MIT-CSAIL-TR-2006-035 May 11, 2006

A Machine-Checked Safety Proof for a CISC-Compatible SFI Technique

Stephen McCamant

A Machine-Checked Safety Proof for a CISC-Compatible SFI Technique

Stephen McCamant
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139
smcc@csail.mit.edu

Abstract

Executing untrusted code while preserving security requires that the code be prevented from modifying memory or executing instructions except as explicitly allowed. Software-based fault isolation (SFI) or "sandboxing" enforces such a policy by rewriting code at the instruction level. In previous work, we developed a new SFI technique that is applicable to CISC architectures such as the Intel IA-32, based on enforcing additional alignment constraints to avoid difficulties with variable-length instructions. This report describes a machine-checked proof we developed to increase our confidence in the safety provided by the technique. The proof, constructed for a simplified model of the technique using the ACL2 theorem proving environment, certifies that if the code rewriting has been checked to have been performed correctly, the resulting program cannot perform a dangerous operation when run. We describe the high-level structure of the proof, then give the intermediate lemmas with interspersed commentary, and finally evaluate the process of the proof's construction.

1 Introduction

Software-based fault isolation (SFI) or "sandboxing" is a technique to enforce memory and control-flow safety policies on untrusted machine code by rewriting it at the instruction level. The original sandboxing technique of Wahbe et al. [WLAG93] is applicable only to RISC architectures; our previous work [MM05] presents an SFI technique that is applicable to CISC architectures like the IA-32 (Intel i386 and successors, or "x86" for short). The technique, embodied in an implementation named PittSFIeld, is optimized to reduce run-time overhead, and achieves a small trusted computing base by using a separate verification pass. To further validate the safety of the technique, we have constructed a machine-checked proof to verify that when applied to a subset of the x86 instruction set, the technique enforces the intended security policy. The present report is a companion to [MM06], expanding on the brief summary of the proof given there, explaining the design decisions behind it and presenting the steps of the proof in almost complete detail.

The key obstacle preventing the application of SFI to an architecture like the x86 is the possibility of jumping into the middle of an instruction. Unlike RISC architectures, where instructions must be aligned (typically on a 4-byte boundary), x86 instructions can potentially begin at any byte, and are a variable number of bytes long. Typically code has a single stream of intended instructions, each following directly after the last, but by starting at a byte in the middle of an intended instruction, the processor can read an alternate stream of instructions, generally nonsensical. If code were allowed to jump to any byte offset, the SFI implementation would need to check the safety of all of these alternate instruction streams; but this would be infeasible. To avoid this problem, the PittSFIeld tool artificially enforces its own alignment constraints on the x86 architecture. Conceptually, we divide memory into segments we call *chunks* whose size and location is a power of two, say 16, bytes. PittSFIeld inserts no-op instructions as padding so that no instruction crosses a chunk boundary; every 16-byte aligned address holds a valid instruction. Instructions

```
nop           mov addr, %eax      xchg %eax, %ebx
inc %eax      mov %eax, addr      xchg %eax, %ebp
jmp addr      and $immed, %ebx    mov %eax, (%ebx)
jmp *%ebx     and $immed, %ebp    mov %eax, (%ebp)
```

Figure 1: The subset of x86 instructions considered in the proof

that are targets of jumps are put at the beginning of chunks, and jump instructions are checked so that their target addresses always have their low 4 bits zero. This transformation means that each chunk is an atomic unit of execution with respect to incoming jumps: it is impossible to execute the second instruction of a chunk without executing the first. To ensure that an otherwise unsafe operation and the check of its operand cannot be separated, PittSFIeld additionally enforces that such pairs of instructions do not cross chunk boundaries, making them atomic. For instance, to prevent illegal memory writes, indirect stores are replaced by a pair of instructions, the first of which ensures that the address used by the second is in bounds; as long as the instructions are in the same chunk, the check cannot be avoided.

The formal proof described here consists of two parts. The first is a simplified model of the behavior of an x86 processor, and of the code properties the technique enforces; these are expressed in a restricted programming language. The second is a proof of a specific property enjoyed by the combination of the technique's checks and the processor's behavior: namely, if a sequence of instructions is approved by the checks, the execution of those instructions will not perform an unsafe operation. The proof proper takes the form of a sequence of lemmas, each of which can be mechanically checked to follow from the preceding ones. The first part, the model, is the one that must be understood to gauge the significance of the result: the proof is meaningful only if the model reflects the most important aspects of the real system. In some sense the details of the proof proper are irrelevant: once they have been checked by a trusted machine verifier, the truth of the result is independent of the proof techniques used. However, understanding why the safety property holds can be useful for understanding the system, for instance when contemplating modifications.

The remainder of this report is organized as follows. Section 2 gives an informal description of the model and the statement of the safety property, followed by the corresponding formal details in Section 3. Section 4 gives an informal description of the proof proper; the formal details then follow in Section 5. Finally, Sections 6, 7, and 8 briefly evaluate the proof's development process, mention related work, and conclude.

2 About the model

We have constructed the model and proof using ACL2, a system whose syntax and semantics are based on a subset of Common Lisp. ACL2's modeling language feels like a programming language, and is executable, but has a number of restrictions that make proofs more tractable: there are only a few data types (we use integers and lists), there are no side effects, higher-order functions are unavailable, and all recursion must be provably terminating. The ability to execute the model is useful for testing, but the model for which a proof is easiest is not necessarily one that executes efficiently: our model generally favors ease of proof.

The first half of the model is a simulator for a subset of the x86 instruction set. A list of the instructions covered in the present version of the proof is shown in Figure 1. The state of the processor in our model consists of 2^{32} bytes of memory (stored sparsely), eight general-purpose registers (some not currently used), and the program counter. (The flags register is included in the data structure for future expansion but not modelled because it cannot be observed by any of the current instructions.) A step function named `step-x86` is the core of the simulator: it maps from one machine state to the subsequent one.

To model the safety policy, two kinds of actions by the processor are treated specially. First are unsafe operations, such as writing in an illegal area, jumping to an illegal address, or attempting to execute an unknown instruction. Unsafe operations cause the processor to immediately halt: `step-x86` returns a distinguished false value `nil`. Second are safely trapped operations: ones that should not be performed by a correct program, but which the real system assumes can be caught by the hardware when they occur. For instance, the technique assumes that areas of memory called *guard regions* cause a trap when read from or written to (in the real system, their pages are left unmapped); such access causes the untrusted code to terminate safely. Since termination is already used to indicate an unsafe state, safe termination is modelled by nontermination: specifically, by a looping state whose successor state is itself.

PittSFIeld is designed so that the security-critical component is not the rewriting tool used by the untrusted code author, but a separate verifier that examines the untrusted code just before execution to check that it has been rewritten correctly. It is this verifier that is included in the model, in the form of a predicate `mem-sandbox-ok` on the contents of memory. The predicate checks that no instruction overlaps a chunk boundary (to be precise, that a sequential disassembly from the beginning of the code region includes an instruction at each chunk-aligned address), and that each instruction satisfies the safety predicate `insn-sandbox-ok`. However, instructions cannot be checked purely in isolation: their safety depends on the preceding instructions. This dependence is captured by a *static invariant*, which represents facts that should hold on any execution upon reaching a particular point in the code. The static invariant is computed by a simple state machine, which examines the instructions in the code region sequentially.

3 Model

This section, as well as Section 5.1, shows portions of the ACL2 code for the proof in typewriter font, interspersed with commentary in roman font. Commentary applies to the subsequent code fragment.

The types `byte` and `addr` represent 8-bit and 32-bit quantities represented as unsigned integers. A memory (abbreviated "mem") is a sparse mapping from 32-bit addresses to 8-bit bytes, implemented as an association list. Missing entries represent `0x00`.

```
(defun byte-p (b) (unsigned-byte-p 8 b))
(defun addr-p (a) (unsigned-byte-p 32 a))
(defalist mem-p (l) (addr-p . byte-p))

(defun make-mem () nil)
(defun mem-fetch (mem addr)
  (if (bound? addr mem)
      (binding addr mem)
    #x00))
(defun mem-store (mem addr val)
  (bind addr val mem))
```

The relevant processor state consists of the eight general-purpose registers, the program counter `eip`, the flags register `eflags`, and the memory. (`eflags`, `ecx`, `edx`, `esi`, `edi`, and `esp` are not used in the instruction subset currently formalized.) Among the things omitted include the floating-point registers, the multimedia extension registers, the segment pointers, and various debugging, system, and tracing registers. The `:assert` clauses in this definition specify that to be a legal `x86-state` (i.e., to satisfy `x86-state-p`), each field of the state must have the appropriate type.

```
(defstructure x86-state
  (eip (:assert (addr-p eip)))
  (eflags (:assert (addr-p eflags)))
  (eax (:assert (addr-p eax)))
  (ebx (:assert (addr-p ebx)))
  (ecx (:assert (addr-p ecx)))
  (edx (:assert (addr-p edx)))
  (esi (:assert (addr-p esi)))
  (edi (:assert (addr-p edi)))
  (ebp (:assert (addr-p ebp)))
  (esp (:assert (addr-p esp)))
  (mem (:assert (mem-p mem))))

(defun x86-state-fetch (x86 addr)
  (mem-fetch (x86-state-mem x86) addr))
```

In the real system, we have the code region runs by default from `0x10000000` to `0x10ffffff` (16MB in total). To allow faster explicit calculation in the model, its code region is only 256 bytes long. Symbolic constants such as the address of the code region are represented by zero-argument functions: since the language of ACL2 is purely functional, the can be relied upon to always have the same value. This also allows the use of ACL2's enable/disable features to choose whether to treat the constants as symbolic or use their values in any particular proof.

```
(defun code-start () #x10000000)
(defun code-size () (expt 2 8))
(defun code-end () (+ (code-start) (code-size)))
(defun guard-size () (expt 2 16))
```

Chunks in the model are 16 bytes long and 16-byte aligned, so the low 4 bits of the first address in each chunk are zero.

```
(defun chunk-size-bits () 4)
(defun chunk-size () (expt 2 (chunk-size-bits)))

(defun code-region-p (addr)
  (and (addr-p addr) (>= addr (code-start)) (< addr (code-end))))
```

