Published in "Logic in Computer Science", LICS, 2003.

A Sound Framework for Untrusted Verification-Condition Generators

George C. Necula* Robert R. Schneck
University of California, Berkeley
necula@cs.berkeley.edu schneck@math.berkeley.edu

Abstract

We propose a framework called configurable proof-carrying code, *which allows the untrusted producer of mobile code to provide the bulk of the code verifier used by a code receiver to check the safety of the received code. The resulting system is both more flexible and also more trustworthy than a standard proof-carrying code system, because only a small part of the verifier needs to be trusted, while the remaining part can be configured freely to suit the safety policy on one hand, and the structure of the mobile code on the other hand.*

In this paper we describe formally the protocol that the untrusted verifier must follow in the interaction with the trusted infrastructure. We present a proof of the soundness of the system, and we give preliminary evidence that the architecture is expressive enough to delegate to the untrusted verifier even the handling of loop invariants, indirect jumps and calling conventions.

1. Introduction

Proof-carrying code (PCC) [16, 19] is a method that allows a software system to verify easily that code received from an untrusted party has certain desired properties. The code producer generates a proof to that effect; the code receiver verifies that this proof is correct and also that it pertains to the code in question. The key advantage of PCC is that it enables the code receiver to check even complicated properties using a small trusted verifier.

At present there are several implementations of PCC. On one hand, there is the Touchstone system [6, 17], which verifies that optimized native machine code produced by a special Java compiler is memory safe. Touchstone uses a *verification-condition generator* (VCGen) to examine the program and produce a theorem, a proof of which guarantees the safety of the program. Touchstone automates the generation of the safety proof, and most notably from the point of view of this paper, it is able to verify quickly even programs of up to one million instructions. This level of scalability has been achieved through careful engineering of the data structures used in the implementation of the verifier along with a number of novel algorithmic "tricks". This does mean that the verifier code, while still much smaller than an alternative trusted compiler, is far from being easy to understand and trust. In fact, there were errors [13] in that code that escaped the thorough testing of the infrastructure.

Another family of PCC implementations, known as Foundational Proof-Carrying Code (FPCC) [2, 1, 15, 11], are designed to reduce the trusted computing base to a minimum, by eliminating the verification-condition generator, and instead requiring that all such program analysis (and its correctness) be incorporated logically into the safety proof. Once completed, an FPCC implementation is expected to contain about an order of magnitude less trusted code than Touchstone; moreover such an implementation would have increased flexibility by having no restriction on the type system or other method used to guarantee safety (since whatever method is used, it must be described in foundational logic and proven formally sound). However, the development of FPCC is much more labor-intensive than that of Touchstone. This is to be expected since besides programming the verifier one has to prove formally its correctness. And the advance from a basic implementation of FPCC to one that incorporates all of the complex scalability improvements in Touchstone is likely to be very costly as well.

Touchstone and FPCC constitute two extremes in the PCC design spectrum. We propose in this paper an alternative design, called *configurable proof-carrying code (CPCC)*, which has the potential to combine the efficiency, scalability and low development cost of Touchstone with the enhanced trustworthiness and flexibility of FPCC. We want to be able to keep the complex algorithms and their implementations that make Touchstone so efficient. Yet we

*This research was supported in part by the National Science Foundation Career Grant No. CCR-9875171, and ITR Grants No. CCR-0085949 and No. CCR-0081588, Air Force contract no. F33615-00-C-1693, and gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

do not want to have to trust those implementations. And we realize that proving formally their correctness would be a gargantuan task. An alternative is suggested by research results in run-time result checking [24] and translation validation [23, 18]: instead of formally proving the correctness of a software system, verify the correctness of each individual run. The difficulty in such an approach is to define precisely the correctness criterion for an execution of the untrusted system, especially in this context where we desire to retain efficiency of construction, transmission, and checking of the correctness proofs. But once this is done, the actual verification of the result can be done without any knowledge of the internal data structures, algorithms and heuristics that the untrusted system uses.

In essence we propose that the code producer supply not only the code but also most of the verifier for the code. We show a strategy for architecting a verifier such that only a small core part needs to be trusted; the core verifier will check certain critical aspects of the results obtained from the untrusted verifier.

Such an architecture has software engineering advantages for any application that uses verification-condition generation, such as theorem provers or symbolic program analyzers. A more significant potential application is for generic safe virtual machines, which can be customized by untrusted clients to verify diverse safety policies. At present, one has to choose among hosts featuring a small number of distinct code verifiers (e.g., for the Java Virtual Machine (JVM) [14], or for the Microsoft Common Language Infrastructure (CLI)). Alternatively, those hosts could provide only a core generic verifier allowing the clients to upload the rest, and thus to create their own verifiers, for JVM or CLI or anything that fits the mobile code of interest. Of course, in order for this vision to become reality we must solve other configuration problems besides that of the verifier, for example that of the generic runtime system.

In Section 1.1 we describe at high level the proposed architecture for configurable proof-carrying code. We continue in Section 2 with a precise description of both the trusted and untrusted components of the verifier, and the protocol of interaction between them; in Appendix A we prove the soundness of the verification process. For reasons of space and presentation, we omit certain refinements that are useful for engineering reasons and present the system at a relatively high level. In the final part of the paper we demonstrate the flexibility of the architecture by designing untrusted verifiers for handling loops and function calls.

1.1. Configurable proof-carrying code

The architecture of the proposed configurable proof-carrying code (CPCC) system is shown in Figure 1. Below the double line are shown the untrusted elements (data

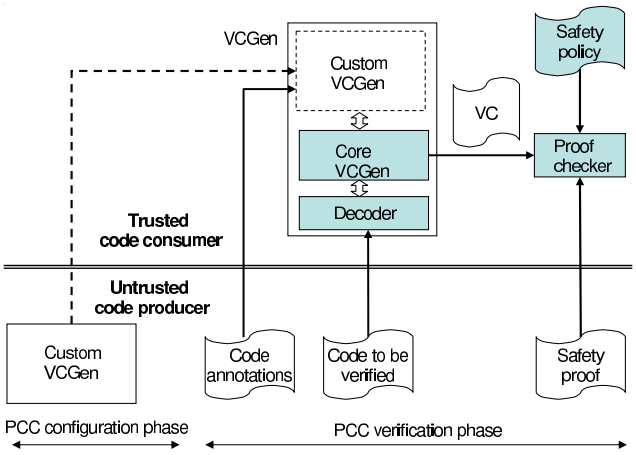

Figure 1. Configurable proof-carrying code.

as wavy boxes and PCC sub-systems as square boxes) provided by the code producer, and above it are the trusted subsystems of the code consumer (shown as shaded).

The operation of CPCC can be described as a progression from left to right in Figure 1. In the configuration phase the code producer supplies a code fragment (the *custom verification-condition generator*) that along with a trusted *core VCGen* and instruction *decoder* form the *verification-condition generator* (VCGen), the most complex part of a PCC verifier. The VCGen scans the untrusted code, decodes all instructions and follows both the data flow and the control flow of the code. In the process, it makes some quick syntactic checks (e.g. that only valid instructions are used and that direct jumps do not jump outside the code segment) and it produces a predicate (the *verification condition*) whose validity entails the safety of the code. We also refer to the custom VCGen as a *VCGen extension*.

