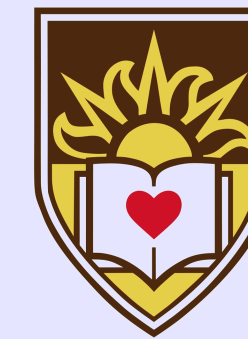
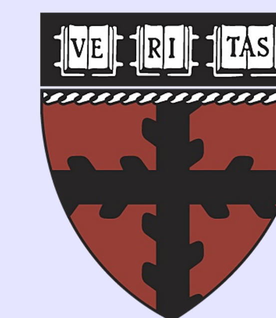
RockSalt: A Formally Verified Machine Code Security Checker





PIs: Gang Tan (Lehigh University), Greg Morrisett (Harvard University)

Participants: Joseph Tassarotti, Jean-Baptiste Tristan, Edward Gan

Introduction and Overview

Motivation

•Google's Native Client (NaCl) uses software-based fault isolation (SFI) to safely run untrusted binaries.

•Policy conceptually simple, but easy to get the details wrong.

Goal

•Build checker with smaller trusted computing base.

Methodology

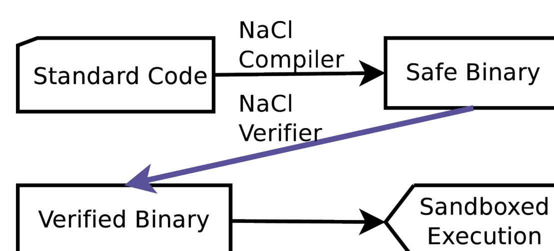
•Develop formal model of x86 in Coq using declarative DSLs. Two components: Syntax (instruction decoding) and Semantics.

•Automatically generate from x86 specifications a DFA to conduct SFI verification

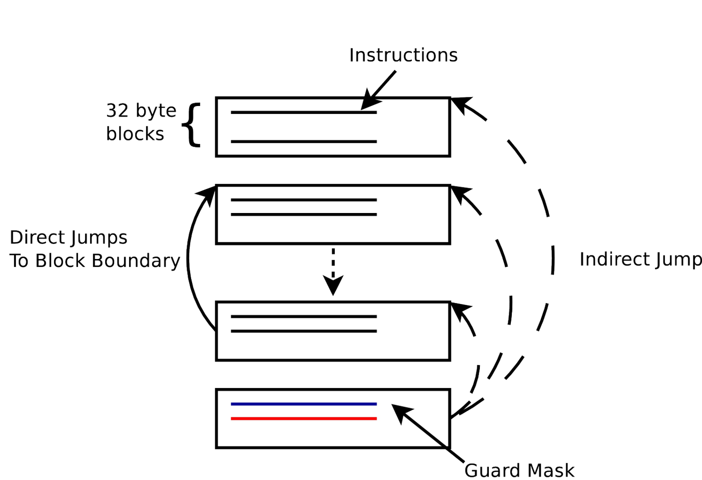
•Prove DFA generator respects safety policy

SFI Policy

Compilation, Verification, Execution

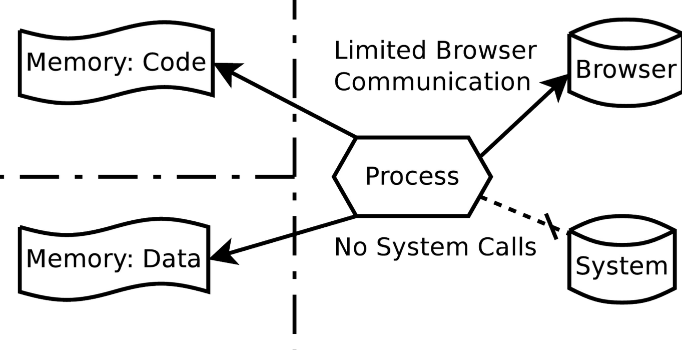


Restricting Parse Offsets



Sandbox Constraints

Compartmental Memory:
Segment Registers



•Problem: How to ensure correct instruction parsing for different parse offsets?

Valid Code Example

```
eb 0e        jmp   100055b <fib+0x5b>
90           nop
90           nop
90           nop
 ...
90           nop
90           nop
83 c4 14     add   $0x14,%esp
5b           pop   %ebx
5d           pop   %ebp
59           pop   %ecx
83 e1 e0     and   $0xffffffe0,%ecx
ff e1        jmp   *%ecx
```

Results and Conclusions

Deliverables:
•Extensible and Declarative DSL for specifying parsers and interpreters
•Parser for 130 instructions, semantic specifications for 70
•Fast, small, DFA tables proven correct

Future Work:
•More instructions:
•More sophisticated runtime model
•Verified C code
•Richer policies: XFI