The code and data regions each have *guard regions* directly before and after them. The presence of the guard regions ensures that small offsets from legal addresses are also safe.

```
(defun code-guard-p (addr)
  (or (and (>= addr (code-end)) (< addr (+ (code-end) (guard-size))))
      (and (>= addr (- (code-start) (guard-size))) (< addr (code-start)))))
```

`(loghead k n)` is the k low bits of n, i.e. $n \,\&\, (2^k - 1)$. Thus, this function checks whether the low 4 bits are all zero.

```
(defun chunk-aligned-p (addr)
  (= (loghead (chunk-size-bits) addr) 0))
```

The real system does not require that the code and data regions be the same size, but it is convenient to have the data region be small as well.

```
(defun data-start () #x20000000)
(defun data-size () (expt 2 8))
(defun data-end () (+ (data-start) (data-size)))

(defun data-region-p (addr)
  (and (addr-p addr) (>= addr (data-start)) (< addr (data-end))))

(defun data-guard-p (addr)
  (or (and (>= addr (data-end)) (< addr (+ (data-end) (guard-size))))
      (and (>= addr (- (data-start) (guard-size))) (< addr (data-start)))))
```

The *zero-tag region* plays a role similar to the guard regions: it allows the technique to enforce memory safety just by turning address bits off. It always starts at address 0, and must be at least as large as either of the code or data regions.

```
(defun zero-tag-size () (expt 2 8))
(defun zero-tag-end () (zero-tag-size))
```

```
(defun zero-tag-region-p (addr)
  (and (addr-p addr) (>= addr 0) (< addr (zero-tag-end))))
```

One subtle point is that the zero tag region also requires its own guard regions. Because address arithmetic wraps around, the guard region 'before' the zero-tag region actually lies at the top of memory.

```
(defun zero-tag-guard-p (addr)
  (or (and (>= addr (zero-tag-end)) (< addr (+ (zero-tag-end) (guard-size))))
      (and (>= addr (- (expt 2 32) (guard-size))) (< addr (expt 2 32)))))
```

Addition modulo 2^{32}.

```
(defun +-32 (x y) (loghead 32 (+ x y)))
```

Compute the length in bytes of the instruction starting at `eip`. Here and in the function `regular-step`, the real x86 instruction encoding is used. See that definition for comments giving the mnemonics for each instruction.

Unknown instructions are treated as having length 1. An alternative would be to return a distinguished value (such as `nil`) in this case, but this would complicate many later definitions. It seems sufficient to treat what is in fact a single unrecognized instruction as a sequence of unrecognized ones: the safety check will eventually reject such code in any case.

```
(defun insn-len (mem eip)
  (let ((byte1 (mem-fetch mem eip))
        (byte2 (mem-fetch mem (+-32 1 eip)))
        (byte3 (mem-fetch mem (+-32 2 eip))))
    (cond ((= byte1 #x90) 1)
          ((and (= byte1 #x89) (= byte2 #xf6)) 2)
          ((and (= byte1 #x8d) (= byte2 #x76) (= byte3 #x00)) 3)
          ((= byte1 #xa1) 5)
          ((= byte1 #xa3) 5)
          ((= byte1 #xe9) 5)
          ((= byte1 #xeb) 2)
          ((and (= byte1 #x81) (= byte2 #xe3)) 6)
          ((and (= byte1 #x81) (= byte2 #xe5)) 6)
          ((= byte1 #x93) 1)
          ((= byte1 #x95) 1)
          ((and (= byte1 #x89) (= byte2 #x03)) 2)
          ((and (= byte1 #x89) (= byte2 #x45) (= byte3 #x00)) 3)
          ((and (= byte1 #xff) (= byte2 #xe3)) 2)
          (t 1))))
```

Read a 32-bit little-endian word from the specified address, as an unsigned integer.

```
(defun fetch-word-le (mem addr)
  (+ (mem-fetch mem addr)
     (* (expt 2 8) (mem-fetch mem (+-32 addr 1)))
     (* (expt 2 16) (mem-fetch mem (+-32 addr 2)))
     (* (expt 2 24) (mem-fetch mem (+-32 addr 3)))))
```

Store a 32-bit word at the specified address. The `logtail` function is the natural counterpart to `loghead`: `(logtail k n)` computes $\lfloor n/2^k \rfloor$.

```
(defun store-word-le (mem addr word)
  (mem-store
   (mem-store
    (mem-store
     (mem-store mem addr (loghead 8 word))
     (+-32 addr 1) (loghead 8 (logtail 8 word)))
    (+-32 addr 2) (loghead 8 (logtail 16 word)))
   (+-32 addr 3) (loghead 8 (logtail 24 word))))
```

Interpret an unsigned byte as a signed byte in two's complement.

The ACL2 functions `nfix` and `ifix` ensure that their result are a natural number or an integer, respectively, by replacing any argument that is not with zero. This ensures that the function's result has the desired type, even if one of the arguments does not.

```
(defun sign-byte (byte)
  (if (<= byte #x7f)
      (nfix byte)
      (ifix (- byte #x100))))
```

Interpret an unsigned word as a signed word.

```
(defun sign-word (word)
  (if (<= word #x7fffffff)
      (nfix word)
      (ifix (- word #x100000000))))
```

This function bundles up the common checks that happen every time the model writes to memory. Writing to a guard region is assumed to cause execution to halt in a safe way, modelled by having the step function loop right before the write. Writing any where else outside the data or zero-tag region causes the execution to halt unsafely, modelled by the step function returning nil.

```
(defun do-write (x86 loop eaddr mem value)
  (cond ((or (data-region-p eaddr) (zero-tag-region-p eaddr))
         (update-x86-state x86 :mem (store-word-le mem eaddr value)))
        ((data-guard-p eaddr) loop)
        (t nil)))
```

Execute one step of the idealized processor, corresponding to a single instruction. If the instruction is unsafe (including if it is unknown), returns nil. If the instruction would in reality cause safe termination, it returns the state unchanged. Otherwise, it returns the state after execution of the instruction, which includes incrementing the program counter `eip` and updating registers or memory appropriately.

```
(defun regular-step (x86)
  (let* ((eip (x86-state-eip x86))
         (mem (x86-state-mem x86))
         (len (insn-len mem eip))
         (next-eip (+-32 eip len))
         (x86-2 (update-x86-state x86 :eip next-eip))
         (insn1 (x86-state-fetch x86 eip))
         (insn2 (x86-state-fetch x86 (+-32 1 eip)))
         (insn3 (x86-state-fetch x86 (+-32 2 eip))))
```

```
(cond
 ;; nop
 ((= insn1 #x90) x86-2)
 ;; 2-byte nop: mov %esi, %esi
 ((and (= insn1 #x89) (= insn2 #xf6)) x86-2)
 ;; 3-byte nop: lea 0x0(%esi),%esi
 ((and (= insn1 #x8d) (= insn2 #x76) (= insn3 #x00) x86-2))
 ;; inc %eax
 ((= insn1 #x40)
  (update-x86-state x86-2 :eax (+-32 1 (x86-state-eax x86))))
 ;; Direct read: mov addr, %eax
 ((= insn1 #xa1)
  (update-x86-state x86-2 :eax (fetch-word-le
                                mem
                                (fetch-word-le mem (+-32 1 eip)))))
 ;; Direct write: mov %eax, addr
 ((= insn1 #xa3)
  (do-write x86-2 x86 (fetch-word-le mem (+-32 1 eip)) mem
            (x86-state-eax x86)))
 ;; Direct jump: jmp addr (32-bit offset)
 ((= insn1 #xe9)
  (update-x86-state
   x86 :eip (+-32 next-eip
                  (sign-word (fetch-word-le mem (+-32 1 eip))))))
 ;; Direct jump: jmp addr (8-bit offset)
 ((= insn1 #xeb)
  (update-x86-state
   x86 :eip (+-32 next-eip
                  (sign-byte (mem-fetch mem (+-32 1 eip))))))
 ;; and <32-bit immed>, %ebx
 ((and (= insn1 #x81) (= insn2 #xe3))
  (update-x86-state x86-2 :ebx (logand (fetch-word-le mem (+-32 2 eip))
                                       (x86-state-ebx x86-2))))
 ;; and <32-bit immed>, %ebp
 ((and (= insn1 #x81) (= insn2 #xe5))
  (update-x86-state x86-2 :ebp (logand (fetch-word-le mem (+-32 2 eip))
                                       (x86-state-ebp x86-2))))
 ;; xchg %eax, %ebx
 ((= insn1 #x93)
  (update-x86-state x86-2 :eax (x86-state-ebx x86-2)
                    :ebx (x86-state-eax x86-2)))
 ;; xchg %eax, %ebp
 ((= insn1 #x95)
  (update-x86-state x86-2 :eax (x86-state-ebp x86-2)
                    :ebp (x86-state-eax x86-2)))
 ;; mov %eax, (%ebx)
 ((and (= insn1 #x89) (= insn2 #x03))
  (do-write x86-2 x86 (x86-state-ebx x86) mem
            (x86-state-eax x86)))
 ;; mov %eax, (%ebp)
 ((and (= insn1 #x89) (= insn2 #x45) (= insn3 #x00))
  (do-write x86-2 x86 (x86-state-ebp x86) mem
            (x86-state-eax x86)))
 ;; jmp *%ebx
 ((and (= insn1 #xff) (= insn2 #xe3))
  (update-x86-state x86 :eip (x86-state-ebx x86)))
 ;; Otherwise, crash
 (t nil))))
```

The complete processor step function includes a number of checks on the program counter, which can cause execution to halt safely or unsafely even before the instruction is decoded. (Put another way, an unsafe jump causes execution to terminate not on the jump instruction, but immediately before the target instruction; this choice reduces the number of checks needed.)

```
(defun step-x86 (x86)
  (let* ((eip (x86-state-eip x86)))
    (cond ((code-guard-p eip) x86)
          ((zero-tag-region-p eip) x86)
          ((not (code-region-p eip)) nil)
          (t (regular-step x86)))))
```

