Analysis of x86 Application and System Programs via Machine-Code Verification

Shilpi Goel

Abstract

Ensuring the reliability of high-level programs is becoming more difficult with the ever-increasing complexity of computing systems. Although automatic program analysis tools have found great success in industrial applications, they have limited scope because they target specific kinds of undesirable behavior, such as buffer overflows. Incorrect compiler transformations and the absence of analysis systems for some high-level languages suggest that a lowest-common-denominator strategy, namely machine-code analysis, will facilitate extensive program verification.

For this dissertation project, we are developing a general framework for the mechanical verification of software by analyzing its corresponding machine (binary) code. To this end, we are specifying a formal and executable model of the x86 instruction set architecture (ISA) using the ACL2 theorem-proving system. Notably, this model can be used for the functional verification of application as well as system programs. If successful, this project will provide the capability to mechanically verify a wide variety of properties of x86 programs, including their correctness with respect to behavior, security, and resource requirements. We have already used this x86 ISA model to prove the correctness of some application programs, such as a binary-level word-count program that makes system calls to obtain its input. This binary program was obtained by compiling the corresponding source program using the LLVM compiler.

We will demonstrate the capabilities of this framework by verifying system programs that access and modify system data structures, such as memory management structures.

1 Overview

The need for scalable tools for analyzing program behavior is indisputable, given the rising complexity and use of software systems. Faulty software can jeopardize human life; defective programs can adversely impact our security and can incur enormous cost to medical, financial, and transportation industries. Static and dynamic analysis tools [1, 2, 3, 4, 5, 6] for software bug-hunting are available commercially. Despite successes in detecting serious vulnerabilities, like buffer overflows [7, 8, 9], stronger guarantees about software are needed; we
need a general, scalable mechanism to ensure that programs behave as expected at run-time.

Most program analysis frameworks are targeted at high-level languages [10, 11, 12, 13, 14]. In the case where only a machine-code program is available (e.g., an executable downloaded from the Internet), these frameworks cannot be used directly for analysis. When high-level source code is procurable, the utility of verifying it in isolation is often questioned; for example, the memory semantics being offered by a programming language are eventually provided by the underlying memory system. If the semantics of the underlying system are not considered, we cannot guarantee that a program will behave as expected. Also, the assurances obtained by verifying high-level code are only valid if the compiler transformations are correct. Even with the advent of verified mainstream compilers [15], it is not possible to guarantee that a program does not access or modify unauthorized memory locations; this is because programs rely on low-level operating system (OS) code for services like memory management [16, 17, 18, 19], and often, such low-level code is written in assembly language. Therefore, it is prudent to analyze machine (binary) code for modern commercially available processors. A big benefit of machine-code analysis tools is their universal applicability; they can be used to verify all programs that can compile down to the supported hardware platform.

Through my dissertation project, I propose to demonstrate that machine-code analysis, coupled with an efficient verification infrastructure, is capable of reasoning about code that has not been (or rather, cannot be) analyzed by existing automatic program analysis tools due to their limited scope. My focus is on verification via mechanical proof that programs meet their specifications.

We are developing a formal and executable model of the x86 instruction set architecture (ISA) using the ACL2 theorem-proving system [20, 21], for both simulation and formal verification of x86 machine code. There are three main thrusts in this project: to develop an accurate formal model of the x86 ISA without any simplification in its semantics, to reduce the trade-off between simulation efficiency and effective mechanical reasoning, and to make machine-code verification practical. We have used the current version of the x86 ISA model to simulate and verify application programs, including those that exhibit non-determinism [22]. We will demonstrate the capabilities of the framework by verifying system programs that access and update system data structures—in particular, those used by the memory management facilities like paging and segmentation. A successful project will yield a robust framework for formal analysis of machine code that will provide the capability to mechanically verify a wide variety of complex properties of x86 programs, including their correctness with respect to behavior, security, and resource requirements.

The rest of this document is organized as follows. I present my proposal in Section 2. I describe the current status of my project in Section 3, and discuss future work in Section 4. Related work is described in Section 5. I conclude with a discussion of evaluation criteria in Section 6.
2 Proposed Project

Modern programs rely on many complex properties for their correct execution. Such properties demand formal scrutiny, simply due to their complexity. Some examples are:

- Are kernel addresses inaccessible from the user space?
- Is it possible for the user data to be inadvertently altered by the kernel?
- Are processes isolated from each other?
- Is a page frame in the physical memory mapped to exactly one page in the linear memory, unless memory sharing is desired?
- Is it possible for the stack or the heap to overwrite the program in some execution?
- Can the memory resource usage be expressed as a function of program inputs, and if so, can we determine that function?

In this dissertation project, we will address the problem of verifying such properties. We will develop a formal, executable model of a significant subset of the x86 ISA in the ACL2 programming language. This ISA model will execute x86 machine-code programs, including those generated by compilers like GCC [23] and LLVM [24], and we will reason about these machine-code programs using the ACL2 theorem-proving system.

There are three main tasks in this project:

1. Building a formal, executable x86 ISA model,
2. Developing a framework for the formal analysis of x86 machine-code programs based on this model, and
3. Employing this framework to verify application and system programs.

This ACL2-based x86 ISA model will enable the formal verification of code that cannot be verified by existing automatic tools such as static and dynamic analyzers. Many of these tools, though successful in carrying out their particular functions, like detecting memory safety violations, do not take specifications as input. Thus, they cannot even state the complex properties mentioned above. Such properties are difficult, if not impossible, to verify using existing automatic tools, including SAT/SMT solvers and model checkers. Interactive theorem-proving, coupled with such automated strategies, has the potential to address the verification of these properties and thus, it can facilitate the development and delivery of secure systems.

If successful, this project will provide the capability, for the first time, to mechanically verify both application and system programs using a robust formal framework based on an accurate model of the x86 ISA. We will verify unmodified x86 machine-code programs—no simplification of the semantics of x86 instructions or mechanisms like memory management will be made. The success of this project will demonstrate that mechanical verification of complex properties, including functional correctness of programs, can be made practical.
by using a machine-code analysis framework based on a theorem-proving system such as ACL2.

In spite of its complexity, there are two main reasons for choosing the x86 ISA over other available ISAs for this project.

- Since x86 is the dominant processor architecture for non-embedded devices, a simulation and verification framework based on the x86 ISA can find immediate practical application.
- The x86 ISA is one of the most complicated processor architectures, and hence, successful completion of the project will demonstrate that our formal methods technology is mature enough to handle real-world, industrial problems.

