Proof-Carrying Code for x86 Architectures

Kevin W. Hamlen

Advisors: Peter Lee and George C. Necula

Undergraduate Senior Research Thesis
Carnegie Mellon University
School of Computer Science
May 11, 1998

Abstract

This paper presents an extension of Necula and Lee's Proof-Carrying Code (PCC) system to support the x86 architecture. PCC is a security scheme which allows the safe execution of untrusted code. Untrusted code to be executed is required to be coupled with a proof that the code satisfies certain safety properties. This code-proof pair is statically checked by the client system prior to execution. If the check succeeds, then the code is deemed "safe" and is accepted and executed by the client. This x86 adaptation rejects all programs which could potentially terminate with an unhandled exception or memory fault. It accepts most x86 programs which satisfy the standard conventions of the architecture, but conservatively rejects some safe programs. Most reasonable programs can be translated into a form which will be accepted.

1 Introduction

Many common computing tasks require the execution of sub-programs which are provided to the system or application by some external source. For example, compilers allow linking of external binaries, perhaps compiled with some other compiler or from some other source language, into the program being compiled. Web browsers allow the transfer and execution of small programs or "applets" supplied by foreign systems. Operating systems run user programs as processes. In such cases it is advantageous for the system to somehow ensure that these sub-programs do not violate certain safety properties as they execute. These safety properties may dictate, for example, that the sub-program terminates within some pre-specified number of instruction cycles, performs no invalid memory accesses, adheres to the standard calling convention of the machine (it preserves a particular set of machine registers across function calls, for example), or perhaps other requirements.

Proof-Carrying Code (PCC) is a solution to this problem wherein a *code consumer* can verify that code provided by an untrusted *code producer* adheres to a certain *safety policy*. The safety policy is a set of rules chosen by the code consumer. These rules define which programs will be accepted by the code consumer and which will be rejected. A code consumer's safety policy is made known in advance to code producers wishing to supply code. The code consumer requires that code producers supply a proof that the code being provided satisfies the safety policy. This proof is then checked by the code consumer to be sure that it is valid. Based on the results of this proof verification step, the supplied code is then either accepted or rejected.

A working PCC system was developed and implemented by Necula and Lee [9, 10] for the DEC Alpha architecture. My research explores an adaptation of this scheme for x86-based architectures. Because of the non-RISC nature of the architecture, it can be difficult to state and prove safety policies for x86 machine language programs. I demonstrate that it is possible to implement PCC for a reasonable subset of all x86 programs. Common safety policies can be stated, proved, and verified for programs in this subset. Programs not in this subset can be detected and rejected by the code consumer. I also propose some ways in which this x86 PCC scheme could be improved to allow a larger set of x86 programs and propose an informal method of showing that the code consumer side of a PCC implementation is correct.

2 The PCC Scheme

In its most general form, a PCC interaction consists of three entities: a *code producer*, a *proof producer* (often the same as the code producer), and a *code consumer*. Each of these entities has a different goal. The code producer's goal is to convince the code consumer that a particular piece of code that it provides satisfies the consumer's safety policy. The code producer works with the proof producer to accomplish this. The proof producer's goal is simply to provide proofs of whatever is requested. The code consumer trusts neither the code producer nor the proof producer. The consumer's goal is to accept code-proof pairs and verify that the proof proves that the code satisfies the safety policy.

A PCC interaction generally proceeds as follows: We begin with the code producer holding a piece of code which it wishes to provide to a code consumer. The code producer first discovers the code consumer's safety policy (which is assumed to be publicly available). The code producer then uses both the safety policy and the code together to generate a statement called a *verification condition*. This verification condition is a logical statement which, if proved, would be sufficient to ensure that the code satisfies the safety policy. As a simplistic example, suppose that the safety policy consists of a single rule: nothing may be written to memory location 10. And suppose that the code consists of two operations: calculate a mathematical expression E, then write to memory

location E. In that case, a verification condition for this code under this safety policy would be $E \neq 10$. If proved, this would ensure that the code satisfies the safety policy.

Once the verification condition has been generated, the code producer passes it to the proof producer and requests a proof. The proof producer uses a theorem-proving algorithm to search for a proof of the given verification condition. Upon finding a proof, it is returned to the code producer. Continuing the example provided in the previous paragraph, suppose the expression E that our sample code calculates is $2 * R1 + 1$ where $R1$ is a machine register which holds integers. Then the resulting proof might be:

$$
\begin{aligned}
& R1 \text{ is an integer} \\
\Rightarrow\ & 2 * R1 \text{ is even} \\
\Rightarrow\ & 2 * R1 + 1 \text{ is odd} \\
\Rightarrow\ & 2 * R1 + 1 \neq 10 \text{ (because 10 is not even)} \\
\Rightarrow\ & E \neq 10.
\end{aligned}
$$

Thus we've constructed a proof of the verification condition which uses only fundamental assumptions about the architecture and about arithmetic.

The resulting proof and the original code are then paired together and passed to the code consumer. The code consumer must now verify this code-proof pair. First, there is the obvious syntactic check that the code constitutes a legal "code object" and the proof constitutes a legal "proof object." This is a straightforward format check. Next there are three more interesting checks that must be performed: (1) The proof must have a conclusion which proves that the safety policy has been satisfied, (2) the proof must be valid, and (3) the proof must match the code. The code consumer checks these things by following a procedure similar to that of the code producer. First it uses its own safety policy together with the supplied code to construct the verification condition independently. This verification condition is compared against the conclusion of the supplied proof to be sure that they match. This completes checks (1) and (3). The code consumer then walks through the proof and checks that each proof rule is applied properly and that no invalid assumptions are made. This completes check (2).

The following informal argument illustrates why the checking scheme described above is sufficient to guarantee that the safety policy will not be violated by the provided code. By generating the verification condition from the safety policy and the code independently, the code consumer formulates a condition which, if proved, is sufficient to guarantee that the safety policy is not violated. It then verifies that the provided proof is a proof of the condition. Assuming these two checks succeed, the code consumer has therefore proved that the code does not violate the safety policy. Thus no matter what tampering or manipulation of the proof and/or code may have taken place, the code consumer accepts the code only if the code does not violate the safety policy.

The PCC scheme has a number of advantages over other methods. First, there are few restrictions imposed on the language in which programs are written. The code consumer accepts programs in machine language so that theoretically programs could be compiled by the code producer from any source language or even written by hand in assembly code for maximum efficiency. Second, there are also few restrictions imposed on the nature of the safety policy. Many different notions of program safety can be represented as PCC safety policies, and different safety policies may be used by different code consumers. Third, the PCC scheme eliminates the need for inserting potentially unnecessary runtime checks in the code to ensure safety. All safety checking is performed before execution begins so that the program can run at full speed once it has been verified. Fourth and finally, with PCC there is no need to pre-negotiate trust between systems with tools like encryption or binary signatures. Thus there is no need to guard against theft of these encryption keys or signa-

tures. Furthermore, systems which have gained the code consumer's trust cannot then abuse that trust and violate the safety policy, whether intentionally or unintentionally. The code consumer trusts no foreign systems and verifies each code-proof pair it receives using its own proof-checker.

Another major advantage of the PCC scheme is that the *trusted computing base* (TCB) is very small. A security scheme's trusted computing base is the set of programs or code that the consumer must trust without formal verification. This generally includes things like the consumer's operating system and whatever program is performing the consumer's verification of foreign-supplied programs. With PCC the code consumer's verification program is quite small and simple. This is because although in general proof generation can be a difficult and complex task, verification of arbitrary existing proofs can be performed by a very elementary algorithm. Minimizing the TCB is important because it allows us to trust the security scheme based on the integrity of as small and as simple a program as possible. We can trust a PCC implementation without trusting the code producer's compiler or the proof producer's theorem-prover, for example. We need only trust the code consumer's small proof-verifier.

2.1 Practical Implementation Issues for PCC

The simplistic PCC scheme described above has several problems which make it somewhat unwieldy and difficult to implement. In order to resolve these difficulties, a few extra details must be introduced to the scheme.

First, it is in general impractical (and unnecessary) to assume that arbitrary proofs can be generated by the proof producer upon request. Proof search can be an intensive process but fortunately it will need to be performed only rarely. In the common case, code producers will wish to supply one of a relatively small number of pre-compiled programs to arbitrary code consumers. These code consumers, while numerous, will usually each hold one of a small number of common safety policies. Thus the code consumer can, for each of its programs and each commonly encountered safety policy, generate the necessary proof in advance and store it for later reference. Pre-generated proofs can then be served upon request without the need to generate them on the fly.

Another problem with the proof-generation step as it was described earlier is that it is often too difficult to automatically generate proofs of verification conditions without any additional information provided. In practice, proof-generators will often wish to make use of the original source code from which the machine code was compiled. The source code can then be used to guide the proof search effort. In some cases it may even be desirable to insert code or otherwise modify the code to simplify proof generation. For this reason it is often advantageous to incorporate the proof-generator into the compiler itself. Such an augmented compiler is called a *certifying compiler* [8]. When the code producer uses a certifying compiler, the code producer and proof producer are combined into a single entity.

In order to allow extra information provided by the source code to be used in this way, the code producer needs a mechanism for guiding the direction of both proof search and proof verification. We must take care, however, not to introduce anything which requires the code consumer (the entity performing the proof verification) to trust something provided by an external source. For this reason, code provided by the code producer is augmented with some "hints" called *annotations* which are not trusted by the code consumer. These commented programs are called *annotated executables*. Annotations can be thought of as claims about the code that must be proved along with the safety policy. In effect, the code producer is saying, "Not only does this code satisfy your safety policy, but these extra invariants also hold and I will prove it." At first it may seem like this makes a difficult problem worse. If it is difficult to prove and verify the safety policy, how will adding more claims to be proved help? Annotations have the effect of guiding and aiding

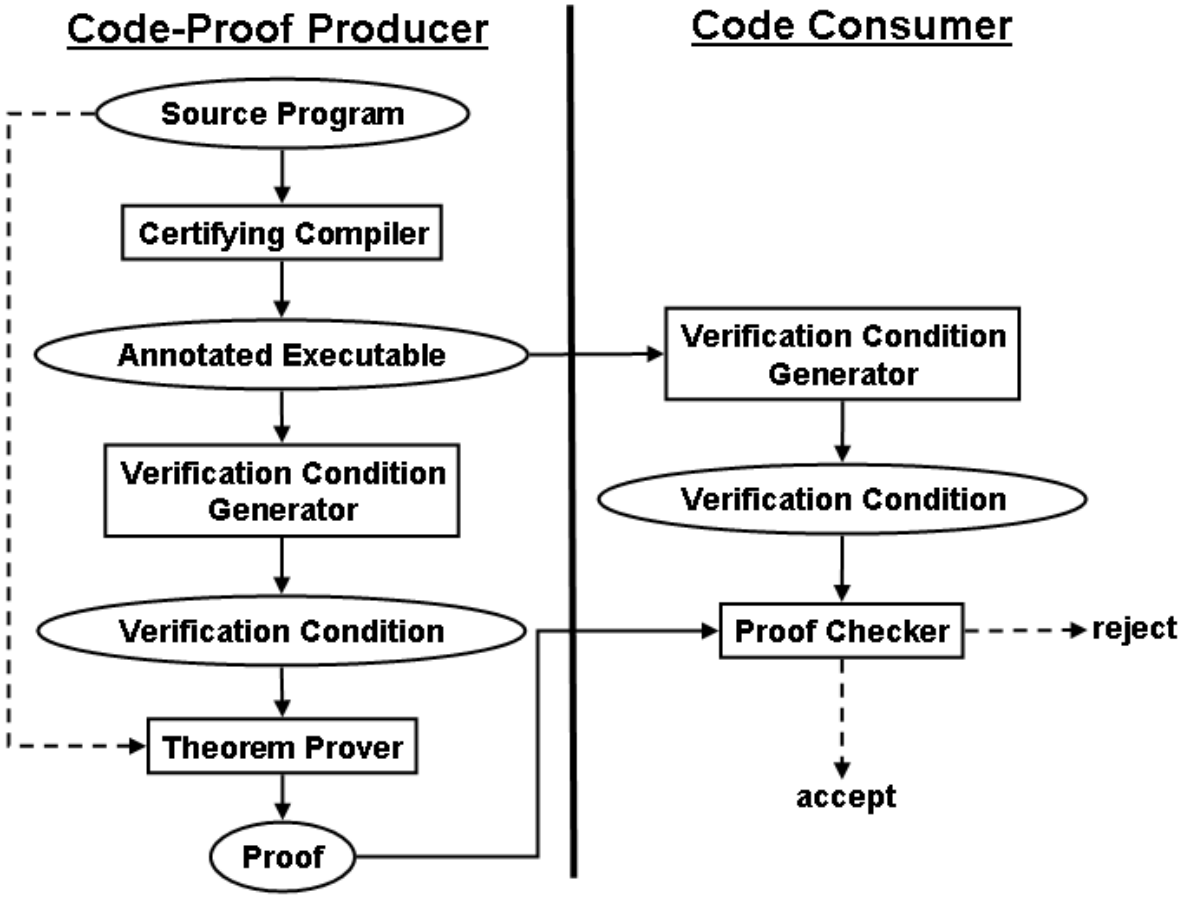

Figure 1: Flowchart representation of a common PCC scheme.

proof generation and verification because, once proved, they can be used as assumptions which help to prove that the safety policy holds. In essence, the annotations can be thought of as small, specialized lemmas.

