2011 Ninth Annual International Conference on Privacy, Security and Trust

An Implementation of a Verification Condition Generator for Foundational Proof-Carrying Code

Jiangong Weng and Amy Felty
School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, Canada
{jweng014,afelty}@site.uottawa.ca

***Abstract*—Proof-carrying code (PCC) is a technique that addresses the problem of mobile code safety. It is a mechanism in which a code producer provides both code and a proof certifying that the code will run safely on a code consumer's machine. The code consumer or the host system will validate the proof against a safety policy before executing the source code. Foundational proof-carrying code (FPCC) aims to minimize the amount of code that must be trusted (the "trusted computing base" or TCB) with the goal of providing more flexibility and increased security. In both PCC and FPCC, the verification-condition generator (VCG) constructs the statement of the safety theorem from the source code, and is an important part of the TCB. This paper presents an implementation of a VCG based on a sound set of Hoare-style rules for machine instructions in the context of FPCC. The implementation in OCaml is described and examples illustrating the approach are given. The output of our VCG is a list of verification conditions that are directly inserted into a proof script that serves as input to the Coq proof assistant, and represents an important part of the safety proofs of our programs. We also present examples showing how these verification conditions are used to complete the proofs of safety. This work represents an important step in automating proofs for PCC.**

I. Introduction

The starting point for our work is foundational proof-carrying code (FPCC) as it is presented in [1], which adopts the approach introduced in [2], [3]. In [1], several examples are presented in detail based on proofs done interactively using the Coq Proof Assistant [4]. In particular, example machine language programs are given, a set of Hoare-style rules are presented and used to generate verification conditions by hand, and the Coq proofs are described in some detail. The examples illustrated the general structure of such proofs, much of which is repetitive and can be automated. In this paper, we make significant progress toward automating such proofs. Our goal is to automate as much as possible, with full automation for a large class of programs, but leaving open the possibility of completing proofs interactively instead of giving up when automation is difficult. Thus we build a tool that currently generates Coq script that can be read into Coq as a proof with holes that can be filled in by the user. We should ultimately be able to completely automate a large class of proofs, but we don't want to give up when automation does not fully succeed. In the latter case, the automatically generated incomplete proof must be readable by users who will fill in the details. This requirement has an important impact on the design of our tool.

Our verification condition generator (VCG) is implemented in Objective Caml (OCaml), chosen primarily because this kind of functional language is good for manipulating formulas and programs as data. The input is a machine code program, a precondition, a postcondition, and possibly some hints, which are formulas representing preconditions for particular lines of code that are generated automatically by a certifying compiler. The work described in this paper directly extends the original proof-carrying code work [5], which includes a VCG and a certifying compiler. Our VCG can be viewed as an extension of the one in [5], [6] that handles the extensions required for a semantic approach to FPCC. The actual implementation of the VCG consists of the following three steps:

1) We build a lexer and parser using a parser generator. This implementation is straightforward. We define data types in OCaml for instructions and formulas. The input is translated to the corresponding internal data structures. In particular, the program becomes a list of instructions, and the other inputs become a list of formulas annotated by line numbers. The hints are annotated with the (relative) address in memory where the corresponding machine instruction is stored, the precondition is annotated with 0, and the postcondition is annotated with the address of the last instruction.
2) Starting with the last machine instruction and the postcondition (VCpost), we first generate an output precondition (VCpre) for this line using Hoare-style rules for machines instructions. The output VCpre is seen as the new input VCpost for next instruction moving backward, and the rest of VCs are generated automatically using a bottom-up strategy. Together, the calculated VCs represent a kind of program invariant, where each formula is associated with a line of code and represents the part of the invariant that holds just before that line is executed. Any input hints become part of the invariant for the line of code that they are associated with. Generating these invariants is the main step that we discuss throughout this paper.
3) We use a pretty-print method to produce Coq script from the internal representation of the machine code and verification conditions, to be used directly as input to Coq. Again, this implementation is straightforward.

