Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls

Shilpi Goel Warren A. Hunt, Jr. Matt Kaufmann Soumava Ghosh
Department of Computer Science
The University of Texas at Austin

***Abstract*—We present an approach to modeling and verifying machine-code programs that exhibit non-determinism. Specifically, we add support for system calls to our formal, executable model of the user-level x86 instruction-set architecture (ISA). The resulting model, implemented in the ACL2 theorem-proving system, allows both formal analysis and efficient simulation of x86 machine-code programs; the *logical mode* characterizes an external environment to support reasoning about programs that interact with an operating system, and the *execution mode* directly queries the underlying operating system to support simulation. The execution mode of our x86 model is validated against both its logical mode and the real machine, providing test-based assurance that our model faithfully represents the semantics of an actual x86 processor. Our framework is the first that enables mechanical proofs of functional correctness of user-level x86 machine-code programs that make system calls. We demonstrate the capabilities of our model with the mechanical verification of a machine-code program, produced by the GCC compiler, that computes the number of characters, lines, and words in an input stream. Such reasoning is facilitated by our libraries of ACL2 lemmas that allow automated proofs of a program's memory-related properties.**

I. Introduction

To enable the formal verification of x86 machine-code programs, we are developing a tool suite based on our formal, executable model of the x86 instruction-set architecture (ISA). The x86 ISA has been modeled in the ACL2 programming language; we have formalized the semantics of most user-level instructions with an interpreter that can execute x86 machine-code programs. We have extended our x86 model with a formalization of an x86 system call instruction, namely, `syscall`. The execution of system calls is not provided directly by the x86 ISA; it is provided by a contemporary operating system, like Linux, FreeBSD, Windows, or MacOS, to a user process. Our extension to the x86 ISA model includes the semantics of various system calls, thereby allowing us to prove properties of user-level x86 machine-code programs that rely on an operating system for system call service.

As is the case for all other instructions specified, our extended model enables not only the formal analysis of system calls, but also supports their simulation. In fact, our extended model provides the capability to simulate and verify non-deterministic computations in general, including system calls and x86 instructions like `rdrand`. We achieve this by way of two modes in our model: the *logical mode* that supports reasoning and the *execution mode* that allows simulation.

Our evolving x86 ISA model includes specifications for 64-bit segmentation, paging, supervisor calls/returns, system registers, and many other system-level features. This model is intended to mimic the ISA-level behavior of an x86 processor; it does not currently include a specification of specialized hardware, such as the APIC and RTC.

One might wonder why we choose to formally analyze machine-code programs. In situations where source programs are unavailable, such as executables downloaded from the Web or many software distributions, we have no alternative but to analyze machine-code programs. Compilers may produce incorrect machine-code from higher-level programs, so it is important to verify the actual code that is executed on a processor. Also, programmers often optimize their high-level programs by embedding assembly code in them; the verification of such programs is impossible without the ability to analyze machine code. It quickly becomes intractable to build and maintain tools targeting various aspects of software verification; our approach provides a single, unified model that can serve multiple purposes.

Our contributions are in three areas: one, a highly-validated formal, executable model of the x86 ISA extended with system calls; two, a framework that, for the first time, provides the capability both to formally analyze and to efficiently simulate user-level x86 machine-code programs that exhibit non-determinism; and three, ACL2 libraries of lemmas that facilitate automated machine-code proofs. We present a case study to demonstrate the capabilities of our tool suite: the proof of correctness of a machine-code, word-counting program much like Linux *wc*. This case study suggests the viability of interactive theorem-proving for complex interpreter-based models with non-determinism, as in the case of our x86 model extended with system calls. All the specification and verification of programs in our tool suite is done using the ACL2 logic and its associated mechanical theorem-proving system; we know of no comparably rigorous environment for the analysis of x86 machine-code programs.

We emphasize the difference between our inference-based approach and flow-based static analysis approaches. Though flow-based approaches are being successfully used to detect vulnerabilities like buffer overflows, they can not guarantee that a given program meets its specifications. Indeed, the lack of requirement of specifications as input is considered to be the biggest strength of these approaches, thereby making them accessible to the average programmer. Our approach falls under "heavyweight" verification; given a program's specifications, our focus is on building automated tools to verify whether the program behaves as intended. Note that it is possible to *prove* the absence of vulnerabilities in our approach.

In Section II, we describe our x86 ISA model and its validation process. We discuss the extension of our model with system calls in Section III. We introduce our example program in Section IV, and present its proof of functional correctness in

Section V. We conclude with discussions of related and future work in Sections VI and VII.

II. x86 ISA Model

Our x86 ISA model [1] implements an interpreter-style operational semantics [2]. Our x86 model's state contains registers like the general-purpose registers (`rax`, `rbx`, etc.), segment registers, flags register, model-specific registers, control registers, instruction pointer `rip`, and memory. Each machine instruction is specified by a *semantic function* that takes an x86 state and returns an appropriately modified next state. A *step* function fetches, decodes, and then executes an instruction by calling the appropriate instruction semantic function. Finally, a *run* function takes the number of instructions, `n`, to be executed and an initial x86 state, and returns a resulting x86 state; the run function either takes `n` steps or stops if an irrecoverable error is encountered, whichever comes first.