I explain the reasons behind our choice to design and develop a verification framework based on a formal and executable x86 ISA model in Section 2.1, and I describe in detail the tasks of this project in Section 2.2.

2.1 Motivation for Our Approach

Inspection of the behavior of machine code is often done by employing instruction set simulators [25, 26], which use sophisticated techniques to achieve high-speed simulation of machine instructions and are written in efficient programming languages like C and C++. Even though such simulators are commonly used as ISA reference models, they are not formal specifications of these ISAs and hence, do not directly provide an infrastructure for formal code analysis.

ISA models written in formal specification languages allow the direct application of formal reasoning tools. Verification using these formal models indeed increases confidence in the reliability of machine code, but this analysis can not be fully trusted until the model is known to be faithful to the processor. One way to increase confidence in the accuracy of a model is by performing co-simulations to validate it against a real machine. Co-simulation is the process of executing a machine program on the processor as well as on the model; if the state of the processor and the model are the same after every instruction, then the model is faithful to the processor for at least those instructions (and data). Unfortunately, formal specification languages do not usually provide an efficient execution environment. Model validation can also be done by performing extensive code reviews—an approach that is independent of the model’s execution efficiency. However, code reviews are time and labor intensive. The reviewers would need to study the vendors’ ISA manuals, which are long documents mostly consisting of prose [27, 28]. There is always the danger of making subjective judgments about the processor’s behavior, especially if the processor has a complicated ISA and if some ISA features are under-specified in the vendor documents. Therefore, code reviews can supplement co-simulations, but not supplant them.

This comparison of ISA simulators and formal ISA specifications justifies the need to develop a formal ISA model whose execution speed allows efficient
co-simulations for model validation. A validated model increases trust in the applicability of the results of formal verification. Another benefit of such a unified model is that it reduces the overhead associated with designing, developing, and maintaining two separate models—in particular, maintaining them consistently. The ACL2 theorem-proving system is an excellent fit for this endeavor. ACL2 is a mathematical logic (a first-order logic of recursive functions) based on an applicative subset of Common Lisp as well as a mechanical theorem prover used to prove theorems in that logic. ACL2 is also a programming language, which offers the execution efficiency provided by underlying Lisp compilers. ACL2 has proved its mettle as an industrial-strength system, and is regularly used in both academic and commercial applications [29].

An alternative approach to developing an ISA model for program verification is to build translators that convert high-speed simulators into formal specifications, which can be used for reasoning. Although this approach of employing translators is in use in the hardware and software industry [30, 31, 32], we focus on the direct approach of mathematically specifying the system under consideration. Although this direct approach requires model validation, but unlike the translator approach, it does not require trusting (or verifying) a translation process and trusting the simulator itself.

It should be mentioned that for all its benefits discussed in Section 1, formal verification of machine code has few practitioners [33, 34, 35]. Information contained in the abstractions provided by high-level languages, like the layout or “shape” of data structures, is mostly lost at the machine-code level, thereby making machine-code analysis detail-intensive and tedious. Work has been done to provably lift low-level verification to a higher-level reasoning process [36]. In this project, we will ameliorate this problem by developing and providing ACL2 libraries that aid in automating machine-code verification, as discussed in Section 2.2.2.

2.2 Proposed Tasks

2.2.1 x86 ISA model

Model Development and Validation

The utility of a formal ISA model for performing machine-code verification depends directly on the model’s completeness (with respect to the ISA features specified), accuracy, and reasoning and execution efficiency. Developing an x86 ISA model will require considerable engineering effort. Intel software developer’s manuals [27] will be used as specification documents, and ambiguities can be resolved by cross-referencing AMD manuals [28] and running tests on real machines to understand processor behavior. Although formally specifying the x86 ISA in its entirety would be ideal, it is beyond the scope of this dissertation project. Given the multitude of features in the x86 ISA that include those left over from its early days to ensure backward compatibility, even deciding which subset of this large and complicated architecture to model is a challenge. Since most modern software relies on 64-bit computing, the focus of this dissertation
is the 64-bit sub-mode of Intel's IA-32e mode [37]. Some components of the x86 ISA state that need to be modeled in order to make progress towards the program verification task, discussed in Section 2.2.3, include:

- General-purpose registers (rax, rcx, etc.), segment registers (cs, ss, etc.), control registers (cr0, cr3, etc.), machine-specific registers (ia32.efer, ia32.fs.base, etc.)
- 64-bit instruction pointer (rip)
- 64-bit flags (rflags)
- Memory management registers (gdtr—global descriptor table register, ldtr—local descriptor table register, etc.)
- Physical and linear memory address space

Other capabilities that need to be supported include all the addressing modes, memory management via paging and segmentation, and the instruction fetch-decode-execute cycle.

Specifying this subset of the x86 ISA is a formidable task in itself. Though these features and capabilities are described in the Intel manuals (which are around 3500 pages long), this description is mostly in English prose. How can we know that our model faithfully represents the x86 ISA? To increase trust in the model’s accuracy, this task also requires developing a co-simulation framework to validate the model against the real processor. Debugging utilities like the GNU Debugger (GDB) [38] and program instrumentation tools like Pin [39] will be used to examine the registers and memory on the real processor. Similar tools to observe changes to the state need to be built for the x86 model. Comparison between the states of the processor and the model should be possible at any level of granularity, be it at the termination of the program or after every x86 instruction or at certain breakpoints. In order to facilitate efficient co-simulations, the model should have a high execution speed.

Supporting Reasoning and Execution

This task will lead to scientific investigations about designing and organizing large models that are both formal and executable. The benefits of a unified model for simulation and formal verification come at a price: we need to mitigate the trade-off between execution and reasoning efficiency. Simple definitions enable easier reasoning but generally offer poor execution performance, and definitions optimized for execution efficiency are typically more difficult to verify because they use a sufficiently different algorithm from the one used in the specification. Abstraction techniques, either provided by ACL2 in the form of features like mbe [40] and abstract stobjs [41] or those developed over the course of this task, need to be leveraged to build an efficient model for both reasoning and execution.
Modes of Operation

The complexity of the x86 ISA model will increase as more features are added to it, and this added complexity will make reasoning inevitably more involved. The issue of balancing verification effort and verification utility is a very pertinent one. For example, users might not want to reason about an application program at the level of physical memory, i.e., taking into account address translations and access rights management provided by the memory management data structures. This is because it is customary for application programs not to have direct access to the system data structures. The memory model seen by application programs is that of linear memory, which is an OS-constructed abstraction of the complicated underlying memory management mechanisms like paging and segmentation that are based on physical memory. Therefore, verification of application programs can be performed at the level of linear memory, if the OS routines that manage the linear memory abstraction can be either trusted or proved correct. However, the verification of system programs, like kernel routines, must necessarily be done at the level of physical memory since these programs can access/modify system data structures.