The generated Coq file contains a series of definitions as well as a few simple lemmas whose proofs can be generated

978-1-4577-0584-7/11/$26.00©2011 IEEE

automatically. This file imports four libraries that were used as the basis for the work in [1], two of which are modified versions of libraries developed for the Coq formalization of syntactic FPCC [7]. Our system, like any FPCC system, is built in layers so that reasoning about particular programs is done at a fairly high level. At the highest level is the *safety theorem*, which is the main theorem proved in these libraries. It states that (1) if the invariant holds before executing the program, and the program satisfies the properties of (2) progress and (3) preservation, then the program is safe to execute whenever the precondition holds. In PCC in general, the definition of safety can vary, and ideally depends on what code consumers designate as safe execution on their own machines. The safety policy is specified as a set of inference rules. Here, we take memory safety as an example, which allows the code consumer to restrict the parts of memory that can be read from and written to. The Coq libraries contain the necessary infrastructure for proving programs are safe with respect to this safety policy. These libraries along with the Coq file generated by our VCG provide all the infrastructure that is needed to complete proofs of the above three properties for a specific program. Completing a proof of safety means instantiating the three hypotheses of the safety theorem with a particular program and proving them, thus establishing that the program is safe to execute. Progress states that whenever the invariant holds, it is possible for execution to proceed with a safe step. Preservation states that whenever a safe step is made, the new state that is reached also satisfies the invariant. The proofs of all three of these properties can be mostly automated, but this remains as future work.

In Section II, we present the internal data structures that are used to encode machine instructions and formulas. Then in Section III, we discuss the VCG algorithm and give details for two machine instructions, which is followed in Section IV by an example illustrating the algorithm. In Section V, we discuss the Coq proofs of safety, illustrating how the results of the VCG fit into proofs as a whole, and in Section VI, we conclude and discuss related and future work.

II. Data Type Definitions

We begin by introducing formal definitions for basic terms, machine instructions, and formulas used in defining invariants. We present them in Fig. 1 with syntax close to OCaml's recursive definitions. The `MemExp` constructor in the type for terms will be composed of a list of terms of the form `HasValue(`n`,`v`)` and a term that denotes a memory. Here, `HasValue` has two arguments n and v, which express a mapping from address n to value v, and it is used to describe memory states, e.g., after execution of a store instruction which updates the memory. We use the mathematical notation $m[n \mapsto v]$ to define such memory updates, where m is the original memory, n is an address in m and v is its new value. When we want to look up the value in memory m at address a, we write $m(a)$. The expression $m[n \mapsto v](a)$ represents the value at address a in a new memory obtained by updating m at address a with value v. (Thus if a is n, then the value is v.) In addition, if there are several updates, e.g., three store instructions in a row, then a new memory can be expressed as $m[n_1 \mapsto v_1, n_2 \mapsto v_2, n_3 \mapsto v_3]$. We abbreviate this new memory as m' and refer to it in the examples below. In our OCaml notation, m' is represented as an ordered list: `[HasValue(n3,v3)), HasValue(n2,v2), HasValue(n1, v1)]`. Note that m does not appear in the OCaml term. In our programs, m is implicit and can always be recovered from context.

```
type term =
  num of int
| reg of int.
| Var of string
| Plus of term * term
| HasValue of term * term
| MemExp of term list * term

type instr =
  Addc of reg * reg * num
| Add of reg * reg * reg
| St of reg * num * reg
| Mov of reg * num
| Ld of reg * reg * num
| Jmp of reg
| Beq of reg * reg * num
| Bgt of reg * reg * num

type form ->
  True
| False
| And of form * form
| Imp of form * form
| Forall of term * form
| Exists of term * form
| Geq of term * term              ( >= )
| Eq of term * term               (=)
| Noteq of term * term            (!=)
| Gthan of term * term            (>)
| Leq of term * term              (<=)
| Preds of string * term list
| HasType of term * term list * term * string
```

Fig. 1. OCaml Definitions for Basic Data Types

In the definition of type `instr`, the types `reg` and `num` are actually synonyms for `term`, used to indicate what form of term appears in each argument. For example, consider the instruction $ADDC\ r_1 := r_8 + 0$. The first constructor `Addc` of type `instr` in Fig. 1 takes three arguments. For this example, the first represents r_1 as `Reg(1)`, the second term represents r_8 as `Reg(8)`, while the third represents 0 as `Num(0)`. Hence, the corresponding OCaml term would be `Addc(Reg(1), Reg(8), Num(0))`. The other instructions are similar.

In the definition of the type `form`, clearly `And` and `Imp` represent conjunction and implication each taking two arguments. The quantifiers `Forall` and `Exists` both have a term argument to represent the bound variable, built using constructor `Var` from the definition of `term`. The second argument contains a formula that may have occurrences of this bound

variable. For instance, the statement $\forall x, x = 1 \Rightarrow x > 0$ has the OCaml notation:

```
Forall(Var(x), Imp(Eq(Var(x), Num(1)),
                   Gthan(Var(x), Num(0)))).
```

Constructors for binary predicates on terms are also given in Fig. 1 along with the mathematical notation for the operators they represent. Note that `Preds` takes a string and a term list. The former represents a predicate name which takes as arguments all the terms in the latter. For example, we can write `Preds("writable", [Num(n)])` to represent the fact that the memory address n is writable. The `Preds` constructor is not limited to defining memory attributes, but could also be used to express any user-defined predicates used in the safety proof. Thus, introducing `Preds` allows our implementation to be parameterized.