Our current modeling focus is on the 64-bit mode of Intel's IA-32e architecture (x86-64). We have a specification of all addressing modes, 121 user-level instructions (223 opcodes), IA-32e paging, and `FS`/`GS`-based segmentation. Our x86 ISA model is around 40,000 lines of code, which includes proofs about the specification, but does not include our tools for binary analysis. The model can execute most user-level integer programs emitted by the GCC/LLVM compiler — notable exceptions are media and floating-point instructions, which we plan to model in the near future.

Our model can be used in either a supervisor-level or programmer-level mode of operation. The supervisor-level mode includes support for IA-32e paging. In this mode, our memory model characterizes a 2^{52}-byte physical address space, which is the largest address space provided by modern x86 implementations. This mode can be used to simulate and verify system software. The programmer-level mode of our model attempts to provide the same environment to a programmer for reasoning as is provided by an OS for programming; it allows the verification of an application program while assuming that services like paging and I/O operations are provided reliably by the operating system. In this mode, our memory model supports the 64-bit linear addresses specified for IA-32e machines.

The simulation speed of our model in programmer-level mode is $\sim$3.3 million instructions/second and in supervisor-level mode, with a two-level page table configuration, is $\sim$920,000 instructions/second on a machine with a 3.50GHz Intel Xeon E31280 CPU. Achieving high simulation speeds facilitates the use of our formal processor model as an instruction-set simulator, which enables its validation against the real machine, as we discuss below. It is a challenge to support efficiency for both reasoning and simulation; specification functions written to maximize simulation efficiency, like those for our memory model specification [3], can be hard to reason about and those written to enable simpler reasoning can run slowly. We use abstraction techniques [4] to attain both reasoning and simulation efficiency. For the rest of this paper, we focus on the programmer-level mode of our x86 model.

ISA Model Validation: How can we trust that our model faithfully represents the x86 ISA? A benefit of using ACL2 to develop our x86 model is that its efficient executability enables validation of the model against the real machine using co-simulation. We compile high-level programs using GCC/LLVM and compare each run of a resulting x86 program on the real machine to the corresponding run on our x86 model. Our model is capable of running unmodified x86 machine-code programs because we do not simplify the semantics of x86 instructions. For example, we have successfully simulated a contemporary SAT solver on our x86 model[1]. When given an instance of the SAT'09 Competition Application benchmark (cmu-bmc-barrel6.cnf), 9,142,833,444 machine instructions are executed at run-time for the solver to run to completion. On all these instructions, our model produced exactly the same effects on the memory and registers as those produced by the real machine.

Our model validation framework uses GDB and Intel's dynamic instrumentation library, Pin [5], to extract the machine state while running programs on the processor. In the execution mode of the x86 model, the framework uses our own dynamic instrumentation library, written entirely in ACL2, to extract our model's state so that it can be compared to the real machine state at a desired level of granularity, be it on a per-instruction or a per-breakpoint basis. This framework is largely automated — it spawns off the GDB/Pin process on the real machine, uses the information captured by GDB/Pin to initialize our x86 model appropriately, runs the model in its execution mode on concrete data, and produces a report containing the differences observed, if any, between the real machine state and the model's state. This automated and easy-to-use framework makes it convenient to run many co-simulations, thereby facilitating fast and thorough model validation.

We have invested several person years of work in our x86 model. We use the Intel manuals [6] as specification documents; ambiguities are resolved by running tests on the real processor and by consulting with processor architects. Our model is a formal specification for the x86 ISA, and it can also serve as the target specification for RTL design verification.

III. System Call Model

User-level programs, either directly or through higher-level interfaces provided in libraries, often make system calls to the underlying operating system to request services such as file I/O and memory management. Though the x86-64 architecture provides other instructions to invoke and return from system calls, we focus on `syscall` and `sysret`; these instructions are the most common and efficient interface between the kernel and a user application. The `syscall` instruction is used by user-level code to call system-level procedures at the highest privilege level by loading the `rip` with the appropriate address from a model-specific register. The companion instruction, `sysret`, returns control from the system procedures to user-level code at the application privilege level. These instructions allow fast privilege-level transitions during the system call invocation and return process by keeping all the information required for the transition in general-purpose and model-specific registers, thereby avoiding the overhead of table references in memory.

[1]This SAT solver was developed by Marijn J. H. Heule; its performance is comparable to those of state-of-the-art solvers.

From the perspective of a user-level program, system calls are non-deterministic — different runs can yield different results on the same machine. Since our x86 model serves both as an executable instruction-set simulator and a formal specification that is used to do proofs about machine code, we need to be able to do the following:

1) Efficiently simulate runs of a program with system calls on concrete data, and
2) Formally reason about such a program given symbolic data.

Ideally, to accomplish both these tasks, modeling enough features of the x86 would allow an operating system to be loaded on the model to service system calls. Consequently, we could both simulate and reason about system calls due to the executable and formal nature of our ACL2-based model. However, loading a modern OS on a processor model is non-trivial; the added complexity of the low-level interaction of the OS with the processor would not only make reasoning about user-level programs harder, but also slow down the simulation speed of concrete program runs.