The resulting final picture of a PCC interaction is shown in Figure 1. On the code/proof-producer side, the certifying compiler takes a source program and generates an annotated executable. A verification condition is generated from this annotated executable and then a theorem-prover takes this condition (and possibly the original source code) and generates a proof. The annotated executable and proof are sent to the code consumer. On the code consumer end, a verification condition is generated from the annotated executable and then a proof-verifier checks that the supplied proof is valid and that it proves the verification condition.

2.2 Necula and Lee's PCC Implementation

The PCC scheme implemented by Necula and Lee [9, 10] uses a procedure like the one depicted in Figure 1. On the code/proof producer end, a certifying compiler takes source programs written in a simplified version of the C programming language as input and outputs annotated DEC Alpha assembly code. The annotations are represented as assembly data directives.

The resulting assembly language program is then assembled into an annotated executable. Annotated executables take the form of DEC Alpha COFF (Common Object File Format) files [3].

The annotations in the COFF file are logical statements or "predicates" written in first-order logic which are stored in a binary form and placed in a special section in the object file. Each annotation uses pointers to addresses in the code and data segments of the file to refer to parts of the program. The logical language used to represent the annotations is LF (Logical Framework) [12].

The verification condition generator program (VCGen) takes these annotated Alpha COFF files as input, parses the Alpha machine code, data declarations, and annotations, and emits a verification condition. This verification condition is also represented in LF. In the worst case, the VCGen algorithm can be trans-exponential, but in practice it runs in linear time.

Finally, the proof-producer program uses a theorem-proving algorithm based on Greg Nelson's *Techniques for Program Verification* [11]. In the worst case this algorithm can be hyper-exponential, but in practice the algorithm runs in linear time.

The code consumer uses the same VCGen program as the code/proof producer. VCGen is used to independently generate the verification condition from the provided code. The proof-checker program then verifies that the supplied proof is a valid proof of the condition. The proof-checker is a very simple linear-time algorithm written in LF.

The trusted computing base (TCB) for Necula and Lee's PCC implementation is quite small and fast. The VCGen and proof-checker programs together comprise only about 4000 lines of C and LF code and both tend to run in linear time.

3 Porting PCC to a New Platform

Before addressing the issue of how to modify Necula and Lee's PCC implementation to support the x86 architecture, it is worthwhile to consider the more general question of how to port an existing PCC scheme of the form depicted in Figure 1 to a new architecture. There is reason to expect that much of a PCC implementation can remain unchanged when moving from platform to platform.

As with a traditional modularly designed compiler, a certifying compiler can be adapted to output assembly code for a new architecture by providing it with a new "back end." That is, a new code generation module together with perhaps minor changes to the intermediate language(s) should be sufficient. The differences between a certifying compiler and a traditional compiler concern the generation of appropriate annotations for the annotated executable. As was mentioned previously, annotations are primarily tools for propagating pertinent information from the source code all the way through the compilation process to the final object code. Since the source language is the same no matter which architecture is being targeted, we expect that the much of the content of annotations will not need to be modified for a new architecture. However, more annotations and new types of annotations may need to be added to annotated executables in order to make verification conditions for the new architecture provable. A new machine language will often have a new and different set of "weaknesses" which user programs could potentially exploit in order to violate safety policies. Annotated executables for this new machine language will need to include annotations sufficient to allow the construction of proofs that demonstrate that these weaknesses are not so exploited.

Both the theorem-prover and the proof-checker modules should be almost completely portable. Both deal strictly with the architecture-independent logical language used to represent verification conditions, annotations, and proofs. A new set of allowable assumptions which describe the specific nature of the new architecture will need to be provided, but the internal proof-search and proof-verification algorithms can remain untouched. In rare cases a new architecture may require the generation of verification conditions which include new mathematical operators or new logical connectives. If this is the case, the theorem-prover and proof-checker will need to be provided with

extra rules of inference explaining the proper use of these operations and connectives. Occasionally it may also be advantageous to provide the theorem-prover with specialized heuristics or methods of proof search in order to deal with particular operations that arise often on the new architecture.

The module remaining to be discussed is the verification condition generator, the one module that exists on both the code/proof producer and the code consumer ends of the scheme. The verification condition generator is in many ways the PCC component which plays the greatest role in dealing with specifics of particular architectures. It will therefore usually need to be almost entirely rewritten when porting to a new architecture. A parser for the new machine language must be written. This parser must take in object files of a format that is likely to be specific to the new architecture and output a verification condition which encodes every aspect of the code which may possibly violate the safety policy.

Most of the remainder of this paper will explain the theory and implementation of a verification condition generator for x86 machine language. New annotation types will be added to the scheme as they are motivated in the context of describing the x86 VCGen program.

4 An x86 Safety Policy

Before embarking on a discussion of the implementation of an x86 VCGen, it is necessary to first define the safety policy which it will enforce. The precise definition of a safety policy is often very architecture-specific. This is to be expected since safety policies should generally be based in part on the set of "weaknesses" (conditions under which hardware exceptions will be thrown, etc.) of the architecture being supported. The x86 safety policy which I will enforce in this paper will be based upon standard conventions for the 32-bit execution model for the x86 architecture [5]. Specific applications of PCC may require extensions to this safety policy, however since the 32-bit execution model is standard, most x86-based applications will probably include the safety requirements dictated by this safety policy.

An informal description of this x86 safety policy will be given here. A rigorous definition of the safety policy will be provided by the x86 VCGen algorithm itself which will be defined in the section 5.

4.1 Segment Register Assignment Safety

The x86 architecture uses a register file consisting of eight "general-purpose" registers, six "segment" registers, and a selection of status and control registers. Some of these registers have special meanings and the safety policy will require that no illegal values be assigned to them. In particular, the segment registers may only be assigned legal "segment selector" values. When an x86 machine language program is to be executed, it is the loader's responsibility to initialize the segment registers to point to appropriate code, data, and stack segments. Generally assignments to segment registers during execution only occur when the value of one segment register is copied to another. An assignment of a value which does not correspond to a legal "segment selector" causes an exception to be thrown and will not be permitted by the safety policy. I will therefore define the set of legal segment selector values by the set of values held in the segment registers at the start of execution. These are the only values which may be assigned to segment registers.

4.2 Calling Convention Safety

I will also require the standard CALL-RET calling convention for 32-bit x86 applications to be observed. This convention dictates that function calls in x86 programs be performed in the following

way:

1. The caller procedure begins by pushing zero or more 32-bit data items onto the stack which are to be passed to the callee procedure as parameters. The current stack pointer is held in the ESP general-purpose register and is decremented as the stack grows. Each data item placed on the stack must occupy exactly 32 bits. (Data items larger than 32 bits must be passed via a 32-bit pointer to a memory address in another segment.)

2. The caller procedure then executes a CALL instruction which pushes the current program counter address onto the stack and passes control to the callee procedure.

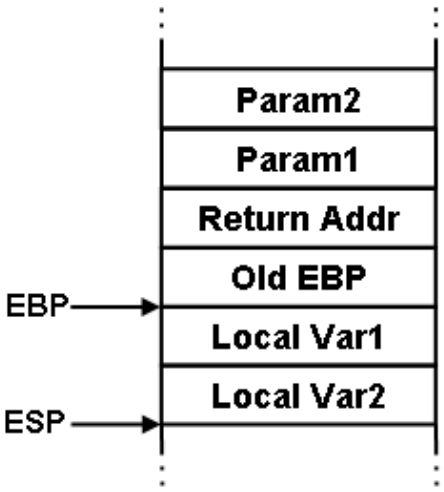

Figure 2: Stack configuration during execution of the body of the callee procedure.

3. The callee is expected to preserve the contents of the segment registers, the ESP register (stack pointer), and the EBP register (frame pointer). Other registers are generally assumed to be volatile and the caller may not assume that they are preserved across the function call. Upon being called, the callee procedure usually saves the contents of the EBP register by pushing it onto the stack and then copies the contents of the ESP register to EBP.

4. The callee procedure may then add local variables to the stack by decrementing the ESP register. The new EBP register contents remains unchanged, serving as a base address from which to access stack parameters (positive offsets from EBP) and local variables (negative offsets from EBP).

5. At the conclusion of the subroutine, the callee pops all of its local variables from the stack by assigning the address in EBP to ESP. The old value of EBP is next popped off the stack using a POP instruction.

6. Finally, a RET instruction executed which pops the return address off of the stack and jumps to it, transferring control back to the calling procedure.

4.3 Memory Safety

Under this safety policy, an x86 application may read and write to/from its data and stack segments but may only read from its code segment. Memory read/writes to/from the data and code segments may only be performed on addresses which correspond to data structures which have been pre-defined in the object file. Annotations must be used to define the alignment type and size of each such data structure. Access of each such data item must then conform to its prescribed alignment and size restrictions. Essentially this means that program data must be used in a consistent way and typecasting from one bit width to another is not permitted. 16-bit integers must be read and written as 16-bit integers, similarly for 32-bit integers, etc.

Stack data does not require any pre-defined data typing because on x86 machines stack data is "boxed" by convention. That is, each stack read/write must be 32-bit aligned and must access exactly 32 bits. Memory read/writes to/from the stack segment may only be performed within the

range of addresses delimited by the start of the parameter list above and the ESP pointer below. This enforces the local scope of each procedure. In addition, the callee may not alter the return address pushed onto the stack by the original CALL instruction.

For now, dynamic memory allocation will not be supported. This can be added in the future.

4.4 Control Flow Safety

Conceptually, this safety policy considers each procedure in the program as a separate sequence of instructions. Control may be passed only into and out of a procedure block via a CALL or RET instruction. In addition, all branch and jump destination addresses must be absolute. That is, no branch, jump, or procedure call may target a destination address which is stored in a register or in writable memory. Thus higher-order functions are not supported. Support for function pointers and other higher-order programming techniques can probably be added in future work.

