A CompCert Backend with Symbolic Encryption

Paolo Torrini, Sylvain Boulmé

INP-UGA, Verimag

GT-MFS, Frejus, 21.03.2022

NanoTrust/IntrinSec

NanoTrust Project: protecting execution by secure compilation with binary code encryption

- compiling C code, targeting RISC-V 32-bits instruction set
- collaboration between Verimag (Marie-Laure Potet, David Monniaux) and CEA (Olivier Savry)
  - co-developed at CEA (prioprietary artifacts):
    - LLVM compiler (multi-level encryption)
    - matching RISCY processor
  - developed at Verimag (discussed in this talk):
    IntrinSec – our certified compiler based on CompCert RISC-V backend extended with symbolic instruction-level encryption

CompCert

- Certified C compiler formalized and verified in Coq
- Compiles to Asm, different backends including RISC-V
- Memory model: split into blocks, address = (block, offset), separates code from data, a code block for each function
- Chain of passes as translations between intermediate languages, simulation proof for each (forward simulation, reversible to backward by determinism of the target)
- Correctness: compiled code behaviour is source behaviour

CompCert lower backend

- Linear, Mach: linerarized languages
  - list of instructions, sequential execution
  - structured state (normal, call, return)
  - structured call stack, pop and push,
    function parameters in caller stackframe
  - Linear: stack as inductive datatype
  - Mach: inductive stack matched by linked list of stackframes in memory (translation from Linear to Mach enforces slot separation)
- Asm:
  - state: memory and registers
  - stack: stored in data memory
  - PC points to next instruction (either sequential or jump), SP to stack, RA to return address

Code and Control Flow Integrity

- attacks either exploit hardware faults or buffer overflows
- code insertion/reuse attacks (code integrity issue): trick the processor into executing external code, or modified internal code
- stack overflow attacks (control flow integrity issue): divert control flow by altering return addresses in the stack
- general mitigating policy:
  separate executable code and rewritable data.
  however: the stack contains control-flow relevant data
- Abadi's work on CFI: constrain control flow to a statically computed control-flow graph (CFG) by instrumenting Asm code with node labels and dynamic checks

CFI vulnerabilities

1. Preservation of CFG forward edges (jumps)
   1. direct jumps: destination known at compile time
   2. indirect jumps: destination only known at runtime
2. Preservation of CFG backward edges (returns): involves protecting the stack
   - up to Mach: stack is inductively structured
   - beyond: backlink stack pointer and return address are vulnerable

Code vulnerabilities

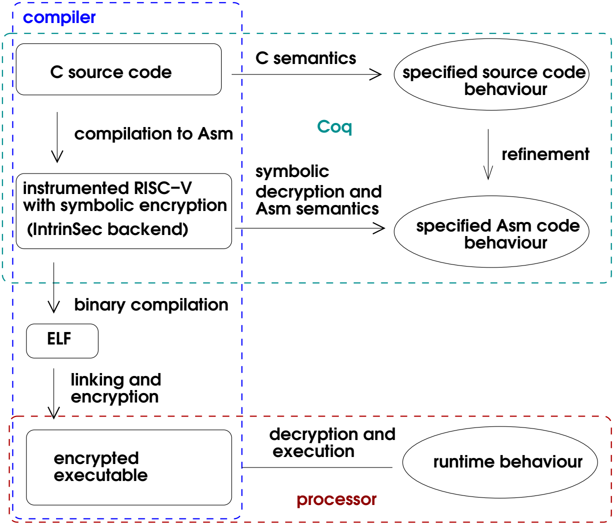
1. Preservation of executable code (CI)
2. Preservation of non-structural stack data: function call parameters, local variables (memory data already in Mach)
3. Preservation of function entry points
    - up to Mach: language ensures function code only accessed from the start
    - problem: compilation to Asm disrupts this guarantee
    - goal: hardening Asm to prevent such disruption