In contrast, the Touchstone implementation of PCC has no configuration phase; the entire VCGen is trusted and hard-wired into the code consumer. Thus, the CPCC system is more trustworthy, since it has a smaller trusted infrastructure. CPCC is at the same time more flexible, because the code producer can create a VCGen that exploits the particular structure and conventions of the mobile code in order to generate fewer and simpler proof obligations. Furthermore, the code producer has complete freedom to establish and use a system of code annotations in order to provide hints to the custom VCGen on a per-program basis. In exchange, the custom verifier has to prove to the core VCGen that many proof obligations can be discharged locally.

It is worth pointing out that in a related paper [22] we show how the code producer can customize the code decoder as well, in order to give the code producer further flexibility in proof design. Essentially, by specializing the decoder, the code producer can specify customized *local* pre-

conditions and postconditions for individual instructions, which in turn can be used to decrease the size of the proofs considerably. A custom decoder must be proven correct with respect to a reference decoder before it can be used. In contrast, a custom VCGen, as described in this paper, can be used to handle special *global* aspects of the code, such as loops and calling conventions. Another difference is that the custom VCGen is provided in the form of executable code, sometimes using complex internal algorithms and data structures. This is why instead of providing a proof of its correctness we have chosen to design an architecture in which only the results of the untrusted VCGen need be verified.

2. The VCGen algorithm

As shown in Figure 1 the VCGen has three components. The decoder is responsible for determining the meaning of individual instructions. Given a program counter and a symbolic description of the execution state, it produces a set of possible new values of the program counter, each accompanied by a description of the modified state (we also refer to these next states as *continuations*). The decoder is also responsible for identifying the particular safety preconditions for each instruction. The decoder is trusted in our system.

The core VCGen is the other trusted component, responsible for orchestrating the execution of both the decoder and the untrusted extension. In the absence of an extension, it queries the decoder and records the safety preconditions as part of the verification condition. Then it simply iterates on all continuations returned by the decoder. This approach works on code without loops, function calls, or indirect jumps (such as those that arise from virtual method dispatch or exception handling). This is where the untrusted VCGen extension comes into the picture. In the presence of such an extension, the core VCGen queries both the extension and the trusted decoder. The extension may return fewer continuations (e.g., in the case of loops, when it detects that some continuations have been handled before), or simpler continuations (e.g., in the case of indirect jumps, when it knows the set of possible concrete jump targets). It is not important how the VCGen extension comes up with its answers. Most likely, it uses internal state and the code annotations. But since the extension is constructed by the code producer, it might also have special knowledge of special code patterns that are used in code generation. The core VCGen does use the continuations from the extension *instead* of the trusted ones from the decoder, but it also emits as part of the verification condition proof obligations which, when met, guarantee the soundness of the verification.

We begin with some needed preliminaries and then we continue with a formal description of all the components.

2.1. The logic

The verification condition and the continuations are expressed in a typed logic, which needs the following features in order to express the necessary reasoning about machine states. We will use the name `Prop` to refer to the type of propositions. We assume that the logic has a unary type constructor `list` for characterizing finite sequences of elements. There will be a type `val` ("values") used for both values and addresses. We assume that `val` is equipped with arithmetic and comparison operations. There is another type `mem` for memory states, which comes equipped with `upd` and `sel` functions for memory update and memory select.

We use a type `state` of concrete machine states, and use the metavariable ρ to range over concrete machine states. The program counter has special importance so we treat each state ρ as a pair, with first component $(\texttt{pc}\ \rho)$ of type `val` and second component $(\texttt{rest}\ \rho)$ of type `reststate`. The type `reststate` will be treated as abstract in the formal development, but in examples it will comprise a finite number of registers of type `val`, together with one memory of type `mem`. A *symbolic expression* is a function of type $\texttt{state} \rightarrow \texttt{val}$. Thus if e is a symbolic expression and ρ a state then $e\ \rho$ is the value of e in state ρ. For notational convenience we will also write symbolic expressions as expressions involving register names. For example, we write $\texttt{r}_1 + \texttt{r}_2$ for the function that given a state yields the sum of the values of registers $\texttt{r}_1$ and $\texttt{r}_2$ in that state. Consequently we write $\texttt{r}_1\ \rho$ for the value of register $\texttt{r}_1$ in state ρ.

We will also use *symbolic predicates* (or simply predicates), which are functions of type $\texttt{state} \rightarrow \texttt{Prop}$, and we write $A\ \rho$ to say that the predicate A holds in state ρ.

For convenience, we overload the arithmetic operators and the boolean connectives to work on symbolic expressions and predicates. For example, the notation $\exists x.\texttt{r}_1 + \texttt{r}_2 \geq x$ denotes a symbolic predicate whose de-sugared form is $\lambda\rho.\exists x.\texttt{r}_1\ \rho + \texttt{r}_2\ \rho \geq x$. We write $\rho[\texttt{r}_i \mapsto v]$ for the state ρ altered by setting the value of $\texttt{r}_i$ to v. We write $A[\texttt{r}_i \mapsto v]$ to denote the symbolic predicate obtained from A by replacing references to $\texttt{r}_i$ by v, namely $\lambda\rho.\big(A\ (\rho[\texttt{r}_i \mapsto v])\big)$.

2.2. Continuations

VCGen cannot work statically with concrete machine states. It works instead with symbolic predicates that are known to hold about concrete machine states. We call such predicates *continuations*.

Most of the time VCGen knows the precise value of the program counter in a state. A *direct continuation* is a pair whose first component (the program counter) is of type `val` and whose second component is of type $\texttt{reststate} \rightarrow$

Prop. We use the type dircont for direct continuations.

By *indirect continuations* we mean any other predicate of states, and we introduce the type indcont as an abbreviation for state → Prop. We use cont for the sum type of direct and indirect continuations; thus each continuation comes flagged either as direct or indirect. To say that ρ *satisfies* a continuation C, written $\rho \models C$, means that $(C\ \rho)$ if C is indirect, or that $(\texttt{pc}\ \rho) = n \wedge \big(A\ (\texttt{rest}\ \rho)\big)$ if $C = (n, A)$ is direct.

Let $\Rightarrow$ represent the state transition relation of the machine, restricted so that only *safe* transitions are allowed, according to the safety policy of the code consumer; thus for example memory operations will only be executed if they are memory-safe. We assume that the machine is deterministic; if there is any transition from a concrete state, then there is only one transition. As usual $^+$ indicates a sequence of one or more transitions. We define an indexed family of predicates safe, such that $(\texttt{safe}_n\ C)$ means that any state satisfying the continuation C can make at least n steps of safe progress:

Definition 1. *For any natural number n, let $\texttt{safe}_n$ be a predicate of continuations, defined as follows. Any continuation is said to be $\texttt{safe}_0$. For any natural number i, a continuation C is said to be $\texttt{safe}_{i+1}$ iff any state satisfying C can make safe progress to a new state satisfying some continuation which is $\texttt{safe}_i$, i.e.*

$$(\texttt{safe}_{i+1}\ C) \iff \forall \rho : \texttt{state}.(\rho \models C) \Rightarrow \exists \rho' : \texttt{state}.(\rho \Rightarrow^+ \rho') \wedge \exists D : \texttt{cont}.(\texttt{safe}_i\ D) \wedge (\rho' \models D).$$

It will become useful to allow continuations to refer to the safety of other continuations; for instance, at the beginning of a function we would like to assume that it is safe to jump to the return address. A technical device is required to have a continuation refer to $\texttt{safe}_i$ at various i; we introduce a notion of continuations indexed by natural numbers, *indexed continuations*. An indexed continuation is a family of ordinary continuations, with the intention that the ordinary continuation at index i will make use of the predicate $\texttt{safe}_i$. A *direct indexed continuation*, type xdircont, is a pair whose first component is of type val and whose second component is of type nat → reststate → Prop. Observe that the program counter is independent of the index. The type xindcont of *indirect indexed continuations* is an abbreviation for nat → state → Prop, and xcont is the union of these two types.

