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 $M$ of the system
  2. A property $\varphi$ expressed in a formal logic
  3. Techniques to prove and check that $M$ satisfies $\varphi$

- 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 $\varphi$, and proofs that $\mathcal{M}$ satisfies $\varphi$
  - An automatic way to check proofs
  - An interactive way to build 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 $M$: a function in the Coq language
- Property $\varphi$: a statement in the Coq language
- Verification that $M$ satisfies $\varphi$: 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 $M$: the formal model of an imperative program
- Property $\varphi$: a statement in the Coq language
- Verification that $M$ satisfies $\varphi$: by interactive proof
Verification of Imperative Programs
Hoare Logic (1/2)

Empty statement axiom

\[
\{ P \} \text{ skip } \{ P \}
\]

Assignment axiom schema

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

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

\[
\begin{align*}
\{ E \land P \} & C \{ Q \} & \{ \neg E \land P \} & D \{ Q \} \\
\{ P \} & \text{if } E \text{ then } C \text{ else } D \text{ endif } \{ Q \}
\end{align*}
\]

While rule

\[
\begin{align*}
\{ E \land \boxed{\text{Inv}} \} & C \boxed{\text{Inv}} \\
\boxed{\text{Inv}} & \text{while } E \text{ do } C \text{ done } \{ \neg E \land \boxed{\text{Inv}} \}
\end{align*}
\]

Rule of consequence

\[
\begin{align*}
P & \Rightarrow P' \\
\{ P' \} & C \{ Q' \} & Q' & \Rightarrow Q \\
\{ P \} & C \{ Q \}
\end{align*}
\]
Verification of Imperative Programs

Example

\[
\begin{aligned}
\{ a > 0 \land b > 0 \} \\
x &:= a; y := b; \\
\{ x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \}
\end{aligned}
\]

while \( x \neq y \) do

\[
\begin{aligned}
\{ x \neq y \land x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \}
\end{aligned}
\]

if \( x < y \) then

\[
\begin{aligned}
x &< y \land x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \\
y &:= y - x \\
\{ x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \}
\end{aligned}
\]

else

\[
\begin{aligned}
x &> y \land x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \\
x &:= x - y \\
\{ x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \}
\end{aligned}
\]

endif

\[
\begin{aligned}
\{ x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \}
\end{aligned}
\]

done

\[
\begin{aligned}
\{ x = y \land x > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, b) \}
\end{aligned}
\]

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:

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

- The buffer may be smaller than the string:

  ```
  X X C O D E _ _ _ A B C D E ∅
  ```

- 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:
    
    \[ h \models (E \leftrightarrow E') \text{ iff } \text{dom}(h) = E \land h(E) = E' \]

- Memory accesses:
  
  **Mutation**
  
  \[
  \{ E \mapsto ? \} \ [E] := E' \ {E \mapsto E'}
  \]

  Example:
  
  \[
  \{ \begin{array}{c}
  ? \\
  x
  \end{array} \} \ [x] := 4 \ \{ \begin{array}{c}
  4 \\
  x
  \end{array} \}
  \]

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

  **Lookup**
  
  \[
  \{ E \mapsto E' \} \ x := [E] \ {E \mapsto E' \land x = E'}
  \]
Verification of Memory Management
Separation Logic (2/2)

- Compositional reasoning using a logic extension
  - Compound heap:
    \[
    h \models P \star Q \iff \\
    \exists h_1, h_2 \text{ s.t. } h_1 \perp h_2 \land h_1 \oplus h_2 = h \land h_1 \models P \land h_2 \models Q
    \]

- Frame Rule
  \[
  \{P\} C \{Q\} \land \text{modified}(C) \cap \text{free}(R) = \emptyset \\
  \{P \star R\} C \{Q \star R\}
  \]

Example:
\[
\begin{array}{c}
\{ \begin{array}{c}
2 \\
\hline
x
\end{array} \} \\
\end{array}
\]
\[x := 4 \]
\[
\begin{array}{c}
\{ \begin{array}{c}
4 \\
\hline
x
\end{array} \} \\
\end{array}
\]
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

\[
\begin{align*}
\{buf \Rightarrow B_0 \cdots B_{n-1} \ast str \Rightarrow S_0 \cdots S_{m-1}\} \\
c1 := buf; c2 := str; tmp := [c2]; \\
\begin{cases} 
buf \Rightarrow S_0 \cdots S_{i-1}B_i \cdots B_{n-1} \ast str \Rightarrow S_0 \cdots S_{m-1} \\
c1 = buf + i \land c2 = str + i \land tmp = S_i 
\end{cases} \\
\text{while } tmp \neq 0 \text{ do} \\
[c1] := tmp; \\
c1 := c1 + 1; \\
c2 := c2 + 1; \\
tmp := [c2] \\
\text{done;}
\end{align*}
\]

\[
\begin{align*}
\\left\{ \begin{array}{l}
tmp = 0 \land \text{buf } \Rightarrow S_0 \cdots S_{i-1}B_i \cdots B_{n-1} \ast str \Rightarrow S_0 \cdots S_{m-1} \\
c1 = buf + i \land c2 = str + i \land tmp = S_i 
\end{array} \right. \\
[c1] := tmp \\
\{buf \Rightarrow S_0 \cdots S_{m-1} \ast T \ast str \Rightarrow S_0 \cdots S_{m-1}\}
\end{align*}
\]