Return the kth instruction among those that the sandboxing check considers (when `(code-start)` is passed for `eip`). If k is large enough that there is no kth instruction, returns nil. The `declare` annotation is a hint that lets ACL2 know how to be sure that this recursive function eventually terminates: the value of the parameter k decreases on each recursive call. The test `(zp k)` is an ACL2 idiom that tests the condition $k = 0$ but treats any value of k that is not a natural number as zero. This is also needed to verify termination: ACL2 functions must terminate even if their arguments have the wrong type.

```
(defun kth-insn-from (mem eip k)
  (declare (xargs :measure (acl2-count k)))
  (cond ((not (code-region-p eip)) nil)
        ((zp k) eip)
        (t
         (let ((prev (kth-insn-from mem eip (- k 1))))
           (if (not (code-region-p prev))
               nil
             (let ((ret (+-32 prev (insn-len mem prev))))
               (if (code-region-p ret) ret nil)))))))
```

Here and elsewhere, the suffix `-rec` refers to a recursive function with an additional argument that implements a similarly named function without the suffix. Such loops are often needed where a more declarative specification would use universal quantification.

```
(defun seq-reachable-rec (mem eip k)
  (if (zp k) (if (= eip (code-start)) 0 nil)
    (let ((kth-insn (kth-insn-from mem (code-start) k)))
    (or (and kth-insn (= eip kth-insn) k)
        (seq-reachable-rec mem eip (- k 1))))))
```

Is the instruction at `eip` among those that the sandboxing check considers, namely one of the instructions in the sandboxed code region that could be reached by a sequential decompilation starting from `(code-start)`?

```
(defun seq-reachable-p (mem eip)
  (and (code-region-p eip)
       (not (null (seq-reachable-rec mem eip (code-size))))))
```

This kind of structure represents the state of the static verification state machine at any point in the instruction sequence.

```
(defstructure static-inv
  (ebx-data-safe (:assert (booleanp ebx-data-safe)))
  (ebx-code-safe (:assert (booleanp ebx-code-safe)))
  (ebp-data-safe (:assert (booleanp ebp-data-safe))))
```

The static invariant that holds before the beginning of the code.

```
(defun base-static-inv ()
  (static-inv nil nil t))
```

Check that a dynamic machine state satisfies the given static invariant. These functions are not actually used in the statement of the safety theorem, only in the proof, but we include them here as an aid to understanding the meanings of the static invariant.

```
(defun check-ebx-data-inv (inv x86)
  (implies (static-inv-ebx-data-safe inv)
           (or (data-region-p (x86-state-ebx x86))
               (zero-tag-region-p (x86-state-ebx x86)))))

(defun check-ebx-code-inv (inv x86)
  (implies (static-inv-ebx-code-safe inv)
           (and (chunk-aligned-p (x86-state-ebx x86))
                (or (code-region-p (x86-state-ebx x86))
                    (zero-tag-region-p (x86-state-ebx x86))))))

(defun check-ebp-inv (inv x86)
  (implies (static-inv-ebp-data-safe inv)
           (or (data-region-p (x86-state-ebp x86))
               (zero-tag-region-p (x86-state-ebp x86)))))

(defun check-static-inv (inv x86)
  (and (check-ebx-data-inv inv x86)
       (check-ebx-code-inv inv x86)
       (check-ebp-inv inv x86)))
```

Turn off all the aspects of the static invariant that hold only for the duration of a single instruction: for instance, `ebx` must be checked in the instruction immediately before its use.

```
(defun reset-static-inv (inv)
  (static-inv nil nil (static-inv-ebp-data-safe inv)))
```

Because the verification is sequential, the processor must be in a relatively safe state before any jump. For instance, the frame pointer `ebp` must point at the data region.

```
(defun ok-to-jump (inv)
  (static-inv-ebp-data-safe inv))
```

Update the static invariant according to the effect of a single instruction (this is the transition function if you think of it as a state machine). This function recognizes the three special `and` instructions that are used to sandbox pointers. It also recognizes the one instruction in the implemented subset that can corrupt `ebp`. Modifications of `ebx` do not need to be recognized because the safety invariants for `ebx` expire after a single instruction in any case.

```
(defun static-inv-trans (mem eip old-inv)
  (let ((byte1 (mem-fetch mem eip))
        (byte2 (mem-fetch mem (+-32 1 eip))))
    (cond ((and (= byte1 #x81) (= byte2 #xe3)
                (= (fetch-word-le mem (+-32 eip 2)) #x200000ff))
           (update-static-inv old-inv :ebx-data-safe t))
          ((and (= byte1 #x81) (= byte2 #xe3)
                (= (fetch-word-le mem (+-32 eip 2)) #x100000f0))
           (update-static-inv old-inv :ebx-code-safe t))
          ((and (= byte1 #x81) (= byte2 #xe5))
           (update-static-inv old-inv :ebp-data-safe
                         (= (fetch-word-le mem (+-32 eip 2)) #x200000ff)))
          ((= byte1 #x95)
           (update-static-inv old-inv :ebp-data-safe nil))
          (t old-inv))))
```

Compute the static invariant that must hold after the execution of the kth instruction: conceptually, the invariants exists in the points between instructions. Instructions k here and elsewhere count from zero, so the invariant at the beginning of the code corresponds to $k = -1$.

```
(defun static-inv-after-kth-insn (mem k)
  (declare (xargs :measure (acl2-count (+ k 1))))
  (cond ((not (integerp k)) nil)
        ((< k -1) nil)
        ((= k -1) (base-static-inv))
        ((not (kth-insn-from mem (code-start) k)) nil)
        (t (static-inv-trans mem (kth-insn-from mem (code-start) k)
            (reset-static-inv (static-inv-after-kth-insn mem (- k 1)))))))
```

The local sandboxing rules apply only to a single instruction. This function determines whether the instruction at `eip` is legal according to those rules.

```
(defun insn-sandbox-ok-local (mem eip)
  (let ((byte1 (mem-fetch mem eip))
        (byte2 (mem-fetch mem (+-32 1 eip)))
        (byte3 (mem-fetch mem (+-32 2 eip))))
    (cond ((= byte1 #x90) t)
          ((and (= byte1 #x89) (= byte2 #xf6)) t)
          ((and (= byte1 #x8d) (= byte2 #x76) (= byte3 #x00) t))
          ((= byte1 #x40) t)
          ((= byte1 #xa1) t)
          ((= byte1 #xa3)
           (data-region-p (fetch-word-le mem (+-32 eip 1))))
          ((and (= byte1 #x81) (= byte2 #xe3) t))
          ((and (= byte1 #x81) (= byte2 #xe5) t))
          ((= byte1 #x93) t)
          ((= byte1 #x95) t)
          (t nil))))
```

As above, but for those rules that depend on the static invariant: for instance, an indirect write via `ebx` is safe only if `ebx` has been checked to point at the data region.

```
(defun insn-sandbox-ok-inv (mem eip inv)
  (let ((byte1 (mem-fetch mem eip))
        (byte2 (mem-fetch mem (+-32 1 eip)))
        (byte3 (mem-fetch mem (+-32 2 eip)))
        (len (insn-len mem eip)))
    (cond ((and (= byte1 #x89) (= byte2 #x03))
           (static-inv-ebx-data-safe inv))
          ((and (= byte1 #x89) (= byte2 #x45) (= byte3 #x00))
           (static-inv-ebp-data-safe inv))
          ((and (= byte1 #xff) (= byte2 #xe3))
           (and (ok-to-jump inv)
                (static-inv-ebx-code-safe inv)))
          ((= byte1 #xe9)
           (let ((eaddr (+-32 (+ eip len)
                              (sign-word (fetch-word-le mem (+-32 eip 1))))))
             (and (ok-to-jump inv)
                  (code-region-p eaddr) (chunk-aligned-p eaddr))))
          ((= byte1 #xeb)
           (let ((eaddr (+-32 (+ eip len)
                              (sign-byte (mem-fetch mem (+-32 eip 1))))))
             (and (ok-to-jump inv)
                  (code-region-p eaddr) (chunk-aligned-p eaddr)))))))
```

This function converts from the representation of an instruction as a pointer to its first byte (a program counter value) into its index in the sequential order of instructions in the code region. If the instruction is not in the sequence, returns `nil`.

```
(defun eip-to-k (mem eip)
  (seq-reachable-rec mem eip (code-size)))
```

The static invariant that must hold before an instruction is the invariant that held after the previous instruction, with transient facts expired if the instruction is aligned (might be a jump target). Note that the use of `reset-static-inv` here is not redundant with the use in `static-inv-after-kth-insn`: that one ensures that the effect of a sandboxing instruction applies to at most one subsequent instruction in any case, while this one ensures that it applies to no subsequent instructions if the sandboxing instruction appears at the end of a chunk.

```
(defun static-inv-before (mem eip)
  (let ((inv-after (static-inv-after-kth-insn
                    mem (- (eip-to-k mem eip) 1))))
    (if (chunk-aligned-p eip)
        (reset-static-inv inv-after)
       inv-after)))
```

The local and invariant-based parts of `insn-sandbox-ok` apply to disjoint sets of instructions, but the default for unrecognized instructions is false, so it is more convenient to combine them with `or` than with `and`.

```
(defun insn-sandbox-ok (mem eip)
  (or (insn-sandbox-ok-local mem eip)
      (insn-sandbox-ok-inv mem eip (static-inv-before mem eip))))

(defun if-aligned-then-reach (mem addr)
  (or (not (code-region-p addr))
      (not (chunk-aligned-p addr))
      (seq-reachable-p mem addr)))
```

Check that each chunk-aligned address (up to the kth byte) in the code region is sequentially reachable. Though somewhat inefficient, counting by ones here makes the proof easier.

```
(defun all-aligned-reachable-rec (mem k)
  (if (zp k) (if-aligned-then-reach mem (code-start))
    (and (all-aligned-reachable-rec mem (- k 1))
         (if-aligned-then-reach mem (+ (code-start) k)))))
```

```
(defun mem-sandbox-ok-rec (mem k)
  (if (zp k) (insn-sandbox-ok mem (code-start))
    (let ((addr (kth-insn-from mem (code-start) k)))
      (and (if (and addr (code-region-p addr))
               (insn-sandbox-ok mem addr) t)
           (mem-sandbox-ok-rec mem (- k 1))))))
```

Does the entire code region in the memory pass the static sandboxing checks?

```
(defun mem-sandbox-ok (mem)
  (and (mem-sandbox-ok-rec mem (code-size))
       (all-aligned-reachable-rec mem (code-size))))
```