5 The Semantic Translation Scheme

The algorithm used by the x86 VCGen program to implement the above safety policy can be considered as a semantic translation which maps annotated machine language programs to *logical predicates* (statements in first-order logic). The mapping of a particular program to a particular predicate will be called a *judgment*. One way to formally represent an algorithm which constructs proper judgments is to define it as a set of *derivation rules* which take zero or more judgments as hypotheses and conclude a single judgment. A sequence of derivation rules is then termed a *derivation*. A derivation is like a "proof" whose conclusion is a judgment which defines a safety predicate. The derivation rules are defined in such a way that it is clear from the machine language program being processed which rules must be applied in what order. The resulting derivation then serves as a recipe for generating the verification condition from the annotated executable.

The next few sections will define the form and meaning of judgments for this algorithm. Derivation rules which manipulate these judgments will then be defined and explained. Finally, an example derivation will be given which strings these derivation rules together to generate a verification condition for a simple program.

5.1 Judgments

I will begin by defining the main judgment for this derivation system:

$$S \vdash K_1 \ / \ I \ / \ K_2 \hookrightarrow P.$$

In this judgment, S will denote a machine state, K_1 and K_2 will denote (possibly empty) sequences of machine language instructions, I will denote a single machine language instruction, and P will denote a logical predicate. I define this judgment to mean that if a machine language program consisting of the instructions in K_1 followed by instruction I followed by instructions in K_2 is given, and if this program is to be executed starting with instruction I and with an initial machine state S, then a verification condition ensuring that this program does not violate our x86 safety policy is given by P.

5.1.1 Expressions and Predicates

Before this definition is complete, a definition for each individual component of the above judgment must be provided. The syntax of logical predicates is an extension of that given by Necula and Lee

in [10]:

Variables	X
Expressions	$E ::= X \mid N \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \cdot E_2 \mid E \bmod N \mid E \gg N \mid$ $E_1 \oplus E_2 \mid E_1 \ominus E_2 \mid E_1 \odot E_2 \mid$ $sel(E_1, E_2) \mid upd(E_1, E_2, E_3)$
Expresion Predicates	$EP ::= true \mid false \mid E_1 = E_2 \mid E_1 \neq E_2 \mid E_1 \geq E_2 \mid E_1 < E_2$
Predicates	$P ::= EP \mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall X.P \mid$ $dWrite32(E) \mid dRead32(E) \mid cRead32(E) \mid dWrite16(E) \mid \cdots$

Here I have listed only the syntactic elements that will be needed for the examples provided in the remainder of this paper. An actual implementation of the algorithm would include more operators and connectives.

In the syntax for expressions I distinguish between normal addition, subtraction, and multiplication ($+,-,\cdot$) and unsigned addition, subtraction, and multiplication with 32-bit wrapping ($\oplus,\ominus,\odot$). I.e. 0xFFFFFFFF + 0x1 = 0x100000000 but 0xFFFFFFFF $\oplus$ 0x1 = 0x0. The $\gg$ operator denotes right-shifting (division by a constant power of two). There are also expressions to facilitate memory reads and writes. $sel(E_1, E_2)$ returns the value stored in memory address E_2 when the state of memory is described by E_1. $upd(E_1, E_2, E_3)$ denotes the memory state produced when initial memory state E_1 is altered by writing the value of expression E_3 into memory address E_2. Thus the expression $sel(upd(m, a, v), a)$ evaluates to v.

Several of the predicates also deserve special comment. The *xWritenn* and *xReadnn* predicates are statements about the writability and readability (respectively) of memory addresses. For example, *dWrite32*(E) indicates that the expression E refers to the address of a 32-bit wide writable memory address in the data segment, as indicated by the data annotations in the executable. (See the safety policy described in section 4.3.) There are 32-, 16-, and 8-bit *dWrite* and *dRead* predicates for accesses to the data segment and a 32-, 16-, and 8-bit *cRead* predicate for reads from the code segment. Stack segment accesses do not require special predicates and will be explained later.

5.1.2 Instruction Sequences

Next I will consider the components K_1, I, and K_2 which together comprise the machine language program being examined. Individual instructions will be described by a label followed by an optional "loop marker" followed by an assembly language mnemonic with appropriate arguments:

Labels	$L ::= N$
Assembly Mnemonics	$M ::= \textbf{mov}\ eax, ebx \mid \textbf{add}\ eax, N \mid \textbf{push}\ ebp \mid \cdots$
Instructions	$I ::= L : M \mid L : \surd M$

A label is simply the numerical memory address where the first byte of the given instruction is stored in the code segment.

The optional loop marker $\surd$ is an annotation provided by the code producer which marks some instruction within the body of each loop in the program. Recall from section 3 that the code producer can supply "comments" about the code called *annotations* in a separate section of the object file. Each annotation has a pointer to some memory address either corresponding to a location in the executable code or corresponding to the location of a data item in the data segment. The loop markers are therefore represented in the object file as annotations which point to the given instruction. The code producer will be required to mark each loop by attaching a

loop marker annotation to some instruction within the loop. Loops which are not marked will be detected by the code consumer and will cause the code to be rejected. More about loops and how they are handled will be discussed later.

For the machine instructions themselves, assembly language mnemonics are used here for purposes of readability. In the actual VCGen implementation, instructions are parsed directly as binary machine code without using an assembly language representation.

Instruction sequences will be denoted by semicolon-separated lists of instructions:

$$\text{Instruction Sequences} \qquad K ::= \cdot \mid I_1; I_2; \ldots; I_k$$

In practice they are represented in the annotated executable as consecutive binary machine code instructions.

5.1.3 Machine States

The final remaining component to be defined is S, the machine state:

$$\begin{aligned} \text{Machine State} \qquad S ::= [\![\; & eax = E_1, ebx = E_2, \ldots, cs = E_3, ss = E_4, \ldots, \\ & s_1 = E_5, s_2 = E_6, \ldots, \\ & of = EP_1, sf = EP_2, zf = EP_3, af = EP_4, cf = EP_5, df = EP_6, \\ & m = E_7, \\ & h = \{L_1, L_2, \ldots\} \\ & inv = \{(L_3, S_1), (L_4, S_2), \ldots\} \quad]\!] \end{aligned}$$

The machine state is a collection of symbolic expressions and predicates which represent the contents of the machine registers (*eax*, *ebx*, *cs*, *ss*, etc.), stack variables (s_1,s_2, etc.), status flags (*of*, *sf*, *zf*, *af*, *cf*, and *df*), and memory (m). It also includes a (possibly empty) set (h) of labels and another (possibly empty) set (*inv*) of label–machine state pairs. These final two members will denote certain key points in the program which have already been visited by the VCGen algorithm and, in the case of the *inv* member, the machine states previously seen at those points. This will be helpful later in the analysis of loops. The contents of memory are represented as a single expression (a nested set of *upd* memory state expressions) assigned to the member m. Stack variables are listed separately in the machine state instead of being represented in the memory expression because on the x86 the stack is used like an extended register file. At any point during execution, access to the stack is limited by the safety policy to a relatively small local scope consisting exclusively of 32-bit values. In addition to the set of machine registers, the stack, and memory, the x86 architecture also includes a selection of status flags which can be either true or false (1 or 0). These status flags are represented in the machine state as expression predicates of the same form as those defined previously. The flags supported will be the overflow flag (*of*), sign flag (*sf*), zero flag (*zf*), adjust flag (*af*), carry flag (*cf*), and direction flag (*df*). (See [5] for a description of each flag's meaning.) The x86 architecture also includes a parity flag, but it will not be supported in this scheme. Any programs whose adherence to the safety policy depend upon the value of the parity flag will be rejected by the code consumer.

It will be helpful to have a convenient notation for examining the value of a particular member of a state and for describing the new state that results from assigning a value to a particular member of the state. The notation $S[eax = E]$ will denote a state S such that the *eax* member holds the expression E. The notation $S[eax \leftarrow E]$ will denote the state that results from taking state S and assigning expression E to the *eax* member.

5.1.4 An Example Judgment

Now that all components of the main judgment have been defined, let us return to the simple example program described in section 2 for an example of such a judgment. The proposed example program takes the value in the EAX register, multiplies it by 2, adds 1, and then stores a value to the memory address defined by the result. The following is an implementation of such a program expressed in x86 assembly language. The leftmost column lists the program address for each instruction. The middle column shows the binary machine language disassembly expressed in hexadecimal. The rightmost column expresses the program in standard x86 assembly mnemonics.

```
40A438 : 6B C0 02          IMUL EAX, EAX, 2
40A43B : 83 C0 01          ADD EAX, 1
40A43E : C6 00 00          MOV [EAX], 0
40A441 : C3                RET
```

Under the stated x86 safety policy, one judgment which might be derived for this program would be

$$S_0 \vdash \cdot \;/\; \texttt{40A438} : \textbf{imul}\ eax, eax, 2 \;/\; \texttt{40A43B} : \textbf{add}\ eax, 1;$$
$$\texttt{40A43E} : \textbf{mov}\ [eax], 0;$$
$$\texttt{40A441} : \textbf{ret} \qquad \hookrightarrow \forall x.\mathit{dWrite32}(x \odot 2 \oplus 1)$$

where S_0 is some machine state consisting of register, stack, and status flag values which are known to be initialized by the x86 program loader prior to execution. The judgment states that the given program consisting of an IMUL, ADD, MOV, and RET instructions in that order is to be executed starting with the IMUL instruction and with an initial machine state S_0. It concludes that a verification condition sufficient to guarantee that the x86 safety policy given in section 4 is honored by the program is given by the logical predicate $\forall x.\mathit{dWrite32}(x \odot 2 \oplus 1)$. Thus the proof producer must prove that no matter what EAX is, the memory write will end up writing to a valid memory location.

The derived safety predicate in this case guarantees the memory safety aspect of the x86 safety policy (section 4.3). It should be pointed out that the calling convention safety requirements imposed by section 4.2 have been ignored for now. (Normally, in order to prove that the sub-program above satisfies the x86 safety policy, the code producer would have to prove that the sub-program conforms to the standard calling convention.) For now, note that by visual inspection the program performs no stack accesses and that the values of the ESP and EBP registers remain untouched. One would expect that this should mean that there is little, if anything, for the code producer to prove in order to show that the calling convention safety requirements have been satisfied. This is the informal reason why nothing about calling convention safety appears in this verification condition. A more formal justification will be provided in section 7.4 which concerns the enforcement of the standard calling convention.

5.1.5 Annotation Judgments

To complete the judgment system I must introduce one more judgment type. This final judgment allows us to refer to the remaining (non-assertion) annotations in the annotated executable. I define the judgment

$$A \lhd L$$

to mean that the annotation A is attached to the instruction at label L. Arbitrary numbers of annotations may be attached to each program label. Annotations will be described using the following syntax:

Register Sets	$R \subseteq \{eax, ax, \ldots, s_1, s_2, \ldots, cf, zf, \ldots\}$
Annotations	$A ::= \mathit{Func}(P_1, P_2, N, R) \mid \mathit{Loop}(P, R)$

An annotation consists of a "type" (*Func* or *Loop*) followed by a list of parameters. The number and types of the parameters depends on the annotation type. One possible parameter type is a "register set" (R) which consists of some subset of the member fields of a machine state (excluding the h and *inv* members). The other annotation parameters consist of predicates (P) and integers (N). The meaning of each annotation type and its parameters will be explained later.

5.2 Derivation Rules

Now that the judgments have been defined, the next step is to show how a derivation is constructed. A derivation consists of an upside-down "tree" of derivation rules. Each node in this tree is an application of a derivation rule. A derivation rule takes zero or more judgments as hypotheses and concludes a single judgment from those hypotheses. As an example, consider the simplest derivation rule, the NOP Introduction rule:

$$\frac{S \vdash K_1; \mathbf{nop} \ / \ I \ / \ K_2 \hookrightarrow P}{S \vdash K_1 \ / \ \mathbf{nop} \ / \ I; K_2 \hookrightarrow P}$$

The judgment on top is the single hypothesis for this rule. The bottom judgment is the conclusion. The label which should precede the NOP instruction has been omitted for brevity. Labels will generally be omitted unless they have particular importance in the rule.

