National Institute of Advanced Industrial Science and Technology
AIST

Research Center for Information Security

Towards a Certified Implementation of a Cryptographically Secure Pseudorandom Bit Generator

暗号学的に安全な擬似乱数生成器の正しい実装に向けて

(Work in progress)

Kiyoshi YAMADA
(Joint work with David NOWAK)

Research Center for Information Security, AIST
Mar. 7, 2009

JSIAM-FAIS at Kyoto

Outline

- **Introduction**
  - Background
  - Our goal
  - How to verify the security of implementation?
  - Blumb Blumb Shub(pseudo random bit generator)
  - The proof assistant Coq
- **Formalization and Verification**
- **Related work**
- **Conclusion**

Background

- **Cryptographic primitives**
  - Basic cryptographic algorithms
    - cipher, hash, signature, pseudo random bit generator, etc.
  - Building blocks for cryptographic systems construction
  - Well tested by cryptographers
  - No guarantee of implementation security
    - usual programmers are not expert in cryptography
    - programming is error prone work

Our goal

- **Develop a toolbox to verify the security of cryptographic primitive implementations**
  1. Establish a framework to verify properties on the program code
  2. Verify the correctness of cryptographic primitives implementation
- **First target: Blum Blum Shub(BBS) in x86-64**

  cryptographically secure pseudo random bit generator
  - why BBS?

    it has a strong security proof based on a mathematical problem
  - why X86-64?

    code is often written in assembly language for efficiency

Ongoing work

How to verify the security of implementation?

Two Step approach

Step 1: verify the security of cryptographic algorithm

Step 2: verify the implementation follows its algorithm

Use the proof assistant Coq

Roadmap and the progress

Step 1:

✓Verify BBS algorithm security [Nowak, 2008]

Today's Talk

Step 2:

✓Implement BBS in x86-64 assembly language

➡Formalize x86-64 architecture in proof assistant Coq

➡Prove correctness of BBS implementation in Coq

Blum Blum Shub (BBS)

- **A cryptographically secure pseudorandom bit generator [Blum et al, 1986]**
  - Algorithm: $x_{i+1} = x_i^2 \bmod n$
    - i-th output = least significant bit of x_i
    - n is a 'Blum integer' i.e.

      a product of two big prime numbers

      each prime number is congruent to 3 modulo 4
  - Cryptographically secure
    - satisfy the "next-bit unpredictability"

      no polynomial-time algorithm can distinguish between an output sequence of the generator and a truly random sequence
  - BBS security relies on the quadratic residuosity problem

The proof assistant Coq

A formal proof management system

- developed by INRIA since 1984

Coq allows:

- formalization

  define functions or predicates, state mathematical theorems and software specifications

- formal verification

  develop interactively formal proofs of these theorems, check these proofs by a relatively small certification "kernel"

Lot of results using Coq

- Formalization of semantics of subset of C, certified mini-ML compiler, formal verification of incremental GC, ...

National Institute of Advanced Industrial Science and Technology
AIST

Research Center for Information Security

Outline

- ✓ **Introduction**
- **Formalization and Verification**
  - Implementing BBS in x86-64
  - Formalizing x86-64 in Coq
  - Verifying properties of x86-64 program
- **Related work**
- **Conclusion**

Implementing BBS in x86-64 (1/2)

Implementation of BBS

```
inline void square_asm_triangle(
  ulong* w, ulong* u, ...) { ... }
inline void square_asm_diagonal(
  ulong* w, ulong* u, ...) { ... }

inline void square_asm(ulong* w, ulong* u, ...) {
  /* compute w = u * u */
  square_asm_triangle(w, u, ...);
  shiftLeft1Bit(w, ...);
  square_asm_diagonal(w, u, ...); }

inline void div_mod_asm(ulong* u, ulong v, ...) {
  /* compute (u, v) = (u mod v, u div v) */ }

void bbs_step(ulong* y, ulong* x, ulong* m, ...) {
  /* y = (x * x) mod m */
  square_asm(y, x, ...); div_mod_asm(y, m, ...); }
```

```
          a  b  c  d
       x  a  b  c  d
       ------------
          ad bd cd dd
       ac bc cc dc
    ab bb cb db
aa ba ca da

abcd ^ 2 = triangle * 2
  + diagonal
```

- original implementation is in assembly language, but here we show a decompiled to C-like language version for the sake of simplicity

Implementing BBS in x86-64 (2/2)