Execute for up to k instructions, returning `nil` if anything unsafe happens along the way.

```
(defun step-for-k (x86 k)
  (declare (xargs :measure (acl2-count k)))
  (cond ((zp k) x86)
        (t (let ((k-minus-1 (step-for-k x86 (- k 1))))
             (and k-minus-1
                  (step-x86 k-minus-1))))))
```

True if `x86` is the sort of state that we guarantee will execute safely. This predicate is in fact used only in the proof.

```
(defun is-start-state (x86)
  (and (x86-state-p x86)
       (equal (x86-state-eip x86) (code-start))
       (data-region-p (x86-state-ebp x86))
       (mem-sandbox-ok (x86-state-mem x86))))
```

The following statement is the final safety result for the system. Together with the above definitions, it represents the precise fact being proved. It states that whatever the initial configuration of the registers and memory, if the code in memory passes the static safety check, then execution of the code can continue for any number k of instructions without violating the safety policy. The ACL2 function `consp` is true for a state object (represented as a list), but false for `nil`. The hypothesis `(data-region-p ebp)` is necessary because we have defined the invariant on `ebp` to hold at the beginning of execution (see `base-static-inv` above); this matches the choice made by the real system. If the verification were defined to require the sandboxed code to establish this fact, the hypothesis could be removed.

```
(defthm safety
  (implies (and (mem-p mem)
                (mem-sandbox-ok mem)
                (data-region-p ebp)
                (addr-p eax) (addr-p ebx) (addr-p ecx) (addr-p edx)
                (addr-p esi) (addr-p edi) (addr-p ebp) (addr-p esp)
```

```
          (addr-p eflags))
     (consp
      (step-for-k
       (x86-state (code-start) eflags eax ebx ecx edx
                  esi edi ebp esp mem)
       k))))
```

4 About the proof

The proof follows a standard structure for proving a safety property of an infinite-state system: it introduces a predicate on the machine state, proves that the predicate holds in the initial state, proves that if the predicate holds in a state, it holds in the successor of that state, and finally proves that if the invariant holds on a step, the safety policy will not be violated on that step. The safety of the system for any number of steps then follows by induction. This invariant, defined by the function `safety-invariant`, is referred to as the *dynamic invariant*, to distinguish it from the static invariants that appear in the checks performed by the verifier. The relationship between the static checks performed by the verifier and the behavior captured in the dynamic invariant is analogous to the relation between a static program analysis (or a type system) and an operational semantics for a programming language: our security result is like a soundness result for a type system.

The safety invariant is essentially a conjunction of a number of properties of the machine state. The most interesting property concerns the interpretation of the static invariant: for each property set in the static invariant just before the instruction about to execute, the corresponding interpretation of that property in terms of the machine state must really hold. The remaining properties making up the invariant are much as one would expect: the program counter must be in the code region, the current instruction must be one of those found by disassembly and legal according to the sandboxing checks, and the contents of the code region must continue to pass the sandboxing checks. The high-level structure of much of the proof consists of checking that when the complete invariant holds for a machine state, then each property holds for the subsequent state: these lemmas have names ending in `-preservation`.

The proof also includes a few substantial sections devoted to proving facts that would appear immediate to a human reader, but require many steps for ACL2. One section concerns the bitwise operations used in sandboxing and their relation to the addition and ordering operations on addresses considered as integers: though ACL2 has a large library of lemmas about bitwise operations from its use in low-level verification, it lacks a powerful decision procedure for these operators, so a number of ad-hoc lemmas are still required. Another notably long derivation is needed for the fact that a store to the data region does not affect the static check performed over the code region. This would be immediate if the two areas were represented by separate variables, but they are mixed together in a single data structure representing memory. What is needed therefore is a series of lemmas that mirror the inductive structure of the safety check, proving that a data write is irrelevant at each level.

5 Safety proof

This section gives the formal statement of almost of all of the lemmas appearing in the safety proof (a few particularly uninformative technical lemmas have been omitted.) Since the purpose of these lemmas is to direct ACL2's search for a proof, the details of which it recreates on the fly, we also refer to these lemmas as a 'proof script.' In the interests of improving readability, we omit the 'hints' that provide additional direction to ACL2 as to how to prove some lemmas, though some more interesting hints will be mentioned in the commentary. We also omit extra hypotheses on the types of variables, when those types are apparent from the variable names; specifically, readers may assume the following: Variables named `mem` represent memories. Variables named `addr` or `word` or named after a 32-bit register such as `eip` or `eax` are 32-bit unsigned integers. Variables named `x86` or `x86-state` represent processor state objects satisfying x86-state-p. Variables named byte represent 8-bit unsigned bytes. Variables named inv represent static invariant objects satisfying static-inv-p. Variables named j, k, n and m represent natural numbers.

The lemmas are grouped roughly into subsections, but to preserve the property that each lemma depends only on the preceding ones, and to better enable comparison with the complete machine-checkable proof, we have not reordered lemmas. For this reason some miscellaneous lemmas appear just before their first use.

5.1 Bitwise operation lemmas

These lemmas concern bitwise operations (conventionally named with the prefix log for 'logical'), and give connections between bitwise operators applied to words and arithmetic operators applied to the values of those words as integers.

The first three lemmas are special cases of the distributivity of bitwise and over bitwise or.

```
(defthm expand-data-mask
 (equal (logand addr #x200000ff)
        (logior (logand addr #x20000000)
                (logand addr #x000000ff))))

(defthm expand-code-mask-all-bits
 (equal (logand addr #x100000ff)
        (logior (logand addr #x10000000)
                (logand addr #x000000ff))))

(defthmd expand-code-mask-rounded
  (equal (logand addr #x100000f0)
         (logior (logand addr #x10000000)
                 (logand addr #x000000f0))))
```

Facts about recursive functions in ACL2 are most easily proved if the induction needed for the proof matches the structure of the function's recursive definition. To take advantage of this, we define several alternate versions of simple functions, and prove that they are equivalent to ACL2's built-in definitions. A particularly convenient perspective for bitwise operations is to consider integers as lists of bits, least-significant first; ACL2 provides functions logcons, logcar, and logcdr for this purpose. The following function is equivalent to (expt 2 n).

```
(defun two-to-the-logcons (n)
  (if (zp n) 1
    (logcons 0 (two-to-the-logcons (- n 1)))))
```

This alternative definition of logand makes its structure of parallel bitwise operations obvious.

```
(defun my-logand* (n m)
  (if (or (zp n) (zp m)) 0
    (logcons (b-and (logcar n) (logcar m))
             (my-logand* (logcdr n) (logcdr m)))))
```

Though the logic of ACL2 is untyped, the proof engine uses a simple type system to determine whether rules are applicable, so it is often helpful to give lemmas that look like type-checking rules when introducing a new function.

```
(defthm logcons-0-to-pos-is-pos
 (implies (posp n)
          (< 0 (logcons 0 n))))
```

```
(defthm logcons-1-to-nat-is-pos
 (implies (natp n)
          (< 0 (logcons 1 n))))

(defthm two-to-the-logcons-is-pos
  (and (< 0 (two-to-the-logcons k))
       (integerp (two-to-the-logcons k))))

(defthm my-logand*-is-nonneg
  (and (<= 0 (my-logand* m n))
       (integerp (my-logand* m n))))
```

This theorem requires a somewhat unusual induction hint: we tell ACL2 to reduce `addr` by `logcdr` and `k` by decrementing in parallel at each step.

```
(defthm my-logand*-with-tttl-is-tttl-or-0-depending-on-logbitp
  (equal (my-logand* addr (two-to-the-logcons k))
         (if (logbitp k addr) (two-to-the-logcons k) 0)))
```

ACL2 uses equality theorems as directed rewrite rules, so the order of the arguments to `equal` is important. Using `defthmd` instead of `defthm` directs ACL2 to treat the generated rewrite rule as ‘disabled’: it will be used only in those proofs where it is specifically requested (as by a hint).

```
(defthmd rewrite-logand-as-my-logand*
  (equal (logand m n) (my-logand* m n)))

(defthmd rewrite-expt-2-as-two-to-the-logcons
  (equal (expt 2 k) (two-to-the-logcons k)))
```

A hint for this theorem tells ACL2 to use the two preceding rewrite rules.

```
(defthm logand-with-bit-is-bit-or-0-depending-on-logbitp
  (equal (logand addr (expt 2 k))
         (if (logbitp k addr) (expt 2 k) 0)))

(defthm logand-with-data-bit-is-bit-or-0-depending-on-logbitp
  (equal (logand addr #x20000000)
         (if (logbitp 29 addr) #x20000000 0)))

(defthm logand-with-code-bit-is-bit-or-0-depending-on-logbitp
  (equal (logand addr #x10000000)
         (if (logbitp 28 addr) #x10000000 0)))

(defun my-logor* (n m)
  (cond ((zp n) (nfix m))
        ((zp m) (nfix n))
        (t (logcons (b-ior (logcar n) (logcar m))
                    (my-logor* (logcdr n) (logcdr m))))))

(defthm my-logor*-is-nonneg
  (and (<= 0 (my-logor* m n))
       (integerp (my-logor* m n))))
```

The usual `loghead` is defined in terms of the modulus operator, but this recursive definition is more useful for proofs that consider integers just as bit strings.

```
(defun my-loghead* (k bits)
  (if (zp k) 0
    (logcons (logcar bits) (my-loghead* (- k 1) (logcdr bits)))))

(defthm my-loghead*-is-noneg
  (and (<= 0 (my-loghead* m n))
       (integerp (my-loghead* m n))))

(defthmd rewrite-logior-as-my-logor*
  (equal (logior m n) (my-logor* m n)))

(defthmd rewrite-loghead-as-my-loghead*
  (equal (loghead k bits) (my-loghead* k bits)))

(defthm logior-and-plus-same-way-beyond-loghead
 (implies (natp bit)
          (equal (logior (expt 2 (+ k bit)) (loghead bit addr))
                 (+ (expt 2 (+ k bit)) (loghead bit addr)))))

(defthm logior-plus-loghead-special-case-1
  (equal (logior #x20000000 (loghead 8 addr))
         (+ #x20000000 (loghead 8 addr))))

(defthm logior-plus-loghead-special-case-2
  (equal (logior #x10000000 (loghead 8 addr))
         (+ #x10000000 (loghead 8 addr))))

(defthm my-logand*-w-aligned-mask
 (implies (equal (my-loghead* bits mask) 0)
          (equal (my-loghead* bits (my-logand* mask addr)) 0)))

(defthm logand-w-aligned-mask
 (implies (and (natp mask)
               (natp bits)
               (equal (loghead bits mask) 0))
          (equal (loghead bits (logand mask addr)) 0)))

(defthm code-mask-is-aligned
 (equal (loghead (chunk-size-bits) #x100000f0) 0))
```

