RockSalt: Better, Faster, Stronger SFI for the x86

Greg Morrisett [*]
greg@eecs.harvard.edu

Gang Tan
gtan@cse.lehigh.edu

Joseph Tassarotti
tassarotti@college.harvard.edu

Jean-Baptiste Tristan
tristan@seas.harvard.edu

Edward Gan
egan@college.harvard.edu

Abstract

Software-based fault isolation (SFI), as used in Google's Native Client (NaCl), relies upon a conceptually simple machine-code analysis to enforce a security policy. But for complicated architectures such as the x86, it is all too easy to get the details of the analysis wrong. We have built a new checker that is smaller, faster, and has a much reduced trusted computing base when compared to Google's original analysis. The key to our approach is automatically generating the bulk of the analysis from a declarative description which we relate to a formal model of a subset of the x86 instruction set architecture. The x86 model, developed in Coq, is of independent interest and should be usable for a wide range of machine-level verification tasks.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms security, verification

Keywords software fault isolation, domain-specific languages

1. Introduction

Native Client (NaCl) is a new service provided by Google's Chrome browser that allows native executable code to be run directly in the context of the browser [37]. To prevent buggy or malicious code from corrupting the browser's state, leaking information, or directly accessing system resources, the NaCl loader checks that the binary code respects a *sandbox* security policy. The sandbox policy is meant to ensure that, when loaded and executed, the untrusted code (a) will only read or write data in specified segments of memory, (b) will only execute code from a specified segment of memory, disjoint from the data segments, (c) will not execute a specific class of instructions (*e.g.*, system calls), and (d) will only communicate with the browser through a well-defined set of entry points.

Ensuring the correctness of the NaCl checker is crucial for preventing vulnerabilities, yet early versions had bugs that attackers could exploit, as demonstrated by a contest that Google ran [25]. A high-level goal of this work is to produce a high-assurance checker for the NaCl sandbox policy. Thus far, we have managed to construct a new NaCl checker for the 32-bit x86 (IA-32) processor (minus floating-point) which we call RockSalt. The RockSalt checker is smaller, marginally faster, and easier to modify than Google's original code. Furthermore, the core of RockSalt is automatically generated from a higher-level specification, and this generator has been proven correct with respect to a model of the x86 using the Coq proof assistant [9].

We are not the first to address assurance for SFI using formal methods. In particular, Zhao *et al*. [38] built a provably correct verifier for a sandbox policy similar to NaCl's. Specifically, building upon a model of the ARM processor in HOL [13], they constructed a program logic and a provably correct verification condition generator, which when coupled with an abstract interpretation, generates proofs that assembly code respects the policy.

Our work has two key differences: First, there is no formal model for the subset of x86 that NaCl supports. Consequently, we have constructed a new model for the x86 in Coq. We believe that this model is an important contribution of our work, as it can be used to validate reasoning about the behavior of x86 machine code in other contexts (*e.g*., for verified compilers).

Second, Zhao *et al*.'s approach takes about 2.5 hours to check a 300 instruction program, whereas RockSalt checks roughly 1M instructions per second. Instead of a general-purpose theorem prover, RockSalt only relies upon a set of tables that encode a deterministic finite-state automaton (DFA) and a few tens of lines of (trusted) `C` code. Consequently, the checker is extremely fast, has a much smaller run-time trusted computing base, and can be easily integrated into the NaCl runtime.

1.1 Overview

This paper has two major parts: the first part describes our model of the x86 in Coq and the second describes the RockSalt NaCl checker and its proof of correctness with respect to the model.

The x86 architecture is notoriously complicated, and our fragment includes a parser for over 130 different instructions with semantic definitions for over 70 instructions[1]. This includes support for operands that include byte and word immediates, registers, and complicated addressing modes (*e.g*., scaled index plus offset). Furthermore, the x86 allows prefix bytes, such as operand size override, locking, and string repeat, that can be combined in many different ways to change the behavior of an instruction. Finally, the instruction set architecture is so complex, that it is unlikely that we can produce a faithful model from documentation, so we must be able to validate our model against implementations.

To address these issues, we have constructed a pair of domain-specific languages (DSLs), inspired by the work on SLED [30] and λ-RTL [29] (as well as more recent work [11, 19]), for specifying

[*] This research was sponsored in part by NSF grants CCF-0915030, CCF-0915157, CNS-0910660, CCF-1149211, AFOSR MURI grant FA9550-09-1-0539, and a gift from Google.

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06…$10.00

[1] Some instructions have numerous encodings. For example, there are fourteen different opcode forms for the `ADC` instruction, but we count this as a single instruction.

$$\begin{array}{lll}
\text{Register} & reg & ::= \ \mathtt{EAX} \mid \mathtt{ECX} \mid \mathtt{EDX} \mid \cdots \\
\text{Segment Reg.} & sreg & ::= \ \mathtt{ES} \mid \mathtt{CS} \mid \mathtt{SS} \mid \cdots \\
\text{Scale} & scale & ::= \ \mathtt{1} \mid \mathtt{2} \mid \mathtt{4} \mid \mathtt{8}
\end{array}$$

Operand

$$\begin{array}{ll}
op \ ::= & int_{32} \mid reg \\
& int_{32} \times \mathtt{option}\, reg \times \mathtt{option}(scale \times reg)
\end{array}$$

Instruction

$$\begin{array}{ll}
i \ ::= & \mathtt{AAA} \mid \mathtt{AAD} \mid \mathtt{AAM} \mid \mathtt{AAS} \mid \mathtt{ADC}(\mathtt{bool} \times op_1 \times op_2) \\
\mid & \mathtt{ADD}(\mathtt{bool} \times op_1 \times op_2) \\
\mid & \mathtt{AND}(\mathtt{bool} \times op_1 \times op_2) \mid \cdots
\end{array}$$

Figure 1. Some Definitions for the x86 Abstract Syntax

the semantics of machine architectures, and have embedded those languages within Coq. Our DSLs are declarative and reasonably high-level, yet we can use them to generate OCaml code that can be run as a simulator. Furthermore, the tools are architecture independent and can thus be re-used to specify the semantics of other machine architectures. For example, one of the undergraduate co-authors constructed a model of the MIPS architecture using our DSLs in just a few days.

The Decoder DSL provides support for specifying the translation from bits to abstract syntax in a declarative fashion. We were able to take the tables from Intel's manual [14] and use them to directly construct patterns for our decoder. Our embedding of the Decoder DSL includes both a denotational and operational semantics, and a proof of adequacy for the two interpretations. We use the denotational semantics for proving important properties about the decode stage of execution, and the operational semantics for execution validation.

The RTL (register transfer list) DSL is a small RISC-like core language parameterized by a notion of machine state. The RTL library includes an executable, small-step operational semantics. Each step in the semantics is specified as a (pure) function from machine states to machine states. We give meaning to x86 instructions by translating their abstract syntax to appropriate sequences of RTL instructions, similar to the way that a modern processor works. Reasoning about RTL is much easier than x86 code, as the number of instructions is smaller and orthogonal.

In what follows, we describe our DSLs and how they were used to construct the x86 model. We also describe our framework for validating the model against existing x86 implementations. We then describe the NaCl sandbox policy in detail, and the new RockSalt checker we have built to enforce it. Next, we describe the actual verification code and the proof of correctness. Finally, we close with a discussion of related work, future directions, and lessons learned.