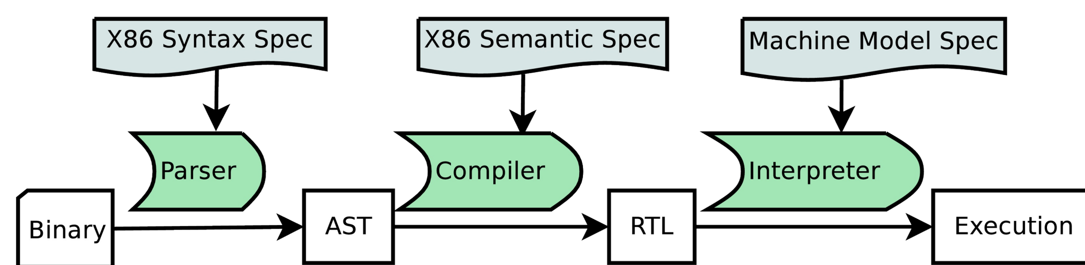
	NaCl	RockSalt
How many lines of C code do you trust?	30,000?	$\leq$ 100
How fast? (200k lines of C code)	0.90s	0.24s
Updating policy?	re-test	re-check

Lessons Learned:

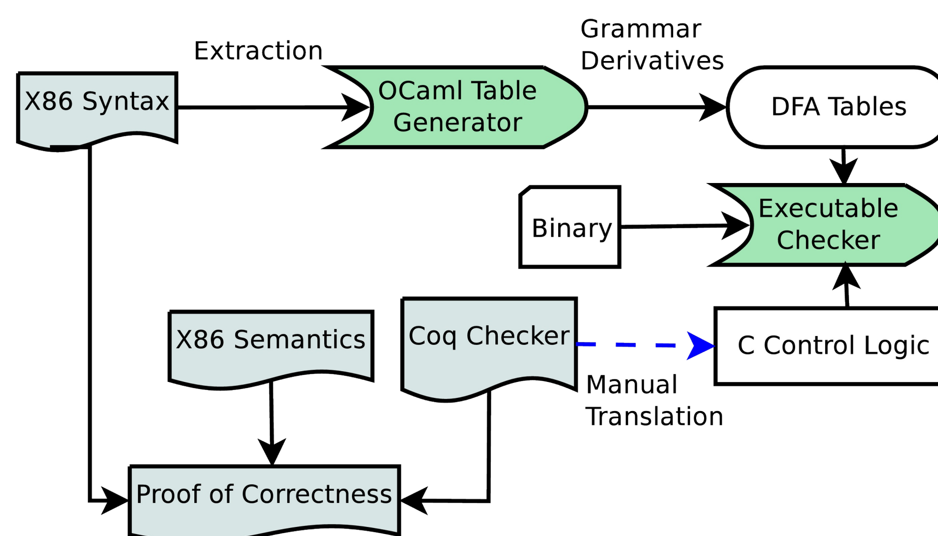
•Compiling from DSLs to a core language simplifies reasoning and allows code reuse
•Use of derivatives with parser combinators allows for natural syntactic reasoning

Model Architecture

Semantic Pathway



Verified Checker



Modular DSLs for Specification of x86

Syntax: Parser Combinators with Derivatives

Datatype for grammars:

```
Inductive grammar : Type → Type
| Char: char → grammar char
| Any: grammar char
| Eps: grammar unit
| Cat:∀T1 T2,grammar T1 → grammar T2 → grammar (T1*T2)
| Void: ∀T, grammar T
| Alt: ∀T, grammar T → grammar T → grammar T
| Star: ∀T, grammar T → grammar (list T)
| Map: ∀T1 T2, (T1 → T2) → grammar T1 → grammar T2
```

Sample Specification:

Concatenation — User Defined Nonterminal — Map: Parse Action

```
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).
```

AST Constructor

Semantics: Monadic Compilation to Low Level Register Transfer Language (RTL)

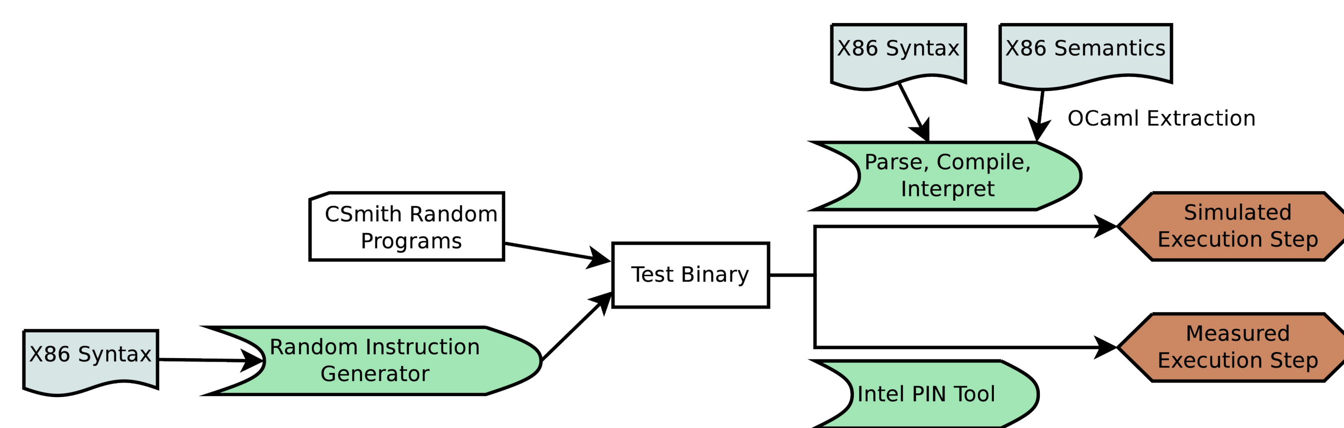
Sample Specification:

Fresh Var Binding — Specialize Memory ops for current

```
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 i
  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;;
  b0 ← test lt zero p0;
  b1 ← test lt zero p1;
  b2 ← test lt zero p2;
  b3 ← arith xor b0 b1;
  b3 ← arith xor up b3;
  b4 ← arith xor b0 b2;
  b4 ← arith and b3 b4;
  set_flag OF b4;;
  ...
```

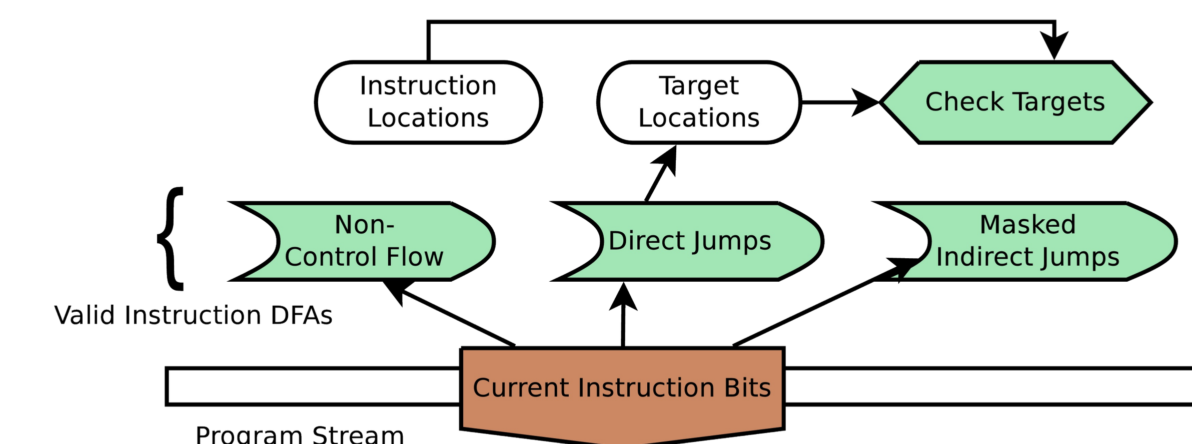
Model Validation

Two Tracks: Simulation and Execution



The Verified Checker

Design of Checker:



Main routine of our NaCl Checker:

```
 1. Bool verifier(DFA *NoControlFlow,
 2.               DFA *DirectJump, DFA *MaskedJump,
 3.               uint8_t *code, uint size)
 4. {
 5.  uint pos = 0, i, saved_pos;
 6.  Bool b = TRUE;
 7.  valid = (uint8_t *)calloc(size,sizeof(uint8_t));
 8.  target = (uint8_t *)calloc(size,sizeof(uint8_t));
 9.
10.  while (pos < size) {
11.    valid[pos] = TRUE;
12.    saved_pos = pos;
13.    if (match(MaskedJump,code,&pos,size)) continue;
14.    if (match(NoControlFlow,code,&pos,size)) continue;
15.    if (match(DirectJump,code,&pos,size) &&
16.        extract(code,saved_pos,pos,target)) continue;
17.    free(target); free(valid);
18.    return FALSE;
19.  }
20.
21.  for (i = 0; i < size; ++i)
22.    b = b && ( !(target[i]) || valid[i] ) &&
23.             ( i & 0x1F || valid[i] );
24.
25.  free(target); free(valid);
26.  return b;
27. }
```

•Transition table for DFAs generated automatically from Coq x86 specifications. C program mechanically follows tables.

DFA match routine:

```
 1. Bool match(DFA *A, uint8_t *code,
 2.            uint *pos, uint size)
 3. {
 4.    uint8_t state = A->start;
 5.    uint off = 0;
 6.
 7.    while (*pos + off < size) {
 8.      state = A->table[state][code[*pos + off]];
 9.      off++;
10.      if (A->rejects[state]) break;
11.      if (A->accepts[state]) {
12.        *pos += off;
13.        return TRUE;
14.      }
15.    }
16.    return FALSE;
17. }
```

Proof of Correctness

•Above C checker was written based on a Coq version

•We show that if the checker written in Coq returns true when run on a program, then that program adheres to the desired security policy

•If a particular DFA returns true, then the corresponding instruction is in the appropriate class

•We say that a machine state is appropriate when:
(1) the data and code segments are disjoint
(2) the DS, SS, and GS segment registers point to the segments they initially did
(3) CS segment register points to initial code segment
(4) the program counter points within the code segment
(5) the original bytes of the program are stored in the code segment

•A state is locally-safe when it is appropriate and the program counter holds an address corresponding to the start of an instruction matched by one of the 3 DFAs

•An appropriate state is k-safe when k>0, and for any s' such that s s', either s' is locally-safe or s' is (k-1)-safe

•We show that if the checker returns true, and a machine is in a locally-safe state, then it is k-safe fo