This function rounds n down to a multiple of $2^{\texttt{bits}}$.

```
(defun loground (bits n)
  (if (zp bits) n
    (logcons 0 (loground (- bits 1) (logcdr n)))))
```

While unsigned integers can be considered as lists of bits with infinitely many zeros at the most-significant tail, signed integers can be considered in two's complement as having a tail of either 0s are 1s (0 or -1). The induction is then not on natural numbers decreasing to 0, but on integers decreasing in absolute value. The `zip` function is the signed-integer counterpart to `zp`, treating non-integers to zero.

```
(defun my-lognot* (n)
  (cond ((zip n) -1)
        ((= n -1) 0)
        (t (logcons (b-not (logcar n)) (my-lognot* (logcdr n))))))
```

```
(defun my-logand2 (n m)
  (cond ((zip n) 0)
        ((zip m) 0)
        ((= n -1) m)
        ((= m -1) n)
        (t (logcons (b-and (logcar n) (logcar m))
                    (my-logand2 (logcdr n) (logcdr m))))))

(defun my-logmask* (size)
  (if (zp size) 0
    (logcons 1 (my-logmask* (- size 1)))))

(defthm my-logand2-of-my-lognot*-of-my-logmask*-is-loground
 (implies (natp bits)
          (equal (my-logand2 (my-lognot* (my-logmask* bits)) n)
                 (loground bits n))))

(defthm my-logand2-of-my-logmask*-is-loghead
  (equal (my-logand2 (my-logmask* k) addr)
         (my-loghead* k addr)))

(defthm my-logand2-w-partial-mask-is-extract-bitfield
  (equal (my-logand2 (my-lognot* (my-logmask* j))
                     (my-logand2 (my-logmask* k)
                                 addr))
         (loground j (my-loghead* k addr))))
```

This function requires an induction schema that reduces `a`, `b`, and `c` by `logcdr` simultaneously, which is helpfully provided in ACL2's library.

```
(defthm associativity-of-my-logand2
 (implies (and (integerp a) (integerp b) (integerp c))
          (equal (my-logand2 a (my-logand2 b c))
                 (my-logand2 (my-logand2 a b) c))))

(defthm my-logand2-with-240-is-4-bits
  (equal (my-logand2 #xf0 addr)
         (loground 4 (my-loghead* 8 addr))))

(defthmd rewrite-logand-as-my-logand2
  (equal (logand m n) (my-logand2 m n)))

(defthm logand-with-240-is-4-bits
  (equal (logand #xf0 addr)
         (loground 4 (loghead 8 addr))))

(defthm commute-my-loghead*-with-loground
  (equal (loground j (my-loghead* k addr))
         (my-loghead* k (loground j addr))))

(defthm loground-of-nat-is-nat
 (implies (and (natp addr)
               (natp j))
          (natp (loground j addr))))
```

```
(defthm commute-loghead-with-loground
  (equal (loground j (loghead k addr))
         (loghead k (loground j addr))))

(defthm logand-w-code-mask-region-cases
 (implies (and (not (equal (logand addr #x100000f0)
                           (loground 4 (loghead 8 addr)))))
          (equal (logand addr #x100000f0)
                 (+ #x10000000 (loground 4 (loghead 8 addr))))))
```

The intuitive statement of the following lemma would be a disjunction, but such a lemma would not work well as a rewrite rule, so we arbitrarily choose one disjunct to be the conclusion and rewrite it as an implication. (When applying the rule, ACL2 effectively makes this transformation automatically.)

```
(defthm logand-w-code-mask-is-code-or-zt
  (implies (not (zero-tag-region-p (logand addr #x100000f0)))
           (code-region-p (logand addr #x100000f0))))
```

5.2 Type-like lemmas

Though ACL2's logic is untyped, we use and define predicates to capture facts that would be part of a type system for a tool that had one: for instance, `integerp` and `x86-state-p`. The equivalent of type-checking a definition is then to prove that a function's result has the expected type whenever its arguments do.

```
(defthm fetch-byte-is-byte
 (implies (mem-p mem)
          (and
           (integerp (mem-fetch mem addr))
           (>= (mem-fetch mem addr) 0)
           (< (mem-fetch mem addr) 256))))

(defthm x86-mem-is-mem
  (implies (x86-state-p x86)
           (mem-p (x86-state-mem x86))))
```

256, 65536, 16777216, and 4294967296 are 2^8, 2^{16}, 2^{24}, and 2^{32} but they cannot be written as `(expt 2 8)` (etc.) in the statement of a theorem, because this would prevent the generated rewrite rule from ever being applicable.

```
(defthm assemble-bytes-is-word
 (let ((word (+ byte1 (* 256 byte2) (* 65536 byte3)
                (* 16777216 byte4))))
   (implies (and (byte-p byte1) (byte-p byte2)
                 (byte-p byte3) (byte-p byte4))
            (and (integerp word)
                 (>= word 0)
                 (< word 4294967296)))))

(defthm fetch-word-is-word
 (implies (mem-p mem)
          (and (integerp (fetch-word-le mem addr))
               (>= (fetch-word-le mem addr) 0)
               (< (fetch-word-le mem addr) 4294967296))))
```

```
(defthm +-32-of-ints-is-addr
 (implies (and (integerp a1) (integerp a2))
          (and (integerp (+-32 a1 a2))
               (<= 0 (+-32 a1 a2))
               (< (+-32 a1 a2) 4294967296)
               (addr-p (+-32 a1 a2)))))

(defthm store-word-is-mem
 (implies (and (mem-p mem)
               (addr-p addr)
               (addr-p word))
          (mem-p (store-word-le mem addr word))))

(defthm x86-state-int-regs
 (implies (x86-state-p x86-state)
          (and (integerp (x86-state-eax x86-state))
               (integerp (x86-state-ebx x86-state))
               (integerp (x86-state-ecx x86-state))
               (integerp (x86-state-edx x86-state))
               (integerp (x86-state-esi x86-state))
               (integerp (x86-state-edi x86-state))
               (integerp (x86-state-ebp x86-state))
               (integerp (x86-state-esp x86-state))
               (integerp (x86-state-eip x86-state))
               (integerp (x86-state-eflags x86-state)))))

(defthm state-p-type-preservation
 (implies (step-x86 x86-state)
          (x86-state-p (step-x86 x86-state))))
```

5.3 Properties of instruction counting

Instructions in the code region can be identified by two numbering schemes: the processor itself uses the address of the first byte of the instruction (variables named `eip`, after the program counter register), but for tasks like computing the predecessor of an instruction, it more useful to number the instructions sequentially from 0 (as in the argument `k` to `kth-insn-from`). Because of the possibility of overlapping instructions, such a sequential numbering may not exist in arbitrary x86 code, but it does exist for our system as long as only *sequentially reachable* instructions are considered, those instructions that would be encountered on a sequential disassembly starting at the beginning of the code region. (In the names of functions and theorems, 'sequentially reachable' is abbreviated `seq-reach`).

Periodically during development, we used a script to check whether the proof could still be verified with any lemmas removed. If so, we usually removed them; besides the aesthetic benefits of simplification, this sometimes significantly improved ACL2's running time, by pruning unproductive parts of its proof search. Earlier versions of the proof had `-1`, `-2`, and `-3` variants of this lemma, but they turned out to be unneeded.

```
(defthm code-step-doesnt-wrap-4-lh
 (implies (and (code-region-p addr)
               (<= k #x10000))
          (equal (+-32 addr k) (+ addr k))))

(defthm code-start-code-region
 (code-region-p (code-start)))
```

```
(defthm kth-insn-is-code-from-start
 (implies (kth-insn-from mem (code-start) k)
          (code-region-p (kth-insn-from mem (code-start) k))))
```

The purpose of the following lemma is to provide a bound on the length of an instruction that can be used in proofs where we do not wish to expand the definition of `insn-len` (since it consists of many cases).

```
(defthm weak-insn-len-bound
 (<= (insn-len mem insn) 65536))
```

This and later lemmas are often formulated using the recursive helper versions (names ending in `-rec`) of functions from the model rather than their fewer-argument wrapper versions. This is encouraged by ACL2's default rewriting policy, which is to expand non-recursive functions but leave recursive functions unexpanded.

```
(defthm seq-reach-of-next-insn-w-next-bound-nowrap
 (let ((next (+ eip (insn-len mem eip))))
   (implies (and (code-region-p eip)
                 (code-region-p next)
                 (seq-reachable-rec mem eip k))
            (seq-reachable-rec mem next (+ 1 k)))))
```

In this and subsequent lemma names, `ito` is an abbreviation for "in terms of".

```
(defthm kth-insn-loc-ito-k
 (implies (and (kth-insn-from mem (code-start) k))
          (and
           (>= (kth-insn-from mem (code-start) k)
               (+ (code-start) k))
           (< (kth-insn-from mem (code-start) k)
              (code-end)))))

(defthm kth-insn-loc-ito-k-beyond-end
 (implies (and (not (and
                     (>= (kth-insn-from mem (code-start) k)
                         (+ (code-start) k))
                     (< (kth-insn-from mem (code-start) k)
                        (code-end)))))
          (not (kth-insn-from mem (code-start) k))))

(defthm no-such-beyond-end
  (implies (and (>= k (code-size))
                (>= (kth-insn-from mem (code-start) k)
                    (+ (code-start) k)))
           (not (< (kth-insn-from mem (code-start) k)
                   (code-end)))))
```