`HasType` is used for the typing judgment, which in mathematical notation is $(w :_{m,A} t)$, meaning that w is a term having type t in memory m with allocation set A (denoting a set of allocated memory addresses). Reasoning about types is important for proving memory safety, which as mentioned, is the central component of the safety policy considered in this paper. In this context, typing judgments are fairly low-level, and depend on the contents of memory and on the set of allocated memory addresses. For instance, the typing judgment $(r_1 :_{m',A'} \mathit{intlist})$ states that the contents of r_1 is an address whose valuse has type $\mathit{intlist}$ in m' with A', where we define A' to be the allocation set $\{w | r_8 < w \wedge w < \mathit{start}\}$ for some fixed address start and some allocation pointer r_8. This judgment is represented in OCaml as:

```
HasType(Reg(8), [HasValue(n1, v1),
                 HasValue(n2, v2),
                 HasValue(n3, v3)],
        Reg(1),"intlist")
```

where the allocation set is represented simply by the allocation pointer `Reg(8)` as the first argument and memory m' is represented as discussed earlier as the second argument.

III. Implementation of the VCG

As mentioned earlier, a bottom up approach is used for generating each verification condition (step 2 in Section I), and the generation is based on Hoare-style rules for machine instructions, which are shown here in Fig. 2. There is one rule for each machine instruction and these rules are axioms. The mov rule is a version of the usual assignment rule where the precondition is obtained by starting with the postcondition and replacing occurrences of the destination register r_d with constant c. The two rules for addition are similar to the mov rule. In the ld rule, $r_s + c$ is an address, and the value at that address in memory is the value that replaces r_d. In addition $r_s + c$ must be a readable address, i.e., $\mathit{readable}(r_s + c)$ must be provable from the safety policy. In the st rule, it is the entire memory in the postcondition that is replaced by an expression representing a new memory, to obtain the precondition. This new memory is the same as the old except for an update at one address (address $r_d + c$). In addition, the precondition assures that address $r_d + c$ is writable. The formula P_c in the rules for the conditional jump instructions (bgt and beq) is the precondition of the statement at relative location c, which is the precondition of the line of code at the jump point. In fact, we require input hints for all jump points, which insures that the precondition for these jump instructions can always be computed.

Our formal proofs in Coq do not implement these rules directly. Instead, the Coq libraries implement the FPCC approach which encodes the low-level semantics of machine instructions directly, and defines safety directly from this semantics. The rules are presented here to illustrate the form of the invariant generated by our VCG. This invariant is used directly in the proof; it is needed to complete the proofs of progress and preservation.

The input for each step of the VCG is a machine instruction and its postcondition (the second and third elements of the Hoare triples in Fig. 2), and from these we compute the precondition (the first element of a Hoare triple), which is then used as the postcondition for the preceding instruction. Thus our VCG starts with the postcondition of the program as a whole and the last machine instruction, and continues backward, until it computes the precondition of the first instruction. The output is a list containing all of these verification conditions. We present the details for two instructions. The others implement the rules from Fig. 2 in a similar fashion.

Fig. 3 contains the top-level function, which calls a helper function specific to the input instruction. Note that for the ST instruction, there is an additional conjunct added to express the constraint that the memory location that will be written to is indeed a writable location. Similarly, the clause for the LD instruction includes the constraint that reading from the specified memory location is allowed. Figures 4 and 5 contain the two helper functions specific for the $ADDC$ and ST instructions, each having auxiliary functions used to process terms that appear as arguments to `Preds` and `HasType`. When the input instruction has the form $ADDC\ r_d := r_s + c$, note that the main work is done by `updAddcTerm` in the clause for `Reg(i)`. In the case when `i` is d, r_d is replaced by $r_s + c$. The definition of `updAddcTerms` is omitted. This function is a standard map function on lists, in this case applying `updAddcTerm` to each of the elements of the input list.

Note that most of the definition of `updSt` in Fig. 5 is omitted since its structure is the same as `updAddc` in Fig. 4. In the definition of `updSt`, the main work is done on subformulas of the form `HasType(ta,ts,tb,ty)` or subterms of the form `MemExp(ts,t)` occurring in the postcondition. In these cases `updStMem` is called to add a new `HasValue` pair to the list representing memory. The function `append` is the standard one for concatenating lists.