Instead, for simulation of system calls, we set up the *execution mode* of our x86 model to interact directly with the underlying OS. ACL2 provides a mechanism [7] for allowing arbitrary Common Lisp code to be defined in *raw Lisp*, outside ACL2. The system call service is provided by raw Lisp functions to obtain "real" results from the OS [8]. Simulation of all instructions other than `syscall` happens within ACL2 (and hence, Lisp). Note that since we are abstracting away the system-level procedures that are invoked by the OS when a system call is made, we do not need to make a similar arrangement for the `sysret` instruction.

These raw Lisp functions should not be used for reasoning since they are *impure*: they are not axiomatized logically, and indeed, are not even functions in the logical sense since repeating the same call can yield different results. It is critical for our framework to prohibit proofs of theorems that state that some system call returns a specific value. If that were the case, then due to the non-determinism inherent in system calls, we might be able to prove that the same system call returns some other value in a different ACL2 session. Or perhaps worse yet, we could prove an instance of $\mathtt{x} \neq \mathtt{x}$ by instantiating `x` with a term that invokes the system call. Another disturbing scenario would be when such theorems contradict results observed in a program run simulation.

Thus, for reasoning about machine-code programs we use the *logical mode* of our model, which incorporates into the state an *environment* field to represent the part of the external world that affects or is affected by system calls. To reason about a system call's effects, we simply consult that `env` field. A well-formed `env` field contains sub-fields that describe a subset of the file system and an *oracle* that provides information that, though a part of the real environment, cannot be inferred from our model of the file system. An example of such information is the file descriptor of a file to be opened; an OS assigns the file descriptor depending on the number of files already opened for a particular process at the time the open system call is made.

The contents of `env` can be *abstract*. For example, to verify a program like `grep` that searches for occurrences of a pattern in an input file, the pattern can be specified as an arbitrary string and the file can be specified as an arbitrary file in the file system (or not, if we wish to reason about the case when the file does not exist). This ability to reason about arbitrary elements in the environment is precisely what makes reasoning about non-determinism possible. Of course, it is also possible to reason about specific elements in the environment, e.g., `grep` with a specific pattern on a specific file, by simply initializing the `env` field with these elements.

Consider two runs of our model with the same initial x86 state, where one is in execution mode with real environment `ENV` and the other is in logical mode with environment field `env`. We say that `env` *corresponds* to environment `ENV` if the execution of system calls produces the same results in the logical mode as in the execution mode.

The execution mode does not unduly impact the logical mode, since the raw Lisp functions do not influence the reasoning process. Conversely, the `env` field does not interfere with the impure functions in the execution mode. However, the logical and execution modes are far from completely independent, as noted by the following three properties.

(L) For reasoning, all the functions in the logical mode of the x86 model are pure.

(E) The execution mode allows the use of raw Lisp functions that directly interact with the underlying OS to provide system call service. Note that the logical mode and execution mode are identical for all instructions except `syscall` — all other instructions have the same definitions in both these modes.

(C) The following connection exists between the logical mode and the execution mode. Let $\mathtt{x}_0$ be an x86 state. Suppose in the execution mode, the evaluation of $\mathtt{(run\ x_0)}$ returns $\mathtt{x}_1$ and updates the real environment from `ENV` to `ENV'`. Then, the following is true for the logical mode: if `env` corresponds to `ENV`, and $\mathtt{x}_0'$ refers to $\mathtt{x}_0$ augmented with `env`, then the evaluation of $\mathtt{(run\ x_0')}$ in the logical mode produces $\mathtt{x}_1$ augmented with `env'`, for some `env'` corresponding to `ENV'`.

See Figure 1 for an illustration of a program run in both the execution and logical modes. We discuss property (C) in some detail later in this section. Due to property (C), we know that evaluation results produced by raw Lisp functions will not be contradicted by theorems proved about system calls; in fact, each program run in the execution mode produces a theorem under a hypothesis about the well-formedness of the environment in the logical mode. Thus, observations made while performing simulation in the execution mode hold in the logical mode as well. Our method facilitates the maintenance of an integrated software base for the logical and execution modes of the x86 model.

Our framework makes reasoning about non-deterministic computations in programs tractable. As we will see in Section V, the proof of correctness of a program is not complicated by the presence of system calls. We have used this approach to model and implement the following system calls: `read`, `write`, `open`, `close`, `lseek`, `dup`, `link`, and `unlink`. We support the simulation of these system calls on both

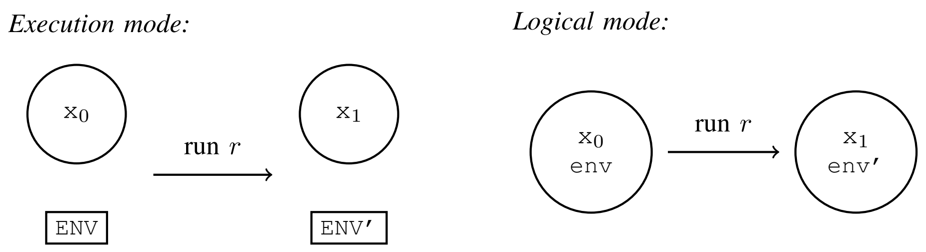

Figure 1. Illustration of a run in the execution mode (left) and in the logical mode (right). A run, r, from an initial state $\texttt{x}_0$ in the execution mode gives a final state $\texttt{x}_1$, and the environment ENV on the real machine transitions to ENV′. In the logical mode, env corresponds to ENV, and r produces the same final state, $\texttt{x}_1$, augmented with env′, which corresponds to ENV′.