The NOP mnemonic stands for "No OPeration." As its name implies, it has no effect except to let execution of the program pass to the next instruction. The above derivation rule models the behavior of the NOP instruction by saying the following: If P is a sufficient verification condition for a particular program given by $K_1; \mathbf{nop}; I; K_2$ when that program is executed starting at instruction I with machine state S, then P is also a sufficient verification condition for the program when it is executed with the same machine state S starting at the NOP instruction which appears immediately before instruction I. The rule is called the "NOP Introduction rule" because its effect is to introduce the NOP instruction to the flow of execution of the program. All rules in the derivation system described in this paper will be introduction rules.

Note that this rule cannot be applied to an illegal use of the NOP instruction in a program. It is difficult to misuse an instruction as benign as the NOP instruction, but there is one circumstance where use of a NOP would result in an error. Namely, if no legal instruction follows the NOP, then control would be passed to a potentially illegal opcode and the program would abort with an exception. Observe that the hypothesis for this rule requires that the legal instruction I immediately follow the NOP. Thus a program with no legal instruction after the NOP violates the stated assumptions and the rule is not applicable.

5.2.1 Changes to the Machine State

Some rules will involve changes to the machine state. Consider the MOV-Immediate Introduction rule:

$$\frac{S[eax \leftarrow N] \vdash K_1; \mathbf{mov}\ eax, N \ / \ I \ / \ K_2 \hookrightarrow P}{S \vdash K_1 \ / \ \mathbf{mov}\ eax, N \ / \ I; K_2 \hookrightarrow P}$$

The `MOV EAX,`*imm* instruction moves a 32-bit integer constant, *imm*, (called an "immediate argument") into the EAX register. The rule above models this behavior by stating the following: If P is a sufficient verification condition for the program $K_1; \mathbf{mov}\ eax, N; I; K_2$ when it is executed starting at instruction I with a machine state S in which *eax* has been assigned the expression N, then P is also a sufficient verification condition for the same program when execution begins at the MOV instruction immediately preceding instruction I with the same machine state S but with any value stored in *eax*. (Any value may be stored in *eax* because the MOV is about to overwrite that value when it stores N there.)

Memory writes are a special case where not only is the state modified, but an extra predicate is added to the verification condition as well. Consider the Indirect MOV-Immediate Introduction rule:

$$\frac{S[m \leftarrow upd(E, EP, N)] \vdash K_1; \mathbf{mov}\ [eax], N\ /\ I\ /\ K_2 \hookrightarrow P}{S[eax = EP, m = E] \vdash K_1\ /\ \mathbf{mov}\ [eax], N\ /\ I; K_2 \hookrightarrow dWrite32(EP) \wedge P}$$

The `MOV [EAX],`*imm* instruction, instead of assigning imm to EAX, assigns *imm* to the memory address pointed to by EAX. (EAX is in this case an offset into the data segment.) The rule above models the change in memory by making an appropriate assignment to the m member of the machine state. A new *dWrite32* predicate is also conjoined with the old condition, requiring the code producer to prove that the memory write targets a legal address.

5.2.2 Status Flags

Unfortunately, assignments on an x86 platform often involve a number of state changes in addition to the specific assignment requested. One way in which this occurs is that assignments are made to the status flags as a side-effect of many operations. Although the MOV instructions illustrated in the previous section do not have these side-effects, most x86 arithmetic and bitwise instructions do. I will use the ADD-Immediate Introduction rule as an example.

The `ADD EAX,`*imm* instruction increments the value in the EAX register by *imm*. After performing the calculation, the status flags are set with various boolean values to indicate the nature of the result. The following is a statement of the ADD-Immediate Introduction rule:

$$\frac{S' \vdash K_1; \mathbf{add}\ eax, N\ /\ I\ /\ K_2 \hookrightarrow P}{S[eax = EP] \vdash K_1\ /\ \mathbf{add}\ eax, N\ /\ I; K_2 \hookrightarrow P}$$

where

$$S' := S[eax \leftarrow (EP \oplus N), cf \leftarrow (EP + N > \text{0xFFFFFFFF}), zf \leftarrow (EP \oplus N = 0), \ldots].$$

Thus not only has the EAX register been assigned the result of the 32-bit addition, but the various status flags are set with expression predicates indicating the nature of the result. The CF flag is set to true if, treating the addition as unsigned, the sum would have been larger than 32 bits. It is cleared otherwise. The ZF flag is set to true if the result is 0 and otherwise cleared. The OF, SF, and AF flags are also affected by the ADD instruction and their respective state members must be set with appropriate expression predicates.

Every rule which introduces an instruction which modifies the status flags must include all of these state changes in its hypotheses. Because the assignments are numerous, I will generally omit them and replace them with ellipses in this document.

5.2.3 Overlapping Registers

The other state changes that take place as a side-effect of any particular register assignment are due to the fact that x86 registers "overlap." I will use the EAX, AX, AH, AL family of registers

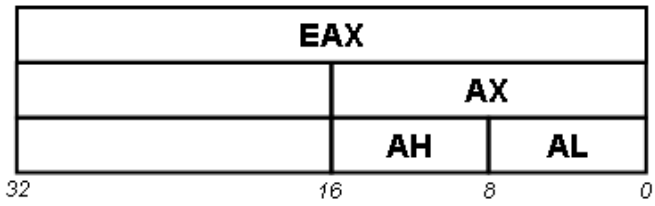

Figure 3: x86 "overlapping" registers.

as an example. The EAX register is a 32-bit register. The lower 16 bits of this register are also known as the AX register. An assignment to the AX register automatically changes the lower 16 bits of the EAX register. Similarly, an assignment to the EAX register will alter the contents of the AX register. In addition, the upper 8 bits of the AX register are also known as the AH register while the lower 8 bits are known as the AL register. Represented diagrammatically, this family of registers looks like Figure 3.

Thus an assignment to the EAX register must be modeled by an assignment to the *eax*, *ax*, *ah*, and *al* state members. An assignment to the AX register involves changes to the EAX, AH, and AL registers. Assignments to AH and AL change both EAX and AX.

One way of modeling this feature of the architecture is to have only a single state member, *eax*, representative of the entire family and then represent writes and reads of the other registers using suitable expressions. For example, assignment of N to AX would be modeled by a state change described by $S[eax = E][eax \leftarrow E - (E \bmod 65536) + N]$. However, this means that even simple computations performed using the smaller registers will result in large, complex symbolic expressions to be proved as part of the verification condition. The vast majority of x86 machine language programs make little or no use of the overlapping nature of the register families. So a 32-bit computation will make use of the 32-bit registers without reading from or writing to the smaller registers. Similarly, a 16-bit computation will ignore the existence of the 32-bit and 8-bit registers which overlap it.

Therefore I use an alternative representation which results in smaller verification conditions for most programs. All registers in the family are represented as separate members in the state. An assignment to one of these registers is modeled by making a straightforward assignment to the stated register and then assigning suitable expressions to registers which overlap it. If the program never refers to a register unless it assigns directly to it first, these more complex expressions will never appear in the verification condition. An assignment of N to AX would therefore be modeled by the following assignments (where *eax* originally holds the expression E):

$$
\begin{aligned}
&eax \leftarrow E - (E \bmod 65536) + N \\
&ax \leftarrow N \\
&ah \leftarrow (N \gg 8) \bmod 256 \\
&al \leftarrow N \bmod 256
\end{aligned}
$$

When stating derivation rules in this document, side-effect assignments to overlapping registers will not be given explicitly. Rather, whenever a register assignment is stated, appropriate corresponding assignments to overlapping registers should be assumed.

5.3 Derivations

At this point we have enough background on derivation rules to make a first attempt at formally deriving the verification condition for the program given in section 5.1.4. As mentioned previously, derivations consist of an upside-down "tree" of derivation rules. So far we have seen only derivations with exactly one hypothesis, so this tree structure will not be seen in this example. (It will become evident starting in section 7.3.2.) We have also not discussed derivation rules which model function calls and returns. For now we will ignore calling convention safety and assume that a RET instruction by itself constitutes a legal program. Our resulting simplified RET-Introduction derivation rule is a leaf rule, having no hypotheses:

$$\frac{}{S \vdash K_1 \;/\; \textbf{ret} \;/\; K_2 \hookrightarrow \mathit{true}}$$

Given this rule and the rules provided in section 5.2, we may now construct a derivation of the verification condition for the program given in section 5.1.4 and restated below.

```
40A438 : 6B C0 02          IMUL EAX, EAX, 2
40A43B : 83 C0 01          ADD EAX, 1
40A43E : C6 00 00          MOV [EAX], 0
40A441 : C3                RET
```

I begin with a statement of the judgment which we wish to derive.

$$S_0 \vdash \cdot \;/\; \textbf{imul}\ \mathit{eax}, \mathit{eax}, 2 \;/\; \textbf{add}\ \mathit{eax}, 1; \textbf{mov}\ [\mathit{eax}], 0; \textbf{ret} \hookrightarrow P$$

for some predicate P which we wish to determine. S_0 is the initial state of the machine prior to execution as set up by the system's program loader. On x86 we are guaranteed that when the program starts, the CS, DS, and SS segment registers are set to appropriate code, data, and stack segment selector values, the ESP register is pointing to the top of the stack, and the top element on the stack is a valid return address. Therefore we will model this initialization by setting each of the corresponding members of S_0 to some special constants: cs_0, ds_0, ss_0, and esp_0. Other register contents are unknown. Thus we will set all of the remaining register members of S_0 to unbound variables. The number of available stack variables will, for now, be assumed to be 1 (the return address). This stack variable, s_1, will also be set to an unbound variable as will the memory state variable, m. The values of the status flags are also unknown at the start of execution, but we cannot model this by simply setting them to unbound variables since a variable is not a valid expression predicate. Instead, we will set them equal to expression predicates whose truth values depend upon unbound variables. Thus we define S_0 to be

$$\begin{aligned}[\![\quad & \mathit{eax} = x, \mathit{ebx} = y, \ldots, \mathit{esp} = \mathit{esp}_0, \mathit{cs} = \mathit{cs}_0, \mathit{ds} = \mathit{ds}_0, \mathit{ss} = \mathit{ss}_0, \\ & s_1 = z, \\ & \mathit{cf} = (u = 0), \mathit{zf} = (v = 0), \ldots, \\ & m = w, h = \{\}, \mathit{inv} = \{\} \quad]\!]\end{aligned}$$

where u, v, w, x, y, and z are all unbound variables. The state members h and inv are set to the empty set since no points in the program have yet been visited.

I now wish to construct a derivation of the judgment. Judgments will be derived "bottom-up." That is, I start with the conclusion to be derived (the judgment given above) with its missing

verification condition predicate, and assume that it is the bottom judgment of some derivation rule. I then infer the identity of this unknown rule and what the hypothesis judgment(s) of this rule would have to be. Next I recursively attempt to construct a derivation of the inferred hypothesis judgment(s) in the same way. The process continues until there are no more judgments to be derived.

As an example of this "bottom-up" process, I will construct a derivation of the judgment for the sample program. Any derivation of this judgment must end with an IMUL-Immediate Introduction rule. This is because the IMUL instruction is the first instruction being executed in the program and hence must be the last instruction introduced to the head of the instruction stream by a rule in the derivation. I haven't yet stated the IMUL-Immediate Introduction rule, but it is a standard arithmetic rule like the ADD-Immediate Introduction example given in section 5.2.1, except that it multiplies EAX by 2 instead of incrementing it by *imm*. Applying this IMUL-Immediate Introduction rule, I arrive at

$$\frac{S_0[eax \leftarrow x \odot 2, \ldots] \vdash \textbf{imul}\ eax, eax, 2\ /\ \textbf{add}\ eax, 1\ /\ \textbf{mov}\ [eax], 0; \textbf{ret} \hookrightarrow P}{S_0 \vdash \cdot\ /\ \textbf{imul}\ eax, eax, 2\ /\ \textbf{add}\ eax, 1; \textbf{mov}\ [eax], 0; \textbf{ret} \hookrightarrow P}$$

For brevity, I have omitted the various assignments made to the state flags, replacing them with an ellipsis.