2. A Coq Model of the x86

Our Coq model of the x86 instruction set architecture has three major stages: (1) a decoder that translates bytes into abstract syntax for instructions, (2) a compiler that translates abstract syntax into sequences of RTL instructions, (3) an interpreter for RTL instructions. The interface between the first two components is the definition of the abstract syntax, which is specified using a set of inductive datatype definitions that are informally sketched in Figure 1.

2.1 The Decoder Specification

The job of the x86 model's decoder is to translate bytes into abstract syntax. We specify the translation using generic grammars constructed in a domain-specific language, which is embedded into Coq. The language lets users specify a pattern and associated semantic actions for transforming input strings to outputs such as abstract syntax. Our pattern language is limited to regular expressions, but the semantic actions are arbitrary Coq functions.

```
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).
```

Figure 2. Parsing Specification for the CALL instruction

Figure 2 gives an example parsing specification we use for the CALL instruction. At a high-level, this grammar specifies four alternatives that can build a CALL instruction. Each case includes a pattern specifying literal sequences of bits (*e.g.*, "1110"), followed by other components like word or modrm2 that are themselves grammars that compute values of an appropriate type. The "@" separates the pattern from a Coq function that can be used to transform the values returned from the pattern match. For example, in the first case, we take the word value and use it to build the abstract syntax for a version of the CALL instruction with an immediate operand.

We chose to specify patterns at the bit-level, instead of the byte-level, because this avoids the need to introduce or reason about shifts and masks in the semantic actions. Furthermore, we were able to take the tables from the Intel IA-32 instruction manual and translate them directly into appropriate patterns.

Our decoding specifications take advantage of Coq's notation mechanism, as well as some derived forms to make the grammar readable, but these are defined in terms of a small set of constructors given by the following type-indexed datatype:

```
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
```

A value of type grammar T represents a relation between lists of chars[2] and semantic values of type T. Alternatively, we can think of the grammar as matching an input string and returning a set of associated semantic values. Formally, the denotation of a grammar is the least relation over strings and values satisfying the following equations:

$$\begin{array}{rcl}
[\![\mathtt{Char}\, c]\!] & = & \{(c :: \mathtt{nil}, c)\} \\
[\![\mathtt{Any}]\!] & = & \bigcup_c \{(c :: \mathtt{nil}, c)\} \\
[\![\mathtt{Eps}]\!] & = & \{(\mathtt{nil}, \mathtt{tt})\} \\
[\![\mathtt{Void}]\!] & = & \emptyset \\
[\![\mathtt{Alt}\, g_1\, g_2]\!] & = & [\![g_1]\!] \cup [\![g_2]\!] \\
[\![\mathtt{Cat}\, g_1\, g_2]\!] & = & \{((s_1 s_2), (v_1, v_2)) \mid (s_i, v_i) \in [\![g_i]\!]\} \\
[\![\mathtt{Map}\, f\, g]\!] & = & \{(s, f(v)) \mid (s, v) \in [\![g]\!]\} \\
[\![\mathtt{Star}\, g]\!] & = & [\![\mathtt{Map}\, (\lambda\_.\, \mathtt{nil})\, \mathtt{Eps}]\!] \cup \\
& & [\![\mathtt{Map}\, (::)\, (\mathtt{Cat}\, g\, (\mathtt{Star}\, g))]\!]
\end{array}$$

Thus, Char c matches strings containing only the character c, and returns that character as the semantic value. Similarly, Any matches a string containing any single character c, and returns c. Eps matches only the empty string and returns tt (Coq's unit).

[2] Grammars are parameterized by the type char.

The grammar `Void` matches no strings and thus returns no values. When g_1 and g_2 are grammars that each return values of type T, then the grammar $\texttt{Alt}\, g_1\, g_2$ matches a string s if either g_1 matches s or g_2 matches s. It returns the union of the values that g_1 and g_2 associate with the string. $\texttt{Cat}\, g_1\, g_2$ matches a string if it can be broken into two pieces that match the sub-grammars. It returns a pair of the values computed by the grammars. `Star` matches zero or more occurrences of a pattern, returning the results as a list.

The last constructor, `Map`, is our constructor for semantic actions. When g is a grammar that returns $\texttt{T}_1$ values, and f is a function of type $\texttt{T}_1 \to \texttt{T}_2$, then $\texttt{Map}\, f\, g$ is the grammar that matches the same set of strings as g, but transforms the outputs from $\texttt{T}_1$ values to $\texttt{T}_2$ values using f. If a grammar forgoes the use of `Map`, then the semantic values represent a parse tree for the input. `Map` makes it possible to incrementally transform the parse tree into alternate semantic values, such as abstract syntax.

As noted above, we use Coq's notation mechanism to make the grammars more readable. In particular, the following table gives some definitions for the notation used here:

$$\begin{array}{rcl|rcl} g_1 \,\texttt{||}\, g_2 & := & \texttt{Alt}\, g_1\, g_2 & g_1 \,\texttt{\$}\, g_2 & := & \texttt{Cat}\, g_1\, g_2 \\ g \,\texttt{@}\, f & := & \texttt{Map}\, f\, g & g_1 \,\texttt{\$\$}\, g_2 & := & (g_1 \,\texttt{\$}\, g_2)\,\texttt{@}\,\texttt{snd} \end{array}$$

We encode the denotational semantics in Coq using an inductively defined predicate, which makes it easy to symbolically reason about grammars. For example, one of our key theorems shows that our top-level grammar, which includes all possible prefixes and all possible integer instructions, is deterministic:

$$(s, v_1) \in [\![\texttt{x86grammar}]\!] \wedge (s, v_2) \in [\![\texttt{x86grammar}]\!] \Longrightarrow v_1 = v_2$$

This helps provide some assurance that in transcribing the grammar from Intel's manual, we have not made a mistake. In fact, when we first tried to prove determinism, we failed because we had flipped a bit in an infrequently used encoding of the `MOV` instruction, causing it to overlap with another instruction.

2.2 The Decoder Implementation

While the denotational specification makes it easy to reason about grammars, it cannot be directly executed. Consequently, we define a parsing function which, when given a record representing a machine state, fetches bytes from the address specified by the program counter and attempts to match them against the grammar and build the appropriate instruction abstract syntax.

Our parsing function is defined by taking the *derivative* of the `x86grammar` with respect to the sequence of bits in each byte, and then checking to see if the resulting grammar accepts the empty string. The notion of derivatives is based on the ideas of Brzozowski [5] and more recently, Owens *et al*. [26] and Might *et al*. [24]. Reasoning about derivatives is much easier in Coq than attempting to transform grammars into the usual graph-based formalisms, as we need not worry about issues such as naming nodes, equivalence on graphs, or induction principles for graphs. Rather, all of our computation and reasoning can be done directly on the algebraic datatype of grammars.

Semantically, the derivative of a grammar g with respect to a character c is the relation:

$$\texttt{deriv}_c\, g = \{(s, v) \mid (c :: s, v) \in [\![g]\!]\}$$

That is, $\texttt{deriv}_c\, g$ matches the tail of any string that starts with c and matches g.