- **about the implementation of BBS**
  - about 400 lines
  - consists of multiple-precision integer arithmetic operation multiplication and residue operation
- **practical implementation**
  - square is not implemented as a simple multiplication,
  - special cases are treated with special code
- **but, a few restrictions for easier verification**
  - no subroutines, do not treat negative value, no absolute addressing

Formalizing: Store (1/2)

Store

- model of flags, registers and memory

```
(* Store definition *)
Record Store : Set := {
  get_cf     : bool;
  get_zf     : bool;
  get_regs   : list Z;
  get_memory : list (list Z)
}.
```

```
(* register definitions *)
Definition RAX : nat := 0%nat.
Definition RCX : nat := 2%nat.
Definition RDX : nat := 3%nat.
Definition RSI : nat := 4%nat.
Definition RDI : nat := 5%nat.
```

- only elements needed for implementing BBS are modeled
- elements of register and memory cells are integers

  their values are restricted to 64-bit unsigned integer by the semantics of instructions
- registers are modeled as a list of integer and their names are natural numbers which correspond to their index in this list

Formalizing: Store (2/2)

Memory model

- the memory is modeled as a list of memory blocks
- a memory block is modeled as a list of integers

```
(* Store definition *)
Record Store : Type := {
  get_cf     : bool;
  get_zf     : bool;
  get_regs   : list Z;
  get_memory : list (list Z)
}.
```

- assume no overlap between memory cells

  this property might be verified independently of our work

  we are currently focusing on the correctness of the arithmetic part of BBS

Formalizing: Address

Address

- memory cell is selected through Addr

```
(* address definition *)
Inductive Addr : Set :=
| addr : Z (*displacement*) ->
         nat (*base reg*)   ->
         nat (*index reg*)  ->
         Addr.
```

```
(* Semantics of Addr *)
Definition sem_addr
  (s:s)(a:Addr) : nat*nat :=
  match a with
  | addr dis bas ind => (
    Zabs_nat(get_reg s bas),
    Zabs_nat(dis + get_reg s ind))
  end.
```

- example

  address (Addr -1 RCX RDX) points to the cell at RDX-1 of memory block at RCX

  this corresponds to -1(RCX, RDX, 8) in X86-64

Formalizing: Instruction, Code

Instruction and Code

```
Inductive Cond : Set :=
| carry : Cond
| zero  : Cond
| not   : Cond -> Cond.

Inductive Instr : Set :=
| clc   : Instr
| rcl_a : Addr -> Instr
| dec_r : nat -> Instr
... .
```

```
Inductive BCode : Set :=
| instr : nat -> Instr -> BCode
| goto  : nat -> nat -> BCode
| cgoto : nat -> Cond -> nat ->
          BCode.

Inductive Code : Set :=
| empty : Code
| bcode :> BCode -> Code
| comp  : Code -> Code -> Code.
```

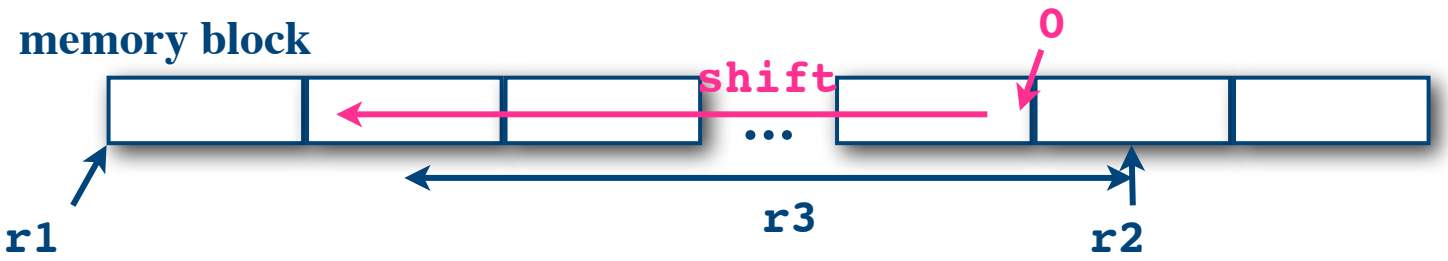
- Cond models conditions for conditional jump instruction
- Instr models non-jump instructions
- BCode models all the Instructions with labels
  - a label corresponds to the address in assembly language
  - a label is modeled as a natural number
- Code models a cluster of instructions