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:

    ![Linked List Diagram]

- Separation property:
  "Newly allocated blocks do not override old ones"

- Related problem found during verification of existing code:
  - Memory exhaustion:

    ![Memory Exhaustion Diagram]
Verification of the Implementation of Cryptosystems

- Algorithms and 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:
    ```c
    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
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”

```coq
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::tlb => listbit_lt tla tlb
    | _ => false
  end
  | _ => false
end.
```
Signed integers in two’s complement notation:

Definitions:

\[
\begin{align*}
(a_n \ldots a_0)_u &= a_n 2^n + \cdots + a_0 \\
(a_n \ldots a_0)_s &= -a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_0
\end{align*}
\]

Examples:

- \((0001)_u = (0001)_s\) but \((1111)_u \neq (1111)_s\)
- In Coq:
  
  \[
  \begin{align*}
  ([[ \text{o::o::i::nil }]]_u &= 1 & [[ \text{i::i::i::i::nil }]]_u &= 15 \\
  ([[ \text{o::o::o::i::nil }]]_s &= 1 & [[ \text{i::i::i::i::nil }]]_s &= -1
  \end{align*}
  \]

We retrieve the “expected” properties:

- \(-1 \nmid 1\)
- In Coq:
  
  \[
  \text{listbist_lt } (\text{i::i::i::i::nil}) (\text{o::o::o::i::nil}) = \text{false}
  \]
 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:

\[
\begin{cases}
X, Y, M & \text{such that } |X|, |Y|, |M| = k \text{ and } X, Y < M \\
Z & \text{such that } |Z| = k + 1 \text{ and } Z = 0 \\
\alpha & \text{such that } \alpha \cdot M_0 \equiv -1[\beta]
\end{cases}
\]

\[
\text{montgomery } X \ Y \ M \ Z \ \alpha \\
\{ \beta^k \cdot Z \equiv X \cdot Y[M] \text{ and } Z < 2 \cdot M \}
\]

- Example: \(10^5 \cdot 39796 \equiv 5792 \cdot 1229 [72639]\)

- Basic idea: zero the least significant word of partial products

\[
\begin{array}{ccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 & 8 & 3 & 5 & 7 & 0 \\
0 & 0 & 0 & 0 & 6 & 5 & 0 & 5 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & 5 & 0 & 9 & 4 & 9 & 0 & 0 & 0 & 0 \\
0 & 0 & 3 & 4 & 7 & 6 & 5 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 3 & 9 & 7 & 9 & 6 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

- 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 with 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

\[ \text{Reachable} (\text{Init}) \cap \text{Bad} = \emptyset \]
Specification of Path Properties

Path properties are better expressed with temporal logics

- A path is a sequence of states:

  ![Path Diagram]

- Sample path properties
  - Stability: “There will be a state from which $\varphi$ is always true.”
    ![Stability Diagram]
    Linear Temporal Logic (LTL) notation: ♦□$\varphi$
  - Response:
    "Always, whenever there is a request, there will be eventually a reply."
    ![Response Diagram]
    LTL notation: □($Request \rightarrow \diamond Reply$)
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: $\Box \Diamond (client\_version = server\_version)$

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

```plaintext
typedef Message {
    int file_version;
    mtype signature
} mtype = { client_key, server_key }

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

int client_version = 100;
int server_version = 102;

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:

```
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:

```plaintext
/******************************************************
    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:

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

/* 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 $\square\Diamond (client\_version = server\_version)$:

```plaintext
/****************************
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
}
```

⇒ 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 cryptographic 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
References


Georges Gonthier. A computer-checked proof of the Four Colour Theorem.


In 17th IEEE Symposium on Logic in Computer Science (LICS 2002).

http://savannah.nongnu.org/projects/seplog/

In 8th International Conference on Formal Engineering Methods (ICFEM 2006).

IPSJ Transactions on Programming, 46(SIG 1 (PRO 24)):110-120, 2005.

L. C. Paulson. The inductive approach to verifying cryptographic protocols.

In 17th IEEE Symposium on Logic in Computer Science (LICS 2002).


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).


In 23rd Workshop of the Japan Society for Software Science and Technology (JSSST 2006).