IV. Examples

To illustrate the operation of the VCG, we give some examples. The first illustrates the execution of the algorithm

$$\frac{}{\{P[c/r_d]\}\ MOV\ \ r_d := c\ \{P\}}\ \text{mov}$$

$$\frac{}{\{P[r_s + c/r_d]\}\ ADDC\ r_d := r_s + c\ \{P\}}\ \text{addc} \qquad \frac{}{\{P[r_{s_1} + r_{s_2}/r_d]\}\ ADD\ r_d := r_{s_1} + r_{s_2}\ \{P\}}\ \text{add}$$

$$\frac{}{\{P[m(r_s + c)/r_d] \wedge \mathit{readable}(r_s + c)\}\ LD\ r_d := m(r_s + c)\ \{P\}}\ \text{ld}$$

$$\frac{}{\{P[m[r_d + c \mapsto r_s]/m] \wedge \mathit{writable}(r_d + c)\}\ ST\ m(r_d + c) := r_s\ \{P\}}\ \text{st}$$

$$\frac{}{\{(r_{s_1} > r_{s_2} \rightarrow P_c) \wedge (\neg(r_{s_1} > r_{s_2}) \rightarrow P)\}\ BGT\ (r_{s_1} > r_{s_2})\ c\ \{P\}}\ \text{bgt}$$

$$\frac{}{\{(r_{s_1} = r_{s_2} \rightarrow P_c) \wedge (r_{s_1} \neq r_{s_2} \rightarrow P)\}\ BEQ\ (r_{s_1} = r_{s_2})\ c\ \{P\}}\ \text{beq}$$

$$\frac{\{A\}S_1\{C\} \quad \{C\}S_2\{B\}}{\{A\}S_1; S_2\{B\}}\ \text{sequence} \qquad \frac{A \Rightarrow A' \quad \{A'\}S\{B'\} \quad B' \Rightarrow B}{\{A\}S\{B\}}\ \text{Implied}$$

Fig. 2. Hoare-style Rules for Machine Instructions

Main update algorithm:
Input: 1) `I` (a machine instruction) 2) `VCpost` (the postcondition of I)
Output: `VCPre` (a verification condition representing the precondition of `I`)

```
match I with
    Addc(_, _, _)          -> updAddc(I,VCpost)
  | Add(_, _, _)           -> updAdd(I,VCpost)
  | St(Reg(r), Num(n), _)  -> And(updSt(I,VCpost), writable(Plus(Reg(r),Num(n))))
  | Mov(_, _)              -> updMov(I,VCpost)
  | Ld(_, Reg(r), Num(n))  -> And(updLd(I,VCpost), readable(Plus(Reg(r),Num(n))))
  | Jmp(_)                 -> updJmp(I,VCpost)
  | Bgt(_, _, _)           -> updBgt(I,VCpost)
  | Beq(_, _, _)           -> updBeq(I,VCpost)
```

Fig. 3. Algorithm for Computing Preconditions

with input instruction $ADDC\ r_3 := r_7 + 1$ and postcondition:

$$(r_3 > 0 \wedge r_1 :_{m[r_2 \mapsto r_3, r_6 \mapsto 0], A'}\ \mathit{intlist})$$

where A' represents the allocation set $\{w | r_8 < w \wedge w < \mathit{start}\}$ as before. We call the main update algorithm in Fig. 3 with the above arguments as inputs `I` and `VCpost`, shown below in OCaml notation.

```
Addc(Reg(3), Reg(7), Num(1))
And(Gthan(Reg(3), Num(0)),
    HasType(Reg(8),
            [HasValue(Reg(2),Reg(3)),
             HasValue(Reg(6), Num(0))],
            Reg(1), Var("intlist")))
```

This function applies `updAddc` in Fig. 4 to `VCPost` to obtain `VCpre`. The steps are shown below. The first argument `I` to all function calls that are shown is omitted, since it does not change. The successive function calls are underlined, and term replacements done by `updAddcTerm` are shown in bold.

```
And(updAddc(Gthan(Reg(3), Num(0))),
    updAddc(HasType(Reg(8),
                [HasValue(Reg(2),Reg(3)),
                 HasValue(Reg(6), Num(0))],
                Reg(1), Var("intlist")))

And(Gthan(updAddcTerm(Reg(3)),
          updAddcTerm(Num(0))),
    HasType(updAddcTerm(Reg(8)),
            updAddcTerms(
              [HasValue(Reg(2),Reg(3)),
               HasValue(Reg(6), Num(0))]),
            updAddcTerm(Reg(1)),
            Var("intlist")))

And(Gthan(Plus(Reg(7),Num(1)), Num(0)),
    HasType(Reg(8),
      [updAddcTerm(
         HasValue(Reg(2),Reg(3))),
```