The development of this lemma (using the three preceding ones) seemed awkward because of the negation, but we were not able to find a more direct proof method.

```
(defthm no-insns-beyond-size
 (implies (>= k (code-size))
          (not (kth-insn-from mem (code-start) k))))
```

```
(defthm seq-reach-next-with-size-nowrap
 (let ((next (+ eip (insn-len mem eip))))
   (implies (and (code-region-p eip)
                 (code-region-p next)
                 (seq-reachable-rec mem eip (code-size)))
            (seq-reachable-rec mem next (code-size)))))

(defthm all-aligned-reach-and-aligned-is-reach-by-k
 (implies (and (all-aligned-reachable-rec mem k)
               (> k (- addr (code-start)))
               (code-region-p addr)
               (chunk-aligned-p addr))
          (seq-reachable-rec mem addr (code-size))))

(defthm all-aligned-reach-and-aligned-is-reach
 (implies (and (all-aligned-reachable-rec mem (code-size))
               (code-region-p addr)
               (chunk-aligned-p addr))
          (seq-reachable-rec mem addr (code-size))))
```

5.4 Dynamic invariants

With the preliminaries of the preceding subsections aside, we now introduce the definitions for the main safety invariant, and prove a few of its simple properties.

```
(defun safe-loop (x86-state)
  (or (code-guard-p (x86-state-eip x86-state))
      (zero-tag-region-p (x86-state-eip x86-state))))
```

The structure of the invariant is slightly more complicated than a simple conjunction, because of the treatment of jumps to the code guard or zero-tag regions. Recall that such jumps are safely trapped, which is modelled by an infinite loop in the simulated processor, but that the loop conceptually occurs after the jump, when the program counter is pointing at the guard region. This means the proof cannot guarantee that the program counter will, for instance, always stay in the code region, as this property would not hold during a safe loop. The invariant is therefore divided into two definitions: `regular-invariant` includes those properties that hold whenever code is executing normally, and `safety-invariant` requires either that they hold, or that the code is in a safe loop. In addition, `safety-invariant` includes some other properties that hold regardless of the program counter's location.

```
(defun regular-invariant (x86-state)
  (let ((eip (x86-state-eip x86-state))
        (mem (x86-state-mem x86-state)))
    (and (code-region-p eip)
         (seq-reachable-p mem eip)
         (insn-sandbox-ok mem eip)
         (check-static-inv (static-inv-before mem eip) x86-state))))

(defun safety-invariant (x86-state)
  (and (x86-state-p x86-state)
       (mem-sandbox-ok (x86-state-mem x86-state))
       (or (safe-loop x86-state)
           (regular-invariant x86-state))))
```

Recall that `insn-sandbox-ok-local` is true only for those instructions whose safety does not depend on the static invariant: for instance, this lemma does not say anything about the safety of an indirect write.

```
(defthm local-sb-ok-implies-safe-reg-step
 (implies (and (x86-state-p x86)
               (seq-reachable-rec (x86-state-mem x86) (x86-state-eip x86)
                                  (code-size))
               (code-region-p (x86-state-eip x86))
               (insn-sandbox-ok-local (x86-state-mem x86)
                                      (x86-state-eip x86)))
          (regular-step x86)))

(defthm regular-inv-implies-safe-reg-step
 (implies (and (x86-state-p x86-state)
               (mem-sandbox-ok-rec (x86-state-mem x86-state) (code-size))
               (code-region-p (x86-state-eip x86-state))
               (regular-invariant x86-state))
          (regular-step x86-state)))

(defthm regular-inv-implies-safe-step
 (implies (and (mem-sandbox-ok-rec (x86-state-mem x86-state) (code-size))
               (regular-invariant x86-state))
          (step-x86 x86-state)))

(defthm step-from-code-is-guard-or-code-nowrap
 (let ((next (+ eip (insn-len mem eip))))
   (implies (and (code-region-p eip)
                 (not (code-guard-p next)))
            (code-region-p next))))

(defthm seq-reach-is-in-code
 (implies (seq-reachable-rec mem eip k)
          (code-region-p eip)))

(defthm seq-reach-in-sb-is-ok
 (implies (and (mem-sandbox-ok-rec mem k)
               (seq-reachable-rec mem eip k))
          (insn-sandbox-ok mem eip)))
```

5.5 Data modification irrelevance

This subsection is devoted to the fact that writing to one of the legal places to write (the data section or the zero-tag region) does not affect the results of the checks performed on the code region. Though intuitively a simple consequence of the fact that the code and data regions are disjoint, this is somewhat cumbersome to prove because both regions are represented in the same memory data structure.

```
(defthm mem-fetch-code-same-after-data-store
 (implies (and (code-region-p eip)
               (or (data-region-p addr) (zero-tag-region-p addr)))
          (equal (mem-fetch (store-word-le mem addr word) eip)
                 (mem-fetch mem eip))))
```

```
(defthm mem-fetch-code-plus-k-same-after-data-store
 (implies (and (< k 65536)
               (code-region-p eip)
               (or (data-region-p addr) (zero-tag-region-p addr)))
          (equal (mem-fetch (store-word-le mem addr word) (+ k eip))
                 (mem-fetch mem (+ k eip)))))

(defthm mem-fetch-code-plus32-k-j-same-after-data-store
 (implies (and (< k 65536)
               (< j 65536)
               (code-region-p eip)
               (or (data-region-p addr) (zero-tag-region-p addr)))
          (equal (mem-fetch (store-word-le mem addr word)
                            (+-32 (+ eip j) k))
                 (mem-fetch mem (+-32 (+ eip j) k)))))

(defthm kth-insn-same-after-data-store
 (implies (and (or (data-region-p addr) (zero-tag-region-p addr)))
          (equal (kth-insn-from (store-word-le mem addr word) (code-start) k)
                 (kth-insn-from mem (code-start) k))))

(defthm nil-not-in-code-region
  (not (code-region-p nil)))

(defthm fetch-word-code-plus-k-same-after-data-store
 (implies (and (< k 65530)
               (code-region-p eip)
               (or (data-region-p addr) (zero-tag-region-p addr)))
          (equal (fetch-word-le (store-word-le mem addr word) (+ k eip))
                 (fetch-word-le mem (+ k eip)))))

(defthm static-inv-trans-same-after-data-store
 (implies (and (or (data-region-p addr) (zero-tag-region-p addr))
               (code-region-p eip)
               (static-inv-p inv))
          (equal (static-inv-trans (store-word-le mem addr word) eip inv)
                 (static-inv-trans mem eip inv))))

(defthm static-inv-trans-is-inv
 (implies (static-inv-p inv)
          (static-inv-p (static-inv-trans mem eip inv))))

(defthm s-i-ebp-data-safe-is-bool
 (implies (or (static-inv-p inv) (not inv))
          (booleanp (static-inv-ebp-data-safe inv))))

(defthm s-i-ebp-data-safe-of-after-kth-is-bool
 (booleanp (static-inv-ebp-data-safe
            (static-inv-after-kth-insn mem k))))

(defthm static-inv-after-kth-same-after-data-store
 (implies (or (data-region-p addr) (zero-tag-region-p addr))
          (equal (static-inv-after-kth-insn (store-word-le mem addr word) k)
                 (static-inv-after-kth-insn mem k))))
```

```
(defthm seq-reach-same-after-data-store
 (implies (or (data-region-p addr) (zero-tag-region-p addr))
          (equal (seq-reachable-rec (store-word-le mem addr word) eip k)
                 (seq-reachable-rec mem eip k))))

(defthm insn-sb-ok-same-after-data-store
 (implies (and (code-region-p eip)
               (or (data-region-p addr) (zero-tag-region-p addr)))
          (equal (insn-sandbox-ok (store-word-le mem addr word) eip)
                 (insn-sandbox-ok mem eip)))

(defthm ebx-code-safe-implies-seq-reach
 (implies (and (static-inv-ebx-code-safe inv)
               (check-static-inv inv x86)
               (all-aligned-reachable-rec (x86-state-mem x86) (code-size))
               (not (zero-tag-region-p (x86-state-ebx x86))))
          (seq-reachable-rec (x86-state-mem x86) (x86-state-ebx x86)
                             (code-size))))

(defthm mem-sb-ok-after-data-store
 (implies (or (data-region-p addr) (zero-tag-region-p addr))
          (equal (mem-sandbox-ok-rec (store-word-le mem addr word) k)
                 (mem-sandbox-ok-rec mem k))))

(defthm all-align-reach-same-after-data-store
 (implies (or (data-region-p addr) (zero-tag-region-p addr)
          (equal (all-aligned-reachable-rec (store-word-le mem addr word) k)
                 (all-aligned-reachable-rec mem k))))

(defthm logand-w-data-mask-is-data-or-zt-region
 (implies (not (data-region-p (logand addr #x200000ff)))
          (zero-tag-region-p (logand addr #x200000ff))))

(defthm safe-loop-implies-stuck
  (implies (safe-loop x86)
           (equal (step-x86 x86) x86)))
```

