan };
chan dns_chan = [0] of { mtype, chan }

mtype = { server_ip }

proctype dns () {
 mtype ip;
 chan reply;
 do
 :: dns_chan ? ip, reply; reply ! server_chan
 od
}
```

Corresponding change in the client model:

```
/* request to the server */
 chan reply_server = [0] of { Message };
 server_chan ! req, reply_server;
```

$\longrightarrow$

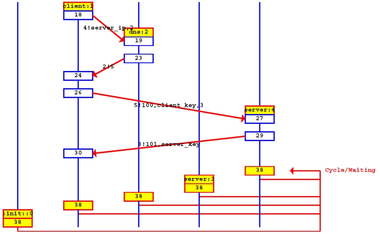
```
/* internet connection */
 chan socket = [0] of { Message, chan };
 chan reply_dns = [0] of { chan };
 dns_chan ! server_ip, reply_dns;
 reply_dns ? socket;

/* request to the server */
 chan reply_server = [0] of { Message };
 socket ! req, reply_server;
```

An Attack found by Model-checking

A spoofed DNS can invalidate $\Box\Diamond(\mathit{client\_version} = \mathit{server\_version})$:

```
/***********************
  model of a spoofed DNS
 ***********************/
chan server_chan = [0] of { Message, chan };
chan bad_server_chan = [0] of { Message, chan };
chan dns_chan = [0] of { mtype, chan }

mtype = { server_ip, bad_server_ip }

proctype dns () {
 mtype ip;
 chan reply;
 do
 :: dns_chan ? ip, reply;
  if
  :: true -> reply ! server_chan
  :: true -> reply ! bad_server_chan
  fi
 od
}
```

Counter-example:



$\Rightarrow$ The application is vulnerable to *replay attacks*

It is possible to enforce a downgrade despite encryption

Applications to Software Security

We have applied model-checking to verification of:

- An existing web-application
- An embedded operating system [Marti et al., 2006]

BTW, verification of crytographic protocols are carried out similarly

Conclusion

In this talk, we had:

- An introduction to formal verification
  - Proof-assistants
  - Model-checking
- Application to software security
  - Memory management in C
  - Implementation of cryptographic devices
  - Verification of internet applications

The slides and the examples are available at `http://staff.aist.go.jp/reynald.affeldt/isss2006/`.

References

The Coq Proof assistant. http://coq.inria.fr.

Interactive Theorem Proving and Program Development. Yves Bertot and Pierre Castéran. Springer. 2004.

Georges Gonthier. A computer-checked proof of the Four Colour Theorem. http://research.microsoft.com/~gonthier/4colproof.pdf.

C. A. R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, 12(10):576–585, 1969.

Richard Bornat. Proof and Disproof in Formal Logic. Oxford University Press, 2005.

J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*.

R. Affeldt and N. Marti. Separation Logic in Coq. http://savannah.nongnu.org/projects/seplog/

N. Marti, R. Affeldt, and A. Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In *8th International Conference on Formal Engineering Methods (ICFEM 2006)*.

R. Affeldt, N. Kobayashi, and A. Yonezawa. Verification of concurrent programs using the Coq proof assistant: a case study. IPSJ Transactions on Programming, 46(SIG 1 (PRO 24)):110-120, 2005.

L. C. Paulson. The inductive approach to verifying cryptographic protocols. Journal of Computer Security, 6:85–128, 1998.

N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A Syntactic Approach to Foundational Proof-Carrying Code. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*.

Samuel P. Harbison, Guy L. Steele Jr. C: A Reference Manual, 4th Edition. Prentice Hall. 1995.

Gérard Le Lann. The Ariane 5 Flight 501 Failure. Rapport de recherche 3079, INRIA, 1996.

MIPS Technologies. MIPS32 4KS Processor Core Family Software User's Manual.

R. Affeldt and N. Marti. Formal Verification of Arithmetic Functions in SmartMIPS Assembly. In *23rd Workshop of the Japan Society for Software Science and Technology (JSSST 2006)*.

Gerard J. Holzmann. The Spin Model Checker. Addison Wesley. 2003.

N. Marti, R. Affeldt, and A. Yonezawa. Model-checking of a multi-threaded operating system. In *23rd Workshop of the Japan Society for Software Science and Technology (JSSST 2006)*.