Update algorithm for $Addc$ (`updAddc`):
Input: 1) `I` (a machine instruction) 2) `VCpost` (the postcondition of I)
Output: `VCPre` (a verification condition representing the precondition of `I`)

```
match VCpost with
    And(P1, P2)          -> And(updAddc(I,P1), updAddc(I,P2))
  | Imp(P1, P2)          -> Imp(updAddc(I,P1), updAddc(I,P2))
  | Forall(Var(str), P)  -> Forall(Var(str), updAddc(I,P))
  | Exists(Var(str), P)  -> Exists(Var(str), updAddc(I,P))
  | Geq(t1, t2)          -> Geq(updAddcTerm(I,t1), updAddcTerm(I,t2))
  | Eq(t1, t2)           -> Eq(updAddcTerm(I,t1), updAddcTerm(I,t2))
  | Noteq(t1, t2)        -> Noteq(updAddcTerm(I,t1), updAddcTerm(I,t2))
  | Gthan(t1, t2)        -> Gthan(updAddcTerm(I,t1), updAddcTerm(I,t2))
  | Leq(t1, t2)          -> Leq(updAddcTerm(I,t1), updAddcTerm(I,t2))
  | Preds(Var(str), ts)  -> Preds(Var(str), updAddCTerms(I,ts))
  | HasType(ta, ts, tb, Var(str)) ->
                            HasType(updAddcTerm(I,ta), updAddCTerms(I,ts),
                                    updAddcTerm(I,tb),Var(str))
```

Update algorithm for terms in `VCpost` of `Addc` (`updAddcTerm`):
Input: 1) `I` (a machine instruction) 2) `term`
Output: an updated term

```
match term with
    Num(n)             -> Num(n)
  | Var(str)           -> Var(str)
  | Reg(i)             -> match I with
                            Addc(Reg(d), Reg(s), Num(c)) ->
                               if (i=d) then Plus(Reg(s), Num(c)) else Reg(i)
  | Plus (t1, t2)      -> Plus(updAddcTerm (I,t1), updAddcTerm (I,t2))
  | HasValue (t1, t2)  -> HasValue(updAddcTerm (I,t1), updAddcTerm (I,t2))
  | MemExp (ts, t)     -> MemExp(updAddcTerms(I,ts), updAddcTerm(I,t))
```

Fig. 4. Computing the Preconditions for the $ADDC$ Instruction

```
       updAddcTerm(
         HasValue(Reg(6), Num(0)))],
      Reg(1), Var("intlist")))

And(Gthan(Plus(Reg(7),Num(1)), Num(0)),
    HasType(Reg(8),
      [HasValue(updAddcTerm(Reg(2)),
                updAddcTerm(Reg(3))),
       HasValue(updAddcTerm(Reg(6)),
                updAddcTerm(Num(0)))],
      Reg(1), Var("intlist")))

And(Gthan(Plus(Reg(7),Num(1)), Num(0)),
    HasType(Reg(8),
      [HasValue(Reg(2),
                Plus(Reg(7),Num(1))),
       HasValue(Reg(6), Num(0))],
      Reg(1), Var("intlist")))
```

Thus, the result obtained as the value of `VCPre` at the last step, in mathematical notation is:

$$(r_7 + 1 > 0 \wedge r_1 :_{m[r_2 \mapsto (r_7+1), r_6 \mapsto 0], A'} \mathit{intlist})$$

The next example illustrates the algorithm applied to an ST instruction. This time, the example input instruction is $ST\ m(r_8 + 1) := r_3$ and the example postcondition is:

$$(r_1 :_{m[r_8+0 \mapsto 5], A'} \mathit{intlist} \wedge m(r_2) \neq 0)$$

with the same allocation set A' as before. The input `I` and `VCpost` to the main update algorithm in Fig. 3 in OCaml notation is:

```
St(Reg(8), Num(1), Reg(3))
And(HasType(Reg(8),
            [HasValue(Plus(Reg(8), Num(0)),
                      Num(5))],
            Reg(1), Var("intlist")),
    Noteq(MemExp([], Reg(2)), Num(0)))
```