Formalizing: Example

Code example

```
Definition ShiftLeft1Bit (l r1 r2 r3 : nat) : Code :=
  (* r1 = base, r2 = offset + length, r3 = length *)
  (comp (instr    l   clc)
  (comp (instr (1+l) (rcl_a (addr (-1) r1 r2))
  (comp (instr (2+l) (dec_r r2))
  (comp (instr (3+l) (dec_r r3))
        (cgoto (4+1) (not zero) l))))).
```

- Memory cells at index r2-r3 to r2-1 are shifted left
- LSB is set to 0



Formalizing: Compositional Semantics

Semantics

- customized compositional semantics of [Saabas and Uustalu, 2007]
- we use its compositionality to split full specifications and proofs into small pieces
- this semantics is equivalent to the usual small-step semantics
- both semantics and equivalence of them are formalized in Coq

$$\frac{}{\texttt{sem\_code(l,s)(instr l i)(S l, sem\_instr s i)}}\ \text{c\_instr}$$

$$\frac{\texttt{l<>l'}}{\texttt{sem\_code(l, s)(goto l l')(l', s)}}\ \text{c\_goto}$$

$$\frac{\texttt{sem\_cond s cond = true}\quad \texttt{l<>l'}}{\texttt{sem\_code(l,s)(cgoto l cond l')(l',s)}}\ \text{c\_cgoto\_true}$$

$$\frac{\texttt{sem\_cond s cond = false}}{\texttt{sem\_code(l,s)(cgoto l cond l')(S l,s)}}\ \text{c\_cgoto\_false}$$

$$\frac{\texttt{l}\in\texttt{dom c1}\quad \texttt{sem\_code(l,s)c1(l',s')}\quad \texttt{sem\_code(l',s')(comp c1 c2)(l'', s'')}}{\texttt{sem\_code(l,s)(comp c1 c2)(l'',s'')}}\ \text{c\_comp\_left}$$

$$\frac{\texttt{l}\in\texttt{dom c2}\quad \texttt{sem\_code(l,s)c2(l',s')}\quad \texttt{sem\_code(l',s')(comp c1 c2)(l'', s'')}}{\texttt{sem\_code(l,s)(comp c1 c2)(l'',s'')}}\ \text{c\_comp\_right}$$

$$\frac{\texttt{l}\notin\texttt{dom c}}{\texttt{sem\_code (l,s)c(l,s)}}\ \text{c\_end}$$

```
Definition sem_instr (s:s)(i:Instr):s.
Definition sem_cond (s:s)(c:Cond):bool.
Fixpoint dom(c:Code) : list nat :=
  match c with
  | empty => nil
  | instr l _ | goto l _ | cgoto l _ _ => l :: nil
  | comp c1 c2 => dom c1 ++ dom c2.
```

Verification: Our goal in Coq

```
Variables p q : Z.
Hypothesis p_prime : prime p.
Hypothesis q_prime : prime q.
Parameter code : Code.

Parameter bbs :
  nat ->
  Zstar (n_gt_1 p_prime q_prime) ->
  list bool.

Parameter sem_code :
  State -> Code -> State.
```

```
Parameter encode :
  nat ->
  Zstar (n_gt_1 p_prime q_prime) ->
  State.
Parameter decode :
  State -> list bool.
```

```
Theorem correct :
  forall len seed final_state,
  sem_code (encode len seed) code
           final_state ->
  decode final_state = bbs len seed.
```

- **`bbs`** is the BBS algorithm modeled in Coq
- **`sem_code`** models x86-64 execution
- **`code`** is an implementation of BBS in x86-64 assembly language
- Theorem **`correct`** is a proposition which states that the implementation follows its algorithm

Verification: ShiftLeft1Bit (1/2)

- **Specify and prove the body of the loop**
  - Body of the loop:

```
Definition ShiftLeft1Bit_123(l r1 r2 r3:nat):Code :=
  (comp (instr    l  (rcl_a (addr (-1) r1 r2 1)))
  (comp (instr (1+l) (dec_r r2))
        (instr (2+l) (dec_r r3)))).
```

  - What we proved
    - Well-formedness
    - Termination
    - Value changes on flag, registers and memories

      Example:

```
Lemma ShiftLeft1Bit_123_correct_r2 :
  forall l r1 r2 r3 s s',
  r2 <> r3 ->
  0 < get_reg s r2 < 2^64 ->
  sem_code (l,s) (ShiftLeft1Bit_123 l r1 r2 r3) ((3+l),s') ->
  get_reg s' r2 + 1 = get_reg s r2.
```