Linux and Darwin systems. Our approach can be used to handle various sources of non-determinism, other than just system calls, that arise in user-level programs. One such example is the rdrand instruction, which is used to provide cryptographically secure random numbers to applications.

System Call Model Validation: For all instructions but syscall, comparing the real machine state to the model state extracted in the execution mode validates the logical mode as well since these modes are identical. However, for the syscall instruction, the execution and logical modes of the x86 model consist of different functions, and are thus distinct. Consequently, two validation tasks need to be performed for the syscall instruction:

1. Validate the execution mode of the x86 model against the real system, i.e., processor plus system call service provided by the operating system, and
2. Validate the execution mode against the logical mode of the x86 model.

We accomplish the first validation task using our model validation framework, as discussed in Section II. Since the raw Lisp functions supporting the execution mode of the syscall instruction interact with the underlying OS and hence, pass on the results of the real machine to our framework, the only functions of the execution mode that need to be validated are those that marshal the input arguments and return values of these raw Lisp functions, and those that capture the effects of a return from the system call. The latter accounts for the effects of the sysret instruction as well; for example, sysret always clears the RF and VM flags, and the programmer's view of the processor after a return from a system call should also depict these flags as cleared.

The second validation task is critical to ensure the property (C) stated earlier. The logical mode for syscall can be thought of as the specification for its execution mode. The specification functions supporting the logical mode are written in accordance with the man pages of the system calls and their more detailed descriptions found elsewhere [9]. We validate the execution mode against the logical mode by performing extensive code reviews, and by comparing program runs in the execution mode to corresponding runs in the logical mode. We illustrate this process by a short example. Consider the following five x86 instructions. This snippet of an assembly program makes a read system call to obtain one byte from a file with descriptor equal to 0, usually the standard input. The arguments needed by the read system call are loaded into appropriate registers, as dictated by the x86-64 Application Binary Interface [10]. The rax register contains the Linux read system call number, the rdi register contains a file descriptor, the rsi register contains the address of the memory buffer where the read bytes will be written, and the rdx register contains the number of bytes to be read.

```
mov $0x0,%rax          /* Syscall number   */
xor %rdi,%rdi          /* File descriptor  */
mov -0x20(%rbp),%rsi /* Buffer address   */
mov $0x1,%rdx          /* Number of bytes  */
syscall
```

In the execution mode, we initialize our x86 model to reflect the state of the real machine when rip points to the address of the first mov instruction. We set up the model to make five steps, i.e., run this snippet. Then, the raw Lisp function for the read system call collects the user's input.

In the logical mode, we initialize the environment field env so that it corresponds to the real environment. As such, the contents of the standard input in the env field should contain the user input that was collected in the corresponding run in the execution mode. After setting up the rest of the fields of the x86 state to be exactly the same as those of the initial state in the execution mode, we run the model to simulate these five instructions. A comparison of the final state obtained in the logical mode and execution mode allows validation of these modes against each other.

IV. Program: Simple Word Count

We analyze the machine code corresponding to a simple word count program taken from "The C Programming Language" by Kernighan and Ritchie [11]. This C program is a bare-bones version of the wc program found on Linux systems. We use this program as a case study to assess the capability of our model to simulate and reason efficiently about programs that make system calls. GCC compilation generated 50 machine instructions (166 bytes) — 17 instructions for the gc procedure, including the syscall instruction, and 36 instructions for the main sub-routine.

The program reads a character from the standard input until the end of input (which is denoted by the character #), each time incrementing the character counter nc. If the character is a newline, then the newline counter nl is also incremented. The word counter nw is incremented at the beginning of every word, i.e., when state transitions from OUT to IN.

```
#define IN   1    /* inside a word  */
#define OUT  0    /* outside a word */
```

```
#define EOF '#'   /* EOF character  */
#include <stdio.h>
int gc(void) {
  char buf[1];
  int n;
  __asm__ volatile
    (
     "mov $0x0, %%rax\n\t"
     "xor %%rdi, %%rdi\n\t"
     "mov %1, %%rsi\n\t"
     "mov $0x1, %%rdx\n\t"
     "syscall"
     : "=a"(n)
     : "g"(buf)
     : "%rdi", "%rsi", "%rdx");
  return (unsigned char) buf[0];
}
/* count lines, words, characters in input */
int main () {
  int c, nl, nw, nc, state;
  state = OUT;
  nl = nw = nc = 0;
  while ((c = gc()) != EOF) {
    ++nc;
    if (c == '\n')
      ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
      state = OUT;
    else if (state == OUT) {
      state = IN;
      ++nw;
    }
  }
  return 0;
}
```

The original program from Kernighan and Ritchie's book used the C standard library (*glibc*) function `getchar` instead of our function `gc`. The machine code corresponding to `getchar` used SIMD (AVX) instructions in some places to speed up execution. Since we do not yet support SIMD instructions in our model, we chose to write `gc` as our own version of `getchar`. The function `gc` can be thought of as an inefficient, unbuffered `getchar`. Every call of `gc` attempts to read one byte from the standard input and stores it in a memory buffer. An alternative to defining `gc` could be to use a portable and lightweight standard library like *newlib* [12] instead of *glibc*.