In light of the above, the x86 ISA model should provide the option to deactivate some features of the ISA, enabling the user to do varying depths of analysis, depending on the kind of programs being considered for verification. Specifically, the x86 model should have the following modes of operation:

1. **Programmer-level Mode:** This mode of the model will attempt to provide the same environment for reasoning as is provided by an OS for programming. It will allow the verification of an application program while assuming that memory management, I/O operations, and services offered via system calls are provided reliably by the underlying OS. The memory model in this mode will support 64-bit linear addresses specified for IA-32e machines. A specification of system calls like `read`, `write`, `open`, and `close` will also be included in the programmer-level mode.

2. **System-level Mode:** This mode will include the specification for IA-32e paging and segmentation; in particular, ISA-prescribed data structures for memory management will be available in this mode. The memory model in this mode will characterize a \(2^{52}\)-byte physical address space, which is the largest physical address space provided by modern x86 implementations. This mode is intended to be used to simulate and verify software that has supervisor privileges and interacts with I/O devices.

An added benefit of having two separate modes of operation will be increased execution speed of programs in the programmer-level mode; this is because executing these programs will not require simulating both the physical address

---

1 The access rights of an application program are ultimately governed by the OS, but it is extremely rare (and inadvisable) for an application program to have direct access to system resources.
space (and hence, accesses/updates to the paging and segmentation data structures) and the linear address space.

It would be beneficial, not to mention interesting, to verify whether the programmer-level mode is an abstraction of the system-level mode, given that the system data structures have been set up correctly. Establishing this relationship between the two modes is out of the scope of this project. However, a future research project could be to state and prove this relationship using the analysis framework developed over the course of this dissertation project (described below in Section 2.2.2).

2.2.2 Machine-Code Analysis Framework

Reasoning about machine code is an arduous process—there are complicated mechanisms like memory and I/O management, fetch-decode-execute cycle, etc. operating on a machine state, which contains a multitude of registers, flags, and a large memory model. As such, a robust program verification framework will be developed in ACL2 to reduce the time and manual effort required to prove properties of machine-code programs. Developing a machine-code analysis framework that is capable of reasoning about arbitrarily complex program properties is both a research and an engineering problem. Under this task, we will focus on the development of a general lemma database and a set of general strategies to reason about memory management mechanisms.

Lemma Database

The heuristics of the ACL2 theorem prover are influenced by the database of known theorems/rules, and developing a lemma database with efficient rules is critical to automate verification. Also, our lemma libraries need to be as general as possible in order to facilitate their re-use.

A program’s behavior can be described by the effects it has to the state of the machine. Given an initial state $x86_i$, the final state $x86_f$ may be described as a nest of updates made in program order to $x86_i$. In order to reason about the behavior of a program, we need to develop lemma libraries to read from, write to, and re-arrange these nests of updates. Broadly speaking, these libraries will contain the following types of theorems:

- **Read-over-Write Theorems**: There are two types of Read-Over-Write theorems. The first describes the independence or non-interference of different components of the x86 state. One example is proving that an update made to a specific register does not modify the value of any other component of the x86 state. The second asserts that reading a component after it is modified returns the value that was written to it during the modification.

- **Write-over-Write Theorems**: Like the Read-Over-Write theorems, there are two types of Write-over-Write theorems. The first asserts that independent writes to the x86 state can commute safely. The second asserts
that if consecutive writes are made to a component, the most recent write is the only visible write.

- **Preservation Theorems**: These types of theorems assert that writing a valid value to a component in a valid x86 state returns a modified x86 state that is still valid.

Libraries dedicated to reasoning about memory accesses and updates deserve special mention. Here we refer to linear memory reads/writes in the programmer-level mode and physical memory reads/writes in the system-level mode of the x86 ISA model. Linear memory reads/writes in the system-level mode are considerably more involved and are discussed later in this section alongside memory management mechanisms. Physical memory reads/writes in the programmer-level mode are unavailable since linear memory is the only specified memory model. Reasoning about a program’s memory reads/writes is a crucial part of establishing whether that program meets its specification. An example of such a memory-related analysis is determining whether a program is reading from and writing to the intended memory locations; e.g., if some data values were previously stored at memory addresses \( x \) to \( (x + y) \), does the machine instruction that intends to modify these data values reference addresses \( x \) to \( (x + y) \)? Reasoning about memory reads/writes is challenging because memory is usually accessed or updated in “clusters”. Symbolic memory addresses are used during the verification of position-independent code, and they can make reasoning even more difficult. Thus, the libraries need to include lemmas about non-interference (or overlap, if applicable) of memory regions represented by symbolic values in addition to those about specific memory addresses.

We expect the libraries to be extended as new lemmas are discovered during the third task (Section 2.2.3), i.e., during the verification of programs.

**Reasoning about Memory Management Mechanisms**

Another important memory-related analysis task is establishing if a program has the appropriate rights to memory regions it needs access to during runtime and, conversely, if it has any rights to regions that should be off limits. The implementation of access control management by the memory management unit on most processors, including x86, is accomplished by segmentation and paging [42, 43]. See Figures 1, 2, and 3 for a pictorial representation of segmentation and IA-32e paging. We briefly describe segmentation and paging on x86 machines below.

**Segmentation** divides a processor’s linear address space into protected segments, which can contain code or data. A logical address is used to reference a byte in a particular segment; a logical address consists of a segment selector and an offset. A segment selector points to a descriptor in segment data structures called Global or Local Descriptor Tables (GDT/LDT). Segment descriptors contain the base address of the segment along with other information related to access control and privilege levels. The offset of the logical address is added to this segment base address to point to a byte in the linear address space. The
Logical Address or Far Pointer
Offset or
Segment Near Pointer or
Selector Effective Address

Segment Selector
Logical Address or Far Pointer
Descriptor Table(s)
Segment Descriptor
Linear Memory
Global or Local
Descriptor Table Register

Figure 1: Segmentation & IA-32e Paging with 1GB page configuration

SEGMENTATION

Control Register
has the base address
of these structures.
Linear Memory
Segment
Selector
Logical Address or Far Pointer
Descriptor
Table(s)
Segment
Descriptor
Global or Local
Descriptor Table Register

Figure 2: IA-32e Paging with 2MB page configuration

Figure 3: IA-32e Paging with 4KB page configuration
x86 ISA specifies six 16-bit segment registers (CS, DS, ES, FS, GS, and SS) to hold segment selectors. However, in 64-bit mode, the segment base address of all segment registers except FS and GS is treated as zero.