Fortunately, calculating the derivative, including the appropriate transformation on the semantic actions, can be written as a straightforward function:

$$\begin{array}{rcl} \texttt{deriv}_c\,\texttt{Any} & = & \texttt{Map}\,(\lambda\,\_.\,c)\,\texttt{Eps} \\ \texttt{deriv}_c\,(\texttt{Char}\,c) & = & \texttt{Map}\,(\lambda\,\_.\,c)\,\texttt{Eps} \\ \texttt{deriv}_c\,(\texttt{Alt}\,g_1\,g_2) & = & \texttt{Alt}\,(\texttt{deriv}_c\,g_1)\,(\texttt{deriv}_c\,g_2) \\ \texttt{deriv}_c\,(\texttt{Star}\,g) & = & \texttt{Map}\,(::)\,(\texttt{Cat}(\texttt{deriv}_c\,g)\,(\texttt{Star}\,g)) \\ \texttt{deriv}_c\,(\texttt{Cat}\,g_1\,g_2) & = & \texttt{Alt}(\texttt{Cat}\,(\texttt{deriv}_c\,g_1)\,g_2) \\ & & \quad(\texttt{Cat}\,(\texttt{null}\,g_1)\,(\texttt{deriv}_c\,g_2)) \\ \texttt{deriv}_c\,(\texttt{Map}\,f\,g) & = & \texttt{Map}\,f\,(\texttt{deriv}_c\,g) \\ \texttt{deriv}_c\,g & = & \texttt{Void} \qquad\qquad \text{otherwise} \end{array}$$

where $\texttt{null}\, g$ is defined as:

$$\begin{array}{rcl} \texttt{null}\,\texttt{Eps} & = & \texttt{Eps} \\ \texttt{null}\,(\texttt{Alt}\,g_1\,g_2) & = & \texttt{Alt}\,(\texttt{null}\,g_1)\,(\texttt{null}\,g_2) \\ \texttt{null}\,(\texttt{Cat}\,g_1\,g_2) & = & \texttt{Cat}\,(\texttt{null}\,g_1)\,(\texttt{null}\,g_2) \\ \texttt{null}\,(\texttt{Star}\,g) & = & \texttt{Map}\,(\lambda\,\_.\,\texttt{nil})\,\texttt{Eps} \\ \texttt{null}\,(\texttt{Map}\,f\,g) & = & \texttt{Map}\,f\,(\texttt{null}\,g) \\ \texttt{null}\,g & = & \texttt{Void} \qquad \text{otherwise} \end{array}$$

Effectively, `deriv` strips off a leading pattern that matches c, and adjusts the grammar with a `Map` so that it continues to calculate the same set of values. If the grammar cannot match a string that starts with c, then the resulting grammar is `Void`. The `null` function returns a grammar equivalent to `Eps` when its argument accepts the empty string, and `Void` otherwise. It is used to calculate the derivative of a `Cat`, which is simply the chain-rule for derivatives.

Once we calculate the iterated derivative of the grammar with respect to a string of bits, we can extract the set of related semantic values by running the `extract` function, which returns those semantic values associated with the empty string:

$$\begin{array}{rcl} \texttt{extract Eps} & = & \{\texttt{tt}\} \\ \texttt{extract}\;(\texttt{Star}\,g) & = & \{\texttt{nil}\} \\ \texttt{extract}\;(\texttt{Alt}\,g_1\,g_2) & = & (\texttt{extract}\;g_1) \cup (\texttt{extract}\;g_2) \\ \texttt{extract}\;(\texttt{Cat}\,g_1\,g_2) & = & \{(v_1, v_2) \mid v_i \in \texttt{extract}\;g_i\} \\ \texttt{extract}\;(\texttt{Map}\,f\,g) & = & \{f(v) \mid v \in \texttt{extract}\;g\} \\ \texttt{extract}\;g & = & \emptyset \qquad\qquad\qquad \text{otherwise} \end{array}$$

To be reasonably efficient, it is important that we optimize the grammar as we calculate derivatives. In particular, when we build a grammar, we always use a set of "smart" constructors, which are functions that perform local reductions, including:

$$\begin{array}{rcl|rcl} \texttt{Cat}\;g\;\texttt{Eps} & \to & g & \texttt{Cat Eps}\;g & \to & g \\ \texttt{Cat}\;g\;\texttt{Void} & \to & \texttt{Void} & \texttt{Cat Void}\;g & \to & \texttt{Void} \\ \texttt{Alt}\;g\;\texttt{Void} & \to & g & \texttt{Alt Void}\;g & \to & g \\ \texttt{Star}\;(\texttt{Star}\,g) & \to & \texttt{Star}\;g & \texttt{Alt}\;g\;g & \to & g \end{array}$$

Of course, the optimizations must add appropriate `Map`s to adjust the semantic actions. Proving the optimizations correct is an easy exercise using the denotational semantics. Unfortunately, the last of these optimizations ($\texttt{Alt}\, g\, g \to g$) cannot be directly implemented as it demands a decidable notion of equality for grammars, yet our grammars include arbitrary Coq functions (and types). To work around these problems, we first translate grammars to an internal form, where all types and functions are replaced with a name that we can easily compare. An environment is used to track the mapping from names back to their definitions, and is consulted in the `extract` function to build appropriate semantic values.

In the end, we get a reasonably efficient parser that we can extract to executable OCaml code. Furthermore, we prove that the parser, when given a grammar g and string s, produces a (finite) set of values $\{v_1, \cdots, v_n\}$ such that $(s, v_i) \in [\![g]\!]$. Since we have proven that our instruction grammar is deterministic, we know that in fact, we will get out at most one instruction for each sequence of bytes that we feed to the parser.

Finally, we note that calculating derivatives in this fashion corresponds to a lazy, on-line construction of a deterministic finite-state transducer. Our efficient NaCl checker, described in Section 3

Machine locations

$loc ::= \texttt{PC} \mid \texttt{EAX} \mid \cdots \mid \texttt{CF} \mid \cdots \mid \texttt{SS} \mid \cdots$

Local variables

$x, y, z \quad \in \text{identifier}$

Arithmetic operators

$op ::= \texttt{add} \mid \texttt{xor} \mid \texttt{shl} \mid \cdots$

Comparison operators

$cmp ::= \texttt{lt} \mid \texttt{eq} \mid \texttt{gt}$

RTL instructions

$$\begin{aligned} rt ::= {} & x \;:=\; y \; op \; z \mid x \;:=\; y \; cmp \; z \\ & \mid x \;:=\; imm \mid x \;:=\; \texttt{load} \; loc \\ & \mid \texttt{store} \; loc \; x \mid x \;:=\; \texttt{Mem}[y] \\ & \mid \texttt{Mem}[x] \;:=\; y \mid x \;:=\; \texttt{choose} \mid \cdots \end{aligned}$$

Figure 3. The RTL Language

is built from a deterministic finite-state automaton (DFA) generated off-line, re-using the definitions for the grammars, derivatives, *etc.* in the parsing library.

2.3 Translation To RTL

After parsing bytes into abstract syntax, we translate the corresponding instruction into a sequence of RTL (register transfer list) operations. RTL is a small RISC-like language for computing with bit-vectors. The language abstracts over an architecture's definition of machine state, which in the case of the x86 includes the various kinds of registers shown in Figure 1 as well as a memory, represented as a finite map from addresses to bytes. Internally, the language supports a countably infinite supply of local variables that can be used to hold intermediate bit-vector values.

The RTL instruction set is sketched in Figure 3 and includes standard arithmetic, logic, and comparison operations for bit vectors; operations to sign/zero-extend and truncate bit vectors; an operation to load an immediate value into a local variable; operations to load/store values in local variables from/to registers; operations to load and store bytes into memory; and a special operation for non-deterministically choosing a bit-vector value of a particular size. We use dependent types to embed the language into Coq and ensure that only bit-vectors of the appropriate size are used in instructions.