Before reasoning about the entire program in the logical mode of our model, we ran simulations in the execution mode. The program behaved as expected on our model, thereby providing confidence that our model faithfully emulates a real x86 system for the instructions of this program.

V. Functional Correctness of Simple Word Count Machine-Code Program

In this section, we discuss the verification of the machine-code program produced by running the GCC compiler on our example program. This machine-code program is structurally quite similar to its C source; in particular, it has a loop that begins with a call to the `gc` sub-routine, which makes a system call. The program variables `nc`, `nl`, `nw`, and `state` are allocated on the stack in consecutive memory locations.

We apply a traditional theorem-proving approach to program verification, since our previous automatic approach using bit-blasting [13] is limited in its handling of loops and large programs. We formally analyze this program using the Boyer-Moore clock function method [14], [15]. We briefly describe this method here. Given a clock function *clock* that specifies the number of steps needed for a program to run to completion, the following theorem states the total correctness of a program: if *x* is an x86 state satisfying specified pre-conditions, then the final state *run(clock(x),x)* satisfies the specified post-conditions. It is the user's responsibility to write these clock functions; there is ongoing research to automate this task in ACL2 [16], comparable to previous work for HOL4 [17], [18].

$$\forall x : \mathit{pre\text{-}conditions}(x) \implies \mathit{final\text{-}state}(\mathit{run}(\mathit{clock}(x), x)) \wedge \mathit{post\text{-}conditions}(x, \mathit{run}(\mathit{clock}(x), x))$$

How can we state functional correctness for our program? We choose to write a trio of simple ACL2 specification functions that compute the character, line, and word counts of a string, respectively. Our post-condition asserts that the values returned by these three specification functions on standard input are found in the expected memory locations of the final x86 state, which is obtained by running the program on our x86 model in its logical mode.

We now outline the proof of functional correctness of the simple word count machine-code program. The program structure can be used as a guide to decompose the proof into two sub-tasks — one, the verification of the initial part of the program when all counters are initialized to 0, and two, the verification of the loop, which begins with a call to the `gc` function. We use the theorems stating correctness of these program components to obtain the final correctness theorem.

We begin by stating the assumptions made about the `env` field in the x86 state, in order to reason about the system call that performs a read in the `gc` function.

1) The file descriptor corresponding to the standard input is `0`. Note that we make this assumption only because the program itself makes this assumption.
2) The contents of the standard input should be terminated by the end-of-file character (# for this program), and thus, be non-empty. We make this assumption because the program does not terminate unless this end-of-file character is encountered.

The read system call has the following interface:

```
ssize_t read (int fd, void *buf, size_t count);
```

This system call tries to read `count` bytes from the file pointed to by the file descriptor `fd` into the memory buffer beginning at `buf` [19]. The read made in the `gc` function of the word-count program has `fd` referencing the standard input, `buf` pointing to a stack address, and `count` equal to one. The specification of the read system call in the logical mode of our x86 model only tells us that one byte is read from the standard input, modeled by the `env` field, and written to the memory buffer unless some error is encountered. We can not deduce the value of this byte. This permits us to reason about our program for all possible bytes that can be returned by one call of `gc`. Various errors, like `buf` pointing to an illegal memory address, are also accounted for by our system call specification.

Let us first focus on the loop. The loop pre-conditions *loop-pre* are as follows.

1) The x86 state is well-formed.
2) The environment assumptions hold for this x86 state.
3) The program is loaded in the memory at its expected location.
4) The instruction pointer, `rip`, points to the first instruction of the loop.
5) The stack pointer, `rsp`, is within a specified range. This guarantees that the stack does not over-write the code during the program's execution.

Pseudo-code for the ACL2 function `loopClk` is shown below. This function is the loop's clock function, which computes the number of steps needed for the loop to complete. The argument `state` of `loopClk` corresponds to the `state` variable of the simple word count program, `offset` corresponds to the position of the next character to be read from standard input, and `strBytes` corresponds to the contents of standard input in bytes. Constants like `cEOF`, `cNL`, `cSpace`, `cTab`, `cOut`, and `cIn` denote the number of instructions that are executed during run-time in one loop iteration, according to which branch of the loop is taken. Thus, the function `loopClk` keeps recurring till `EOF` is encountered; for each recursive call, it adds the number of instructions to be executed at run-time based on the character read.

```
loopClk(state,offset,strBytes):

if !(envAssumptions(offset,strBytes)) then
 // No instructions are run when environment
 // assumptions fail.
 0
else {
 // gcSpec is gc's specification function.
 char = gcSpec(offset,strBytes)
 if (char == EOF) then
   cEOF
 else {
   case (char) {
     newline   :  state = OUT
                  loopSteps = cNL
     space     :  state = OUT
                  loopSteps = cSpace
     tab       :  state = OUT
                  loopSteps = cTab
     otherwise :  if (state == OUT) then
                     state = IN
                     loopSteps = cOut
                   else
                     loopSteps = cIn }
   return(loopSteps +
          loopClk(state,(1+ offset),strBytes))
 } }
```

Given these pre-conditions and loop clock function `loopClk`, the loop correctness theorem is as follows, where we write *l* to abbreviate the application of `loopClk` to the appropriate values stored in the x86 state, *x*.