We will use natural-number subscripts to instantiate the index and produce an (ordinary) continuation from an indexed continuation. Thus for an indirect indexed continuation C, C_i is an ordinary indirect continuation for any i; for a direct indexed continuation $C = (n, A)$, $C_i = (n, A_i)$ is an ordinary direct continuation for any i. Observe also that any ordinary continuation can be treated as indexed by returning the same continuation at each index. In most examples we will not be interested in the ability to index, and will use ordinary continuations, via this correspondence, even in contexts where the typing calls for indexed continuations.

In order to maintain the soundness of the approach we must impose certain restrictions on indexed continuations, and more specifically on how they can refer to the safe predicates. This is unsurprising as what we have described so far allows completely unrelated ordinary continuations at each index. We will discuss these restrictions together with the soundness proof in Appendix A. We also describe there a sufficient and easily enforced syntactic restriction, which is satisfied by all of the extensions that we have considered.

We now turn our attention in turn to each of the components of a VCGen.

2.3. The decoder

Note that, unlike Foundational PCC, our framework does not include the program or the semantics of machine instructions in the logic; reasoning about such is handled by trusted code called the *decoder*. The decoder can be viewed abstractly as a function of the following type:

$$\texttt{decode} : \texttt{xdircont} \to (\texttt{state} \to \texttt{Prop}) \times \texttt{xcont list}$$

Given a direct indexed continuation, it uses the value of the program counter to determine the current instruction. The decoder returns a symbolic predicate that must be satisfied in the current state for the instruction to execute safely, along with a set of continuations describing all possible outcomes of the instruction. The ability to return more than one continuation is useful for branching instructions. The ability to return indirect continuations is useful for indirect jump instructions.

Knowledge of the program, the semantics of machine instructions, and the notion of safety of various individual instructions is entirely embedded in the behavior of the trusted decoder.

The implementation of the decode function is trusted to satisfy the following correctness property:

Property 2 (Decoder Correctness). *Let C be a* direct *indexed continuation. Then*

$$(\texttt{decode}\ C) = (P, \mathit{Conts})$$

where P : state → Prop and Conts is a list of indexed continuations, such that

$$\forall i. \forall \rho : \texttt{state}. (\rho \models C_i) \wedge (P\ \rho) \Rightarrow \bigvee_{D \in \mathit{Conts}} \exists \rho' : \texttt{state}.(\rho \Rightarrow^+ \rho') \wedge (\rho' \models D_i)$$

We show next the expected behavior of the decoder on a few example instructions. We use $p+1$ for the program counter of the instruction following the instruction at p. As mentioned above, these and other examples are written using unindexed continuations, which can be interpreted as indexed continuations in order to type correctly.

Arithmetic operations. If p is the program counter of an instruction "$\mathtt{r}_1 \leftarrow \mathtt{r}_2 + \mathtt{r}_3$" for distinct registers $\mathtt{r}_1$, $\mathtt{r}_2$, and $\mathtt{r}_3$, then

$$\begin{aligned}&\mathtt{decode}\ (p, A) = \big(\mathtt{True},\\ &\quad \{(p+1,\ \exists x_1 : \mathtt{val.r}_1 = \mathtt{r}_2 + \mathtt{r}_3 \wedge A[\mathtt{r}_1 \mapsto x_1])\}\big).\end{aligned}$$

There is no safety requirement for an assignment. The predicate in the new continuation is the usual strongest postcondition for an assignment.

Memory read. If p is the program counter of a memory read "$\mathtt{r}_1 \leftarrow (\mathtt{read}\ \mathtt{r}_2)$" for registers $\mathtt{r}_1$ and $\mathtt{r}_2$, then

$$\begin{aligned}&\mathtt{decode}\ (p, A) = \big(\mathtt{saferd}\ \mathtt{r}_2, \{(p+1,\\ &\qquad \exists x_1 : \mathtt{val.r}_1 = (\mathtt{sel}\ \mathtt{r}_M\ \mathtt{r}_2) \wedge A[\mathtt{r}_1 \mapsto x_1])\}\big).\end{aligned}$$

Here, `saferd` is the predicate that denotes that a memory address is readable. The meaning of this predicate can be customized for each safety policy by giving appropriate rules for proving it. The special register $\mathtt{r}_M$ models the memory state and the symbolic expression "$(\mathtt{sel}\ \mathtt{r}_M\ \mathtt{r}_2)$" denotes the current contents of the memory at the address stored in $\mathtt{r}_2$.

Conditional branch. The result of `decode` for a "branch if zero" instruction "$\mathtt{if}\ \mathtt{r}_1 = 0\ \mathtt{jump}\ L$" at program counter p is:

$$\begin{aligned}&\mathtt{decode}\ (p, A) = \big(\mathtt{True},\\ &\qquad \{(p+1,\ A \wedge \mathtt{r}_1 \neq 0),\ (L,\ A \wedge \mathtt{r}_1 = 0)\}\big).\end{aligned}$$

This is an example of a return value with more than one continuation.

Halt. We assume that the architecture has a special instruction that can be used to perform a safe termination of the execution. In order to use a simple notion of safety as the ability to make progress forever, we assume that the halt instruction is implemented as a self-loop. The decoding of such an instruction, with input continuation C, could be $(\mathtt{True}\ , \{C\})$.

Indirect jump. An indirect jump to the address stored in register $\mathtt{r}_1$ is decoded as

$$\mathtt{decode}\ (p, A) = (\mathtt{True}, \{\mathtt{pc} = \mathtt{r}_1 \wedge A\}).$$

This is one of the few instructions that makes use of the ability to return indirect continuations. Observe that the jump itself is always considered to be safe; further progress might not be possible after the jump, in fact it might not even be possible safely to examine the instruction at the new program counter, but it is not unsafe just to set the program counter to the new value. (In particular, the decoder can return $(\mathtt{False}, \{\})$—indicating there is no way to make safe progress—when it is given a program counter that is outside the code block.)

2.4. The custom VCGen

The core VCGen cannot simply use the decoder to determine where to proceed; some of the returned continuations might be indirect, or they might introduce an infinite loop if followed blindly. So for every continuation at which it queries the decoder, the core VCGen also queries the VCGen extension. The core trusts the decoder results but will actually use the extension results. To ensure soundness of the verification, the core emits as part of the verification condition proof obligations which, if met, guarantee that the extension results are sound with respect to the decoder ones. There are only three requirements for a custom VCGen: (1) its results should be of a certain syntactic form, (2) each query must terminate, and (3) it should be memory safe so that it does not have unexpected side-effects.