Workplan

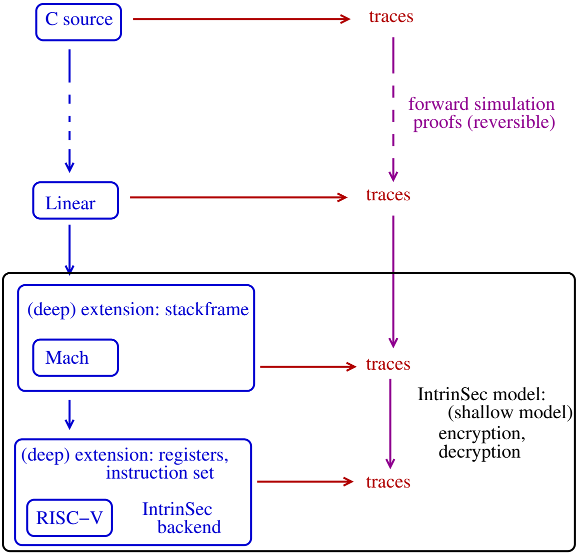
Two main ideas:

1. use encryption of executable code to ensure CI and (partial) CFI
2. make explicit in Asm protection which is implicit in the Mach semantics

NanoTrust/IntrinSec structure



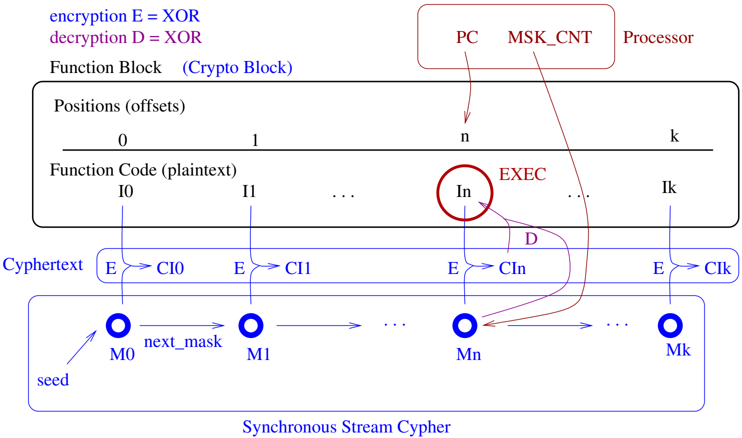
IntrinSec structure



Encryption in NanoTrust

1. instruction-level: single instruction encryption, based on 32 bit masks (one for each instruction, combined with plaintext by XOR) and stream cyphers
   [CEA, IntrinSec explicitly]
2. program-level: whole program encryption using heavy-weight key [CEA, IntrinSec implicitly]
3. pointer-level: code and data encryption based on fat pointers
   [CEA]

Instruction-level encryption



stream cypher (here): finite stream of masks defined by

1. initial mask (generated from a *seed*)
2. *next_mask* function (pseudo-random)

IntrinSec backend

- symbolic instruction-level encryption and decryption model:
  - CompCert verification goes as far as assembly, encryption applied to binary code after linking, so we resort to an axiomatic model

- ISA instrumentation on top of RISC-V:
  - 3 crypto registers:
    1. MSK_CONT: mask for the next instruction
    2. MSK_BRN: destination mask before the jump
    3. RET_MSK: return access mask
  - 4 crypto instructions:
    1. load destination mask known at compile-time
    2. load destination mask known at runtime
    3. store mask to the stack
    4. load mask from the stack

Instruction-level encryption protocol

- crypto block = function code block
- function entry mask stored at block start,
  internal masks sequentially determined by *next_mask*
- when jumping:
  load destination mask,
  store return mask on stack (as with return address)
- when returning:
  load return mask from stack (as with return address)

Example

```c
int fact(int n){
  if (n <= 1) return 1;
  return n*fact(n-1);
}
```

```
  ecr.enter
fact:
  mv        x30, sp
  addi      sp, sp, -16
  sw        x30, 0(sp)
  sw        ra, 4(sp)
  ecr.sw    emr, 8(sp)
  sw        x8, 12(sp)
  mv        x8, ra0
  ecr.load  emb, L100
  addi      x31, x0, 1
  blt       x31, x8, .L100
  addi      ra0, x0, 1
  ecr.load  emb, L101
  j         .L101
.L100:
  addi      ra0, x8, -1
  ecr.load  emb, fact
  call      fact
  mul       ra0, x8, ra0
.L101:
  lw        x8, 12(sp)
  lw        ra, 4(sp)
  ecr.lw    emb, 8(sp)
  addi      sp, sp, 16
  jr        ra
```