This time, the top-level function applies `updSt` in Fig. 5, and we have the following steps applied to `VCpost` to obtain `VCpre`:

```
And(And(updSt(HasType(Reg(8),
                [HasValue(Plus(Reg(8),
                               Num(0)),
                          Num(5))],
```

Update algorithm for St (updSt):
Input: 1) I (a machine instruction) 2) VCpost (the postcondition of I)
Output: VCPre (a verification condition representing the precondition of I)

```
match VCpost with
    And(P1, P2)          -> And(updSt(I,P1), updSt(I,P2))
  | ...
  | HasType(ta, ts, tb, Var(str)) ->
                           HasType(updStTerm(I,ta), updStMem(I,ts),
                                   updStTerm(I,tb), Var(str))
```

Update algorithm for terms in VCpost of St (updStTerm):
Input: 1) I (a machine instruction) 2) term
Output: an updated term

```
match term with
    Num(n)            -> Num(n)
  | Var(str)          -> Var(str)
  | Reg(i)            -> Reg(i)
  | Plus(t1, t2)      -> Plus(updStTerm (I,t1), updStTerm (I,t2))
  | HasValue(t1, t2)  -> HasValue(updStTerm (I,t1), updStTerm (I,t2))
  | MemExp(ts, t)     -> MemExp(updStMem(I,ts), updStTerm(I,t))
```

Update algorithm for memory expressions in VCpost of St (updStMem):
Input: 1) I (a machine instruction) 2) ts (a list of terms)
Output: an updated list of terms

```
match I with
    St(Reg(d), Num(c), Reg(s)) ->
      append(updateStTerms(I, ts),[HasValue(Plus(Reg(d), Num(c)), Reg(s))])
```

Fig. 5. Computing the Preconditions for the ST Instruction

```
                  Reg(1), Var("intlist"))),
        updSt(Noteq(MemExp([], Reg(2)),
                    Num(0)))),
    writable(Plus(Reg(8), Num(1))))
```

Here, new subformulas and subterms are shown in bold as they appear in each step. From this point on, we show only the first 2 conjuncts, since the writable conjunct does not change.

```
And(HasType(updStTerm(Reg(8)),
      updStMem([HasValue(Plus(Reg(8),
                              Num(0)),
                         Num(5))]),
      updStTerm(Reg(1)), Var("intlist")),
    Noteq(updStTerm(MemExp([], Reg(2))),
          updStTerm(Num(0))))

And(HasType(Reg(8),
      [updStTerm(HasValue(Plus(Reg(8),
                               Num(0)),
                          Num(5))),
                 HasValue(Plus(Reg(8),
                               Num(1)),
                          Reg(3))],
      Reg(1), Var("intlist")),
    Noteq(MemExp(updStMem([]),
                 updStTerm(Reg(2))),
          Num(0)))

And(HasType(Reg(8),
      [HasValue(Plus(Reg(8), Num(0)),
                Num(5)),
       HasValue(Plus(Reg(8),Num(1)),
                Reg(3))],
      Reg(1), Var("intlist")),
    Noteq(MemExp([HasValue(Plus(Reg(8),
                                Num(1)),
                           Reg(3))],
                 Reg(2)),
          Num(0)))
```

Thus, the result obtained as the value of VCPre at the last step, in mathematical notation is:

$$r_1 :_{m[r_8+0\mapsto 5, r_8+1\mapsto r_3], A'} intlist \wedge \\ m[r_8+1 \mapsto r_3](r_2) \neq 0 \wedge writable(r_8+1))$$

V. Integrating the VCG Output into a Coq Proof

In this section, we briefly describe how the results of the VCG, which as mentioned represent a program invariant, are used in the Coq proof of safety for the program. As mentioned, FPCC encodes the low-level semantics of machine instructions

$$
\begin{array}{rcl}
safe(R,M,pc) & := & \forall R',M',pc'.[(R,M,pc \mapsto^* R',M',pc') \Rightarrow \\
 & & \quad \exists R'',M'',pc''.(R',M',pc' \mapsto R'',M'',pc'')] \\
Progress(Inv) & := & \forall R,M,pc.[Inv(R,M,pc) \Rightarrow \\
 & & \quad \exists R',M',pc'.(R,M,pc \mapsto R',M',pc')] \\
Preservation(Inv) & := & \forall R,M,pc,R',M',pc'.Inv(R,M,pc) \Rightarrow \\
 & & \quad (R,M,pc \mapsto R',M',pc') \Rightarrow Inv(R',M',pc')
\end{array}
$$

$$
\frac{Inv(R,M,pc) \qquad Progress(Inv) \qquad Preservation(Inv)}{safe(R,M,pc)}\ \textsf{safety}
$$