Paging is used to simulate a large linear address space with a smaller physical RAM. This virtualization of linear address space presents a simpler memory interface to application programs by hiding the details of physical memory resource management, which is managed by OS routines. When IA-32e paging is used, each segment is divided into pages (or frames) of size 4KB, 2MB, or 1GB. An OS tracks these pages via hierarchical data structures (page directories and tables). When a linear memory access is made, these data structures are “walked” to obtain the translation to the corresponding physical address. A page-fault exception is generated if the required page is located in secondary storage. In response to this exception, the OS swaps in the required page to the physical memory; this swapping should be transparent to a program’s execution.

Besides address translation, paging data structures determine the access rights for each translation. For instance, data cannot be written to a linear memory address that is marked read-only.

It should be noted that the segmentation data structures are embedded in the linear address space and the paging data structures are in the physical address space. Also, the format and sizes of these data structures are specified by the ISA, but their initialization and maintenance is the responsibility of the OS.

Due to their central role in memory management, it is critical for these data structures to be set up and managed correctly by the OS. We need to formulate predicates to check for proper configuration of these structures. For example, a valid paging data structure should be 4K-aligned, and a valid entry in that structure is one that does not result in a page-fault exception. Such an entry must satisfy some ISA-prescribed conditions, like the present bit (which indicates that the entry either references another paging entry or maps a page frame) must be set, the reserved bits must be equal to zero, and access rights must not be violated. These predicates will serve as a specification for kernel code that initializes and modifies these structures, and they can also be used to derive assertions for OS code to obtain run-time guarantees.

We need to develop the capability to verify whether arbitrary updates to the paging entries preserve these predicates. Interestingly, we also have to prove that the predicates are preserved when these structures are walked (or traversed), because traversals can cause on-the-fly updates. Specifically, flags called the accessed and dirty flags may be modified during these traversals. If a paging structure walk is done on behalf of a linear memory read, the accessed flag is set in all the entries that are read during the translation of that linear address to its corresponding physical address. If the walk is done on behalf of a linear memory write, the dirty flag is set in the final entry that maps a frame. These flags are used by the memory-management software to manage the swapping of pages and paging data structures to/from the physical memory.

This address translation from linear to physical addresses is done for every memory access/update. This makes the proofs of linear memory theorems, like
Read-over-Write and Write-over-Write, in the context of memory management difficult, since it involves reasoning about the non-interference (or overlap, if applicable) of these hierarchical data structures.

### 2.2.3 Program Verification

Application programs have access to a smaller x86 state than system programs, and they are based on the simpler memory model of linear address space, where the underlying details of physical memory resource management are hidden. Therefore, as mentioned in Section 2.2.1 we provide different modes of operation of the x86 ISA model for the verification of application and system programs.

#### Verification in the Programmer-level Mode

The programmer-level mode of the x86 ISA model will be used for the verification of application programs while assuming that the underlying OS services are reliable. Some of these services are non-deterministic from the point of view of a programmer—different runs can yield different results on the same machine. For example, the \texttt{open} system call might open a file in one execution, but return an error in another execution if the file has been deleted. Thus, reasoning about application programs that request services from the OS or use instructions like \texttt{RDRAND} (read a random number) requires reasoning about non-deterministic behavior.

#### Verification in the System-level Mode

Verification of programs in the system-level mode will take low-level features of the x86 ISA into account. In this project, we will focus on the system features dedicated to x86 memory management. Program verification will be done in the following context: every reference to a linear address made by the program will be translated to a physical address via appropriate paging data structure walks. This will require establishing whether the predicates that recognize well-formed data structures hold for all accesses and updates to the x86 state. Discovering and formalizing properties of paging data structures, as discussed earlier in Section 2.2.2 would aid in the verification of OS kernel code, which would enable verification of application software relying on this code (unless, of course, the user chooses to trust OS routines and verifies application programs exclusively in the programmer-level mode).

Two kinds of verification activities fall under this task:

- **Verification of programs that do not directly modify any memory management data structures**: User-mode programs should not modify any system data structures. Also, apart from the kernel routines dedicated to memory management, most system programs operate in the linear address space and do not modify segment and paging data structures. For such programs, reasoning can be done in terms of the top-level linear memory accesses and updates because address translation will involve only reading
from the system data structures, and the mapping of logical to physical addresses dictated by these structures will remain unchanged. This raises the question of why a user might choose to verify these programs in the system-level mode at all. One reason is that even though address mapping might be unchanged, the system data structures provide other useful information, like access rights or cache types, that is not available directly at the level of linear memory. Another reason is that such programs might need access to parts of the state, like machine-specific registers, that are unavailable in the programmer-level mode.

There is a caveat: even when programs do not explicitly modify system data structures, the memory management unit can modify them by updating the accessed and dirty flags during paging structure walks. Updates to the accessed and dirty flags will not affect the mapping of logical addresses to physical addresses—though, of course, this would need to be proved as a part of this task, and therefore, our reasoning approach will still be applicable.

- **Verification of OS-level routines that initialize and modify the system data structures**: Verification of these routines will require full-blown reasoning about accesses and updates made to the system data structures. Reasoning can be simplified by lifting all the information in the hierarchical data structures to a single list that maps a logical address to its corresponding physical address and associated access rights. Accesses/updates to the hierarchy of data structures can be concisely captured by reads from or writes to this projection, thereby simplifying and speeding up the symbolic simulation process for the theorem prover.

Verification of programs in this mode will permit deducing the memory footprint of a program, i.e., the memory regions a program can access and modify at run-time based on the access rights prescribed in the memory-management data structures. Determining a program’s memory footprint can facilitate reasoning about the absence of unauthorized code and data access by that program. An important outcome will be the ability to compose memory footprints of different programs to determine whether they can be safely resident (linked and executed) in the memory simultaneously.

## 3 Current Status

The following publications describe in detail some of the work we have done so far:


I present the current status of this project by reporting progress in the three tasks outlined in Section 2.2.

3.1 x86 ISA Model

Model Development and Validation

Our model of the x86 ISA is written in the ACL2 specification language. We have a specification of all addressing modes, most user-level instructions, and paging in the IA-32e mode. We currently have limited support for FS- and GS-based segmentation; we discuss segmentation later in this section. As of this writing, we support more than a hundred instructions (around 220 opcodes), and the instructions that are unspecified include those that access co-processor I/O address spaces, rarely used BCD instructions, cache-related instructions, and virtual machine extensions.