A custom VCGen takes as argument the same direct indexed continuation that is passed to the decoder, and yields a new set of *direct* indexed continuations:

$$\mathtt{customVCGen} : \mathtt{xdircont} \rightarrow \mathtt{xdircont\ list}$$

Intuitively, the extension proposes a set of possible continuations for the current state. The custom VCGen is untrusted and it can do whatever it pleases to produce this list, including calling `decode`, inspecting the program itself, maintaining some internal state, using annotations provided by the code producer, etc.

The simplest example of an extension simply returns the same (direct) continuations returned by `decode`. This suffices to handle straight-line code and forward direct jumps and branches (so that there is no non-termination issue). This and more interesting examples of extensions are discussed in Section 3.

We have described the form which the results of the custom VCGen must have in order to be used by the core VCGen. But we also need to ensure that the custom VCGen does yield some results (i.e. terminates) and does not have unwanted side-effects (i.e. the returned value is the only result). To ensure termination we propose to use a timeout mechanism. The alternative of proving the termination of the extension would be too expensive.

There are two possible ways to prevent side-effects. First, the extension could be run in a separate process and

address space and could communicate with the rest of the VCGen using some form of interprocess communication.

The other possibility is to use PCC itself to verify the memory safety of the extension during the configuration phase. Current technology makes this possible and quite painless, as long as the extension is written in a type-safe language (e.g. the Touchstone system for Java programs). Alternatively, for extensions written in C one can use CCured [20] to ensure that the code is memory safe and to produce a proof to that extent.

Interestingly, if we have a proof of the extension's memory safety, we can verify the proof and that it corresponds to the extension by using the extension itself as part of the VCGen. Only while this is done we must run the extension in a different process. This latter condition ensures that the extension is actually memory safe in the run in which it is used to "prove" its own memory safety for *all* runs. After this step we can safely run the extension in the same address space as the rest of the VCGen. This provides a way to bootstrap the process so that we incur the cost of using different address spaces only once during the configuration phase.

2.5. The core VCGen

The other piece of trusted code besides the decoder is the core VCGen, which implements an algorithm to tie together the results of the untrusted custom VCGen and the decoder to create a verification condition. If the verification condition holds, the program is safe, in the sense that any machine state reached during the execution of the program can make *safe* progress to some new state. We now describe this algorithm and in Appendix A we prove it sound, in the sense that proving the verification condition guarantees the safety of the program.

The strategy of VCGen is to build a set $\mathcal{S}$ of indexed direct continuations. The set $\mathcal{S}$ will be built so that if the VC holds, then for all n, each continuation $C \in \mathcal{S}$ satisfies $\texttt{safe}_n\ C_n$. Furthermore, $\mathcal{S}$ will include an initial continuation which holds (at index 0) of any machine state at which the program can begin execution—here, we take the most general continuation $(0, \texttt{True})$, though particular safety policies may allow more initial information. It will follow from these facts that the program can make indefinite progress from the initial state.

The verification condition amounts to the statement of the induction step of an induction over n. Each continuation in $\mathcal{S}$ is assumed $\texttt{safe}_i$ (corresponding to the induction hypothesis) and the necessary facts are proven to ensure that each is in fact $\texttt{safe}_{i+1}$. The base case is trivial given the definition of `safe`.

$$
\begin{array}{l}
Scan(\{\}, \_) = \{\} \\
Scan(\{C\} \cup ToScan, seen) = \\
\quad \texttt{let}\ (p, \_) = C \\
\quad \texttt{in if}\ p \in seen\ \texttt{then error else} \\
\quad \{C\} \cup Scan\big((\texttt{customVCGen}\ C) \cup ToScan, \\
\qquad\qquad \{p\} \cup seen\big)
\end{array}
$$

$$
\begin{array}{l}
VC = \forall i. \Big(\bigwedge_{C \in \mathcal{S}} \texttt{safe}_i\ C_i \Big) \Rightarrow \\
\qquad \bigwedge_{C \in \mathcal{S}} \Big(\forall \rho. (\rho \models C_i) \Rightarrow \\
\qquad\qquad (P_C\ \rho) \wedge \bigwedge_{D \in Conts_C} \texttt{safe}_i\ D_i \Big) \\
\quad \texttt{where}\ (P_C, Conts_C) = \texttt{decode}\ C \\
\qquad\quad \mathcal{S} = Scan(\{(0, \texttt{True})\}, \{\})
\end{array}
$$

Figure 2. The algorithm for core VCGen

The core VCGen constructs the set $\mathcal{S}$ by querying the custom VCGen. It accumulates all the continuations ever returned by the custom VCGen and recursively scans them to ensure that progress can be made. The algorithm implemented by core VCGen is shown in Figure 2. At all times the first argument of $Scan$ contains those direct continuations that have been discovered but not yet scanned, and the second argument contains the set of program counters that have been scanned already. Core VCGen ensures termination by failing if a program counter is seen more than once.

Once $Scan$ constructs $\mathcal{S}$, then each direct continuation in it is passed to the decoder. The core VCGen then adds two assertions to the verification condition: first, that the precondition for the current instruction (as identified by the decoder) is met in the current state, and thus at least one step of safe progress can be made; second, that *each* continuation identified by the decoder is itself safe, and thus enough safe progress can be made to complete the current step of the induction.

Note that as we present it in this paper, proving the VC will require proving facts about our indexed continuations at all indices i. This is likely to be feasible only when the indexed continuations are parametric in i in a relatively simple way, such as in the syntactic restriction presented in Appendix A. Moreover, although the definition of $\texttt{safe}_i$ makes reference to the machine transition relation , we do not expect explicit reasoning about to occur in proofs of verification conditions. Instead, to meet proof obligations that certain continuations are safe, we use the fact that other continuations have been *assumed* safe. In particular, if C and D are continuations such that $\forall \rho : \texttt{state}.\rho \models C \Rightarrow \rho \models D$, then $(\texttt{safe}_n\ D) \Rightarrow (\texttt{safe}_n\ C)$. For a more general statement and the proof, refer to Lemma 3 in Appendix A.

$\mathbb{S}$	$Scan'$	$Conts$	VC'
$A : (0, \texttt{True})$	$B : (1, \texttt{r}_2 = 5)$	same	
$B : (1, \texttt{r}_2 = 5)$	$C : (2, \texttt{r}_1 = (\texttt{sel r}_M\ \texttt{r}_2) \wedge \texttt{r}_2 = 5)$	same	$(\texttt{saferd}\ 5)$
$C : (2, A_1)$	$D : (3, \texttt{r}_1 \neq 0 \wedge A_1)$ $E : (5, \texttt{r}_1 = 0 \wedge A_1)$	same	
$D : (3, \texttt{r}_1 \neq 0 \wedge A_1)$	$F : (4, \exists x.\texttt{r}_2 = x + \texttt{r}_1 \wedge \texttt{r}_1 \neq 0 \wedge \texttt{r}_1 = (\texttt{sel r}_M\ x) \wedge x = 5)$	same	
$F : (4, A_2)$	$F : (4, A_2)$		
$E : (5, \texttt{r}_1 = 0 \wedge A_1)$	$G : (6, \exists x.\texttt{r}_2 = 0 \wedge \texttt{r}_1 = 0 \wedge \texttt{r}_1 = (\texttt{sel r}_M\ x) \wedge x = 5)$	same	
$G : (6, A_3)$	$G : (6, A_3)$	same	

Figure 3. A sample run of the null extension. We use A_1 as an abbreviation for $\texttt{r}_1 = (\texttt{sel r}_M\ \texttt{r}_2) \wedge \texttt{r}_2 = 5$, A_2 for $\exists x.\texttt{r}_2 = x + \texttt{r}_1 \wedge \texttt{r}_1 \neq 0 \wedge \texttt{r}_1 = (\texttt{sel r}_M\ x) \wedge x = 5$ and A_3 for $\exists x.\texttt{r}_2 = 0 \wedge \texttt{r}_1 = 0 \wedge \texttt{r}_1 = (\texttt{sel r}_M\ x) \wedge x = 5$.