Fig. 6. Definitions and Lemmas for Proving Safety

directly. The main building block for this semantics is the definition of a step relation, which defines how the program state changes during one step of computation. The state is represented as a triple of the form (R, M, pc) where R is a register bank, M is a memory, and pc represents the value of the program counter. We write $(R, M, pc \mapsto R', M', pc')$ to denote the step relation, where the triple on the left represents the machine state before executing an instruction, and the triple on the right represents the state after execution. We write $(R, M, pc \mapsto^* R', M', pc')$ to denote its reflexive transitive closure. For example, the state change resulting from a ST instruction is expressed as the following clause in the definition of the step relation:

$$
\begin{array}{l}
ST\ m(r_d + c) := r_s \Rightarrow \\
\quad writable(R_d + c) \wedge R' = R \wedge \\
\quad M' = M[R_d + c \mapsto R_s] \wedge pc' = pc + 1
\end{array}
$$

where R_d and R_s are the values of the contents of r_d and r_s, respectively, before the execution of the instruction. In the new state after execution of a ST instruction, the register bank remains unchanged, the memory is updated at address $R_d + c$, and the program counter is incremented by 1. Note that according to this definition, the step can only take place if the location $R_d + c$ is indeed writable.

The full invariant used in the proof has the general form:

$$
\begin{array}{l}
Inv(R,M,pc) := \\
\quad [prog(M) \wedge \\
\quad ((pc = 1 \wedge P_1(R)) \vee \cdots \vee (pc = n \wedge P_n(R)))] \vee \\
\quad safe(R,M,pc)
\end{array}
$$

Here, $prog(M)$ is a predicate stating which instructions are at which addresses in memory in the program, and is part of the invariant because it is necessary to know that the code does not get modified during execution. The main content of the invariant is a formula with n disjuncts, where n is the number of instructions. For $i = 1, \ldots, n$, P_i represents the verification condition generated for the i^{th} instruction. The last subformula, $safe(R, M, pc)$ is included for the end of the program, which can be viewed as a return statement that is always safe. The part of the proof that uses it is straightforward (and fully automated), so we do not discuss it here. The definitions of $safe$, as well as those of progress and preservation (described informally earlier), are shown in Fig. 6 along with the precise statement of the safety theorem. The definition of $safe$ simply states that execution of a safe program never gets stuck.

In proving the safety theorem, the proof that the invariant follows from the precondition is usually very simple. The only interesting parts of the proof left to be filled in after running our VCG are the proofs of progress and preservation. Both have the invariant as a hypothesis, which is used to break the proof into cases. This is also done automatically, and the user is left only to fill in the cases for each proof. Most cases of the progress proof have the form:

$$
\begin{array}{l}
(prog(M) \wedge pc = i \wedge P_i) \Rightarrow \\
\quad \exists R'\ \exists M'\ \exists pc'\ (R,M,i \mapsto R',M',pc')
\end{array}
$$

and most cases of the preservation proof have the form:

$$
[(R,M,pc \mapsto R',M',pc') \wedge prog(M) \wedge pc = i \wedge P_i] \Rightarrow P_{i+1}.
$$

Usually each case is simple and the proofs for each individual instruction follow a particular pattern. For example, in a proof, a case for a ST instruction is likely to be similar to other cases for other ST instructions. If there were any difficulty in the proof, it is likely to be at the jump points, though even these are very often simple. Thus, as argued earlier, full automation for a large class of programs should be possible.

It is worth noting that since we have implemented a semantic approach to FPCC, primitive typing rules are not included. Instead, we encode a semantics of types from first principles and the typing rules are proved as lemmas. Proofs of safety for individual programs will use typing rules, but whether they are stated as axioms or proved as lemmas does not affect the work described in this paper. Overall, though, since we adopt an approach that replaces axioms with proofs from first principles, the system as a whole is more trustworthy.

VI. Related and Future Work

Though FPCC has the advantage of having a smaller TCB than PCC as it was originally formulated, it still has some components that must be trusted such as the proof checker, which in our setting is Coq. We plan to examine replacing Coq with simpler proof checkers such as the ones mentioned in [8] that have been mechanically verified. Note that our VCG is not part of the TCB, but is instead used to generate invariants used in the premise of the safety theorem.

As discussed in Section III, jump instructions require hints to be provided. Finding such hints by hand, especially for

a large program is a daunting task. Certifying compilers, which can use information gained by analyzing the program during compilation, can help by outputting hints that help with reasoning. However, it might also be plausible for the code producer to provide more details such as global invariants that help can further help the compiler in finding such hints. This is also the subject of future work.