I now have a new judgment to derive. If I could construct a derivation of the judgment on top, I would then have a completed derivation of the original judgment. By the same logic as before, any derivation of the top judgment must end with an ADD-Immediate Introduction rule. The ADD-Immediate Introduction rule was presented in section 5.2.1. Applying it to the top judgment, I obtain

$$\frac{\dfrac{S_0[eax \leftarrow x \odot 2 \oplus 1, \ldots] \vdash \ldots\ /\ \textbf{mov}\ [eax], 0\ /\ \textbf{ret}; \cdot \hookrightarrow P}{S_0[eax \leftarrow x \odot 2, \ldots] \vdash \ldots\ /\ \textbf{add}\ eax, 1\ /\ \textbf{mov}\ [eax], 0; \ldots \hookrightarrow P}}{S_0 \vdash \cdot\ /\ \textbf{imul}\ eax, eax, 2\ /\ \textbf{add}\ eax, 1; \ldots \hookrightarrow P}$$

Again I find myself with a new judgment to derive on top. This time I must use the memory assignment variant of the MOV-Immediate Introduction rule. This is another rule that was presented in section 5.2.1. Here I get my first information as to the identity of the unknown predicate, P. The applicable MOV rule demands that the resulting predicate P be of the form $dWrite32(EP)$ AND Q where EP is the expression assigned to the *eax* state member and Q is an unknown predicate. So I make this required substitution for P, propagating it down the derivation:

$$\frac{\dfrac{\dfrac{S_0[eax \leftarrow x \odot 2 \oplus 1, \ldots] \vdash \ldots\ /\ \textbf{ret}\ /\ \cdot \hookrightarrow Q}{S_0[eax \leftarrow x \odot 2 \oplus 1, \ldots] \vdash \ldots\ /\ \textbf{mov}\ [eax], 0\ /\ \textbf{ret} \hookrightarrow dWrite32(x \odot 2 \oplus 1) \wedge Q}}{S_0[eax \leftarrow x \odot 2, \ldots] \vdash \ldots\ /\ \textbf{add}\ eax, 1\ /\ \textbf{mov}\ [eax], 0; \ldots \hookrightarrow dWrite32(x \odot 2 \oplus 1) \wedge Q}}{S_0 \vdash \cdot\ /\ \textbf{imul}\ eax, eax, 2\ /\ \textbf{add}\ eax, 1; \ldots \hookrightarrow dWrite32(x \odot 2 \oplus 1) \wedge Q}$$

The judgment to be derived on top now requires a RET Introduction rule. I must therefore make use of the simplified rule given at the beginning of this section. Again, this is a rule which identifies part of the unknown predicate. In this case we must substitute *true* for Q in order to apply the rule. The resulting derivation is:

$$\frac{\dfrac{\dfrac{S_0[eax \leftarrow x \odot 2 \oplus 1, \ldots] \vdash \ldots\ /\ \textbf{ret}\ /\ \cdot \hookrightarrow true}{S_0[eax \leftarrow x \odot 2 \oplus 1, \ldots] \vdash \ldots\ /\ \textbf{mov}\ [eax], 0\ /\ \textbf{ret} \hookrightarrow dWrite32(x \odot 2 \oplus 1) \wedge true}}{S_0[eax \leftarrow x \odot 2, \ldots] \vdash \ldots\ /\ \textbf{add}\ eax, 1\ /\ \textbf{mov}\ [eax], 0; \ldots \hookrightarrow dWrite32(x \odot 2 \oplus 1) \wedge true}}{S_0 \vdash \cdot\ /\ \textbf{imul}\ eax, eax, 2\ /\ \textbf{add}\ eax, 1; \ldots \hookrightarrow dWrite32(x \odot 2 \oplus 1) \wedge true}$$

There are no more top judgments to be derived (the simplified RET Introduction rule has no hypotheses) and hence this is a completed derivation of the desired judgment.

Note that in constructing the derivation I have automatically generated the missing verification condition. Also note that the procedure used to generate the derivation was completely deterministic. At every step there was only one possible rule which applied. This means that the search for a derivation never requires backtracking and there is no need to remember a history of rules applied. An automated implementation of this procedure need only remember the set of top judgments to be derived (in general there may be more than one since derivation rules can have multiple hypotheses) and the current version of the final verification condition. At each step, the automated derivation-generator arbitrarily selects and removes a judgment from the set of top judgments to be derived and applies the only applicable derivation rule to it. If a judgment is ever encountered for which no rule applies, the program is illegal and the code consumer should reject it. Applying a rule will generally involve making substitutions to the verification condition being constructed and adding new judgments to the set of top level judgments. When there are no more judgments to be derived, the derivation and the verification condition are complete.

Going back to the verification condition that was just generated, there are two small refinements I can make in order to acquire the exact verification condition predicted in section 5.1.4. First, observe that the conjunction of an arbitrary predicate and true is equivalent to the predicate without the conjunction with *true*. An automated VCGen need not perform any reductions on verification conditions, but simplifications like this one are very easy to detect and can result in smaller predicates to be exchanged between the code producer and code consumer. Eliminating the *true* leaves me with $dWrite32(x \odot 2 \oplus 1)$. Finally, all remaining unbound variables should be universally quantified at the outermost level. The final verification condition is then $\forall x.dWrite32(x \odot 2 \oplus 1)$ which is exactly what was predicted in section 5.1.4.

6 Obtaining Derivation Rules from Pseudo-Code Hardware Specifications

Now that judgments have been introduced and explained and the basic VCGen algorithm has been shown, I can propose a systematic translation scheme which is capable of handling most common x86 programs. Explicitly listing derivation rules for every x86 machine language instruction would not be feasible in a paper of any reasonable length. Categorizing the set of x86 instructions and giving a representative rule from each category would be misleading due to the non-RISC nature of the architecture. Each instruction tends to have very individualistic behavior. However, in a formal specification for the x86 architecture, the behavior of each instruction is often couched in terms of a simple pseudo-code language. See [6] for an example of this. This pseudo-code description is usually directly translatable into appropriate derivation rules. The corresponding VCGen implementation then becomes very easy to verify by inspection, lending more credibility to the trusted computing base.

As an example of translating a pseudo-code description of an x86 instruction into a derivation rule I will use the second machine instruction specified in [6]. The pseudo-code specification given on page 3-12 of [6] for the AAD (ASCII Adjust AX before Division) instruction is reproduced

below.

$$\begin{aligned}
&\texttt{tempAL} \leftarrow \texttt{AL};\\
&\texttt{tempAH} \leftarrow \texttt{AH};\\
&\texttt{AL} \leftarrow (\texttt{tempAL} + (\texttt{tempAH} * \texttt{0AH}))\ \texttt{AND FFH};\\
&\texttt{AH} \leftarrow 0;
\end{aligned}$$

The corresponding AAD Introduction rule is given by

$$\frac{S[al \leftarrow (E_1 + (E_2 \cdot 10)) \bmod 256, ah \leftarrow 0] \vdash K_1; \textbf{aad} \ / \ I \ / \ K_2 \hookrightarrow P}{S[al = E_1, ah = E_2] \vdash K_1 \ / \ \textbf{aad} \ / \ I; K_2 \hookrightarrow P}$$

Notice that this rule closely matches the pseudo-code given above it. The same assignments are made and the same computations are performed with minor adjustments. (In this case the AND operation has been transformed into an equivalent mod.)

This is admittedly a particularly simple example, but even complex pseudo-code descriptions can usually be transformed in this way. An example of a more difficult transformation will be given in section 8.

7 Enforcing the x86 Safety Policy

Recognizing that the behavior of most x86 instructions can be modeled by derivation rules that can be obtained directly from hardware pseudo-code descriptions, we can then turn our attention to derivation rules which are instrumental in enforcing the x86 safety policy outlined in section 4. The next several sections will each center on one of the x86 safety policy components and show how derivation rules which enforce this policy are constructed.

7.1 Enforcing Segment Register Safety

I will begin with the simplest of the four safety policy components from section 4: segment register safety. As stated in section 4.1, no x86 segment register (CS, DS, SS, ES, FS, or GS) may ever be assigned an invalid segment selector value. Most x86 programs should not need to construct their own segment selectors. Segment selector values for the program's code segment, data segment, and stack segment are loaded into the CS, DS, and SS registers respectively by the system loader prior to execution. The program may then need to copy these values into the ES, FS, or GS registers in order to perform memory-copy operations, but generally these are the only segment assignments ever made. CS, DS, and SS are generally not written to at all.

The x86 VCGen implementation will enforce this common behavior. The initial machine state will have the special constants, cs_0, ds_0, and ss_0, assigned to the *cs*, *ds*, and *ss* members. *es*, *fs*, and *gs* will initially be assigned unbound variables. When an assignment is made to a segment register, the symbolic expression being assigned must be identically one of cs_0, ds_0, or ss_0. Thus segment selector values may only be copied directly from register to register or from a register to the stack and back. When a memory access is performed, the segment selector which identifies the segment for the memory access must be exactly cs_0, ds_0, or ss_0. No extra safety predicates are added to the verification condition when these checks are performed. The pertinent segment selectors must each have one of the three legal symbolic expressions for segment registers at the time of verification condition generation or the code will be rejected immediately.

Thus derivation rules which assign to segment registers are limited to rules like the following:

$$\frac{S[es \leftarrow ds_0] \vdash K_1; \mathbf{mov}\ es, eax\ /\ I\ /\ K_2 \hookrightarrow P}{S[eax = ds_0] \vdash K_1\ /\ \mathbf{mov}\ es, eax\ /\ I; K_2 \hookrightarrow P}$$

7.2 Enforcing Memory Safety

Next I will skip to section 4.3: memory safety. Since the rules for segment register safety given in the previous section guarantee that the value of all segment registers are known at all points in the program before runtime, it then becomes a trivial matter to enforce the various memory safety policies associated with each of the different memory segments.

When a memory write is performed, the corresponding segment selector must hold the symbolic expression ds_0 or ss_0. If the segment selector expression is ds_0, then an appropriate memory safety predicate (*dWrite32*, *dWrite16*, etc.) is added to the verification condition. If the segment selector expression is ss_0 then the write must be a 32-bit write to a valid memory address in the current local stack scope. Guaranteeing the validity of stack accesses will be discussed further in section 7.4 as part of the discussion of calling convention enforcement.

When a memory read is performed, the corresponding segment selector must hold the symbolic expression cs_0, ds_0, or ss_0. If the segment selector expression is cs_0 or ds_0, then an appropriate memory safety predicate (*dRead32*, *cRead16*, etc.) is added to the verification condition. Again, stack reads must be 32-bit reads to a properly aligned memory address within the local stack scope. Enforcement of this convention will be discussed in section 7.4 as part of calling convention safety.

7.3 Enforcing Control Flow Safety

Control flow safety is a complex issue which will involve the presentation of a number of new concepts involving derivation rules. The simplest kind of control transfer has already been shown. Most machine instructions simply pass control to the next instruction in the sequence. Ensuring that a next instruction in the sequence exists is achieved by requiring that this next instruction be parsed before the derivation rule which passes control to it can be applied. The first sample derivation rule, NOP Introduction, is an example:

$$\frac{S \vdash K_1; \mathbf{nop}\ /\ I\ /\ K_2 \hookrightarrow P}{S \vdash K_1\ /\ \mathbf{nop}\ /\ I; K_2 \hookrightarrow P}$$

The existence of a legal instruction, I, following the NOP is required in order to apply the rule.