3. Writing a custom VCGen

The custom VCGen is uploaded as executable code. A custom VCGen can have (read-only) access to the code, and can either interpret it itself, or call `decode`. Importantly, a given custom VCGen may expect the code producer to provide certain annotations to the code; important examples are loop invariants, and preconditions and postconditions of functions. These annotations are sent along with the program to the custom VCGen. Annotations are produced by the code producer. We do not specify (and the trusted core VCGen does not care) how these annotations are encoded. Additionally the custom VCGen has a private internal state and can remember for example what answers it has previously produced.

In the rest of this section we give some examples of VCGen extensions inspired by the features present in the trusted Touchstone VCGen. The difference is that these extensions are not trusted anymore. This is especially important when the VCGen must incorporate optimizations for scalability. One such optimization is discussed in Section 3.2.1.

3.1. The null extension

The simplest useful extension simply re-uses the continuations returned by the decoder on the given continuation.

$$\texttt{nullExt}(p, A) = \{C \in Conts \mid C \text{ is direct}\},$$
$$\text{where } (P, Conts) = \texttt{decode}(p, A)$$

This extension simply tells the core VCGen to use the decoder continuations. This suffices to handle straight-line code, plus direct jumps or branches in code without loops and function calls.

Consider the execution of the VCGen on following example program. The command `halt` is considered to be safe; we treat it as a self-loop.

```
0 r2 ← 5
1 r1 ← (read r2)
2 if r1 = 0 jump 5
3 r2 ← r2 + r1
4 halt
5 r2 ← 0
6 halt
```

See Figure 3 for a sample run of VCGen through this program. The column labeled $\mathbb{S}$ shows all direct continuations that are collected by $Scan$. In the row corresponding to a continuation we show the continuations returned by the VCGen extension (in column $Scan'$, labeled with letters), by `decode` (in column $Conts$, which says `same` if the results are identical to those of the extension), and the safety preconditions that are added to the VC (in the column VC', possibly pre-processed into an equivalent form for reasons of presentation). The verification condition is simply the conjunction of the entries in the column VC', or $\texttt{saferd}\ 5$ in this case.

3.2. The loop invariant extension

For a program that loops the core VCGen will fail upon seeing the same value of the program counter twice. The standard technique for dealing with this problem is to provide loop invariants and to set up an inductive argument that requires seeing the body of the loop only once. This can be done entirely within an untrusted extension because the "essence" of induction is implemented by the core VCGen.

We assume that the VCGen extension can come up with loop invariant predicates, possibly based on annotations provided by the producer or maybe based on some special

$\mathcal{S}$	$Scan'$	$Conts$	VC'
$A : (0, \texttt{True})$	$B : (3, \texttt{r}_x = 0 \wedge \texttt{r}_y = 2 \wedge \texttt{r}_z = 0)$	same	
$B : (3, A_1)$	$C : (4, A_1 \wedge \texttt{r}_i = 2)$ $D : (6, A_1 \wedge \texttt{r}_i \neq 2)$	same	
$C : (4, A_1 \wedge \texttt{r}_i = 2)$	$E : (8, I)$	$(8, \texttt{r}_x = 0 \wedge \texttt{r}_y = 1 \wedge \texttt{r}_z = 0 \wedge \texttt{r}_i = 2)$	$A_2 \Rightarrow I$
$D : (6, A_1 \wedge \texttt{r}_i \neq 2)$	$E : (8, I)$	$(8, \texttt{r}_x = 0 \wedge \texttt{r}_y = 2 \wedge \texttt{r}_z = 0 \wedge \texttt{r}_i = 2)$	$A_3 \Rightarrow I$
$E : (8 : I)$	$F : (9, I \wedge \texttt{r}_x < \texttt{r}_i)$ $G : (11 : I \wedge \texttt{r}_x \geq \texttt{r}_i)$	same	
$F : (9, I \wedge \texttt{r}_x < \texttt{r}_i)$	$E : (8, I)$	$(8, \exists x'.I[\texttt{r}_x \mapsto x'] \wedge x' < \texttt{r}_i \wedge \texttt{r}_x = x' + 1)$	$A_4 \Rightarrow I$
$G : (11, I \wedge \texttt{r}_x \geq \texttt{r}_i)$	$H : (12, A_5)$	same	$I \wedge \texttt{r}_x \geq \texttt{r}_i \Rightarrow \texttt{r}_z = 0 \wedge \texttt{r}_x = 2$
$H : (12, A_5)$	$H : (12, A_5)$	same	

Figure 4. A sample execution for the invariant extension. Here I is the invariant predicate, A_1 is an abbreviation for $\texttt{r}_x = 0 \wedge \texttt{r}_y = 2 \wedge \texttt{r}_z = 0$, A_2 for $\texttt{r}_x = 0 \wedge \texttt{r}_y = 1 \wedge \texttt{r}_z = 0 \wedge \texttt{r}_i = 2$, A_3 for $\texttt{r}_x = 0 \wedge \texttt{r}_y = 2 \wedge \texttt{r}_z = 0 \wedge \texttt{r}_i = 2$, A_4 for $\exists x'.I[\texttt{r}_x \mapsto x'] \wedge x' < \texttt{r}_i \wedge \texttt{r}_x = x' + 1$, and A_5 for $I \wedge \texttt{r}_x \geq \texttt{r}_i \wedge \texttt{r}_z = 0 \wedge \texttt{r}_x = 2$.

knowledge that is has about the structure of the code. The invariant extension could be implemented as a filter processing the output of some other extension (e.g., the null extension), replacing the other extension's predicate with an invariant wherever there is an annotation:

$$\begin{aligned}&\texttt{invExt}(\mathcal{E}) = \\ &\quad \{(p, A) \in \mathcal{E} \mid \text{no invariant annotation at } p\} \cup \\ &\quad \{(q, I) \mid (q, B) \in \mathcal{E} \wedge \\ &\qquad\qquad \text{there is an ``invariant } I\text{'' annotation at } p\}.\end{aligned}$$

The execution of the invariant extension on the following program is shown in Figure 4 using the same format as for the null extension. We use a hypothetical `assert` instruction to yield the effect of code which can only be executed safely under certain conditions.

```
0  r_x ← 0
1  r_y ← 2
2  r_z ← 0
3  if r_i = 2 then {
4     r_y ← 1
5  } else {
6     r_i ← r_y
7  }
8  while (r_x < r_i) do {            # invariant I
9     r_x ← r_x + 1
10 }
11 assert r_z = 0 ∧ r_x = 2
12 halt
```

Notice that the invariant extension provides the continuation labeled E whenever the execution reaches line 8. This gives rise to three conjuncts in the VC related to proving the loop invariant (the first two prove that the invariant holds on each of the two ways of entering the loop and the third proves that the invariant is preserved in the loop). The fourth conjunct is introduced by the result of `decode` for the assert instruction; it essentially requires that the invariant is strong enough to prove the assertion.