The model can execute almost all user-level programs emitted by the GCC and LLVM compilers. For example, we have successfully simulated a contemporary SAT solver\(^2\) on our model; our model produced exactly the same effects on the memory and registers as those produced by the real machine. This comparison was performed by our co-simulation framework that uses GDB on a real x86 machine to capture the effects of a program and a GDB-like tool, written entirely in ACL2, to do the same on our model.

The x86 ISA model provides an interpreter-style operational semantics\[^4\], where the meaning of a machine-code program is given by a recursively-defined interpreter over the processor’s state. Components of the state in our model include: general-purpose registers, segment registers, status register, floating-point registers, machine-specific registers, control registers, instruction pointer, and a time- and space-efficient memory model. The effect of each machine instruction is specified by a semantic function, which takes an initial x86 state as input and returns an appropriately modified next state as output. A step function fetches, decodes, and executes an instruction by calling its associated instruction semantic function. Finally, a run function takes the number n of instructions to be executed and an initial x86 state. This run function either takes n steps or terminates early if an unrecoverable error is encountered, returning an appropriately modified final x86 state in either case. The semantics of an x86 machine-code program are given by the semantics of its constituent instructions,

---

\(^2\)This SAT solver, developed by Marijn J. H. Heule, has performance comparable to those of state-of-the-art solvers. However, it does not include any preprocessing techniques.
and we can prove properties about these x86 machine-code programs using the ACL2 theorem-proving system.

**Supporting Reasoning and Execution**

Similar to previous work in the ACL2 community [46, 47], considerable effort was devoted to improving the execution efficiency of our ACL2 functions by using type declarations and guards [48]. These are used to provide type information to the underlying Common Lisp compiler to enable compile-time optimizations. In ACL2, annotating functions with such type information creates proof obligations to justify whether the resulting optimizations are correct. Once these obligations are proved, unspecified behavior during execution is averted.

We use sophisticated features available in ACL2 to avoid trade-offs between reasoning and execution efficiency. One such example is the introduction of a layer of abstraction in our model so that the low-level layer can be optimized for execution performance and the high-level layer for reasoning efficiency. We prove an invariant that a suitable correspondence is maintained between the two layers at all times. The lower layer is built around ACL2 structured objects called concrete stobjs (Single-Threaded OBJects) [49] that are used to model the x86 state. Concrete stobjs are mutable objects with applicative semantics; hence, they offer high performance by allowing destructive assignments while providing copy-on-write semantics for reasoning. Reasoning can be simplified by the use of abstract stobjs [41], which can provide a simple interface to a corresponding concrete stobj. Our top-level functions use the simpler abstract definitions during reasoning and the optimized concrete definitions during simulation.

Measured on a machine with Intel Xeon E31280 CPU @ 3.50GHz, the simulation speed of the model is around 330,000 to 3.3 million instructions/second, depending on whether simulation is being done at the level of physical address space (where address translation is performed via paging) or linear address space, respectively. To our knowledge, this is the fastest formal simulator of its kind.

**Modes of Operation**

As discussed in Section 2.2.1, the x86 ISA model has two modes of operation: programmer-level mode and system-level mode.

**Programmer-level Mode:** This mode is intended to be used for the verification of application programs. It supports the 64-bit linear address space, and also provides some support for 64-bit segmentation. Segment descriptor tables (GDT/LDT) are unavailable in this mode, but if a logical address refers to the FS or GS segments (which are the only segments that can have non-zero bases in the 64-bit mode), the segment base addresses are obtained from the machine-specific registers ia32_fs_base or ia32_gs_base, respectively. Thus, it is assumed that these registers contain the appropriate values. This is a
valid assumption—ia32_fs_base and ia32_gs_base are actually the “hidden” parts (or cache) of these segment registers. All the segment registers are associated with a hidden cache to store the corresponding information from the appropriate segment descriptor in GDT/LDT. This cache is loaded automatically by the processor when the visible part (i.e., the 16-bit selector) is loaded. It is the responsibility of system software to load the visible part of segment registers and, consequently, the hidden part, if the descriptor tables have been modified. Thus, assuming that the registers ia32_fs_base and ia32_gs_base are appropriately initialized is just another instance of this mode providing an environment similar to that provided by the OS for developing and executing application programs.

As system calls are non-deterministic, they pose an interesting challenge both for simulation and verification. Consequently, the x86 instructions syscall and sysret are special in this mode. The syscall instruction is used by application programs to call system procedures at a higher privilege level in order to request services from the underlying OS, and the sysret instruction is used by the system programs to return control to the application-level program that requested some service. In the programmer-level mode, the semantic function of syscall is extended to provide the semantics of system calls like read and write, and the instruction sysret is unavailable for use.

For simulations, we set up an execution mode in the programmer-level mode to use the results returned by functions that invoke the system calls by interacting directly with the underlying OS. That is, simulation of all instructions except syscalls occurs within ACL2; for syscalls, we escape out of ACL2 to get the “real” results. However, these functions are impure, and hence, not logical functions because they can return different values for the same input arguments. For reasoning, we set up a logical mode where an environment field env is incorporated into the x86 state to represent the part of the external world that affects or is affected by system calls. Among other things, the env field consists of an oracle that specifies the result of any non-deterministic or random action. To reason about a system call’s effects, the env field is consulted. More details can be found in our FMCAD’14 paper [22].

The following connection exists between the logical and the execution modes of the programmer-level mode. Let x86_i be an x86 state. Suppose in the execution mode, the evaluation of \texttt{run}(x86_i) returns x86_f and updates the real environment from ENV_i to ENV_f. Then, the following is true for the logical mode: if env_i corresponds to ENV_i, and x86_i' refers to x86_i augmented with env_i, then the evaluation of \texttt{run}(x86_i') in the logical mode produces x86_f', which is x86_f augmented with env_f, for some env_f corresponding to ENV_f. To ensure that this connection holds for our model in the programmer-level mode, we perform co-simulations by comparing program runs in the execution mode to corresponding runs in the logical mode.

System-level Mode: This mode is intended to be used for program verification at the level of physical address space. It models a 52-bit physical address space. It does not support segmentation data structures yet, but includes the specification
of IA-32e paging for all configurations (4KB, 2MB, and 1GB pages) of paging data structures, including predicates that check whether paging structure entries are valid. As mentioned earlier, some requirements for an entry being valid are: the reserved bits must be zero, access rights must not be violated, and the present bit must be set.