For each x86 constructor, we define a function that translates the abstract syntax into a sequence of RTL instructions. The translation is encapsulated in a monad that takes care of allocating fresh local variables, and that allows us to build higher-level operations out of sequences of RTL commands.

Figure 4 presents an excerpt of the translation of the `ADD` instruction. The `ADD` constructor is parameterized by a prefix record, a boolean mode, and two operands. The prefix record records modifiers including any segment, operand, or address override. The boolean mode is set when the default operand size is to be used (*i.e.*, 32-bits) and cleared when the operand size is set to a byte. The operands can be registers, immediate values, or effective addresses.

The first two local definitions specialize the load and store RTL to the given prefix and mode. The third definition selects the appropriate segment. Next, we load constant expressions 0 and 1 (of bit-size 1) into local variables `zero` and `up`. Then we fetch the bit-vector values from the operands and store them in local variables `p0` and `p1`. At this point, we actually add the two bit-vectors and place the result in local variable `p2`. Then we update the machine state at the location specified by the first operand. Afterwards, we set the various flag registers to hold the appropriate

```
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;;
  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;;
  ...
```

Figure 4. Translation Specification for the ADD instruction

1-bit value based on the outcome of the operation. Here, we have only shown the code needed to set the overflow (`OF`) flag.

Occasionally, the effect of an operation, particularly on flags, is under-specified or unclear. To over-approximate the set of possible behaviors, we use the `choose` operation, which non-deterministically selects a bit-vector value and stores this value in the appropriate location.

2.4 The RTL Interpreter

Once we have defined our decoder and translation to RTL, we need only give a semantics to the RTL instructions to complete the x86 model. One option would be to use a small-step operational semantics for modeling RTL execution, encoded as an inductive predicate. However, this would prevent us from extracting an executable interpreter which we need for validation.

Instead, we encode a step in the semantics as a function from RTL machine states to RTL machine states. RTL machine states record the values of the various x86 locations, the memory, and the values of the local variables. To support the non-determinism in the `choose` operation, the RTL machine state includes a stream of bits that serves as an *oracle*. Whenever we need to `choose` a new value, we simply pull bits from the oracle stream. Of course, when reasoning about the behavior of instructions, we must consider all possible oracle streams. This is a standard trick for turning a non-deterministic step relation into a function.

Most of the operations are simple bit-vector computations for which we use the CompCert integer bit-vector library [18]. Consequently, the definition of the interpreter is fairly straightforward and extracts to reasonable OCaml code that we can use for testing.

2.5 Model Validation

Any model of the x86 is complicated enough that it undoubtably has bugs. The only way we can gain any confidence is to test it against real x86 processors (and even they have bugs!). As described above, we have carefully engineered our model so that we can extract an executable OCaml simulator from our Coq definitions. We use this simulator to compare against an actual x86 processor.

One challenge in validating the simulator is extracting the machine state from the real processor. We use Intel's Pin tool [20] to insert dynamic instrumentation into a binary. The instrumentation dumps the values of the registers to a file after each instruction, and the values in memory after each system call. We then take the original binary and run it through our OCaml simulator, comparing the values of the registers after the RTL sequence for an instruction has been generated and interpreted. Unfortunately, this procedure sometimes generates false positives because of our occasional use of the oracle to handle undefined or under specified behaviors.

We use two different techniques to generate test cases to exercise the simulator. First, we generate small, random `C` programs using `Csmith` [36] and compile them using `GCC`. This technique proved useful early in the development stage, especially to test standard instructions. In this way, we simulated and verified over 10 million instruction instances in about 60 hours on an 8 core intel Xeon running at 2.6Ghz.

However, this technique does not exercise instructions that are avoided by compilers, and even some common instructions have encodings that are rarely emitted by assemblers. For example, our previously discussed bug in the encoding of the `MOV` instruction was not uncovered by such testing because it falls in this category.

A more thorough technique is to fuzz test our simulator by generating random sequences of bytes, which has previously proved effective in debugging CPU emulators [21]. Using our generative grammar, we randomly produce byte sequences that correspond to instructions we have specified. This lets us exercise unusual forms of all the instructions we define. For instance, an instruction like add with carry comes in fourteen different flavors, depending on the width and types of the operands, whether immediates are sign-extended, etc. Fuzzing such an instruction guarantees with some probability that all of these forms will be exercised.

3. The RockSalt NaCl Checker

Recall that our high level goal is to produce a checker for Native Client, which when given an x86 binary image, returns true only when the image respects the sandbox policy: when executed, the code will only read/write data from specified contiguous segments in memory, will not directly execute a particular set of instructions (*e.g.*, system calls), and will only transfer control within its own image or to a specified set of entry points in the NaCl run-time.

The 32-bit x86 version of NaCl takes advantage of the segment registers to enforce most aspects of this policy. In particular, by setting the `CS` (code), `DS` (data), `SS` (stack), and `GS` (thread-local) segment registers appropriately, the machine itself will ensure that data reads and writes are contained in the data segments, and that jumps are contained within the code segment. However, we must make sure that the untrusted instructions do not change the values of the segment registers, nor override the segments inappropriately.

At first glance, it appears sufficient to simply parse the binary into a sequence of instructions, and check that each instruction in the sequence preserves the values of the segment registers and does not override the segment registers with a prefix. Unfortunately, this simple strategy does not suffice. The problem is that, since the x86 has variable length instructions, we must not only consider the parse starting at the first byte, but all possible parses of the image. While most programs will respect the initial parse, a malicious or buggy program may not. For example, in a program that has a buffer overrun, a return address may be overwritten by a value that points into the middle of an instruction from the original parse.

To avoid this problem, NaCl provides a modified compiler that rewrites code to respect a stronger *alignment policy*, following the ideas of McCamant and Morrisett [22]. The alignment policy requires that all computed jumps (*i.e.*, jumps through a register) are aligned on a 32-byte boundary. This is ensured by inserting code to mask the target address with an appropriate constant, and by inserting no-ops so that potential jump targets are suitably aligned. In more detail, the aligned, sandbox policy requires that:

1. Starting with the first byte, the image parses into a legal sequence of instructions that preserve the segment registers;
2. Every 32^{nd} byte is the beginning of an instruction in our parse;
3. Every indirect jump through a register r is immediately preceded by an instruction that masks r so that it is 32-byte aligned;
4. The masking operation and jump are both contained within a 32-byte-aligned block of instructions;
5. Each direct jump targets the beginning of an instruction and that instruction is not an indirect jump.

Requirements 4 and 5 are needed to ensure that the code cannot jump over the masking operation that protects an indirect jump.

3.1 Constructing a NaCl Checker

Google's NaCl checker is a hand-written `C` program that is intended to enforce the aligned, sandbox policy. Their checker partially decodes the binary, looking at fields such as the op-codes and mod/rm bits to determine whether the instruction is legal, and how long it is. Two auxiliary data structures are used: One is a bit-map that records which addresses are the starts of instructions. Each time an instruction is parsed, the corresponding bit for the address of the first byte is set. The other is an array of addresses for forward, direct jumps. After checking that the instructions are legal, the bit-map is checked to ensure that every 32nd byte is the start of an instruction. Then, the array of direct jump targets is checked to make sure they are valid according to the policy above.