7.3.1 Unconditional Jumps

Next there are instructions which unconditionally jump to a specified program address. The JMP instruction is an example. Unlike NOP instructions, the JMP instruction does not require that a legal instruction immediately follow it in the instruction sequence. Instead it requires that a legal instruction exist at the destination address specified. The following is a statement of the Forward JMP Introduction rule:

$$\frac{S \vdash K_1; \mathbf{jmp}\ L; K_2\ /\ L : I\ /\ K_3 \hookrightarrow P}{S \vdash K_1\ /\ \mathbf{jmp}\ L\ /\ K_2; L : I; K_3 \hookrightarrow P}$$

7.3.2 Conditional Forward Jumps

Not all control transfers are unconditional like those enacted by the JMP instruction. The x86 instruction set, like most architectures, includes a set of machine instructions which, depending

on a particular conditional expression, either transfer control to a specified destination address or allow control to pass to the next instruction in the sequence. Such instructions are called branch instructions. A verification condition for a program which includes a branch instruction must ensure that whether or not the branch is taken, the safety policy is satisfied. This notion can be formalized in a verification condition by using the $\supset$ connective. The branch condition must imply the verification condition which results from taking the branch. The negation of the branch condition must imply the verification condition which results from allowing control to pass to the next instruction.

An example is the Forward JZ Introduction rule. The JZ assembly mnemonic stands for "Jump on Zero." It examines the contents of the Zero status flag and, if it is set to true, it branches to the indicated address. Otherwise control falls through to the next instruction. The Forward JZ Introduction rule will be the first example of a rule which has multiple hypotheses. Hypotheses are listed in arbitrary order horizontally across the top portion of the rule. Like the Forward JMP Introduction rule, the Forward JZ Introduction rule will only apply to forward branches. The following is a statement of the rule:

$$\frac{S \vdash K_1; \mathbf{jz}\ L; K_2\ /\ L : I_2\ /\ K_3 \hookrightarrow P_1 \quad S \vdash K_1; \mathbf{jz}\ L\ /\ I_1\ /\ K_2; L : I_2; K_3 \hookrightarrow P_2}{S[zf = EP] \vdash\ /\ \mathbf{jz}\ L\ /\ I_1; K_2; L : I_2; K_3 \hookrightarrow (EP \supset P_1) \wedge (\overline{EP} \supset P_2)}$$

The rule above states that if you know the safety predicate resulting from execution of the program starting at the instruction after the branch, and you know the predicate resulting from execution of the program starting at the target of the branch, then you can formulate the predicate resulting from execution starting at the branch instruction itself. This resulting safety predicate will say that the branch condition being true implies whatever predicate results from taking the branch, and the branch condition being false implies whatever predicate results from not taking the branch.

There is one bit of notation used here which has not been seen previously. The expression $\overline{EP}$ denotes the negation of expression predicate EP. This will appear often in rules for instructions whose behavior depends on a particular condition or set of conditions. For reasons of efficient proof search there is no not connective in the language of logical predicates. However, expression predicates are all of a form that can be explicitly negated. $\overline{true} = false$, $\overline{false} = true$, and all of the relational operators have corresponding opposites. Thus the symbolic expression for $\overline{EP}$ can be computed directly and substituted into the verification condition.

Because it has multiple hypotheses, an implementation of the Forward JZ Introduction rule requires multiple recursive calls to the VCGen algorithm. Thus an example derivation which uses this rule will illustrate the tree structure of derivations. Consider the following program fragment:

```
0040A438 74 03          JZ 0040A43D
0040A43A 83 C0 01       ADD EAX, 1
0040A43D C6 00 00       MOV [EAX], 0
```

A verification condition for this program must ensure that if the predicate currently stored in the *zf* state member is true, then EAX is an 8-bit writable address. Otherwise, EAX+1 is required to be an 8-bit writable address. The derivation fragment which constructs this condition is given

below:

$$\dfrac{\dfrac{\vdots}{S \vdash \ldots \ / \ L : \textbf{mov}\ [eax], 0 \ / \ \ldots \hookrightarrow dWrite8(E) \wedge Q_1} \qquad \dfrac{\dfrac{\vdots}{S[eax \leftarrow E \oplus 1] \vdash \ldots \ / \ \textbf{mov}\ [eax], 0 \ / \ \ldots \hookrightarrow dWrite8(E \oplus 1) \wedge Q_2}}{S \vdash \ldots \ / \ \textbf{add}\ eax, 1 \ / \ \ldots \hookrightarrow dWrite8(E \oplus 1) \wedge Q_2}}{S[zf = EP, eax = E] \vdash \ldots \ / \ \textbf{jz}\ L \ / \ \ldots \hookrightarrow (EP \supset dWrite8(E) \wedge Q_1) \wedge (\overline{EP} \supset dWrite8(E \oplus 1) \wedge Q_2)}$$

Here Q_1 and Q_2 denote unknown predicates that would be generated by whatever instructions follow the final MOV instruction in the program fragment. (Note that Q_1 and Q_2 may well be different despite the fact that they are generated from the same sequence of instructions. Q_2 is generated using an initial state in which the EAX register holds a value which is 1 larger than the value of EAX in the state used in the generation of Q_1. This difference in state can later have a profound effect upon the verification condition generated.) EP and E denote the contents of the ZF flag and the EAX register respectively upon execution of the fragment.

Although the derivation looks complex, it is constructed using the same deterministic algorithm demonstrated in section 5.3. Handling of the JZ instruction simply requires two recursive calls to the algorithm instead of one.

7.3.3 Backwards Jumps: Loops

The final type of control flow which needs to be addressed is looping. Recall from section 5.1.2 that annotations called loop markers (denoted with a $\surd$ symbol) may be attached to any instruction in the instruction sequence. Also, recall that in addition to the judgment type that has been used so far, a second judgment type of the form

$$A \lhd L$$

was introduced in section 5.1.5 which has subsequently gone unused. These annotation judgments will permit the construction of derivation rules which require the existence of certain annotations. Annotation judgments never appear as conclusions of derivation rules; they only appear as hypotheses. These hypotheses can be checked ("derived") by simply verifying the existence of the required annotation in the annotated executable.

Each loop in the program is required to be marked by a loop marker annotation ($\surd$) attached to some instruction in the loop body. When an instruction is encountered which has a loop marker attached, a second annotation of type *Loop* attached to the same instruction is also located. (In practice these two annotations are actually one and the same. That is, the *Loop* annotation itself is the loop marker. However, it is convenient for notational purposes to represent them as two separate annotations here.) The second annotation of type *Loop* contains information which will allow the construction of a suitable verification condition for the loop. In order to enforce the restriction that all loops must contain a loop marker, no backwards branch may be traversed twice in a single path of control through the program without visiting a loop marker for that loop at least once.

In section 5.1.5 it was mentioned that *Loop* annotations have the form $Loop(P, R)$ and hence have two parameters: a logical predicate and a register set. The logical predicate is a suggested loop invariant. It is a predicate of the usual form except that machine state member names (registers,

stack variables, and the memory state variable m) may be used as unbound variables. Certain other variable names which will denote the initial contents of registers and memory may also be used. For example, the variable name eax_0 is reserved to represent the initial contents of the EAX register. The complete set of reserved variable names will be given in section 7.4.1 as part of the discussion on calling convention safety. As an example, one valid predicate P would be $(eax < eax_0) \wedge (ebx \bmod 8 = 0)$.

Loop safety is encoded in the verification condition by an inductive argument. First, the loop invariant must hold at the point when its corresponding loop marker is first seen. This is the base case of the induction. An "iteration" of the loop will then be defined as a flow of control through the program which starts and ends at the same loop marker. For an arbitrary such iteration, the invariant is assumed to hold and the verification condition for one iteration must be shown to satisfy the safety policy. Finally, when the loop marker is revisited at the conclusion of this arbitrary iteration, the loop invariant must hold for the next iteration. This completes the inductive argument.

The suggested loop invariant is not in any way trusted by the code consumer. Loop safety demands only that there exist some loop invariant for which the above inductive argument holds. Thus the verification condition described is valid regardless of the suggested invariant supplied by the code producer. Providing a "wrong" invariant can make the verification condition unprovable or overly strict, but it will never allow the safety policy to be violated.

Before this inductive argument can be written as a verification condition, several ideas must be formalized. First, the notion of an "arbitrary iteration" must be expressed as a logical predicate. The register set parameter of the *Loop* annotation facilitates this. The register set is to be regarded as the set of state members which may change from one loop iteration to the next. Thus an "arbitrary iteration" of the loop is expressed by setting each of these state members to a new unbound variable and then generating a verification condition for a single iteration of the loop using these values. At the conclusion of this iteration an altered machine state will have been produced. Each of the state members which are not in the register set are compared with their original values to be sure that they have indeed not been altered by the "arbitrary iteration". This ensures that the register set supplied in the *Loop* annotation cannot be falsified.

The second idea which must be formalized is the notion of enforcing the loop invariant. The predicate provided in the *Loop* annotation uses registers, stack variables, and the memory state variable as its unbound variables. Each of these must be substituted with the corresponding symbolic expression assigned to that state member in the current machine state. I will denote the substitution of the symbolic expressions in state S into a predicate P by $\sigma_S(P)$.

The following, then, is the Loop Start Introduction rule which, upon encountering a loop marker, begins to construct a safety predicate for a proof by induction.

$$\frac{(L, \_) \notin V \qquad Loop(P_1, R) \lhd L \qquad S'[inv \leftarrow V \cup \{(L, S')\}] \vdash K_1 \ / \ L : {}^{(\surd)}I \ / \ K_2 \hookrightarrow P_2}{S[inv = V] \vdash K_1 \ / \ L : \surd I \ / \ K_2 \hookrightarrow \sigma_S(P_1) \wedge \forall y_{eax} \forall y_{ebx} \ldots . (\sigma_{S'}(P_1) \supset P_2)}$$

where

$$S' := S[x \leftarrow y_x | \forall x \in R]$$

Here we see the first example of a use of the *inv* state member. The Loop Introduction rule only applies when this is the first time that label L has been visited. The hypothesis $(L, \_) \notin V$ represents a check that the label L is not in one of the pairs in *inv*. When the recursive call to the VCGen algorithm is performed (third hypothesis), the label is marked as having been visited and the current register state recorded by adding the label–machine state pair (L, S') to *inv*. Also each member of the register set given by R must be assigned an unbound variable. This is denoted

by the construction of state S' with the assignment $[x \leftarrow y_x | \forall x \in R]$. Then, when the resulting predicate is returned from the recursive call, these new variables must be universally quantified. This is denoted by the expression $\forall y_{eax} \forall y_{ebx} \ldots . (\sigma_{S'}(P_1) \supset P_2)$. The informal notation $^{(\surd)}$ has been used in the third hypothesis to denote that the instruction I should then be processed as if the loop marker were not present. However, the loop marker is not erased and should be observed by the algorithm when (if) label L is visited again. (A rule for revisiting loop markers will be given shortly.)

The induction argument is completed when a previously visited loop marker is later revisited. When a loop marker is revisited, the following Loop End Introduction rule applies:

$$\frac{(L, [\![eax = E_{eax}, ebx = E_{ebx}, \ldots]\!]) \in V \qquad Loop(P, R) \lhd L \qquad \forall x \notin R . x = E_x}{S[inv = V] \vdash K_1 \ / \ L : \surd I \ / \ K_2 \hookrightarrow \sigma_S(P)}$$

Note that the jump destination L is extracted from the *inv* state member thus making this rule only applicable to loop markers which have already been visited by a Loop Start Introduction rule. In extracting the jump destination L, I also look up the original machine state that existed when the loop was entered. The third hypothesis uses this machine state to check that no state member not in the register set R has been modified since label L was first visited. The resulting verification condition states that beginning the next iteration of the loop requires that the loop invariant for this next iteration holds. Since it has already been shown that the satisfaction of the loop invariant implies the safety of an iteration of the loop, this is sufficient to complete the inductive argument. No recursive calls to the VCGen algorithm are necessary. Thus the Loop End Introduction rule is one of the base cases for the VCGen algorithm.