Theorem 1: $\forall x : \textit{loop-pre}(x) \implies \textit{halted}(\textit{run}(l, x)) \wedge \textit{post}(x, \textit{run}(l, x))$

where *x* is an x86 state that satisfies the loop pre-conditions *loop-pre*, *post* relates the trio of our specification functions to the values in the expected memory locations of the counters in the halted state *run(l,x)*, and *l* specifies the number of steps the entire loop takes to reach the final state, i.e., *l* is a value computed by `loopClk`.

Proof: This theorem can be proved by strong induction on the value *l* of `loopClk`. If *l* is 0, then `envAssumptions` is false; thus *loop-pre(x)* does not hold, which proves the base case. Otherwise the proof splits into cases according to the character read. Let us address the case that this character is a newline, as the other cases are analogous. By the inductive hypothesis, we may assume the following, which is obtained from the theorem by replacing *x* with *run(*`cNL`*, x)* and *l* with (l − `cNL`), and noting that (l − `cNL`) is the the application of `loopClk` to the appropriate values stored in the x86 state, *run(*`cNL`*, x)*.

$$\textit{loop-pre}(\textit{run}(\texttt{cNL}, x)) \implies \textit{halted}(\textit{run}((l - \texttt{cNL}), \textit{run}(\texttt{cNL}, x))) \wedge \textit{post}(\textit{run}(\texttt{cNL}, x), \textit{run}((l - \texttt{cNL}), \textit{run}(\texttt{cNL}, x))) \quad (1)$$

The following fact is easy to prove by definition of the *run* function.

$$\textit{run}(l, x) = \textit{run}((l - \texttt{cNL}), \textit{run}(\texttt{cNL}, x)) \quad (2)$$

By substituting 2 into 1, we obtain:

$$\textit{loop-pre}(\textit{run}(\texttt{cNL}, x)) \implies \textit{halted}(\textit{run}(l, x)) \wedge \textit{post}(\textit{run}(\texttt{cNL}, x), \textit{run}(l, x)) \quad (3)$$

The proof of Theorem 1 follows from the induction hypothesis if we prove the following lemmas:

L1: $\forall x : \textit{loop-pre}(x) \implies \textit{loop-pre}(\textit{run}(\texttt{cNL}, x))$
L2: $\forall x : \textit{loop-pre}(x) \implies \textit{post}(x, \textit{run}(\texttt{cNL}, x))$
L3: $\forall x : \textit{post}(x, \textit{run}(\texttt{cNL}, x)) \wedge \textit{post}(\textit{run}(\texttt{cNL}, x), \textit{run}(l, x)) \implies \textit{post}(x, \textit{run}(l, x))$

The proof of L1 is conceptually simple, and we lead ACL2 to simplify expressions representing the values of components of the state *run(*`cNL`*,x)* that are relevant to *loop-pre*, such as its `rip`. The proof of L2 proceeds in the same manner. Given our description of *post* as relating stack values for the counters with our specification functions, the proof of L3 follows from the transitivity of *post*. ■

We then prove the following theorem about the initial part of the program, i.e., the part preceding the loop. Here, *pre* has a form similar to *loop-pre* but with obvious differences, for example: the `rip` points to the first instruction of the program instead of to the first instruction of the loop, and the constant `i` is the number of instructions required to take the program to the beginning of the loop.

Theorem 2: $\forall x : \textit{pre}(x) \implies \textit{loop-pre}(\textit{run}(i, x))$

The proof is similar to the proof of L1. We finally prove total correctness for this program by using Theorems 1 and 2. Here, *c* is the value computed by the clock function for the entire program on initial state *x*, that is, $c = i + l$.

Theorem 3: $\forall x : \textit{pre}(x) \implies \textit{halted}(\textit{run}(c, x)) \wedge \textit{post}(x, \textit{run}(c, x))$

Though the proof of correctness of the word-count program is straightforward, it is worth emphasizing that it was done on a large interpreter-based model of the x86, where the semantic functions of instructions are, on an average, ~200 lines of ACL2. This proof makes heavy use of compositional reasoning and would have been harder to do had we not developed our own libraries to automate reasoning about reads and writes made to the x86 state. We proved many lemmas about registers, flags, etc. Here we briefly discuss one such library that facilitates reasoning about memory accesses and updates.