There are two disadvantages with Google's checker: it is difficult to reason about because it is somewhat large (about 600 statements of code)[3] and the process of partial decoding is intertwined with policy enforcement. In particular, it is difficult to tell what instructions are supported and with what prefixes, and even more difficult to gain assurance that the resulting code enforces the appropriate sandbox policy. Furthermore, it is difficult to modify the code to *e.g.*, add new kinds of safe instructions or combinations of prefixes.

In contrast, the RockSalt checker we constructed and verified is relatively small, consisting of only about 80 lines of Coq code. This is because the checker uses table-driven DFA matching to handle the aspects of decoding, following an idea first proposed by Seaborn [33]. The basic idea is to break all instructions into four categories: (1) those that perform no control-flow, and are easily seen as okay; (2) those that perform a direct jump—we must check that the target is a valid instruction; (3) those that perform an indirect jump—we must check that the destination is appropriately masked; and (4) those instructions that should be rejected. Each of these classes, except the third one, can be described using a simple regular expression. The third class can be captured by a regular expression if we make the restriction that the masking operation must occur directly before the jump, which in practice is what the NaCl compiler does.

It is possible to extract OCaml code from our Coq definitions and use that as the core of the checker, but we elected to manually translate the code into `C` so that it would more easily integrate into the NaCl run-time. This avoids adding the OCaml compiler and run-time system to the trusted computing base, at the risk that our translation to `C` may have introduced an error. However, at under 100 lines of `C` code, we felt that this was a reasonable risk, since the vast majority of the information is contained in the DFA tables which are automatically generated and proven correct. Of course, one could try to use a verification tool, such as Frama-C/WP [10]

[3] To be fair, this includes CPU identification and support for floating-point and other instructions that we do not yet handle.

```
 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. }
```

Figure 5. Main Routine of our NaCl Checker

```
Bool match(DFA *A, uint8_t *code,
           uint *pos, uint size)
{
   uint8_t state = A->start;
   uint off = 0;

   while (*pos + off < size) {
     state = A->table[state][code[*pos + off]];
     off++;
     if (A->rejects[state]) break;
     if (A->accepts[state]) {
       *pos += off;
       return TRUE;
     }
   }
   return FALSE;
}
```

Figure 6. The DFA match routine

or VCC [8], to prove the correctness of this version, in which case the functional code in Coq could serve as a specification.

Figure 5 shows the C code for the high-level verifier routine. This function relies upon two sub-routines, `match` and `extract` that we will explain later, but intuitively handle the aspects of decoding. Like Google's checker, the routine uses two auxiliary arrays: the `valid` array records those addresses in the code that are valid jump destinations, whereas the `target` array records those addresses that are jumped to by some direct control-flow operation. We used byte arrays instead of bit arrays to avoid having to reason about shifts and masks to read/write bits.

The main loop (line 10) iterates through the bytes in the `code` starting at position 0. This position is marked as `valid` and then we attempt to `match` the bytes at the current position against three patterns. The first pattern, `MaskedJump`, matches only when the bytes specify a mask of register r followed immediately by an indirect jump or call through r. Note that a successful match increments the position by `size` which records the length of the instruction(s), whereas a failure to match leaves the position unmodified. The second pattern, `NoControlFlow`, matches only when the bytes specify a legal NaCl instruction that does not affect control flow (*e.g.*, an arithmetic instruction). The third pattern, `DirectJump` matches only when the bytes specify a direct `JMP`, `Jcc` or `CALL` instruction. The routine `extract` then extracts the destination address of the jump, and marks that address in the `target` array. If none of these cases match, then the checker returns `FALSE` indicating that an illegal sequence of bytes was found in the code.

After the main loop terminates, we must check that (a) if an address is the target of a direct jump, then that address is the beginning of an instruction in our parse (line 22), and (b) if an address is aligned on a 32-byte boundary, then that address is the beginning of an instruction in our parse (line 23).

The process of matching a sequence of bytes against a pattern is handled by the routine `match` which is shown in Figure 6. The function simply executes the transitions of a DFA using the bytes at the current position in the code. The DFA has four fields: a starting state, a boolean array of accepting states, a boolean array of rejecting states, and a transition table that maps a state and byte to a new state.

3.2 DFA Generation

What we have yet to show are the definitions of the DFAs for the `MaskedJump`, `NoControlFlow`, and `DirectJump` patterns, and the correctness of our checker hinges crucially upon these definitions. These are generated from within Coq using higher-level specifications. In particular, for each of the patterns, we specify a grammar re-using the parsing DSL described in Section 2.1, and then compile that grammar to appropriate DFA tables. For example, the grammar for a `MaskedJump` is given below:

```
Definition nacl_MASK_p (r: register) :=
  "1000" $$ "0011" $$ "11" $$ "100"
  $$ bitslist (register_to_bools r)
  $  bitslist (int_to_bools safeMask).

Definition nacl_JMP_p  (r: register) :=
  "1111" $$ "1111" $$ "11" $$ "100"
  $$ bitslist (register_to_bools r).

Definition nacl_CALL_p (r: register) :=
  "1111" $$ "1111" $$ "11" $$ "010"
  $$ bitslist (register_to_bools r).

Definition nacljmp_p (r: register) :=
  nacl_MASK_p r $ (nacl_JMP_p r || nacl_CALL_p r).

Definition nacljmp_mask :=
  nacljmp_p EAX || nacljmp_p ECX || nacljmp_p EDX ||
  nacljmp_p EBX || nacljmp_p EBP || nacljmp_p ESI ||
  nacljmp_p EDI.
```

The `nacl_MASK_p` function takes a register name and generates a pattern for an "`AND` r, `safeMask`" instruction. The `nacl_JMP_p` and `nacl_CALL_p` functions take a register and generate patterns for a jump or call instruction (respectively) through that register. Thus, `nacljmp_mask` and the top-level grammar match any combination of a mask and jump through the same register (excluding `ESP`).

We compile grammars to DFAs from within Coq as follows: First, we strip off the semantic actions from the grammars so that we are left with a regular expression r_0. This regular expression corresponds to the starting state of the DFA. We use the `null` routine to check if this is an accepting state and a similar routine to check for rejection, and record this in a table. We then calculate the derivative of r_0 with respect to all 256 possible input bytes. This yields a set of regular expressions $\{r_1, r_2, \cdots, r_n\}$. Each r_i corresponds to a state in the DFA that is reachable from r_0. We assign each regular expression a state, and record whether that state is an accepting or rejecting state. We continue calculating derivatives of each of the r_i with respect to all possible inputs until we no longer create a new regular expression. The fact that there are a finite number of unique derivatives (up to the reductions performed by our smart constructors) was proven by Brzozowski [5] so we are ensured that the procedure terminates.

In practice, calculating a DFA in this fashion is almost as good as the usual construction [26], but avoids the need to formalize and reason about graphs. The degree to which we simplify regular expressions as we calculate derivatives determines how few states are left in the resulting DFA. In our case, the number of states is small enough (61 for the largest DFA) that we do not need to worry about further minimization.

3.3 Testing the C Checker

In the following section, we discuss the formal proof of correctness for the Coq version of the RockSalt checker. But as noted above, in practice we expect to use the `C` version, partially shown in figures 5 and 6. Although this code is a rather direct translation from the Coq code, to gain further assurance, we did extensive testing, comparing both positive and negative examples against Google's original checker.