One of the main areas of future work is to further automate the proofs of progress and preservation. As discussed, these two proofs are generally constructed by a series of rule applications that follow a similar pattern. Exploiting these patterns will allow more automation work to be done. In particular, some additional proof tactics to automate this task in Coq could be introduced.

Chang et. al. [9] argue that because there exist a variety of code verification strategies, it is best to use a verifier that is best suited to the code verification strategy. Most examples of safety policies, including the ones considered here, have been simple. The setting described here is general and flexible and may be a good starting point for handling a variety of strategies. This is also a subject of future work.

One subject of active current work is to extend the kinds of safety policies that can be handled by this approach. We are currently working on including pointers to our machine language and defining the safety policy to rule out dangling pointers. We also plan to incorporate the kinds of extended policies considered in [8], which include non-interference and resource control. As mentioned in that paper, FPCC has the potential to support a wider range of security policies than other approaches, and in addition can easily support various different verification methods.

An important issue in PCC is the size of proofs (also called *certificates*). It would be interesting to incorporate the kind of certificate used in the *reflective* PCC approach [8], which replaces deduction by computation (also using Coq), and tends to result in smaller certificates. Certificate size is also addressed in [10], where a proof is replaced by a smaller proof generator program that is executed in a secure virtual machine and is part of the TCB. It should be possible to apply this kind of compression to the proofs generated in our setting.

Another goal is to leverage work on certificate translation [11]. In our setting, this would allow the difficult parts of the proof to be done interactively using the higher-level source program instead of the lower level machine program. Certificate translation is used to translate complete proofs at the higher level to complete proofs at the lower level. In doing so, we may also be able to leverage the automation capabilities of the JACK tool [12], which provides proof automation for properties of Java programs, and translates them to the bytecode level. JACK extends earlier work on the successful automation in the LOOP [13] and ESC/Java2 [14] tools, with the goal of obtaining a tool that fits into a general PCC system usable by application developers.

Working in another direction, we can also exploit the power of Coq to prove more complex properties of programs. We do so by simply expressing the desired property as a postcondition. If the property is difficult to prove, the user will have to complete more of the proof interactively, but will have access to Coq's full power to do so. Here again, certificate translation may help, since it would allow the user to complete the proof at the level of the source program.

ACKNOWLEDGMENT

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] A. P. Felty, "Tutorial examples of the semantic approach to foundational proof-carrying code," *Fundamenta Informaticae*, vol. 77, no. 4, pp. 303–330, 2007.

[2] A. W. Appel and A. P. Felty, "A semantic model of types and machine instructions for proof-carrying code," in *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 2000, pp. 243–253.

[3] A. W. Appel, "Foundational proof-carrying code," in *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, Jun. 2001, pp. 247–256.

[4] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[5] G. Necula, "Proof-carrying code," in *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 1997, pp. 106–119.

[6] G. C. Necula, "Compiling with proofs," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, Sep. 1998.

[7] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, "A syntactic approach to foundational proof-carrying code," *Journal of Automated Reasoning*, vol. 31, no. 3–4, pp. 191–229, Nov. 2003.

[8] G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie, "The MOBIUS proof carrying code infrastructure (an overview)," in *Seventh International Symposium on Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, vol. 5382. Springer, 2007, pp. 1–24.

[9] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck, "The open verifier framework for foundational verifiers," in *ACM SIGPLAN International Workshop on Types in Language Design and Implementation*. ACM Press, Jan. 2005, pp. 1–12.

[10] H. Pirzadeh, D. Dubé, and A. Hamou-Lhadj, "An extended proof-carrying code framework for security enforcement," in *Transactions on Computational Science XI*, ser. Lecture Notes in Computer Science, vol. 6480. Springer, 2010, pp. 249–269.

[11] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk, "Certificate translation for optimizing compilers," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 5, pp. 1–45, 2009.

[12] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet, "JACK—a tool for validation of security and behavior of java applications," in *Sixth International Symposium on Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, vol. 4709. Springer, 2007, pp. 152–174.

[13] J. van den Berg and B. Jacobs, "JACK—a tool for validation of security and behavior of java applications," in *Seventh International Conference on Tools and Algorithms form the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 2031. Springer, 2001, pp. 299–312.

[14] J. van den Berg and B. Jacobs, "ESC/Java2: Uniting ESC/Java and JML," in *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, ser. Lecture Notes in Computer Science, vol. 3362. Springer, 2005, pp. 108–128.