Reasoning about memory usage is challenging, simply because memory is so large. Moreover, code and data share the memory, which requires establishing that each write to the stack or heap during the program's execution does not overwrite program and data. Verifying position-independent code entails reasoning about disjointness of memory regions that are specified by symbolic or computed memory addresses. As such, a lot of tedious low-level arithmetic reasoning about inequalities and equalities involving these symbolic addresses is required. Our library lifts this problem to reasoning about membership of addresses in lists instead, and list-based reasoning is done largely automatically. An example is the automated proof of the disjointness of the program and the stack done in our word-count case study. Another proof that was discharged automatically was that the word-count program does not modify unintended regions of memory, i.e., the only writes that occur during the program's execution are to the stack, and the rest of the memory is the same as it was before the execution. This is an important theorem because it rules out one kind of potential "evilness" of our word-count program. Other kinds of memory guarantees, like ruling out stack smashing and buffer overflows, can also be established using our library.

In order to facilitate re-use, our proof libraries are designed to be as general as possible. As we verify progressively more complicated programs, we discover new lemmas and extend these libraries. Below is some empirical evidence that illustrates how our libraries can reduce manual effort:

Lines of ACL2 needed to verify the word count program:

- Without the libraries: $\sim$20K
- With the libraries: $\sim$8K

$\sim$8K lines of ACL2 might still seem excessive. However, at least half of these lines were generated by ACL2 in response to commands to simplify specific symbolic expressions. The simplified expressions are large because there are many updates to different components of the x86 state to symbolically run even a small program.

VI. Related Work

Machine-code verification has long been an area of active research. As such, many formal models of contemporary processor ISAs have been developed to enable reasoning about machine-code [20]–[23]. Our strategy of modeling an external environment to account for non-determinism in programs is similar to Moore's work [24] in ACL2 to model non-determinism in a pedagogical multiprocessor system. There has been considerable research on the verification of system calls from a micro-kernel point of view [25], [26]. In this paper, we concern ourselves with reasoning about user-level x86 programs that interact with a contemporary operating system. Here, we mention some recent work with goals similar to ours.

Morrisett et al., while working on software fault isolation [27], developed an x86 ISA specification in the Coq [28] proof assistant that can also be used for machine-code verification. Morrisett's x86 specification is not directly executable; an executable OCaml simulator has to be extracted from the x86 specifications in Coq. The resulting simulator has an execution rate of $\sim$50 instructions/second; it simulates $\sim$10 million instructions in 60 hours on an 2.6 GHz, 8 core Intel Xeon machine. This work is concerned with restricting certain kinds of computations that can be performed natively on the host machine in order to avoid information leaks to a web browser. It is not designed to handle the verification of general user-level programs that employ system calls. Feng et al. [29] use the Coq proof assistant [28] to prove the functional correctness of machine-code on a formal model of a processor that can handle asynchronous events like signals and interrupts. However, this processor model is a simplified version of the x86, and does not handle 64-bit x86 machine-code programs. Dowse et al. [30] used the Sparkle proof assistant [31] to verify programs that perform I/O. This verification effort is targeted at higher-level programs, specifically lazy functional programs. Malecha et al. [32] also verified high-level Coq programs that perform I/O. Reps et al. [33] have developed a sophisticated system called TSL, that can create retargetable tools for different types of machine code analyses, especially data-flow analyses. We do not know of a TSL-created tool that can prove whether a given machine code program meets its specification.

VII. Conclusion and Future Work

We mechanically verify user-level x86 machine-code programs with our ACL2-based ISA model extended with a specification of system calls. Our effort is the first mechanical verification of a user-level x86 machine-code program that includes the use of system calls.

Our extended model has two modes: (a) a logical mode that formally axiomatizes an external environment to enable reasoning about programs that include instruction-based non-determinism and that make system calls, and (b) an execution mode that supports program simulation by interacting with the underlying OS to produce results just as if executing a user-level machine-code program natively on an x86 processor with contemporary OS support. We regularly validate the accuracy of our x86 model using co-simulation, having already done so for many billions of instructions.

Our approach avoids any special treatment for system calls when proving the functional correctness of a user program. More generally, our framework makes formal analysis of non-determinism in programs tractable. This effort has led to the development of ACL2 libraries that automate machine-code verification, in particular for reasoning about memory reads and writes. Automating such tedious reasoning activities considerably speeds up the proof development process.

Our case study of the verification of the word count program provides compelling evidence that there is much more potential for automating x86 machine-code proofs in our framework. The proof for this program was tedious; similar kinds of theorems were needed to reason about different parts of the program. However, these proofs were already largely automated due to the support provided by our libraries. We continue to develop tools to support automation in order to make machine-code verification in our framework accessible to those unfamiliar with formal verification of programs on interpreter-based models.

We should note that our model of the file system does not account for concurrent updates by external processes. Our verification work assumes that the input being processed will not be changed during the execution of our program; thus, our specification states the behavior of our programs in the absence of such concurrent updates. Exploring program correctness in view of possible interference by other programs would require, at the very least, a more subtle model of the environment being provided for our verification effort.

We believe that our specification of the x86 ISA, coupled with the ACL2 system, can facilitate regular verification of x86 machine-code programs. To realize this goal, we would begin by verifying various programs in standard libraries; then, we would verify programs that make use of these standard libraries. Such compositional methods can provide a scalable way to prove the functional correctness of machine-code programs.