For testing purposes, the `ncval` (Native Client Validator) command line tool was modified so that our verification routine can be used instead of Google's. We ensured that both verifiers reject a set of hand-crafted unsafe programs, and we also ensured that they both accept a set of benchmark programs once processed by the NaCl version of GCC which inserts appropriate no-ops and mask instructions. To work around the lack of floating-point support in our checker, we use the "-msoft-float" flag so that GCC avoids generating floating-point instructions. The benchmark programs were drawn from the same set as used in CompCert [18] and include an implementation of AES, SHA1, a virtual machine, fractal computation, a Perl interpreter, and 16 other programs representing more than 4,000 lines of code. We also used Csmith [36] to automatically generate `C` programs, and compiled them with NaCl's version of GCC. We then verified that our driver and Google's always agreed on a program's safety. Using this method we have verified over two thousand small C programs.

Finally, we measured the time it takes to check binaries using both our `C` checker and Google's original code. For the small benchmarks mentioned above, there was no measurable difference in checking times. However, on an artificially generated C program of about 200,000 lines of code, running on a 2.6 GHz Intel Xeon core, Google's checker took 0.90 seconds and our checker took 0.24 seconds (averaging over one hundred runs). Consequently, we believe that RockSalt is competitive with Google's approach.

4. Proof of Correctness for the Checker

After building and testing the checker, we wanted to prove its correctness with respect to the sandbox policy. That is, we wanted a proof that if the checker returns `TRUE` for a given input binary, and if that binary is loaded and executed in an appropriate environment (in particular, where the code and data segments are disjoint), then executing the binary would ensure that only the prescribed data segments are read and written, and control only transfers within the prescribed code segment.

At a high-level, our proof shows that at every step of the program execution, the values of the segment registers are the same as those in the initial state, and furthermore, that the bytes that make up the code-segment are the same bytes that were analyzed by the checker. These invariants are sufficient to show NaCl's sandbox policy is not violated. Furthermore, the checker should have ruled out system calls and other instructions that are not allowed. But of course, formalizing this argument requires a much more detailed set of invariants that connect the matching work done in the checker to the semantics, along with the issues of alignment, masking, and jump-destination checks.

We begin by defining the notion of an *appropriate* machine state:

DEFINITION 1. *A machine state is* appropriate *when:*

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.*

Appropriateness captures the key data invariants that we need to maintain throughout execution of the program. We augment these data invariants with a predicate on the program counter to reach the definition of a *locally-safe* machine state:

DEFINITION 2. *A machine state is* locally-safe *when it is appropriate and the program counter holds an address corresponding to the start of an instruction that was matched by the* `verify` *process using one of the three generated DFAs.*

In other words, for a locally safe state, the `pc` is marked as `valid`.

We would like to argue that, starting from a locally-safe state, we can always execute an instruction and end up in a locally-safe state. This would imply that the segment registers have not changed, that the code has not changed, that any read or write done by the instruction would be limited to the original data segments, and that control remains within the original code segment.

Alas, we do not immediately reach a locally-safe state after executing one instruction. The problem is that our `MaskedJump` DFA operates over two instructions (the mask of the register, followed by the indirect jump). Thus, we introduce the notion of a *k-safe* state:

DEFINITION 3. *An appropriate state s is k-safe when $k > 0$ and, for any s' such that $s \longrightarrow s'$, either s' is locally-safe or s' is $(k-1)$-safe.*

With the definitions given above, it suffices to show that if a state is locally-safe, then it is also k-safe for some k (and in fact, k is either 1 or 2). Indeed, each locally-safe state s should be k-safe for some k: if $s \longrightarrow s'$ then either s' is locally-safe or we executed the mask of a `MaskedJump` and we should be in an appropriate state, ready to execute a branch instruction that will target a masked (and therefore valid) address. Then, assuming the computation starts in a locally-safe state (*e.g.*, with the `pc` at any valid address), it is easy to see that the code cannot step to a state where the segment registers have changed, or the bytes in the code segment have changed.

THEOREM 1. *If s is locally-safe, then it is also k-safe for some k.*

Since a locally-safe instruction has a program counter drawn from the set of `valid` instructions, and since the verifier did not return `FALSE`, we can conclude that a prefix of the bytes starting at this address matches one of the three DFAs. We must then argue that for each class of instructions that match the DFAs, after executing the instruction, we either end up in a locally-safe state or else after executing one more instruction, end up in a locally safe-state.

In the Section 4.1, we sketch the connection we formalized between the `DFA`s and a set of inversion principles that characterize the possible instructions that they can match. These principles allow us to do a case analysis on a subset of the possible instructions. For example, in the case that the `MaskedJump` DFA matches, we know that the bytes referenced by the program counter must decode into a masking operation on some register r, followed by bytes that decode into a jump or call to register r. The proof proceeds by case analysis for each of the three `DFAs` utilizing these inversion principles.

The easiest (though largest) case is when the `NoControlFlow` DFA has the successful match. We prove three properties for each non-control-flow instruction I that the inversion principle gives us:

(1) executing I does not modify segment registers;

(2) executing I modifies only the data segments' memory;

(3) after executing I, the new program counter is equal to the old program counter plus the length of I.

For the most part, arguing these cases is simple: For the first property, we simply iterate over the generated list of `RTLs` for I and ensure there are no writes to the segment registers. The second property follows from the inversion principles which forbid the use of a segment override prefix, and the third property follows from the semantics of non-control-flow instructions. From these three facts, it follows that after executing the instruction, we are immediately in a locally-safe state. That is, the original state was 1-safe.

In the case where I was matched by `DirectJump`, we must argue that the final loop in `verify` ensures that the target of the jump is `valid`. Of course, we must also show that the segment registers are preserved, the code is preserved, *etc*. But then we can again argue that the original state was 1-safe.

For the `MaskedJump` case, we must argue that the state is 2-safe. The inversion principle for the DFA restricts the first instruction to an `AND` of a particular register r with a constant that ensures after the step, the value of r is aligned on a 32-byte boundary, the segments are preserved, and the `pc` points to bytes within the code segment that decode into either a jump or call through r. We then argue that this state is 1-safe. Since the destination of the jump or call is 32-byte aligned, the final loop of the verifier has checked that this address is `valid`. Consequently, it is easy to show that we end up in a locally-safe state.

4.1 Inverting the DFAs

A critical piece in our proof of correctness is the relationship between the DFAs generated from the `NoControlFlow`, `DirectJump`, and `MaskedJump` regular expressions and our semantics for machine instructions. We sketch the key results that we have proven here.

One theorem specifies the connection between a regular expression, the DFA it generates, and the `match` procedure:

THEOREM 2. *If r is a regular expression, and D is the DFA generated from that regular expression, then executing* `match` *on D with a sequence of bytes $b_1, \ldots, b_n$ will return* `true` *if there is some $j \leq n$ such that the string $b_1, \ldots, b_j$ is in the denotation of r.*

The theorem requires proving that our DFA construction process, where we iteratively calculate all derivatives, produces a well-formed DFA with respect to r. Here, a well-formed DFA basically provides a mapping from states to derivatives of r that respect certain closure properties. Fortunately, the algebraic construction of the DFA makes proving this result relatively straightforward. The theorem also requires showing that running `match` on D with $b_1, \ldots, b_n$ is correct which entails, among other things, showing that the array accesses are in bounds, and that when we return `TRUE`, we are in a state that corresponds to the derivative of the regular expression with respect to the string $b_1, \ldots, b_j$.