It is still necessary to ensure that at least one loop marker exists in every loop. Otherwise the VCGen algorithm would not terminate on some inputs. This is accomplished by remembering backwards jumps traversed during each flow of control through the program. No backwards jump should ever be traversed more than once if a loop marker exists in every loop. The following Backwards JMP Introduction rule enforces this idea:

$$\frac{L_2 \notin H \qquad S[h \leftarrow H \cup \{L_2\}] \vdash K_1 \ / \ L_1 : I \ / \ K_2; L_2 : \ \mathbf{jmp} \ L_1; K_3 \hookrightarrow P}{S[h = H] \vdash K_1; K_1 : I; K_2 \ / \ L_2 : \ \mathbf{jmp} \ L_1 \ / \ K_3 \hookrightarrow P}$$

The rule is essentially identical to a regular unconditional forward jump rule as seen in section 7.3.1 except that it jumps backwards instead of forwards and it requires that the jump not have been previously traversed. The set of backwards jumps previously traversed is stored in the h state member. There is no corresponding rule for the case when the jump at L_2 has already been taken because this case only occurs when no loop marker exists in the loop. When this is the case, the program should be rejected by the code consumer because no rule applies.

Labels are added to h only when backwards jumps are taken, not just when they are visited by the algorithm. The following Backwards JZ Introduction rule demonstrates this:

$$\frac{\begin{array}{ll} & S[h \leftarrow H \cup \{L_2\}] \vdash \qquad\qquad\qquad\qquad S \vdash K_1; L_1 : I_1; K_2; L_2; \mathbf{jz} \ L_1 \ / \ I_2 \ / \ K_3 \\ L_2 \notin H & \quad K_1 \ / \ L_1 : I_1 \ / \ K_2; L_2; \mathbf{jz} \ L_1; K_3 \hookrightarrow P_1 \qquad\qquad\qquad\qquad \hookrightarrow P_2 \end{array}}{S[zf = EP, h = H] \vdash K_1; L_1 : I_1; K_2 \ / \ L_2 : \mathbf{jz} \ L_1 \ / \ I_2; K_3 \hookrightarrow (EP \supset P_1) \wedge (\overline{EP} \supset P_2)}$$

The label L_2 is added to h only in the recursive call where the conditional jump is taken, not in the recursive call where the jump is not taken and control is passed to the next instruction in the sequence.

It should be noted that there is nothing which prevents a malicious code producer from inserting loop markers and loop annotations which do not correspond to any loop in the program. This is not a concern. The effect of such falsified loops is that the entire remainder of the program will be treated as a single arbitrary iteration of a very large loop. This interpretation, while strange and overly abstract, is more than sufficient to uphold the safety policy. Inserting more than one loop marker into a single loop has a similar effect but, again, cannot result in a verification condition insufficient to uphold the safety policy.

Similarly, jumps which escape a loop body are also not a concern. Such jumps are interpreted as a jump out of an arbitrary iteration of the loop and a suitably abstract version of the verification condition for the remainder of the program will be generated. No special case rules are necessary; the existing rules as they are stated will generate the proper condition.

This completes the enforcement of control flow safety. The one remaining issue of constraining jump destinations to addresses within the same procedure block will be handled by the methods of enforcing calling convention safety.

7.3.4 Instruction Sequence Aliasing

At this point, readers already familiar with the peculiarities of the x86 architecture may be concerned about the problem of handling jumps whose targets correspond to the "middle" of an instruction in the instruction sequence. Although in this document I have represented instruction sequences as semicolon-separated lists of assembly mnemonics, this formalism cannot represent all legal x86 programs. In practice, each machine instruction in an x86 program consists of one or more consecutive bytes in the code segment. Since each byte has its own memory address, it is possible to write an x86 program with a jump target that corresponds to the middle of an instruction. Different machine instructions take up different numbers of bytes, so it is not entirely obvious when a program contains such a jump. Further complicating matters, a program which does jump to the middle of an instruction may actually execute without error. When a jump instruction is executed by the hardware, the bytes starting at the destination memory address are interpreted as the beginning of a new machine instruction regardless of any previous interpretation of those same bytes as, perhaps, the middle of an instruction. The following x86 program is an example of this.

```
0040101A : 05 90 B8 00 00          ADD EAX, B890h
0040101F : EB FA                   JMP 40101B
00401021 : 74 F7                   JZ 40101A
00401023 : C3                      RET
```

Notice that the JMP instruction jumps to an address which corresponds to the middle of the ADD instruction. Disassembling this same series of bytes starting at the jump destination, we actually obtain a legal sequence of x86 instructions which will then be executed when this jump is taken:

```
0040101B : 90                      NOP
0040101C : B8 00 00 EB FA          MOV EAX, FAEB0000h
00401021 : 74 F7                   JZ 40101A
00401023 : C3                      RET
```

I refer to programs like this as examples of *instruction sequence aliasing* because the same sequence of bytes is interpreted as two different instruction sequences. Although such programs cannot be represented using the notational conventions of section 5.1.2, they are nonetheless handled

perfectly well by the derivation rules prescribed in sections 7.3.1 through 7.3.3. I will use the program above as an example.

The VCGen algorithm will accept this program only if a loop marker and *Loop* annotation are both attached to address 40101A, 40101B, 40101C, 40101F, or 401021. (If not, there will exist a flow of control which will result in the backwards branch at 401021 being taken twice without a loop marker having been seen.) Suppose such annotations are attached to address 40101C. Then a verification condition sufficient to guarantee that the program adheres to the safety policy will be generated. The sequence of rules applied, from bottom to top, during the course of the algorithm will be: ADD Immediate Introduction, Backwards JMP Introduction, NOP Introduction, Loop Start Introduction, MOV Immediate Introduction, Backwards JZ Introduction, and then two different branches in the proof tree begin. One simply consists of the RET Introduction rule. The other, from bottom to top, consists of: ADD Immediate Introduction, Backwards JMP Introduction, NOP Introduction, and Loop End Introduction.

The algorithm essentially treats the program like a jump into the middle of a loop body. In actuality there was no explicit "jump." Instead, execution managed to get past the loop marker at address 40101C via instruction sequence aliasing. The effect, however, is the same. Once the loop marker is visited for the first time and the loop officially "starts," the backwards jump at 40101F is simply treated as a natural continuation of the loop iteration. A new iteration does not begin until the loop marker is revisited.

This example illustrates the versatility of the VCGen algorithm. Even pathological examples such as the one above result in accurate (and even reasonable!) verification conditions to be proved.

7.4 Enforcing Calling Convention Safety

Calling convention safety is the final point of the x86 safety policy presented in section 4 that must be enforced. A treatment of calling convention safety has been saved until last because it takes all of the individual pieces provided in the preceding sections and constructs a "big picture" of how to generate a verification condition for a full x86 program.

7.4.1 Function Calls and Returns

The beginning and end of the block of code assigned to each procedure in an annotated executable must be clearly defined in the object file. The x86 COFF object file format includes this information already. If an object file format to be supported does not include this information, it must be added with new annotation types. Each procedure block must be disjoint; none may overlap in memory.

In addition, the first instruction in each procedure must have an annotation of the form $Func(P_1, P_2, N, R)$ attached where P_1 and P_2 are predicates, N is an integer, and R is a register set. The predicates P_1 and P_2 are, respectively, a precondition and postcondition for the procedure. They are expressed in the same manner as loop invariant predicates: in the language of regular predicates but using machine state member names as unbound variables. The integer N denotes the number of parameters which the procedure expects to receive on the stack. The register set R is the set of registers which the procedure does not promise to preserve. If the calling convention described in section 4.2 is to be honored, R may not contain the *cs*, *ds*, *ss*, *esp*, or *ebp* state members. Those registers are required to be preserved across all function calls.

A separate verification condition is generated for each procedure using the VCGen algorithm described previously. Each procedure is treated as a separate sequence of instructions. Thus control may not flow out of one procedure block and into another without the use of a CALL or RET instruction. The initial machine state, S_0, supplied to the VCGen algorithm for each procedure is constructed similar to the method used in section 5.3. The segment register members *cs*, *ds*, and

ss are assigned the constants cs_0, ds_0, and ss_0. All other registers are assigned unbound variables. I will use the following naming scheme for these unbound variables: *eax* will initially be assigned the unbound variable z_{eax}, *ebx* will be assigned z_{ebx}, etc. Each of the status flags are assigned an expression predicate containing an unbound variable: $z_{zf} = 0$, $z_{cf} = 0$, etc. S_0 will initially have $N + 1$ stack variables where N is the integer provided in the *Func* annotation. Stack variables 1 through N correspond to the N available parameters. Stack variable $N + 1$ corresponds to the return address. Each of these are initialized with an unbound variable. Also the stack variable s_{N+1} may never appear in the register set R since the return value must be preserved by the callee. Finally, the memory state member m is also initialized with an unbound variable z_m.

The VCGen algorithm is then used to derive the judgment $S_0 \vdash \cdot \ / \ I \ / \ K \hookrightarrow VC$ where I is the first instruction of the procedure, K is the rest of the instruction sequence, and VC is the verification condition produced by the algorithm. The verification condition which the proof producer is responsible for proving for this procedure is then given by $\forall z_{eax} \forall z_{ebx} \ldots (P_1 \supset VC)$. That is, given that the function precondition holds, the proof producer must show that the verification condition for the function body holds. All unbound variables in the verification condition for the function body are universally quantified. The proof producer must prove a statement of this form for every procedure in the program. Thus a complete verification condition for a full x86 program has the form

$$(\forall \ldots P_1 \supset VC_1) \wedge (\forall \ldots P_2 \supset VC_2) \wedge \cdots \wedge (\forall \ldots P_k \supset VC_k)$$

where k is the number of procedures in the program.

The postcondition and the preservation of registers not in R must also be checked. This is done within the VCGen algorithm at the termination of the procedure. Since the calling convention demands that all procedures terminate with a RET instruction, these checks are performed by the following RET Introduction rule

$$\frac{\mathit{Func}(P_1, P_2, N, R) \lhd L \qquad \forall x \notin R.x = z_x}{S \vdash L : K_1 \ / \ \mathbf{ret} \ / \ K_2 \hookrightarrow \sigma_S(P_2)}$$

where the first hypothesis denotes that the given *Func* annotation is assigned to the current procedure and the second hypothesis performs the register preservation check. As in section 7.3.3, $\sigma_S(P_2)$ represents the predicate P_2 with state member names substituted with their associated symbolic expressions from the current machine state S.

Correspondingly, procedure calls require that the precondition of the called function be satisfied and that the necessary number of stack variables be available in the local scope. After the call completes, the caller may assume that the postcondition of the called procedure has been satisfied and that the registers not in R have been preserved.

Before providing a rule which accurately represents such a procedure call, I must introduce a new substitution. Across a procedure call, stack variables in the machine state get renamed. The caller's top N stack variables (S_{k+1-N} through S_k where k is the number of stack variables in the current machine state) are the callee's bottom N stack variables (S_1 through S_N). The substitution $\delta_{N,k}$ will denote the mapping of the names of the callee's bottom N stack variable names to the caller's top N names where the caller has a total of k stack variables. So for example, $\delta_{N,k}(\mathit{dWrite32}(S_1) \wedge (S_2 < 10))$ would reduce to $\mathit{dWrite32}(S_{k+1-N}) \wedge (S_{k+2-N} < 10)$.