A linear memory address, coupled with information necessary to check access rights like the origin \(^3\) of the address, is used as input to the paging mechanism. A linear memory access will report a failure in our model if a page-fault exception is encountered—exceptions are currently unsupported by our x86 ISA model. We have specified a paging walk by defining it in terms of traversals of each of the hierarchical paging structures. These traversal functions also perform on-the-fly updates to the accessed and dirty flags, as specified in the x86 ISA.

For example, let \(la\) be the linear address to be translated. The paging specification functions perform the following main actions; Figures 1, 2, and 3 represent these actions pictorially. We assume that \(pa\) is the physical address that corresponds to \(la\), i.e., the translation of \(la\) does not result in a page-fault exception.

- Read the control register \(cr3\) to obtain the physical address that points to the base of the first structure, called the Page Map Level 4 (PML4) table.
- A portion of \(la\) is used to index into the PML4 table to locate the PML4 entry that governs the translation of \(la\). If this entry is valid, we read a portion of it to obtain the base address of the next paging structure, called Page Directory Pointer (PDP) table.
- Another portion of \(la\) is used to index into the PDP table to locate the relevant PDP entry. If this entry is valid and if its PS (page size) bit is equal to one, then this entry contains the base address of a 1GB page. The rest of \(la\) is added to this base address to obtain \(pa\). If PS is zero, this entry contains the base address of the next paging structure, called the Page Directory.
- Another portion of \(la\) is used to index into the Page Directory to locate the relevant entry. If this entry is valid and if its PS bit is equal to one, then this entry contains the base address of a 2M page. The rest of \(la\) is added to this base address to obtain \(pa\). If PS is zero, then this entry contains the base address of the next paging structure, called the Page Table.
- Another portion of \(la\) is used to locate the relevant entry in the Page Table. If this entry is valid, it contains the base address of the 4K page, and the rest of \(la\) is added to it to obtain \(pa\).

The paging specification functions that model the above actions have been optimized for execution efficiency by using type declarations and guards, which

\(^3\)Determining the origin of a linear memory address constitutes answering questions like: is the linear memory reference on behalf of a read, a write, or an instruction fetch/decode/execute? What is the privilege level, as determined by the segmentation data structures, of this linear memory reference?
need to be verified in ACL2. For these specification functions, this requires verifying properties of arithmetic and bit-vector operations. We use pre-existing ACL2 libraries [50] to aid in this task.

Currently, our x86 ISA model is more than 45,000 lines of executable ACL2 logic. The complexity of the ISA and the size of our specification makes documentation a critical part of the project. All the currently implemented features in our x86 ISA model have been documented using XDOC [51], a documentation tool for ACL2.

3.2 Machine-Code Analysis Framework

Lemma Database

We have developed macros to automatically generate and prove Read-over-Write, Write-over-Write, and Preservation theorems about different components of the x86 state. Automating this process makes it convenient to add previously unsupported components to the model’s state. We have also developed general libraries to reason about non-interference/overlap of memory regions. These libraries are mature enough to reason about non-trivial properties of a given program automatically; some examples are the independence of the stack and heap from the program and data, modification of only the intended regions of memory by a program, etc.

Reasoning about Memory Management Mechanisms

We have made progress in reasoning about paging data structure traversals. We have formulated predicates that recognize a valid paging structure entry, and proved that if these predicates hold on a given entry, walking that entry will not result in a page-fault exception. We have also proved that despite on-the-fly updates to these data structures during traversals, walking these valid entries repeatedly does not modify the address mapping. These lemmas have allowed us to prove a Read-over-Write theorem about linear memory references in the context of memory management. This Read-over-Write theorem states that reading from a linear address $x$ after some write was made to another “distinct” linear address $y$ returns the same value as it would if no write was made to $y$ in the first place. For this theorem, the notion of $y$ being distinct is more than just $y$ being different from $x$; it also means that the paging entries that govern the translation of $y$ do not overlap with those that govern the translation of $x$. Of course, we would need to prove more general Read-over-Write theorems (e.g., those that say that some entries governing the translation of $y$ are the same as those governing the translation of $x$) in order to support reasoning at the level of linear memory reads/writes in the system-level mode.
3.3 Program Verification

Verification in the Programmer-level Mode

We can automatically verify snippets of straight-line machine code using a BDD or SAT-based symbolic execution proof engine in ACL2 called GL \[52, 53\]. Providing the specification is the primary user requirement in order to prove theorems using this framework. If a conjecture fails, GL can compute counterexamples. We have verified a complicated popcount (bit-count) program, based on Anderson’s bit-twiddling hacks \[54\], using this technique, and we were successful in reporting the bug in its incorrect version \[44\]. Such a “bit-blasting” technique for reasoning about straight-line code snippets facilitates compositional verification, thereby reducing the time and effort required to reason about larger programs.

We have successfully performed the mechanical verification of a word-count program that computes the number of characters, words, and lines in a stream read in using the read system call \[22\]. We wrote a trio of simple ACL2 specification functions that compute the character, word, and line counts of a string. The final theorem 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 symbolically running the word-count program on our x86 model. Our lemma libraries reduced the manual effort required for machine-code verification of this program substantially, as demonstrated by some empirical evidence below.

- Lines of ACL2 needed to verify the word count program:
  - Without the libraries: approximately 20K lines
  - With the libraries: approximately 8K lines

Even though 8K lines may still seem excessive, it should be noted that more than half of these 8K lines were generated by ACL2 in response to requests to simplify specific symbolic expressions. Even these simplified expressions are large because there are many updates to the x86 state.

Using our lemma database, we automated the proof of disjointness of the word-count program and its stack in every execution. We determined that irrespective of the size of the input file, this program always uses a fixed amount of memory on the stack to compute and store the character, word, and line counts. 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. Note that the numbers above for the lines of ACL2 needed to verify this program include all these theorems too.

Verification in the System-level Mode

We have begun preparing the system-level mode of the x86 ISA model for program verification by developing a database of lemmas about memory manage-
ment. However, non-trivial verification effort in this mode is some work away, as discussed in the future work section below (Section 4).

4 Future Work

I describe the work remaining in this dissertation project in terms of the three tasks outlined in Sections 2.2 and 3. I also present a projected timeline of our future work. Other future research directions that are not within the scope of this project are discussed in Section 6.