Another key set of lemmas show that the languages accepted by the regular expressions are subsets of the languages accepted by our `x86grammar`. Additionally, we must prove an inversion principle for each regular expression that characterizes the possible abstract syntax we get when we run the semantics on the bytes. For example, we must show that `DirectJump` only matches bytes that when parsed, produce either (near) `JMP`, `Jcc`, or `CALL` instructions with an immediate operand. Fortunately, proving the language containment property and inversion principles is simple to do using the denotational semantics for grammars.

One of the most difficult properties to prove about the decoder was the uniqueness of parsing. In particular, we needed to show that each bit pattern corresponded to at most one instruction, and no instruction's bit pattern was a prefix of another instruction's bit pattern—*i.e.*, that our `x86grammar` was unambiguous. A naive approach, where we simply explore all possible bit patterns is obviously intractable, when there are instructions up to 15 bytes long. Another approach is to construct a DFA for the grammar and then show that each accepting state has at most one semantic value associated with it. While this is possible, the challenge is getting Coq to symbolically evaluate the DFA construction and reduce the semantic actions in a reasonable amount of time[4].

Consequently, we constructed a simple procedure that checks whether the intersection of two grammars is empty. The procedure, which only succeeds on star-free grammars (stripped of their semantic actions) works by generalizing the notion of a derivative from characters to star-free regular expressions:

$$\begin{array}{rcl}
\texttt{Deriv}\ g\ \texttt{Eps} & = & g \\
\texttt{Deriv}\ g\ (\texttt{Char}\ c) & = & \texttt{deriv}_c\ g \\
\texttt{Deriv}\ g\ \texttt{Any} & = & \texttt{DrvAny}\ g \\
\texttt{Deriv}\ g\ \texttt{Void} & = & \texttt{Void} \\
\texttt{Deriv}\ g\ (\texttt{Alt}\ g_1\ g_2) & = & \texttt{Alt}\ (\texttt{Deriv}\ g\ g_1)\ (\texttt{Deriv}\ g\ g_2) \\
\texttt{Deriv}\ g\ (\texttt{Cat}\ g_1\ g_2) & = & \texttt{Deriv}\ (\texttt{Deriv}\ g\ g_1)\ g_2
\end{array}$$

where

$$\begin{array}{rcl}
\texttt{DrvAny}\ \texttt{Any} & = & \texttt{Eps} \\
\texttt{DrvAny}\ (\texttt{Char}\ c) & = & \texttt{Eps} \\
\texttt{DrvAny}\ \texttt{Eps} & = & \texttt{Void} \\
\texttt{DrvAny}\ \texttt{Void} & = & \texttt{Void} \\
\texttt{DrvAny}\ (\texttt{Alt}\ g_1\ g_2) & = & \texttt{Alt}\ (\texttt{DrvAny}\ g_1)\ (\texttt{DrvAny}\ g_2) \\
\texttt{DrvAny}\ (\texttt{Cat}\ g_1\ g_2) & = & \texttt{Alt}\ (\texttt{Cat}\ (\texttt{DrvAny}\ g_1)\ g_2) \\
 & & \quad (\texttt{Cat}\ (\texttt{null}\ g_1)\ (\texttt{DrvAny}\ g_2))
\end{array}$$

When it is defined, it is easy to show that:

$$\texttt{Deriv}\ g_1\ g_2 = \{s_2 \mid \exists s_1 . s_1 \in g_2 \wedge s_1 s_2 \in g_1\}$$

and thus, when $\texttt{Deriv}\ g_1\ g_2 \rightarrow \texttt{Void}$, we can conclude that there is no string in the intersection of the domains of g_1 and g_2, and furthermore, g_2's strings are not a prefix of those in g_1. This allowed us to easily prove (through Coq's symbolic evaluation) that the `x86grammar` is unambiguous: We simply recursively descend into the grammar, and each time we encounter an `Alt`, check that the intersection of the two sub-grammars is empty.

5. Related Work

With the growing interest in verification of software tools, formal models of processors that support machine-checked proofs have become a hot topic. Often, these models have limitations, not because of any inherent design flaw, but rather because they are meant only to prove specific properties. For example, work on formal verification of compilers [6, 18] only needs to consider the subset of the instructions that compilers use. Moreover, these compilers emit assembly instructions and do not prove semantics preservation all the way down to machine code, so their model leaves out the tricky problem of decoding. The same kinds of limitations exist for the processor models used in the formal verification of operating systems [7]. Some projects focus on one specific part of the model, for

[4] Recall that that the DFAs generated for the NaCl checker strip the semantic actions, so they do not need to worry about reducing semantic actions.

instance the media instructions [16], and some others [22] model just a few instructions mostly as a proof of concept. Even though we are focused here on NaCl verification, our long term goal is to develop a general model of the x86 so we have tried hard to achieve a more open and scalable design.

There are several projects focused on the development of general formal models of processors. Some projects have considered the formalizations of RISC processors [2, 13, 23]. As noted, developing a formal model for the x86 poses many new problems, partly because decoding is significantly more complex, but also for the definition and validation of such a vast number of instructions (over 1,000) with so many variations, from addressing modes to prefixes.

One model close in spirit to our own is the Y86 formalization in ACL2 by Ray [31]. Like our model, Ray's provides an executable simulator. However, the Y86 is a much smaller fragment (about 30 instructions), and has a much simpler instruction encoding (*e.g.*, no prefixes).

Perhaps the closest related research project, and the one from which we took much inspiration in our design, is the work on modeling x86 multi-processor memory models [27, 32, 34]. This work comes with a formal model of about 20 instructions, and we borrowed many of the ideas, such as the use of high-level grammars for specifying the decoder. However, their focus was on issues of non-determinism where it is seemingly more natural to use predicates to describe the possible behaviors of programs. The price paid is that validation requires symbolic evaluation and theorem proving to compare abstract machine states to concrete ones. Although this was largely automated, we believe that our functional approach provides a more scalable way to test the model. Indeed, we have been able to run three orders of magnitude more tests. On the other hand, it remains to be seen how effective our approach will be when we add support for concurrency.

Our decoder, formalized in Coq, uses parsers generated from regular expressions using the idea of derivatives. Others have formalized derivative-based regular expression *matching* [3] but not parsing. However, more general parser generators for algorithms such as SLR and LR have recently been formalized [4, 15].

The original idea for Software-based fault isolation (SFI) was introduced by Wahbe *et. al*. [35] in the context of a RISC machine. This work used an invariant on dedicated registers to ensure that all reads, writes, and jumps were appropriately isolated. Of course, parsing was not a problem because instructions had a uniform length. As noted earlier, McCamant and Morrisett [22] introduced the idea of the alignment constraint to handle variable-length instruction sets. In that paper, they formalized a small subset of the x86 (7 instructions) using ACL2 and proved that their high-level invariants were respected by those instructions, but did not prove the correctness of their checker. In fact, even with the small number of instructions, Kroll and Dean found a number of bugs in the decoder [17], which reinforces our argument that one should be wary of a trusted decoder or disassembler.

Pilkiewicz [28] developed a formally verified SFI checker in Coq for a simple assembly language.

There has been much subsequent work on stronger policies than SFI, including CFI [1] and XFI [12]. Some of this work has been formalized, but typically for RISC machines and in a context where decoding is ignored.