Using this new substitution, a CALL Introduction rule which models the procedure call convention described above is given by

$$\frac{\mathit{Func}(P_1, P_2, N, R) \lhd L \qquad k \geq N \qquad S' \vdash K_1; \mathbf{call}\ L \ / \ I \ / \ K_2 \hookrightarrow P_3}{S^{(k)} \vdash K_1 \ / \ \mathbf{call}\ L \ / \ I; K_2 \hookrightarrow \sigma_S(\delta_{N,k}(P_1)) \wedge \forall y_{eax} \forall y_{ebx} \ldots (\sigma_S(\delta_{N,k}(P_2)) \supset P_3)}$$

where as usual

$$S' := S[x \leftarrow y_x | \forall x \in R].$$

The notation $S^{(k)}$ indicates that S is a machine state in which there are k stack variables. The state assignments in the third hypothesis are again the assignment of unbound variables to all state members which have been declared to be volatile across the function call. Since the precondition P_1 and the postcondition P_2 are written from the called procedure's "point of view," the δ substitution is used to map the stack variable names to their corresponding names in the caller's stack frame.

7.4.2 Stack Usage

Rules governing proper use of the stack within a procedure body are quite restrictive compared to accesses of the data segment. As mentioned in section 5.1.3, it is standard practice for x86 programs to treat the stack as an extended register file. The derivation rules for modifying the ESP register and performing memory accesses within the stack segment will enforce this idea.

The *esp* state member may only be assigned a symbolic expression which is composed solely of the operators $\oplus$ and $\ominus$, integer constants, and the variable z_{esp} (the initial value of the ESP register). When such an expression is assigned, it is immediately reduced to an expression of the form $z_{esp} \ominus c$ where c is a constant. If c is negative, the assignment is invalid and the code is immediately rejected. (This is to prevent a procedure from popping more stack variables off of the stack than it pushes on.) c must also be a multiple of 4, otherwise the code is rejected. (In the 32-bit execution model, the stack pointer must always be 4-byte aligned.) When an assignment to *esp* is made which conforms to these restrictions, the number of stack variables in the machine state is adjusted appropriately. Any new stack variables are initially assigned new unbound variables.

The following are two examples of rules which model this behavior. The first is a PUSH Introduction rule which pushes a value onto the stack.

$$\frac{S^{(k+1)}[esp \leftarrow (E_1 \ominus 4), s_{k+1} \leftarrow E_2] \vdash K_1; \textbf{push } eax \ / \ I \ / \ K_2 \hookrightarrow P}{S^{(k)}[esp = E_1, eax = E_2] \vdash K_1 \ / \ \textbf{push } eax \ / \ I; K_2 \hookrightarrow P}$$

Here, the new state member s_{k+1} gets added and assigned the pushed value E_2 and the stack pointer is decremented. The second is an example of a direct assignment made to the ESP register, such as that which would be executed at the end of a procedure call in order to return the stack pointer to its original value.

$$\frac{E \rightsquigarrow (z_{esp} \ominus c) \qquad Func(P_1, P_2, N, R) \lhd L \qquad \begin{array}{r} S^{(N+c/4+1)}[esp \leftarrow (z_{esp} \ominus c)] \\ [s_j \leftarrow y_j | \forall j \in (k+1..c/4)] \vdash \\ L : K_1; \textbf{mov } esp, ebp \ / \ I \ / \ K_2 \hookrightarrow P \end{array}}{S^{(k)}[ebp = E] \vdash L : K_1 \ / \ \textbf{mov } esp, ebp \ / \ I; K_2 \hookrightarrow P}$$

The $\rightsquigarrow$ symbol denotes the assertion that the given expression reduces to the indicated form where c is required to be a non-negative multiple of 4. The lookup of the *Func* annotation associated with the current procedure (second hypothesis) is performed in order to acquire the value of N which appears in the third hypothesis. The state assignments in the third hypothesis denote the assignment of new unbound variables to any new state members.

Stack memory accesses have similar restrictions. All memory accesses to/from the stack segment must target memory addresses which are given by expressions of the same form as described for *esp*, reducing to $z_{esp} \ominus c$. Again, c must be a multiple of 4 but in this case it can range from $-4(N+1)$ to $4(k-2-N)$ where N is the number of parameters expected by the procedure and

k is the number of stack variables in the current machine state. This range allows access to the parameter list and any local variables. Limiting stack accesses to this range enforces the local scope of the procedure's stack frame.

The following is an example of a rule which writes to a stack variable. Here, the memory being written to is in the stack segment because, by default, EBP is taken to be an offset into the stack segment.

$$\frac{(E \ominus 4) \rightsquigarrow (z_{esp} \ominus c) \qquad S[s_{c/4} \leftarrow 0] \vdash K_1; \mathbf{mov}\ [ebp-4], 0\ /\ I\ /\ K_2 \hookrightarrow P}{S[ebp = E] \vdash K_1\ /\ \mathbf{mov}\ [ebp-4], 0\ /\ I; K_2 \hookrightarrow P}$$

Reads from the stack segment are analogous to the write shown here.

8 A Second Pseudo-Code Translation Example

The basis for the x86 VCGen algorithm has now been shown in its entirety. The collection of derivation rules that have been proposed may appear somewhat complex and difficult to construct, but it was suggested back in section 6 that derivation rules for the various x86 instructions can be obtained fairly straightforwardly from hardware pseudo-code descriptions. Now that the basic ideas behind the derivation rules for various classes of x86 instructions have been explained, a more advanced example of such a pseudo-code translation may be instructive.

The following is the pseudo-code description of the AAA (ASCII Adjust After Addition) instruction given on page 3-11 of [6] (the first instruction listed).

```
IF ((AL AND 0FH) > 9) OR (AF = 1)
    THEN
        AL ← (AL + 6);
        AH ← (AH + 1);
        AF ← 1;
        CF ← 1;
    ELSE
        AF ← 0;
        CF ← 0;
FI;
AL ← AL AND 0FH;
```

As you can see, it includes a number of logical constructs not present in the example in section 6, including an `IF-THEN-ELSE` block with a fairly sophisticated conditional expression. Here is the corresponding AAA Introduction rule:

$$\frac{\begin{array}{ll} S[al \leftarrow (E_1 + 6) \bmod 16, & \\ \quad ah \leftarrow (E_2 + 1) \bmod 256, & S[al \leftarrow E_1 \bmod 16, \\ \quad af \leftarrow true, & \quad af \leftarrow false, \\ \quad cf \leftarrow true] \vdash K_1; \mathbf{aaa}\ /\ I\ /\ K_2 \hookrightarrow P_1 & \quad cf \leftarrow false] \vdash K_1; \mathbf{aaa}\ /\ I\ /\ K_2 \hookrightarrow P_2 \end{array}}{\begin{array}{l} S[al = E_1, ah = E_2, af = EP] \vdash K_1\ /\ \mathbf{aaa}\ /\ I; K_2 \hookrightarrow \\ \qquad ((E_1 \bmod 16 \geq 10) \supset P_1) \wedge ((E_1 \bmod 16 < 10) \wedge \overline{EP}) \supset P_2) \end{array}}$$

Observe that the same technique for handling conditional forward branches in section 7.3.2 has been used to model the `IF-THEN-ELSE` block here. In addition, the technique of representing bitwise `AND`'s with mod operators introduced in section 6 has been used again.

The derivation rule makes the same distinctions. It indicates that assuming either of the two conditional expressions are true, the verification condition corresponding to the first of the two execution paths must hold. Assuming both are false, the verification condition corresponding to the other path must hold. Each of the two possible paths are modeled by hypotheses which make the appropriate assignments to the machine state and generate the needed verification condition.

9 Timing

A brief word should be said about the performance of the VCGen algorithm. Every application of a derivation rule introduces exactly one instruction (or loop marker) to the instruction sequence. Most of the derivation rules stated require only one recursive call to the algorithm. Thus the algorithm is nearly linear in the size of the code being processed. The exception to this are instructions which involve conditional behavior. Conditional branch instructions and the AAA Introduction rule given in section 8 are examples of conditional behavior. Such instructions require two recursive calls to the algorithm causing some instructions in the program to be visited twice as often. However, since each procedure block is processed separately, the effect of each such double recursive call is limited to the scope of a single procedure block. Hence the VCGen algorithm runs in $O(2^b n)$ time where b is the number of instructions per procedure block exhibiting conditional behavior and n is the number of instructions in the entire program.

10 Conclusions and Future Work

The VCGen algorithm that has been presented detects and rejects all programs capable of committing illegal memory accesses, violations of the standard x86 calling convention, or illegal jumps or branches. In order to accomplish this, it conservatively rejects some programs which are safe in all these respects. The major restrictions it imposes on safe programs are:

- Dynamic allocation is not yet supported.
- Typecasting from one bit width to another is not permitted.
- Higher order function programming (function pointers, etc) are not supported.
- Temporary stack variables are limited to 32-bit values which may not "escape" (e.g. get passed by reference to a callee or get accessed from a nested function). Therefore complex datatypes like arrays must be stored as permanent structures in the data segment. (Or, if dynamic allocation is made possible later, they can be represented as temporary allocated structures in the bss segment.)

Each of these current deficiencies could potentially be supported in the future. Dynamic allocation would require the addition of new memory state expressions which represent the allocation and freeing of blocks of memory. Extending the overlapping register solution proposed in section 5.2.3 to memory state expressions could permit typecasting from one bit width to another. Techniques for formally verifying programs which use higher-order functions are known and could be incorporated into the scheme. I also believe that a set of stack-oriented memory predicates like those used to represent structures in the data segment could be combined with the register file

treatment of the stack proposed in section 5.1.3 to permit complex datatypes to be stored on the stack. Such a combination could also probably be used to allow stack variables to "escape." This remains to be explored.

One remaining problem, however, is that the size of any x86 VCGen program is likely to be significantly larger than VCGen programs for RISC architectures. This size increase is due to the need for a rather unique, albeit easily formulated derivation rule for each instruction in the architecture. Space limitations can be partially overcome by storing implementations of the various derivation rules in separate modules. Few x86 programs make use of the full instruction set so that in practice it should not be necessary to load all of these modules into memory for any single program. The trusted computing base, however, remains somewhat large and is therefore perhaps difficult to trust.

Translating pseudo-code hardware specifications into derivation rules is one way to attack this problem. Formalizing this procedure would require the development of a universal architecture specification language capable of expressing the operational semantics of architectures at the instruction set level. Such a language could be used by architecture designers to write specifications which would serve as definitions of correctness during development. These same specifications could then be used by PCC programmers to automatically generate VCGen programs which correctly model the operational semantics of the architectures. Various safety policies for the architectures could then be expressed using the universal hardware specification language. These safety policies could then be incorporated into the VCGen program as part of the VCGen generation process.

Acknowledgements

This work is due in large part to the contributions of George Necula and Peter Lee. I would like to thank them for their introduction to and instruction on Proof-Carrying Code and for guiding me towards important issues to be addressed in an x86 implementation. I would also like to thank them along with James Cheney for their many comments and corrections on earlier drafts of this paper. Finally, I also thank Frank Pfenning for his course on Linear Logic and associated meta-logics which inspired the representation of x86 VCGen as a derivation system.

References

[1] E. W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[2] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[3] G. R. Gircys. *Understanding and Using COFF*. O'Reilly & Associates, Inc., 1988.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[5] Intel Corporation, P.O. Box 7641, Mt. Prospect, IL 60056-7641. *Intel Architecture Software Developer's Manual Vol. 1: Basic Architecture*, 1997. Available as ordering number 243190.

[6] Intel Corporation, P.O. Box 7641, Mt. Prospect, IL 60056-7641. *Intel Architecture Software Developer's Manual Vol. 2: Instruction Set Reference*, 1997. Available as ordering number 243191.

[7] George C. Necula. Proof-carrying code. *24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[8] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, Computer Science Department, Carnegie Mellon University, October 1997.

[9] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *ADM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[10] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Special Issue on Mobile Agents, Lecture Notes in Computer Science, Forthcoming.

[11] G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox, Palo Alto Research Center, June 1981.

[12] Frank Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.