A good choice for I is $\texttt{r}_z = 0 \wedge \texttt{r}_i = 2 \wedge \texttt{r}_x \leq \texttt{r}_i$. The reader can verify that all four conjuncts hold.

3.2.1 An invariant extension with improved scalability

The previous invariant extension works well but it has the disadvantage that it requires very large invariant annotations, which in turn increase the amount of data that has to be communicated between the code producer and the code consumer. In some of the largest experiments with Touchstone we observed invariants that contained several hundred simple conjuncts [6]. It turns out that many of these conjunct can be inferred from the context. For example, we notice in the example program from the previous section that the values of $\texttt{r}_i$ and $\texttt{r}_z$ are the same every time the invariant point is hit. We should not have to specify in invariant annotations what their values are.

Touchstone uses an invariant extension whose annotations specify both an invariant predicate *and* a list of registers whose values are invariant through the loop (without specifying what those values are). This list of registers is represented compactly as a bitmap. For example, in the previous example the Touchstone annotation would

consist of the invariant $I' = \mathtt{r}_x \leq \mathtt{r}_i$ (the interesting part of I) along with the list $\{\mathtt{r}_z, \mathtt{r}_i\}$ of invariant registers. The modified invariant extension would then replace the continuation (q, B) not with (q, I) as before but with $(q, \exists x, y.B[\mathtt{r}_x \mapsto x, \mathtt{r}_y \mapsto y] \wedge I')$, which is logically equivalent to (q, I) but obtained from a smaller annotation. Notice how we inherit the state of the first path that reaches the invariant, except for everything involving the non-invariant registers.

This is an example of the "tricks" that we have found so useful for scalability in Touchstone. We feel that taking such experimental features out of the trusted computing base would go a long way towards making the implementation of PCC trustworthy.

3.3. The function call/return extension

Function calls are handled by associating to each function F (via the annotation data sent with the program) a precondition Pre_F, a list of callee-save registers CS_F, and a postcondition $Post_F$. These are used in a conventional PCC system (as described in [17]) roughly as follows. If a function F is called in a state satisfying A, we require a proof (in the VC) that $A \Rightarrow Pre_F$; then we proceed from the instruction following the call with the assumption $Post_F \wedge \exists\{x_r\}_{r\in\mathcal{R}}.A[r \mapsto x_r]_{r\in\mathcal{R}}$, where $\mathcal{R} = Regs \setminus CS_F$. That is, we assume the postcondition, and that A holds when you replace those registers possibly changed by F with some values (namely, those they had before F). We separately must handle the code within F, assuming only Pre_F at the beginning, and at the return requiring proofs that $Post_F$, and that the return address and the registers in CS_F in fact match the values they had at the beginning of the function.

We now describe an approach to implement this in our framework. We exhibit here the behavior of the custom VCGen at a function call and at a function return; in Appendix B we show that the conjuncts added to the VC are equivalent to the proof obligations in a conventional PCC system, as described above.

For this example we will use indexed continuations. These will be written as continuations referring to `safe`, without a subscript; the subscript is given by the index with which the continuation is instantiated.

For an instruction `jump` F at program counter n, the decoder would return (F, A). Instead, the extension returns two continuations:

$$\Big(F,\ \lambda\rho.(Pre_F\ \rho) \wedge \mathtt{safe}\ \lambda\rho'.\big((\mathtt{pc}\ \rho') = (\mathtt{ra}\ \rho) \wedge (Post_F\ \rho') \wedge \textstyle\bigwedge_{r\in CS_F}(r\ \rho') = (r\ \rho)\big)\Big)$$

$$\big(n+1,\ \lambda\rho'.(Post_F\ \rho') \wedge \exists\{x_r\}_{r\in Regs\setminus CS_F}.(A[r \mapsto x_r]_{r\in Regs\setminus CS_F}\ \rho')\big)$$

The first continuation is shared by all calls to F and states the precondition to F (including the fact that there is a safe continuation for the value of the return address register). The second continuation instructs the core to continue after the call.

The return instruction is implemented as an indirect jump to the return address register. Custom VCGen will emit no continuations for the return instruction; the safety of the continuation returned by the decoder will have to be proven independently. In fact, we will be able to use the assumption, made when we entered the function, that jumping to the return address is safe (under the appropriate conditions). See Appendix B for details.

4. Related work

This work is closely related to work in Foundational Proof-Carrying Code (FPCC) [2, 1, 15, 11]. In FPCC one works with a logic which can express the states and transitions of a given machine architecture, and includes in the safety proof everything needed to formally prove that indefinite progress can be made from the initial state. The analogy to VCGen need do little more than embed the integers which constitute the untrusted code into a logical expression.

On the one hand, our work is intended to bring VCGen-based PCC closer to the ideal of FPCC. We will have a smaller trusted base and a more flexible framework which allows the code producer more leeway in how safety is to be proved and in the actual implementation of the internals of the verifier. In fact, starting with a formalization of the soundness proof of Appendix A, and an implementation of the `decode` function for a real architecture, one might be able to transform this framework into a pure FPCC implementation. (Such an effort would also require the incorporation of the work of [21] and [22] to formally derive the type system of the safety policy.)

On the other hand, we maintain the goal of retaining the ability of VCGen-based PCC systems such as Touchstone to efficiently handle large programs. It seems possible that retaining this scalability may require accepting a larger trusted base.

In order to soundly refer to the safety of a predicate, which itself claims the safety of other predicates, we introduced the technical device of indexed continuations. A similar notion had been developed previously by Appel and McAllester in [3], in order to create a semantic model for recursive types to be used in FPCC proofs.

Recently Bernard and Lee [5] have developed an alternative approach to proof-carrying code in which there is no trusted VCGen. Their proposal is to use temporal logic to encode the control-flow related part of the safety policy in a declarative way as an alternative to using a trusted VCGen

implemented in C. We show here that in fact we do not have to trust most of the VCGen. However, the use of temporal logic has benefits beyond just eliminating VCGen and we expect that a combination of the two approaches might be beneficial.

Our work is also related to the large body of work on program verification using verification-condition generators [4, 8, 12] or equivalently weakest precondition generators. A good reference is Dijkstra's classic "A Discipline of Programming" [9]. One can view our work as an effort to take the formal proofs of correctness (e.g. those from [7, 9]) and to factor out those parts that require global reasoning from those that talk about localized aspects. All such correctness proofs have in common the use of an assume-guarantee or co-inductive reasoning. The core VCGen is responsible for ensuring that all of the components, for which assumptions are being made, meet their guarantees. The custom extension then can be seen as directing the local aspects of the proofs, with reference to a trusted language semantics as given by the trusted decoder.

Although the original motivation for this work was to increase the level of trustworthiness and flexibility of a PCC system, the results that we describe here are applicable to all instances in which verification conditions or weakest preconditions are used to analyze programs. For example, [10] discusses a variation of the standard verification condition generation that produces significantly smaller predicates. This kind of "trick" is similar to those that Touchstone employs and is suited for an implementation as an untrusted extension to the verification condition generator, using the architecture described here. And even if there is trust between the provider of the verification-condition generator and its user, we have found that the ability to perform independent checks on the operation of the infrastructure to be of great software engineering advantage.