5.6 Properties of reachability

This subsection is essentially a continuation of Section 5.3 giving more properties about the reachability of instructions and their appearance in the sequential ordering. Specifically, it proves that `kth-insn-from` and `eip-to-k` are inverse operations (for instructions that are in fact reachable), and that adding one in the sequential ordering corresponds to advancing `eip` to the next instruction.

```
(defthm kth-insn-of-seq-reach-rec-is-insn
 (implies (seq-reachable-rec mem eip k)
          (equal (kth-insn-from mem (code-start) (seq-reachable-rec mem eip k))
                 eip)))

(defthm elim-kth-insn-from-given-seq-reach-rec
 (implies (and (seq-reachable-rec mem eip (code-size))
               (equal k (seq-reachable-rec mem eip (code-size))))
          (equal (kth-insn-from mem (code-start) k) eip)))
```

```
(defthm if-seq-reach-in-k-then-bound-by-kth-insn
 (implies (and (kth-insn-from mem (code-start) k)
               (seq-reachable-rec mem eip k))
          (<= eip (kth-insn-from mem (code-start) k))))

(defthm insn-k-plus-one-not-reachable-in-k-given-kth-exists
 (implies (kth-insn-from mem (code-start) k)
          (seq-reachable-rec
                mem (kth-insn-from mem (code-start) (+ 1 k)) k))))

(defthm insn-k-plus-one-not-reachable-in-k
  (not (seq-reachable-rec
        mem (kth-insn-from mem (code-start) (+ 1 k)) k)))

(defthmd seq-reach-rec-same-with-inc-bound
 (implies (seq-reachable-rec mem eip k)
          (equal (seq-reachable-rec mem eip (+ k 1))
                 (seq-reachable-rec mem eip k))))

(defthmd seq-reach-rec-same-with-added-bound
 (implies (and (seq-reachable-rec mem eip k))
          (equal (seq-reachable-rec mem eip (+ j k))
                 (seq-reachable-rec mem eip k))))

(defthmd rewrite-seq-reach-rec-w-size-bound-to-lower-bound
 (implies (and (seq-reachable-rec mem eip k)
               (< k (code-size)))
          (equal (seq-reachable-rec mem eip (code-size))
                 (seq-reachable-rec mem eip k))))

(defthm seq-reach-rec-of-kth-insn-is-k-bound-code-size-given-reach-k
 (implies (and (seq-reachable-rec mem (kth-insn-from mem (code-start) k) k)
               (< k (code-size))
               (kth-insn-from mem (code-start) k))
          (equal (seq-reachable-rec mem (kth-insn-from mem (code-start) k)
                                    (code-size))
                 k)))

(defthm seq-reach-rec-of-kth-insn-is-k-bound-code-size-given-code-size
 (implies (seq-reachable-rec mem (kth-insn-from mem (code-start) k)
                             (code-size))
          (equal (seq-reachable-rec mem (kth-insn-from mem (code-start) k)
                                    (code-size))
                 k)))

(defthm seq-reach-rec-of-next-as-kth-is-one-plus
 (implies (and (seq-reachable-rec
                mem
                (+ (kth-insn-from mem (code-start) k)
                   (insn-len mem (kth-insn-from mem (code-start) k)))
                (code-size))
               (seq-reachable-rec mem (kth-insn-from mem (code-start) k)
                                  (code-size)))
          (equal (seq-reachable-rec
                  mem (+ (kth-insn-from mem (code-start) k)
                         (insn-len mem (kth-insn-from mem (code-start) k)))
                  (code-size))
                 (+ 1 (seq-reachable-rec
                       mem (kth-insn-from mem (code-start) k) (code-size)))))

(defthm seq-reach-rec-of-next-is-one-plus
 (implies (and (seq-reachable-rec mem (+ eip (insn-len mem eip)) (code-size))
               (seq-reachable-rec mem eip (code-size)))
          (equal (seq-reachable-rec mem (+ eip (insn-len mem eip))
                                    (code-size))
                 (+ 1 (seq-reachable-rec mem eip (code-size))))))
```

5.7 Preservation lemmas

This subsection proves the most central facts of the proof: that each of the elements of the safety invariant is preserved by execution steps.

```
(defthm reset-passes-ebp-data-safe
 (equal (static-inv-ebp-data-safe (reset-static-inv inv))
        (static-inv-ebp-data-safe inv)))

(defthm ebx-data-safe-is-transient
 (not (static-inv-ebx-data-safe (reset-static-inv inv))))

(defthm ebx-code-safe-is-transient
 (not (static-inv-ebx-code-safe (reset-static-inv inv))))

(defthm logand-w-code-mask-is-zt-or-code
  (implies (not (code-region-p (logand #x100000f0 addr)))
           (zero-tag-region-p (logand #x100000f0 addr))))
```

Preservation lemma #1: if the current instruction is sequentially reachable, so is the next one.

```
(defthm seq-reach-preservation
 (implies (and (step-x86 x86-state)
               (not (code-guard-p (x86-state-eip x86-state)))
               (not (code-guard-p (x86-state-eip (step-x86 x86-state))))
               (not (zero-tag-region-p (x86-state-eip (step-x86 x86-state))))
               (mem-sandbox-ok-rec (x86-state-mem x86-state) (code-size))
               (all-aligned-reachable-rec (x86-state-mem x86-state)
                                          (code-size))
               (seq-reachable-rec (x86-state-mem x86-state)
                                  (x86-state-eip x86-state)
                                  (code-size))
               (insn-sandbox-ok (x86-state-mem x86-state)
                                (x86-state-eip x86-state))
               (check-static-inv (static-inv-before (x86-state-mem x86-state)
                                                    (x86-state-eip x86-state))
                                 x86-state))
          (seq-reachable-rec (x86-state-mem (step-x86 x86-state))
                             (x86-state-eip (step-x86 x86-state))
                             (code-size))))
```

```
(defthm seq-reach-of-reg-step
 (implies (and (step-x86 x86-state)
               (not (code-guard-p (x86-state-eip x86-state)))
               (not (code-guard-p (x86-state-eip (step-x86 x86-state))))
               (not (zero-tag-region-p (x86-state-eip (step-x86 x86-state))))
               (regular-step x86-state)
               (not (safe-loop x86-state))
               (mem-sandbox-ok-rec (x86-state-mem x86-state) (code-size))
               (all-aligned-reachable-rec (x86-state-mem x86-state)
                                          (code-size))
               (seq-reachable-rec (x86-state-mem x86-state)
                                  (x86-state-eip x86-state)
                                  (code-size))
               (insn-sandbox-ok (x86-state-mem x86-state)
                                (x86-state-eip x86-state))
               (check-static-inv (static-inv-before (x86-state-mem x86-state)
                                                    (x86-state-eip x86-state))
                                 x86-state))
          (seq-reachable-rec (x86-state-mem (regular-step x86-state))
                             (x86-state-eip (regular-step x86-state))
                             (code-size))))
```

Preservation lemma #2: if the code region passes the sandboxing checks before an instruction, it still does afterwards.

```
(defthm mem-sb-preservation
 (implies (and (step-x86 x86-state)
               (safety-invariant x86-state))
          (mem-sandbox-ok-rec (x86-state-mem (step-x86 x86-state))
                              (code-size))))
```

Preservation lemma #3: if all of the 16-byte aligned instructions in the code region are reachable before an instruction, they still are after it.

```
(defthm all-align-reach-preservation
 (implies (and (step-x86 x86-state)
               (safety-invariant x86-state))
          (all-aligned-reachable-rec (x86-state-mem (step-x86 x86-state))
                                     (code-size))))

(defthm code-region-is-not-guard
 (implies (code-region-p addr)
          (not (code-guard-p addr))))

(defthm check-ebx-code-safe-implies-code-region-p
 (implies (and (static-inv-ebx-code-safe inv)
               (check-static-inv
                inv
                (x86-state eip eflags eax ebx ecx edx
                           esi edi ebp esp mem))
               (not (zero-tag-region-p ebx)))
          (code-region-p ebx)))
```

Preservation lemma #4: if the program counter is in the code region before an instruction, it still is afterwards.

```
(defthm code-region-p-preservation
 (implies (and (step-x86 x86-state)
               (mem-sandbox-ok-rec (x86-state-mem x86-state)
                                   (code-size))
               (all-aligned-reachable-rec (x86-state-mem x86-state)
                                          (code-size))
               (code-region-p (x86-state-eip x86-state))
               (seq-reachable-rec (x86-state-mem x86-state)
                                  (x86-state-eip x86-state)
                                  (code-size))
               (insn-sandbox-ok (x86-state-mem x86-state)
                                (x86-state-eip x86-state))
               (check-static-inv (static-inv-before (x86-state-mem x86-state)
                                                    (x86-state-eip x86-state))
                                 x86-state)
               (not (zero-tag-region-p (x86-state-eip (step-x86 x86-state))))
               (not (code-guard-p (x86-state-eip (step-x86 x86-state)))))
          (code-region-p (x86-state-eip (step-x86 x86-state)))))
```

As previously mentioned, ACL2 does not by default expand the definitions of recursive functions like `static-inv-after-kth-insn` (`siaki` for short). However, it is often necessary to unroll it by one level to expose the calculation of the static invariant for the current instruction (`static-inv-trans`); when enabled, this rule performs that expansion.

```
(defthmd expand-siaki-one-step
  (implies (and (mem-p mem)
                (addr-p eip)
                (seq-reachable-rec mem eip (code-size)))
           (equal (static-inv-after-kth-insn
                   mem (seq-reachable-rec mem eip (code-size)))
                  (static-inv-trans
                   mem eip
                   (reset-static-inv
                    (static-inv-after-kth-insn
                     mem
                     (- (seq-reachable-rec mem eip (code-size)) 1)))))))

(defthm logand-with-jump-mask-is-aligned
 (implies (natp eip)
          (chunk-aligned-p (logand #x100000f0 eip))))
```

The most complicated of the preservation lemmas concerns the meaning of the per-instruction static invariants. To make the proof more manageable for ACL2, we split it up into three smaller cases for the three aspects of the invariant, but it is still relatively time consuming. While most of the lemmas in the proof can be verified in less than a second, this one requires 2-5 minutes depending on the speed of machine used, more than any other.

```
(defthm inv-preservation-ebx-code
 (implies (and (step-x86 x86-state)
               (regular-step x86-state)
               (not (code-guard-p (x86-state-eip x86-state)))
               (not (zero-tag-region-p (x86-state-eip x86-state)))
               (seq-reachable-rec (x86-state-mem (step-x86 x86-state))
                                  (x86-state-eip (step-x86 x86-state))
                                  (code-size))
               (safety-invariant x86-state))
         (check-ebx-code-inv (static-inv-before
                               (x86-state-mem (step-x86 x86-state))
                               (x86-state-eip (step-x86 x86-state)))
                              (step-x86 x86-state))))

(defthm inv-preservation-ebx-data
 (implies (and (step-x86 x86-state)
               (regular-step x86-state)
               (not (code-guard-p (x86-state-eip x86-state)))
               (not (zero-tag-region-p (x86-state-eip x86-state)))
               (seq-reachable-rec (x86-state-mem (step-x86 x86-state))
                                  (x86-state-eip (step-x86 x86-state))
                                  (code-size))
               (safety-invariant x86-state))
          (check-ebx-data-inv (static-inv-before
                                (x86-state-mem (step-x86 x86-state))
                                (x86-state-eip (step-x86 x86-state)))
                               (step-x86 x86-state))))

(defthm inv-preservation-ebp
  (implies (and (step-x86 x86-state)
               (regular-step x86-state)
               (not (code-guard-p (x86-state-eip x86-state)))
               (not (zero-tag-region-p (x86-state-eip x86-state)))
               (seq-reachable-rec (x86-state-mem (step-x86 x86-state))
                                  (x86-state-eip (step-x86 x86-state))
                                  (code-size))
               (safety-invariant x86-state))
          (check-ebp-inv (static-inv-before
                           (x86-state-mem (step-x86 x86-state))
                           (x86-state-eip (step-x86 x86-state)))
                          (step-x86 x86-state))))

(defthm in-code-implies-not-zt
 (implies (code-region-p eip)
          (not (zero-tag-region-p eip))))
```