Verification: ShiftLeft1Bit (2/2)

- **Same properties proved with loop**

```
Definition ShiftLeft1Bit_1234(l r1 r2 r3:nat):Code :=
  (comp (ShiftLeft1Bit_123 l r1 r2 r3)
        (cgoto (3+l) (not zero) l)).

Definition ShiftLeft1Bit(l r1 r2 r3:nat):Code :=
  (comp (instr l clc)
        (ShiftLeft1Bit_1234 (1+l) r1 r2 r3)).
```

  - compositionality is used here

- **We define much reusable lemmas:**
  - inversion lemmas

    Coq's inversion tactic is powerful, but in some cases it does not work well
  - idempotence, symmetry, associativity for `comp`

    for formal manipulation of syntactic tree
  - specifications for each instruction

Oversight in [Saabas and Uustalu, 2007]

B.1 Proof of Theorem 6

Theorem 6 (Preservation of evaluations as stuck reduction sequences)

$$\textit{If } (l,\sigma) \vdash c \Downarrow (l',\sigma'), \textit{then } (l,\sigma) \xrightarrow{c}{}^{*} (l',\sigma') \not\rightarrow$$

Proof. By structural induction on the derivation of $(l,\sigma) \rightarrow (l',\sigma')$.
The case of derivation of $(l,\sigma) \rightarrow (l',\sigma')$ is of the form

$$\frac{\begin{array}{ccc}\vdots & & \vdots \\ l \in \mathrm{dom}(c_i) & (l,\sigma) \vdash c_i \Downarrow (l'',\sigma'') & (l'',\sigma'') \vdash (c_0 \oplus c_1) \Downarrow (l',\sigma')\end{array}}{(l,\sigma) \vdash c_0 \oplus c_1 \Downarrow (l',\sigma')}$$

where $i = 0$ or 1: By the induction hypothesis, we have $(l,\sigma) \xrightarrow{c_i}{}^{*} (l'',\sigma'') \not\rightarrow$ and $(l,\sigma) \xrightarrow{c_0 \oplus c_1}{}^{*} (l'',\sigma'') \not\rightarrow$. By Lemma 3, we have $(l,\sigma) \xrightarrow{c_0 \oplus c_1}{}^{*} (l',\sigma')$. Hence $(l,\sigma) \xrightarrow{c_0 \oplus c_1}{}^{*} (l'',\sigma'') \xrightarrow{c_0 \oplus c_1}{}^{*} (l',\sigma') \not\rightarrow$.

not applicable!

Lemma 3 (Extension of the domain)

$\textit{If } c_0 \subseteq c_1 \textit{ and } l \in \mathrm{dom}(c_0), \textit{then } (l,\sigma) \xrightarrow{c_0} (l',\sigma') \textit{ iff } (l,\sigma) \xrightarrow{c_1} (l',\sigma').$

Reformulate

Lemma 3' (Extension of the domain)

$\textit{If } c_0 \subseteq c_1 \textit{ and } l \in \mathrm{dom}(c_0), \textit{and } (l,\sigma) \xrightarrow{c_0}{}^{*} (l',\sigma') \textit{ then } (l,\sigma) \xrightarrow{c_1}{}^{*} (l',\sigma').$

Outline

- ✓ Introduction
- ✓ Formalization and Verification
- ➡ Related work
- ➡ Conclusion

Related Work

- **Formal verification for montgomery multiplication implemented in assembly language using Coq [Affeldt et al, 2006]**
  1. replace all jumps by while loops
  2. make a proof using standard Hoare logic
  3. convert while loops to jump using a certified translator
- **Verification of machine code implementations of arithmetic functions for cryptography [M. Myreen and M. J. C. Gordon, 2007]**
  1. build a functional implementation of the algorithm
  2. make the correctness proof of the functional program
  3. prove assembly code implements functional program

Conclusion

- **Summary**
  - Background
    - no guarantee of implementation security for cryptographic primitives
  - Our goal
    - develop a toolbox to verify the security of cryptographic primitive implementations
  - Our first results
    - implementation of BBS in x86-64 assembly language
    - formalization of the compositional semantics for assembly language [Saabas and Uustalu, 2007] in Coq
    - formalization of ShiftLeft1Bit, a part of our implementation of BBS, and proofs of its properties