4.1 Proposed Tasks

4.1.1 x86 ISA Model

In order to verify critical kernel routines, the x86 ISA model needs to support more features like supervisor-mode instructions (rdmsr, wrmsr, sysret, etc.), I/O instructions (in, out, etc.), and local, global, and interrupt descriptor tables (LDT/GDT/IDT) so that segmentation, traps, interrupts, and call gates can be implemented. SIMD (single instruction, multiple data) and floating-point instructions might also have to be supported. Note that our current model already includes more than 220 x86 opcodes. We plan to support new features when required—if our target program uses an unsupported feature, we will specify that feature to enable the program’s simulation and verification.

4.1.2 Machine-Code Analysis Framework

We have already verified some application programs using our framework. Before embarking on system code verification, we have to state and prove some key lemmas about paging and segmentation data structure traversals and updates. Some examples are: can we determine that there is no overlap among these data structures? How do these structures, especially the paging structures, change when traversals are made? How can we lift reasoning about traversals of and updates to the paging structures so that top-level Read-over-Write and Write-over-Write theorems about linear memory references in the context of memory management can be proved?

The machine-code analysis framework will also include theorems about the new features that will be added to the x86 ISA model. For example, we may be required to support reasoning about how the system descriptor tables embedded in the linear address space are affected when changes are made to the underlying paging data structures.

4.1.3 Program Verification

Once the x86 ISA model and the machine-code analysis framework are ready for the verification of system programs, verification of a kernel program that
accesses and manipulates the memory management data structures will be attempted. An example is an optimized data-copy program whose specification states that it should successfully copy a chunk of data values from a linear memory location $x$ to a disjoint linear memory location $y$. However, the implementation of this program may be optimized such that the paging data structure entries for both $x$ and $y$ point to the same pages initially. A copy is created when needed, i.e., only if changes to the data are made. This is accomplished by employing the *copy-on-write* technique—the shared pages are marked read-only and an exception is thrown when an attempt is made to write to them. This exception acts as a signal to the kernel to create a copy and modify the paging structure entries of the program accordingly. The optimized data-copy program falls into the category of *zero copy* programs [55, 56, 57] that avoid making redundant data copies when possible. Our verification objective would be two-fold: one, to prove that when data is not modified, this data-copy program returns an x86 state where the data values at locations $x$ and $y$ are the same; and two, if some of the data values located at $x$ (or $y$) are modified, then the data values at locations $x$ and $y$ differ only for those locations where these modifications were made.

During the course of this effort, we expect both to revisit the modeling choices made during the specification of new ISA features and to add to or improve the lemma database. Verification of such a program involves reasoning about complicated x86 features like address translation, access control management, exceptions, and interrupts. Therefore, a successful verification effort will serve as a proof of concept that our libraries indeed provide the capabilities that we proposed.

This project ambitiously aims to make formal analysis of machine code a large step closer to being a practical choice in development of serious software systems. Our work so far has already demonstrated that it is feasible to develop realistic formal models of complicated systems like the x86 ISA without compromising on accuracy; these models can also function as simulators. Progress made in the verification of real application programs provides confidence that practical verification of system programs is an attainable goal.

4.2 Timeline

We envision the following timeline of the future work.

*Spring 2015 – Summer 2015*: Specifying more x86 instructions; modeling the system descriptor tables to support segmentation and interrupts; formulating and proving properties about paging data structure traversals and modifications

*Fall 2015*: Choosing and simulating system program(s), such as an optimized data-copy program; this would identify the x86 features that need to be modeled in order to support the program’s execution and verification
Spring 2016: Verification of the target program(s)—this includes discovering and specifying properties of interest; it may also involve re-visiting modeling choices made earlier

Summer 2016 – Fall 2016: Dissertation writing and final defense

5 Related Work

Though program verification has a long history, with Turing’s 1949 paper [58] being one of the earliest works on program correctness, serious interest in machine-code verification arose only in the last three decades. This section describes efforts pertaining to low-level code verification that are relevant to the proposed work. However, it should be noted that compiler bugs, an issue with source-code verification (as discussed in Section 1), can be resolved by developing verified compilers [59, 60, 61, 62] or verifying compilers [63, 64]. Work has also been done to verify compiler transformations for concurrent programs [65]. Such compilers are not mainstream tools yet because they offer considerably less performance than their counterparts like GCC or LLVM. That said, CompCert [15, 66] is improving the state of the art: it is a verified compiler for a subset of C that targets x86, ARM, and PowerPC processors. Such a solution would, of course, be necessary for every source language of interest, while the x86 machine-code verification approach applies whenever the high-level language has a compiler targeting x86. Another noteworthy work is the “CLI short stack” [67], a stack of mechanically verified systems in the NQTHM theorem prover [68] that includes a gate-level microprocessor design called FM8502 [69], an assembler for an assembly language called Piton [70] that targets this microprocessor, and higher-level language, micro-Gypsy [71], that targets Piton. The CLI stack, though composed of systems that are far simpler than modern ones, set a milestone in the history of designing and verifying computing systems, from software all the way down to hardware.

Clutterbuck and Carré stressed the importance of solving the problem of low-level code verification. They analyzed Intel 8080 machine code by using a proof checker to discharge proof obligations generated by a Verification Condition Generator (VCG) [72]. Bevier developed a simple multitasking operating system kernel for a von Neumann machine and verified some of the services offered by this kernel [73] using the NQTHM theorem prover. Boyer and Yu used NQTHM to formalize most of the user-mode instruction set of a commercial Motorola MC68020 microprocessor, and then verified the binary code produced by compiling the Berkeley String library using GCC [74]. Though proofs done in this undertaking were time and effort intensive, this work is a landmark in the history of machine-code verification performed using theorem-proving techniques.

Attemps to reduce the overhead associated with low-level verification have been made by employing compositional verification techniques and specialized Hoare-style logics. Matthews et al. mechanized assertional reasoning by imple-
menting a VCG in ACL2 and made heavy use of compositional reasoning to verify low-level (e.g., Java bytecode) programs [75]. Feng et al. used the Coq proof assistant [76] to verify machine code on a simplified formal model of the x86 processor using domain-specific and separation logics [77]. Myreen’s “decompilation into logic” technique [36, 33], developed in HOL [78], eases the effort required for low-level verification by reducing the problem of reasoning about machine code to reasoning about simpler logic functions. Decompilation takes machine code as input, and produces logic functions capturing the functional behavior of machine code and a theorem that relates the machine code to these functions. This technique has been used in the Jitawa project [79], to produce a verified Lisp runtime for Milawa [80], a self-verifying theorem prover. Decompilation can be used for verifying machine code on different architectures [81]. It should be noted that unlike this proposed work, Myreen’s technique is used for the verification of user-level programs only, and does not (yet) have a reasoning methodology for non-deterministic behavior or address translation. Moore has developed a tool called Codewalker [82, 83] in ACL2, which also implements decompilation—it allows formal exploration of code in any programming language that has been specified by an ACL2 model.