5. Conclusions and future work

We have created a logically sound method for implementing a PCC system based on a VCGen that is mostly untrusted, using methods to verify each execution of the untrusted VCGen rather than the code itself. This leads to a system which is significantly more secure and flexible than other VCGen-based PCC systems such as Touchstone; however, it also should allow us to re-use many of the engineering refinements and even actual modules which allow Touchstone to efficiently handle large programs.

There are a number of refinements to the verification process which are useful for engineering reasons; these include modifications of the notion of continuation, and the exact form of the VC, in order to increase the efficiency of the process. For reasons of space and presentation we have chosen to omit these considerations in this paper, but intend to present them in an expanded version.

This paper explores the soundness of VCGen extensions; in [21] and [22] we discuss the soundness of custom safety policies for the system. We can indeed combine these two approaches. However, further research is needed on incorporating VCGen extensions that rely heavily on the proof rules of the safety policy. For instance, Touchstone's handling of dynamic dispatch for Java is by using indirect jumps to addresses which, via the safety policy, are known to have a given function type. The core VCGen introduced in this paper will happily emit the proof obligations needed for that handling to guarantee safety; proving those obligations will amount to proving certain facts about the safety policy, e.g. that following the safety policy has enforced a certain global invariant.

The framework presented in this paper was designed to fit the needs of the Touchstone system. Thus we are hopeful that we will be able to eventually produce an actual implementation of the system, at which point it will be possible to evaluate how much of the scalability of Touchstone can be retained in practice. It is certain though that there would be much less trusted code in such a system.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, Jan. 2000.

[3] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *ACM SIGPLAN Notices*, 31(1):1–3, Jan. 2002.

[5] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, July 27-30 2002.

[6] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.

[7] J. W. deBakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.

[8] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec. 1998.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *ACM SIGPLAN Notices*, 36(3):193–205, Mar. 2001.

[11] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 31(1):58–70, Jan. 2002.

[13] C. League. Touchstone soundness bug report. Personal communication, Oct. 2001.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.

[15] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher-order logic. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000.

[16] G. C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, Jan. 1997.

[17] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Sept. 1998. Also available as CMU-CS-98-154.

[18] G. C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PDLI)*, pages 83–94, Vancouver, BC, Canada, 18–21 June 2000. ACM SIGPLAN.

[19] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Second Symposium on Operating Systems Design and Implementations*, pages 229–243. Usenix, Oct. 1996.

[20] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *The 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, Jan. 2002.

[21] G. C. Necula and R. R. Schneck. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, July 27-30 2002.

[22] G. C. Necula and R. R. Schneck. Proof-carrying code with untrusted proof rules. In M. Okada, editor, *Software Security – Theories and Systems. Mext-NSF-JSPS International Symposium, ISSS 2002*, Lecture Notes in Computer Science, 2003. Available from http://raw.cs.berkeley.edu/Papers/sfpol_isss02.pdf.

[23] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 151–166. Springer, 1998.

[24] H. Wasserman and M. Blum. Software reliability via runtime result-checking. *Journal of the ACM*, 44(6):826–849, Nov. 1997.

A. Soundness of VCGen

We begin with a lemma about continuations which, though not directly needed for the soundness proof, is the main tool for proving assertions about safety in verification conditions.

Lemma 3. *Let n be a natural number, and let C be a continuation. If*

$$\forall \rho : \mathtt{state}.(\rho \models C) \Rightarrow \big(\exists D : \mathtt{cont}.(\rho \models D) \wedge (\mathtt{safe}_n\ D)\big),$$

then $(\mathtt{safe}_n\ C)$.

Proof. If $n = 0$ then this is immediate. Suppose $n = i + 1$. Take any ρ such that $\rho \models C$; then there is D such that $\rho \models D$ and $(\mathtt{safe}_{i+1}\ D)$. By the definition of $\mathtt{safe}_{i+1}$, $\exists \rho' : \mathtt{state}.\ (\rho^+\rho') \wedge \exists E : \mathtt{cont}.\ (\mathtt{safe}_i\ E) \wedge (\rho' \models E)$; and this is exactly what is required to conclude $(\mathtt{safe}_{i+1}\ C)$. □

We say that a continuation C is *stronger* than a continuation C' (written $C \Rightarrow C'$) when $\forall \rho : \mathtt{state}.\rho \models C \Rightarrow \rho \models C'$. For direct continuations (p, A) and (p', A') this is equivalent to $p = p' \wedge \forall \rho : \mathtt{reststate}.(A\ \rho) \Rightarrow (A'\ \rho)$.

Corollary 4. *Let n be a natural number, and let C and D be continuations. If $C \Rightarrow D$ and $(\mathtt{safe}_n\ D)$, then $(\mathtt{safe}_n\ C)$.*

We now consider the restriction on indexed continuations which is required for soundness.

Definition 5. *Let C be an indexed continuation. Say that C is* well-behaved *if for all i,*

$$\mathtt{safe}_{i+1}\ C_i \Rightarrow \mathtt{safe}_{i+1}\ C_{i+1}.$$

Though this is the technical condition required by the soundness proof, the following stronger property may provide more intuition.

Definition 6. *Let C be an indexed continuation. Say that C is* monotone *if for all i,*

$$C_{i+1} \Rightarrow C_i.$$

Lemma 7. *If an indexed continuation C is monotone, then it is well-behaved.*

Proof. This follows by Corollary 4. □

In fact, the restriction is only required on the output of the custom VCGen. Formally we require that the extension satisfy the

Property 8 (Custom VCGen Condition). *Every indexed continuation output by custom VCGen is well-behaved.*

In practical use, it is almost certainly better not to have to prove that our indexed continuations are well-behaved. Instead we can use a syntactic restriction, namely, that all direct indexed continuations have the form

$$\big(n, \lambda i.\lambda\rho.\exists\tau : V.A(\tau,\rho) \wedge \bigwedge_j \texttt{safe}_i\ E_j(\tau,\rho)\big),$$

where V is any type, and where A and the various E_j do not depend on the index i. A similar restriction is placed on indirect indexed continuations. Any such indexed continuation is clearly monotone, by the definition of `safe`, and thus well-behaved.

We can now state and prove the soundness of the VCGen algorithm from Figure 2.

Theorem 9 (Soundness of Verification). *Assume that the decoder correctness property (Property 2) holds, and assume that the custom VCGen satisfies the custom VCGen condition (Property 8). If the VCGen algorithm terminates successfully having built the VC, and the VC holds, then the program is safe, in the sense that an arbitrary number of safe execution steps can be made from the initial state.*

Proof. Suppose that the algorithm terminates successfully, having built the VC and the set $\mathcal{S}$.

We prove by induction on n that $\texttt{safe}_n\ C_n$ holds for each $C \in \mathcal{S}$. Then, since the initial state satisfies the initial continuation $(0, \texttt{True})$—with any subscript n, since this continuation is unindexed—and the initial continuation is in $\mathcal{S}$, we have that n steps of safe progress can be made from the initial state for any n, and the theorem follows.

The case $n = 0$ is immediate, so let $n = i + 1$. Then the VC incorporates the induction hypotheses that $\texttt{safe}_i\ C_i$ for each $C \in \mathcal{S}$. So assume that the VC holds. Fix $C \in \mathcal{S}$; let $(P, Conts) = \texttt{decode}\ C$. Take any state ρ such that $\rho \models C_i$. By the VC, we have that $(P\ \rho)$. But by Property 2 this ensures that ρ can make safe progress; in fact we have that for some $D \in Conts$, $\exists\rho' : \texttt{state}.(\rho^+\rho') \wedge \rho \models D_i$. But the VC also ensures that $\texttt{safe}_i\ D_i$. It follows, by the definition of $\texttt{safe}_n$, that $\texttt{safe}_{i+1}\ C_i$.