IntrinSec step relation (extending RISC-V)

New relation (decryption condition): the value in `v2` is the mask for `v1`

```
Inductive valid_mask_at_pc (v1 v2: val) : Prop :=
  | valid_mask_at_pc_intro : ∀ (fb: block) (ofs: ptrofs),
     v1 = Vptr fb ofs ∧ v2 = Vint (encrypt_msk ofs fb) →
     valid_mask_at_pc v1 v2.
```

Step relation – the step at TS is executed only if decryption succeeds (the mask in `MSK_CNT` is right for the value in `PC`):

```
Inductive Asm_step: state → trace → state → Prop :=
| exec_step_internal: ∀ b ofs f i rs m rs' m',
        rs PC = Vptr b ofs →
        find_funct_ptr b = Some (Internal f) →
        find_instr ofs f = (Some i) →
 (*DS*) valid_mask_at_pc (rs PC) (rs MSK_CNT) →
 (*TS*) exec_instr b f i rs m = Next rs' m' →
        Asm_step (State rs m) E0 (State rs' m').
```

IntrinSec simulation theorem

Revised Match state relation:

```
Inductive match_states: Mach.state → Asm.state → Prop :=
| match_states_normal: ...
| match_states_call: ...
| match_states_return: ...
```

Forward simulation theorem:

```
Theorem step_simulation :
  ∀ S1 t S2, Mach_step S1 t S2 →
  ∀ S1' (MS: match_states S1 S1'),
  (∃ S2', plus Asm_step S1' t S2' ∧ match_states S2 S2')
  ∨ (measure S2 < measure S1 ∧ t = E0 ∧ match_states S2 S1').
```

Informally: each Mach step, starting from Mach state matched by Asm state, can be simulated by Asm steps (no stuttering).

Instruction-level encryption: security aspects

- Code integrity ensured by encryption, also against code insertion exploits
- function code only accessed from start (entry mask needed, we assume *next_mask* is secret)
- stack data not protected
- CFG forward edges:
  - direct jumps protected by encryption
  - indirect jumps: access mask stored with the function
- CFG backward edges: stack data not protected, but return address needs is paired with return mask

Generalizing crypto blocks: in progress

- crypto blocks need not be function blocks:
  special labels to reset the stream cypher
- makes encryption model more complex, as encryption function depends on code, not only on position
- helps shifting to different, syntax-based notion of straightline code (code without jumps)

Security aspects: the Mach point of view

Regardless of encryption, how do Mach security properties (function entry points and structural stack character) reflect in Asm?

- PseudoAsm: intermediate language between Mach and Asm
- same instruction set as Mach
- Asm-style semantics, state = memory + registers, memory stack, use of PC, RA and SP
- breakdown of translation from Mach to Asm:
  1) from Mach to PseudoASM
  2) from PseudoASM to Asm

PseudoAsm back-translation

- translating back from PseudoAsm to Mach
- stronger match-state relation, requiring the memory stack to preserve the structure of the inductively defined one (memory-well-formedness, MWF)
- (backward) simulation provable under the stronger match-state relation
- MWF can be enforced in forward translations from Linear to Mach, and from Mach to PseudoAsm
- under the MWF restriction, PseudoAsm programs can only behave as Mach ones, and so preserve CFG as much as them

Conclusions

- certified IntrinSec compiler
- lightweight encryption, low overhead
- main workload: approx 6-7 months person work, approx. 6000 lines code added, Coq 8.10, CompCert 3.8
- main hurdle: loss of reuse wrt standard CompCert backends, due to changes in the notion of straightline code
- improve modularity
- memory model, fat pointers
- proving formally security properties