Some other efforts to analyze machine-code programs include work done by Morrisett et al. in software fault isolation [34]. This entailed developing a Coq-based x86 ISA specification that can be used for machine-code verification. This specification is not directly executable—an executable OCaml simulator (with an execution rate of around 50 instructions/second) has to be extracted from the Coq code. Reps et al. [84] developed a sophisticated system, TSL, that can create re-targetable tools for different types of data flow analyses on machine code. Proof-Carrying Code (PCC) [85, 86] and Typed Assembly Language [87] are used to obtain some fixed safety properties (like type safety) of low-level code.

Efforts targeted towards supervisor-mode code verification deserve special mention; sel4 is the “world’s first operating-system kernel with an end-to-end proof of implementation correctness and security enforcement” [88, 89]. Shao et al. focus on clean-slate development of OS kernels [90, 91] by employing PCC and modular verification. In contrast, the proposed work focuses on developing tools and techniques for the verification of existing software. Alkassar et al. formulated properties of paging data structures and TLBs, and proved shadow page table algorithms correct [92]. Note that this work was done at the level of C language, using VCC [10].

Reasoning about traversals and updates to data structures has received considerable attention in the ACL2 community itself. Greve and Wilding had issued a challenge to the ACL2 users to devise efficient ways to reason about complex and pointer-rich dynamic data structures in a linear address space [93]. Specifically, they called for efficient solutions for proving non-interference properties of data structures. An example of a research question they posed was: does a non-interference proof scale quadratically with the number of entries in the data structure, and if so, can we do better? Sophisticated solutions were posted to answer such questions. Moore used memory taggings [94] to mark how
each memory address of interest will be interpreted by accessor and updater functions. These tags were then used to formulate and prove non-interference theorems. Liu took the approach of proving that accesses and updates do not affect the format and sizes of the data structures under consideration, and lifted reasoning about on-the-fly updates to a sequence of updates in order to separate traversals from explicit modifications [95]. Greve used address enumeration techniques [96], supported by libraries that aid in specification of memory regions [97], to collect a set of all the addresses that specify a data structure so that disjointness properties can be stated and proved about these memory locations of interest. In our work, we intend to leverage these solutions to reason about data structures embedded in both linear and physical address spaces. Our work will be somewhat more involved in the system-level mode of the x86 ISA model because in addition to reasoning about system data structures in the physical address space, we would need to reason about user-defined data structures in the linear address space, which is ultimately implemented on top of those system data structures.

Formal analysis of machine code often entails the development of a formal model of a processor ISA. Such a model can also be used as a target specification for microprocessor design verification [98, 99, 69, 100]. Building accurate models of processor ISAs is a challenging task in itself. There have been investigations into developing domain-specific languages [101, 102] to facilitate clear and precise specification of ISAs, even by non-experts.

Hunt’s ACL2-based Y86 model [103] can be called the precursor of the work proposed in this dissertation project; the Y86 [104] is a simple 32-bit x86-like processor that was developed by Bryant and O’Hallaron for pedagogical purposes. The main contribution of the proposed work will be a unified x86 ISA model for both binary program simulation and formal analysis of application and system programs.

6 Evaluation Criteria

The proposed project will involve the development of ACL2 libraries that contain x86 ISA specifications, strategies to aid in automated machine-code verification, and proofs of program correctness. Exact simulation of programs by our x86 ISA model (i.e., programs producing the exact same effects on the model as on the real processor during execution) will demonstrate its accuracy. Such simulations will provide evidence that no simplification of the semantics of the x86 ISA was done while developing the model. As is the case with conventional instruction set simulators, sophisticated debugging and program instrumentation tools will be available to the user.

This effort should be considered a success if we deliver these libraries, including a correctness proof for a non-trivial program with supervisor privileges, for example, one similar in complexity to a kernel routine used for memory management. These proofs will serve as a proof of concept that our libraries provide the capabilities that we are proposing. Our goal is to reduce the overhead as-
associated with reasoning without compromising on precision. The quality and accuracy of our framework will facilitate its use for future verification efforts. Comprehensive documentation will also contribute towards the usability of our framework.

**Issues Beyond the Scope of this Project:** An interesting research task could be to investigate ways to lift machine-code verification to a higher-level analysis effort. Though our work does involve automating proofs of low-level details to develop a scalable reasoning framework, we will not focus on developing techniques similar to decompilation \[33\] [82] to make machine-code verification accessible to users without any prior machine-code reasoning experience. Another issue that is out of scope of this project is verifying whether the programmer-level mode is an abstraction of the system-level mode, as mentioned before in Section 2.2.1. Since our ISA model is that of a uniprocessor x86 machine, we will not be reasoning about concurrency-related issues in this work. Accounting for the behavior of caches will also not be addressed during the timeline sketched above. It needs to be emphasized that our x86 model will serve as a formal specification of the complicated x86 ISA. Unlike vendor documents, the model is composed of formulas in ACL2; it can be thought of as a “formula manual” which can be referenced to infer concrete information about the x86 ISA. This formal specification can be used to verify machine-code programs (a compile-to specification), as is proposed in this project, as well as to determine whether a micro-architecture implements this ISA (a build-to specification), which should be considered as another possible future direction not in the scope of this project.

The potential for future research using our framework appears to be high.

**Acknowledgments**

This material is supported by DARPA under contract number N66001-10-2-4087. Parts of this material also appear in an NSF proposal titled “TWC: Small: Memory Analysis and Machine-Code Verification Techniques for Multiprocessor Systems” for the program “Secure and Trustworthy Cyberspace” (SaTC). I thank Prof. Warren Hunt and Dr. Matt Kaufmann for their guidance and support. I also thank Cuong Kim Chau, Soumava Ghosh, Dr. Marijn Heule, Prof. Warren Hunt, Dr. Matt Kaufmann, Robert Krug, Prof. J Moore, Ben Selfridge, Nathan Wetzler, and Dr. William Young for their feedback on and/or contributions to the evolving x86 ISA model.

**References**


[23] GCC, the GNU Compiler Collection. [https://gcc.gnu.org/](https://gcc.gnu.org/)


[38] GNU. GDB: The GNU Project Debugger. 

http://www.gnu.org/software/gdb/