To complete the induction step, we use the custom VCGen condition which ensures that C is well-behaved; thus we have that $\texttt{safe}_{i+1}\ C_{i+1}$. This completes the proof. □

B. The function call/return extension

In this section we aim to show that the implementation of function calls and returns described in Section 3.3 is sensible in that the conjuncts added to the VC are in fact equivalent to what we expect to be required to prove. Recall that we are using continuations referring to `safe`, without a subscript, to represent indexed continuations; the subscript is given by the index with which the continuation is instantiated.

We first consider the case of a function call. Given a continuation (n, A) where instruction n is `jump` F, the decoder returns one continuation, (F, A) (a call is just like a jump). So core VCGen will add to the VC a proof obligation that this continuation implies one of the two returned by custom VCGen (for which see Section 3.3). It will be the first one, so the VC will contain

$$\begin{aligned}\forall\rho.(A\ \rho) \Rightarrow (Pre_F\ \rho) \wedge \\ \texttt{safe}\ \lambda\rho'.\big((\texttt{pc}\ \rho') = (\texttt{ra}\ \rho) \wedge \\ (Post_F\ \rho') \wedge \\ \textstyle\bigwedge_{r\in CS_F}(r\ \rho') = (r\ \rho)\big)\end{aligned}$$

That $A \Rightarrow Pre_F$ we expected to have to prove in order to call the function. In order to prove the instance of `safe`, we will use that assumption that everything that ends up in $\mathcal{S}$ is `safe`; so in particular, we have the assumption

$$\begin{aligned}&\texttt{safe}\ \big(n+1,\ \lambda\rho'.(Post_F\ \rho') \wedge \\ &\quad \exists\{x_r\}_{r\in Regs\setminus CS_F}.(A[r \mapsto x_r]_{r\in Regs\setminus CS_F}\ \rho')\big)\end{aligned}$$

It suffices (by Corollary 4) to show that this continuation is weaker than the one we wish to show safe; so it suffices to show

$$\begin{aligned}&\forall\rho.(A\ \rho) \Rightarrow (\texttt{ra}\ \rho) = n+1 \wedge \\ &\quad \forall\rho'.(Post_F\ \rho') \wedge \bigwedge_{r\in CS_F}(r\ \rho') = (r\ \rho) \Rightarrow \\ &\qquad (Post_F\ \rho') \wedge \exists\{x_r\}_{r\in Regs\setminus CS_F}. \\ &\qquad\qquad (A[r \mapsto x_r]_{r\in Regs\setminus CS_F}\ \rho')\end{aligned}$$

but as long as $\forall\rho.(A\ \rho) \Rightarrow (\texttt{ra}\ \rho) = n+1$ the rest follows easily.

Observe that the continuation given to the beginning of the function does not depend on the continuation at the call site. Thus the function is only traversed once. However, each call site will have separately to prove that the continuation actually follows from the one given by `decode`, the hard part of which is proving, at each call site n, that

$$\forall\rho.(A\ \rho) \Rightarrow (Pre_F\ \rho) \wedge (\texttt{ra}\ \rho) = n+1,$$

as expected.

It remains to deal with the return instruction. The custom VCGen returns no continuations, so we will have to prove the safety of the decoder's output directly. If the return instruction is at program counter n and we are scanning the continuation (n, B), we will be required to prove

$$\texttt{safe}\ \lambda\rho.\big((\texttt{pc}\ \rho) = (\texttt{ra}\ \rho) \wedge (B\ \rho)\big). \tag{10}$$

In fact, since $(B\ \rho)$ is unlikely to guarantee that $(\texttt{ra}\ \rho)$ has a specific literal value, we can't get this case of `safe` from the assumption that all continuations in $\mathcal{S}$ are `safe`.

Instead it must come from B. Luckily we assumed a continuation was `safe` at the start of the function and that assumption should still be available, in a modified form. Recall that the continuation at the start of the function is

$$\Big(F,\ \lambda\rho.(Pre_F\ \rho) \wedge \\ \texttt{safe}\ \lambda\rho'.\big((\texttt{pc}\ \rho') = (\texttt{ra}\ \rho) \wedge \\ (Post_F\ \rho') \wedge \\ \textstyle\bigwedge_{r\in CS_F}(r\ \rho') = (r\ \rho)\big)\Big)$$

Now, the continuation here assumed safe includes references to the return address, and all the callee-save registers, at the state ρ, which corresponds to the state at the beginning of the function. If any of these registers are changed during the course of the function, an existential quantifier will be introduced to indicate that there was *some* value (namely the value before the change) which yielded a safe continuation. So by the return instruction, we should have

$$\forall\rho.(B\ \rho) \Rightarrow \exists x_{ra}.\exists\{x_r\}_{r\in CS_F}. \\ \texttt{safe}\ \lambda\rho'.\big((\texttt{pc}\ \rho') = x_{ra} \wedge (Post_F\ \rho') \wedge \\ \bigwedge_{r\in CS_F}(r\ \rho') = x_r\big).$$

Now, we expect to have to prove that the return address and the callee-save registers *match* their values at the beginning of the function. This corresponds to

$$\forall\rho.(\texttt{pc}\ \rho) = (\texttt{ra}\ \rho) \wedge (B\ \rho) \Rightarrow \\ \exists x_{ra}.\exists\{x_r\}_{r\in CS_F}.\texttt{safe}\ \lambda\rho'.\big((\texttt{pc}\ \rho') = x_{ra} \wedge \\ (Post_F\ \rho') \wedge \bigwedge_{r\in CS_F}(r\ \rho') = x_r\big) \wedge \\ (\texttt{ra}\ \rho) = x_{ra} \wedge \bigwedge_{r\in CS_F}(r\ \rho) = x_r.$$

If we can in fact prove that, we will have, by rewriting the various equalities, that

$$\forall\rho.(\texttt{pc}\ \rho) = (\texttt{ra}\ \rho) \wedge (B\ \rho) \Rightarrow \\ \texttt{safe}\ \lambda\rho'.\big((\texttt{pc}\ \rho') = (\texttt{ra}\ \rho) \wedge \\ (Post_F\ \rho') \wedge \bigwedge_{r\in CS_F}(r\ \rho') = (r\ \rho)\big).$$

But clearly we have that

$$\forall\rho.(\texttt{pc}\ \rho) = (\texttt{ra}\ \rho) \wedge (B\ \rho) \Rightarrow \\ (\texttt{pc}\ \rho) = (\texttt{ra}\ \rho) \wedge \bigwedge_{r\in CS_F}(r\ \rho) = (r\ \rho);$$

if we can also prove

$$\forall\rho.(\texttt{pc}\ \rho) = (\texttt{ra}\ \rho) \wedge (B\ \rho) \Rightarrow (Post_F\ \rho),$$

which we expect to have to prove, then we will be able to use Lemma 3 to establish (10).

Thus we have shown that the various proof obligations arising from the use of the extension described in Section 3.3, do in fact reduce to facts which we expect to have to prove following our intuitive conception.