6. Future work & Conclusions

We have presented a formal model for a significant subset of the x86, and a new formally verified checker for Native Client called RockSalt. The primary challenge in this work was building a model for an architecture as complicated as the x86. Although we only managed to model a small subset, we believe that the design is relatively robust thanks to our ability to extract and test executable code. The experience in using the model to reason about a simple but real policy such as NaCl's sandbox, provides some assurance that the model will be useful for reasoning in other contexts.

6.1 Future Work

As explained before, our x86 model is far from complete. We do not yet handle floating-point instructions, system programming instructions, nor any of the MMX, SSEn, 3dNow! or IA-64 instructions. On the other hand, we have managed to cover enough instructions that we can compile real applications and run them through the simulator. Moving forward, we would like to extend the model to cover at least those instructions that are used by compilers.

Our model of machine states is also overly simple. For example, we do not yet model concurrency, interrupts, or page tables. However, we believe that the use of RTL as a staging language makes it easier to add support for those features. For example, to model multiple processors and the total-store order (TSO) memory consistency model [34], we believe that it is sufficient to add a store buffer to the machine state for each processor. Of course, validating a concurrent model will present new challenges.

We believe that the use of domain-specific languages will further facilitate re-use and help to find and eliminate bugs. For example, one could imagine embedding these languages in other proof assistants (HOL, ACL2, *etc*.) to support portability of the specification across formal systems.

We would also like to close the gap on RockSalt so that the `C` code, derived from our verified Coq code, is itself verified and compiled with a proven-correct compiler such as CompCert. In fact, one fun idea is to simply bypass the compiler and write the checker directly in x86 assembly to see how easy it is to turn the process in on itself. Finally, there are richer classes of policies, such as XFI, for which we would like to write checkers and prove correctness.

6.2 Lessons Learned

The basic idea of using domain-specific languages to build a scalable semantics worked well for us. In our first iteration of the model, we tried to directly interpret x86 instructions, but soon realized that any reasoning work would be proportional to the number of distinct instructions. Compiling instructions to a small RISC-like core simplified our reasoning, and at the same time, made it easier to factor the model into smaller, more re-usable components.

One surprising aspect of the work was that the pressure to provide reasoning principles for parsers forced us to treat the problem more algebraically than is typically done. In particular, the use of derivatives, which operate directly on the abstract syntax of grammars, made our reasoning much simpler than it would be with graphs.

It goes without saying that constructing machine-checked proofs is still very hard. The definitions for our x86 model and NaCl checker are about 5,000 lines of heavily commented Coq code, but the RockSalt proofs are another 10,000 lines. Of course, many of these proofs will be useful in other settings (*e.g.*, that the decoder is unambiguous) but the ratio is still quite large. One reason for this is that reasoning about certain theories (*e.g.*, bit vectors) is still rather tedious in Coq, especially when compared to modern SAT or SMT solvers. Yet, the dependent types and higher-order features of the language were crucial for constructing the model, much less proving deep properties about it.

For us, another surprising aspect of the work was the difference that comes with scale. We have a fair amount of experience modeling simple abstract machines with proof assistants. Doing a case split on five or even ten instructions and manually discharging the cases is reasonable. But once you have hundreds of cases, any of which may change as you validate the model, such an approach is no longer tenable. Consequently, many of our proofs were actually done through some form of reflection. For example, to prove that the `x86grammar` is unambiguous, we constructed a computable function that tests for ambiguity and proved its correctness. In turn, this made it easier to add new instructions to the grammar. Frankly, we couldn't stomach the idea of proving the correctness of a hand-written x86 decoder, and so we were forced into finding a better solution. In short, when a mechanized development reaches a certain size, we are forced to develop more automated and robust proof techniques.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the 12th ACM Conf. on Computer and Commun. Security*, CCS '05, pages 340–353. ACM, 2005.

[2] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. of the Workshop on Declarative Aspects of Multicore Programming*, pages 13–24. ACM, 2009.

[3] J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Proc. of the 15th Intl. Conf. on Implementation and Application of Automata*, number 6482 in CIAA '10, pages 59–68. Springer-Verlag, Aug. 2010.

[4] A. Barthwal and M. Norrish. Verified, executable parsing. In *European Symp. on Programming*, ESOP '09, pages 160–174. LNCS, 2009.

[5] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481–494, 1964.

[6] A. Chlipala. A verified compiler for an impure functional language. In *Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–106. ACM, 2010.

[7] D. Cock. Lyrebird: assigning meanings to machines. In *Proc. of the 5th Intl. Conf. on Systems Software Verification*, SSV'10, pages 6–15. USENIX Association, 2010.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42. Springer-Verlag, 2009.

[9] Coq development team. The Coq proof assistant. `http://coq.inria.fr/`, 1989–2012.

[10] L. Correnson, Z. Dargaye, and A. Pacalet. *WP plug-in manual*. CEA LIST.

[11] J. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '10, pages 403–416. ACM, 2010.

[12] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, OSDI '06, pages 75–88. USENIX Association, 2006.

[13] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 243–258. Springer, 2010.

[14] Intel Corporation. *Pentium Processor Family Developers Manual*, volume 3. Intel Corporation, 1996.

[15] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *European Symp. on Programming*, ESOP '12. Springer, 2012. To appear.

[16] W. A. H. Jr. and S. Swords. Centaur technology media unit verification. In *Computer Aided Verification, 21st Intl. Conf.*, volume 5643 of *LNCS*, pages 353–367. Springer, 2009.

[17] J. Kroll and D. Dean. BakerSFIeld: Bringing software fault isolation to x64. `http://www.cs.princeton.edu/~kroll/papers/bakersfield-sfi.pdf`.

[18] X. Leroy. Formal verification of a realistic compiler. *Commun. of the ACM*, 52(7):107–115, 2009.

[19] J. Lim. *Transformer Specification Language: A System for Generating Analyzers and its Applications*. PhD thesis, University of Wisconsin-Madison, May 2011.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005.

[21] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proc. of the 18th Intl. Symp. on Software Testing and Analysis*, pages 261–272. ACM, 2009.

[22] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. of the 15th Conf. on USENIX Security Symp.*, pages 209–224. USENIX Association, 2006.

[23] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher order logic. In *Automated Deduction - CADE-17, 17th Intl. Conf. on Automated Deduction*, volume 1831 of *LNCS*, pages 7–24. Springer, 2000.

[24] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *Proc. of the 16th ACM SIGPLAN Intl. Conf. on Functional Programming*, ICFP '11, pages 189–195. ACM, 2011.

[25] Native Client team. Native client security contest. `http://code.google.com/contests/nativeclient-security/index.html`, 2009.

[26] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19:173–190, March 2009.

[27] S. Owens, P. Böhm, F. Z. Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 363–369. Springer, 2011.

[28] A. Pilkiewicz. A proved version of the inner sandbox. In *native-client-discuss mailing list*, April 2011.

[29] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *LNCS*, pages 176–192. Springer, 1998.

[30] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19(3):492–524, 1997.

[31] S. Ray. Towards a formalization of the X86 instruction set architecture. Technical Report TR-08-15, Department of Computer Science, University of Texas at Austin, March 2008.

[32] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 379–391. ACM, 2009.

[33] M. Seaborn. A DFA-based x86-32 validator for Native Client. In *native-client-discuss mailing list*, June 2011.

[34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, SOSP '93, pages 203–216. ACM, 1993.

[36] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 283–294. ACM, 2011.

[37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. of the ACM*, 53(1):91–99, 2010.

[38] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *11th Intl. Conf. on Embedded Software*. ACM, 2011.