Lecture Notes:
RockSalt: Better, Faster, Stronger SFI for the x86

Michael Brown

March 2, 2017

1. Introduction

Native Client (NaCl) allows users to run untrusted code in a sandbox. The developer compiles code using the NaCl compiler, which adds checks to the code to make sure it doesn't go outside of the sandbox. This makes the job easier for the NaCl checker, which the user invokes to verify code before running it.

NaCl Sandbox Policy:

- Only read/write in allowed memory segments.
- Only execute from code segment, not data segments.
- Do not execute disallowed instructions (e.g. system calls)
- Communicates only using entry points defined by NaCl.

Their new implementation of the NaCl checker, RockSalt, is:

- Verified
- Fast
- Easily modifyable

Note: RockSalt only targets 32-bit x86, not 64-bit.

2. A Coq Model of x86

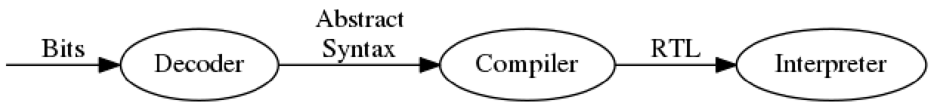

Figure 1: Model Diagram

They first decode bits into an abstract syntax representation, which is basically just a representation of assembly code. They then translate the abstract syntax into RTL, a simple language they represent the effects of instructions with. Their final model is an interpretation of RTL.

Decoder

- Translated instruction tables directly from Intel manual.
- Parser written in their own language that specifies a grammar in regular expressions, and a Coq function to call on a match.
- CALL instruction example:

```
Definition CALL_p : grammar instr :=
    "1110" $$ "1000" $$ word @
    (fun w => CALL true false (Imm_op w) None)
|| "1111" $$ "1111" $$ ext_op_modrm2 "010" @
    (fun op => CALL true true op None)
|| "1001" $$ "1010" $$ halfword $ word @
    (fun p => CALL false false (Imm_op (snd p))
        (Some (fst p)))
|| "1111" $$ "1111" $$ ext_op_modrm2 "011" @
    (fun op => CALL false true op None).
```

Implementation:

- $deriv_c\ g$ - Adjusts grammar g so it matches the same grammar with with a leading c stripped away.
- They parse by taking the derivative of their grammar (with respect to the current character c) until it matches the empty string.

Ex: For a grammar g = "$[ax][by]$" matching the string "xb":

- g1 = $deriv_x\ g$
  - g1 = "$[by]$"
  - g1 does not match empty string
- g2 = $deriv_b\ g1$
  - g2 = ""
  - g2 does match the empty string

RTL

```
* Simple RISC-like language

Definition conv ADD prefix mode op1 op2 :=
    let load := load op prefix mode in
    let set := set op prefix mode in
    let seg := get segment op2 prefix DS op1 op2 in
    zero ← load Z size1 0;
    up ← load Z size1 1;
    p0 ← load seg op1;
    p1 ← load seg op2;
    p2 ← arith add p0 p1;
    set seg p2 op1;;
    ...
```

Model Validation

They instrument binaries, using Intel's Pin tool, then run them to get the register and memory values after each instruction. Then they compare these values from their model. They test with several types of programs:

- C programs generated with Csmith
  - Does not cover instructions not emitted by compilers.
- Random bytes
  - Uses their grammar to test all encodings of all instructions.

3. The RockSalt NaCl Checker

Jumps

Variable length instructions make checking instructions harder, since malicious programs can jump into the middle of an instruction. NaCl imposes a 32-byte jump alignment policy to make verification more feasible:

1. The first byte is the start of a valid sequence of instructions.
2. Every 32 bytes is the start of a valid sequence of instructions.
3. Indirect jumps are masked so they are 32-byte aligned.
4. Indirect jumps and their mask operation do not cross a 32-byte boundary.
5. Direct jumps target the start of a valid sequence of instructions. (cannot target indirect jumps)

Checking Routine

They split instructions into categories, each of which can be matched with a regular expression generated from their model:

- No control-flow
- Direct jumps
- Indirect jumps with preceeding mask
- Rejected instructions

Main loop:

- Starts at first byte
- For each instruction:
  - Mark byte as a valid jump target.
  - Match the instruction DFA
    - MaskedJump: Continue
    - NoControlFlow: Continue
    - DirectJump: Record the target address.
    - Other: Reject the program

After the main loop, the following checks are performed:

- All jump targets are valid.
- Every 32-byte boundary is valid.

They compared RockSalt to the original NaCl checker by testing the validity of many programs.

4. Proof of Correctness for the Checker

Appropriate state:

1. the original data and code segments are disjoint,
2. the DS, SS, and GS segment registers point to their respective original segments,
3. the CS segment registers point to the original code segment,
4. the program counter points within the code segment, and
5. the original bytes of the program are stored in the code segment.

Locally-Safe state - Both appropriate and next instruction matches DFA for MaskedJump, NoControlFlow, or DirectJump.

Starting from a locally-safe state, you should always stay in a locally-safe state. MaskedJump consists of 2 instructions, so it is technically not locally-safe. It is 2-safe though: after executing both instructions it will again be locally-safe.

Proof works by case analysis on each instruction that could have matched each DFA. For each DFA they prove:

- NoControlFlow
  - Does not modify segment registers.
  - Modifies only data segment memory.
  - New program counter points to the next instruction.
- DirectJump
  - Above NoControlFlow properties
  - Target was marked as valid
- MaskedJump
  - Instruction is 2-safe - First instruction will always be a mask, they must prove the second instruction always jumps to a 32-byte boundary.