Acknowledgment

We thank Marijn J. H. Heule for his invaluable feedback on the paper. This work is supported by DARPA under contract number N66001-10-2-4087.

References

[1] Matt Kaufmann and Warren A. Hunt, Jr., "Towards a formal model of the x86 ISA," Department of Computer Science, University of Texas at Austin, Tech. Rep. TR-12-07, May 2012.

[2] J S. Moore, "Mechanized operational semantics," Lectures in the Marktoberdorf Summer School (August 5-16, 2008)., Online; accessed: January 2014. http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html.

[3] Warren A. Hunt, Jr. and Matt Kaufmann, "A formal model of a large memory that supports efficient execution," in *Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012, Cambrige, UK, October 22-25)*.

[4] Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann, "Abstract stobjs and their application to ISA modeling," in *Proceedings ACL2 2013, EPTCS 114, 2013, pp. 54-69*.

[5] Intel, "Pin: A Dynamic Binary Instrumentation Tool," http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[6] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manuals." Order Number: 325462-048US. (September 2013). http://download.intel.com/products/processor/manual/325462.pdf., online; accessed: January 2014.

[7] Matt Kaufmann, J S. Moore, Sandip Ray, and E. Reeber, "Integrating external deduction tools with ACL2," *Journal of Applied Logic*, vol. 7, no. 1, pp. 3–25, Mar. 2009.

[8] CCL, "CCL Manual: Foreign Function Interface," http://ccl.clozure.com/manual/chapter13.html., clozure Common Lisp Manual. Online; accessed: January 2014.

[9] Michael Kerrisk, *The Linux Programming Interface*. No Starch Press, 2010.

[10] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell, "System V Application Binary Interface: AMD64 Architecture Processor Supplement," http://www.x86-64.org/documentation/abi.pdf, online; accessed: January 2014.

[11] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd ed. Prentice-Hall, 1988.

[12] Newlib, "Newlib C Library," https://sourceware.org/newlib/., online; accessed: January 2014.

[13] Shilpi Goel and Warren A. Hunt, Jr., "Automated code proofs on a formal model of the X86," in *Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, Ernie Cohen and Andrey Rybalchenko, Ed., vol. 8164. Springer Berlin Heidelberg, 2014, pp. 222–241. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54108-7_12

[14] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young, "Special Issue on System Verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 409–530, 1989.

[15] Sandip Ray, Warren A. Hunt, Jr., John Matthews, and J S. Moore, "A mechanical analysis of program verification strategies," *Journal of Automated Reasoning*, vol. 40, no. 4, pp. 245–269, May 2008.

[16] J S. Moore, "Code Walker Tool," (presented as a Rump Session Talk at the ACL2 Workshop, 2013, Laramie, Wyoming).

[17] Magnus O. Myreen, *Formal Verification of Machine-code Programs*. British Computer Society., 2008.

[18] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind, "Decompilation into logic - improved," in *Formal Methods in Computer-Aided Design (FMCAD), 2012*, 2012, pp. 78–81.

[19] Linux, "read(2) - Linux manual page," Retrieved from: http://man7.org/linux/man-pages/man2/read.2.html., online; accessed: January 2014.

[20] Warren A. Hunt, Jr., "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, 1989.

[21] J S. Moore, *Piton: A Mechanically Verified Assembly-level Language*. Kluwer Academic Publishers, 1996.

[22] Anthony Fox, "Directions in ISA specification," *Interactive Theorem Proving (ITP)*, pp. 338–344, 2012.

[23] Ulan Degenbaev, "Formal Specification of the x86 Instruction Set Architecture," 2012.

[24] J S. Moore, "A mechanically checked proof of a multiprocessor result via a uniprocessor view," *Formal Methods in System Design*, vol. 14, no. 2, pp. 213–228, 1999.

[25] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer, "Formal verification of a microkernel used in dependable software systems," in *Computer Safety, Reliability, and Security*. Springer, 2009, pp. 187–200.

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and Others, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[27] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan, "Rocksalt: Better, faster, stronger SFI for the x86," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 395–404. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254111

[28] Coq, "Coq proof assistant," http://coq.inria.fr/.

[29] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong, "Certifying low-level programs with hardware interrupts and preemptive threads," *Journal of Automated Reasoning*, vol. 42, no. 2, pp. 301–347, 2009.

[30] Malcolm Dowse, Andrew Butterfield, Marko van Eekelen, and Maarten de Mol, "Towards machine-verified proofs for I/O," *Technical Report 0408 in the Proceedings of Implementation and Application of Functional Languages, 16th International Workshop, IFL'04, Lübeck, Germany.*, pp. 469–480., September 8-10, 2004.

[31] Maarten De Mol, Marko Van Eekelen, and Rinus Plasmeijer, "The mathematical foundation of the proof assistant Sparkle," 2007, Technical Report ICIS-R07025, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands.

[32] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky, "Trace-based verification of imperative programs with I/O," *Journal of Symbolic Computation,*, vol. 46, no. 2, pp. 95–118., 2011.

[33] J. Lim and T. Reps, "Tsl: A system for generating abstract interpreters and its application to machine-code analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, no. 1, p. 4, 2013.