Preservation lemma #5: if the state before an instruction satisfies the static invariant at that instruction, the state after the instruction satisfies the invariant at the next instruction.

```
(defthm inv-preservation
 (implies (and (step-x86 x86-state)
               (regular-step x86-state)
               (code-region-p (x86-state-eip (step-x86 x86-state)))
               (not (code-guard-p (x86-state-eip x86-state)))
               (not (zero-tag-region-p (x86-state-eip x86-state)))
               (safety-invariant x86-state))
```

```
          (check-static-inv (static-inv-before
                             (x86-state-mem (step-x86 x86-state))
                             (x86-state-eip (step-x86 x86-state)))
                            (step-x86 x86-state))))
```

Preservation lemma #6: if the current instruction is legal according to the sandboxing policy, the next one will be too. Since `mem-sandbox-ok` guarantees that every sequentially reachable instruction is legal, this is actually a simple corollary of preservation lemmas 1 and 2.

```
(defthmd insn-sb-ok-preservation
 (implies (and (mem-sandbox-ok-rec (x86-state-mem x86-state)
                                   (code-size))
               (step-x86 x86-state)
               (insn-sandbox-ok (x86-state-mem x86-state)
                                (x86-state-eip x86-state))
               (all-aligned-reachable-rec (x86-state-mem x86-state)
                                          (code-size))
               (seq-reachable-rec (x86-state-mem x86-state)
                                  (x86-state-eip x86-state)
                                  (code-size))
               (check-static-inv (static-inv-before (x86-state-mem x86-state)
                                                    (x86-state-eip x86-state))
                                 x86-state)
               (not (zero-tag-region-p (x86-state-eip (step-x86 x86-state))))
               (not (code-guard-p (x86-state-eip (step-x86 x86-state)))))
          (insn-sandbox-ok (x86-state-mem (step-x86 x86-state))
                           (x86-state-eip (step-x86 x86-state)))))
```

5.8 Final safety induction

Once all of the individual preservation lemmas have been proved, the remainder of the proof is relatively easy.

The statement of the preservation of the complete safety invariant is quite simple:

```
(defthm safety-preservation
 (implies (and (step-x86 x86-state)
               (safety-invariant x86-state))
          (safety-invariant (step-x86 x86-state))))

(defthm mem-sb-ok-means-first-insn-sb-ok
 (implies (mem-sandbox-ok-rec mem k)
          (insn-sandbox-ok mem (code-start))))

(defthm srr-of-start-is-zero
  (equal (seq-reachable-rec mem (code-start) k) 0))

(defthm first-static-inv-is-trivial
  (equal (static-inv-before (x86-state-mem x86) (code-start))
         (base-static-inv)))
```

The base case for the safety induction: the initial state satisfies the safety invariant.

```
(defthm start-state-inv
 (implies (is-start-state x86)
          (safety-invariant x86)))
```

The inductive case for the safety induction: if one state satisfies the invariant, so does the next.

```
(defthm safety-induction
 (implies (safety-invariant x86-state)
          (safety-invariant (step-x86 x86-state))))
```

Safety in the sense of not making any unsafe steps follows at each step from the safety invariant:

```
(defthm safety-inv-implies-safe-to-now
 (implies (safety-invariant x86)
          (consp x86)))
```

The safety invariant holds at each step:

```
(defthm if-sb-then-safety-inv-after-k
  (implies (is-start-state x86)
           (safety-invariant (step-for-k x86 k))))
```

From the above results, the final safety theorem then follows directly.

```
(defthm safety
  (implies (and (mem-p mem)
                (mem-sandbox-ok mem)
                (data-region-p ebp)
                (addr-p eax) (addr-p ebx) (addr-p ecx) (addr-p edx)
                (addr-p esi) (addr-p edi) (addr-p ebp) (addr-p esp)
                (addr-p eflags))
           (consp
            (step-for-k
             (x86-state (code-start) eflags eax ebx ecx edx
                        esi edi ebp esp mem)
            k))))
```

6 Evaluation

We constructed the proof described in this report incrementally, by starting with a minimal machine model, verifier, and safety proof (for a machine with only a single `nop` instruction) and then adding support for new instructions and the corresponding parts of the verification and proof one at a time. To summarize this progress, we measured the state of the proof at a number of milestones, shown in Figure 2 along with examples of what future steps might include. From the statistics shown in the table, it might appear that our proof development strategy was relatively scalable: though the size of the proof script and the time needed to verify it increase as the size of the instruction set increases, the rate of their growth is not excessive, and the largest sizes and times are still manageable. (The size and time do not increase monotonically because work on expanding the proof and on simplifying it proceeded in parallel: sometimes we discovered ways that long or slow proof steps could be eliminated.) However, the count of the amount of proof that changed between steps (counting the number of lines added or deleted, where a modified line is counted as deleted and then added) shows that the process was not as incremental as might be desirable. Though some additions to the instruction set (such as those in step 2) required few changes the proof because their safety policy was similar to existing instructions, new instructions that required changes to the safety invariant tended to require larger proof changes. In the case of the current developer, the effort spent on lemmas that were discarded in later steps was a valuable learning experience, but a more experienced ACL2 user might have be advised to use fewer intermediate steps, since the total number of lines written was significantly larger than the size of our final proof.

Step	Lines	Changed	Time	Instruction(s) added		
1	716	-	35s	1-byte `nop`		
2	765	193	78s	2- and 3-byte `nop`	`inc %eax`	`mov` *addr*`, %eax`
3	583	584	26s	`mov %eax,` *addr*		
4	707	174	42s	`jmp` *addr*		
5	1927	1414	465s	`and $`*immed*`, %ebx`	`xchg %eax, %ebx`	`mov %eax, (%ebx)`
6	1673	978	186s	`jmp *$ebx`		
7	1687	890	771s	`and $`*immed*`, %ebp`	`xchg %eax, %ebp`	`mov %eax, (%ebp)`
8				`and $`*immed*`, %esp`	`xchg %eax, %esp`	`mov %eax, (%esp)`
9				`addl $`*immed*`, %esp`		
10				`call` *addr*	`pushl %eax`	`popl %eax`
11				`andl $`*immed*`, (%esp)`	`ret`	
12				`xchg %eax, %ecx`	`cmp %eax, %ecx`	`jne` *addr*

Figure 2: Steps in the incremental development of the safety proof. For steps that were completed, "Lines" gives the number of non-commnent-non-blank lines in the proof script, "Changed" gives the number of such lines added or deleted since the preceding version, and "Time" gives the time required for ACL2 to verify the proof (on the machine used during development, an 1100MHz AMD Athlon).

The most serious obstacle to the continued scaling of this proof approach, though, cannot be easily quantified: it is that the steps of the proof became more difficult as it grew. For instance, steps 1-6 were complete at the time of the 2005 technical report [MM05], but an attempt at step 7 around that time bogged down and was abandoned; a successful proof of step 7 came only after additional inspiration in 2006. This increase in subjective difficulty occurred even as the developer became more familiar with ACL2, so it is likely that an objective measure of difficulty, if it existed, would have increased even faster. The most likely candidate for the cause of this difficulty is insufficient abstraction: the proofs of too many lemmas required reasoning about the complete definition of the machine's step function and the per-instruction safety predicate. Even though such reasoning was mainly performed automatically by ACL2, the large number of cases increased ACL2's running time, and it became difficult to tell which additional intermediate lemmas should have been introduced to guide the proof search.

7 Related work

A more complete discussion of earlier SFI implementations and other code safety mechanisms appears in [MM06]; here we mention only other proofs.

The Gleipnir project at Microsoft Research has investigated a binary-rewriting security technique called Control-Flow Integrity, or CFI. As suggested by the name, CFI differs from SFI in focusing solely on constraining a program's jumps: in the CFI implementation, each potential jump target is labelled by a 32-bit value encoded in a no-op instruction, and an indirect jump checks for the presence of an appropriate tag before transferring control. The CFI authors have written a human-checked proof [ABEL05] that a CFI-protected program will never make unsafe jumps, even in the presence of arbitrary writes to data memory. However, the proof is formulated in terms of a miniature RISC architecture whose encoding is not specified. This is somewhat unsatisfying, as the safety of the real CFI technique is affected in subtle ways by the x86 instruction encoding (for instance, the possibility that the immediate tag value used in the comparison at a jump site might be itself interpreted as a safe jump target tag.)

More recently, Winwood and Chakravarty developed a machine-checked safety proof in Isabelle/HOL for a model of an SFI-like rewriting technique applicable to RISC architectures [WC05]. To avoid having to move instructions, their approach overwrites indirect jump instructions with direct jumps of the same size to a trusted dispatcher. Unfortunately, on the Alpha processors they consider this puts a 2MB limit on the size of binaries to which their technique is applicable. Their proof has a number of similarities to ours, but is generally more abstract: for instance, rather than reasoning directly about the implementation of the code that checks indirect jumps, they treat it as a separate module with some axioms about its behavior, which can then be composed with the rewritten program.

8 Conclusion

The experience of constructing this proof has confirmed the reputation of machine-checked proof construction as an endeavor on which it is easy to spend a large amount of effort on a seemingly simple task. When explained intuitively, the safety of the PittSFIeld rewriting technique strikes many listeners as immediately clear, yet the proof in this report is time-consuming to read, not to mention to write. However, some other aspects of this experience compared favorably to that reputation: for instance, the ACL2 tool was relatively easy to learn, with the developer accomplishing meaningful results after only a few weeks of use, and the level of automation was such that a number of changes to the model were possible with only minor changes to the proof scripts. The construction of a machine-checked proof can be a pleasant and productive experience, as long as the statement to prove is sufficiently simple and the tool used is appropriate to the task.

References

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *ICFEM 2005, Proceedings of the 7th International Conference on Fromal Engineering Methods*, pages 111–124, Manchester, UK, November 1–4, 2005.

[MM05] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report 2005-030, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, May 2005. (also MIT LCS TR #988).

[MM06] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2–4, 2006.

[WC05] Simon Winwood and Manuel M. T. Chakravarty. Secure untrusted binaries - provably! In *Third International Workshop on Formal Aspects in Security and Trust*, pages 171–186, Newcastle upon Tyne, UK, July 18-19, 2005.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993.