From Typed Assembly Language to Proof Carrying Code

Alex Vaynberg
Advisor: Peter Lee

April 30, 2004

1 Introduction

This paper describes a translation between two methods of certifying binary code. By certified binary code, we mean code that comes with a mathematical justification that the code will not perform operations that will cause undesired effects. Such operations include writing to arbitrary point in memory and jumping to an arbitrary point in the code.

Before the methods for certifying code were developed, two approaches existed for showing that the code was safe to run.

1. The author places a cryptographic signature on the program.[7] This guarantees that the code is the code written by the the person whose signature is shown. Thus, the user can trust that the author did not place dangerous code into his program. However, it is a poor method, as the author may unknowingly produce dangerous code due to bugs and vulnerabilities.

2. The interpreter / virtual machine can babysit the code.[7]. This method, although guaranteeing that all dangerous instruction will not execute, tends to be quite slow.

The certified code approaches differ from the traditional approaches:

- There is no need to trust the author. The author supplies all the information necessary for the software user to easily check the program.

- There is no need to check the code at runtime. All the checking is done before running, leaving pure native binary to run on the actual machine. Thus the overhead is a one-time cost.

The methods for certifying code are Typed Assembly Language (TAL) and Proof Carrying Code (PCC). Other than their goal of producing safe programs, these methods are not very similar, utilizing very different approaches to guaranteeing safety. This is a consequence of how TAL and PCC programs are produced. TAL is typically compiled from languages that are already considered safe. PCC is well suited for the cases where static safety is not so obvious, or the conditions for safety are complex.

The goal of this work is to produce a translation from TAL to PCC. The reasons to why the translation is useful are as follows:

1. One system, but not the other, might be installed on the user's machine. Having a translator would eliminate the absolute need to have both systems. Moreover, it may be undesirable to have both systems since each requires a special program to inspect the code, and verify that it is indeed safe. This means that there would be two systems which may have safety compromising bugs.

2. The alternative to translation is retargetting. This means that a compiler for some language would have to know how to produce code for both systems. Given that the process of generating TAL if quite different from process of generating PCC, it is very likely that a single compiler will target a single system. Hence a translation would be useful.

3. The scientific community has already recognized that there is a connection between TAL and PCC [2]. This translation is another step in formalizing and understanding this connection.

The main outcome of this research is a complete specification of a translation from TAL to PCC. Additionally there is a proof that the translation of any safe program in TAL is a safe program in PCC that performs the same computation. Thus, we show that the PCC method for ensuring safety is at least as powerful as the method used by TAL.

1.1 Typed Assembly Language

1.1.1 Type System

Programs can execute operations that may not be specified, or even dangerous. Consider trying to add a function to a number, or writing to arbitrary location in memory. A program that does this may end up in an undefined state or possibly crash. To prevent this, we use a type system.

A type system is a set of inductively defined rules, such that any program that satisfies the rules can not ever execute an operation that results in an undefined state. This is achieved by keeping track of all data and checking that it is only used in the way that it can be used. The way a specific piece of data can be used is called a type.

The advantage of having a type system is that a simple check that a program satisfies the rules guarantees that the program will never be in an undefined state. There is no need to run or debug the program to determine this. It is a logical impossibility for the program to be well-typed and end up in an undefined state.

1.1.2 Typechecking

Typechecking is a process of checking that a program is valid in a given type system. This involves determining a type for every expression in the program, such that it is consistent with the rules. In some cases it is impossible to automatically determine types. This is resolved by using type annotations, a syntactic construct that explicitly defines a type for a specific expression. These ambiguities frequently occur when determining the types of functions, and some languages require the types of functions to be explicitly specified.

Once the entire program has been typed in accordance with the rules, a program is called well-typed.

1.1.3 Typed Assembly Language

Typed Assembly Language is a generic platform-independent assembly language reinforced with a type system. Since the type system is sound, any valid program written in TAL is safe.

The TAL code that is used in this paper is defined in the *Morissett et al.*[4] The type system allows for integers, tuples, recursive functions, existentials (packages of related information), and polymorphism. [4]

This typed assembly language can and has been made precise for various architectures with small adjustments. There is a working implementation for the x86 architecture: TALx86[3], which includes a compiler for the PopCorn language, which is a safe language that is similar to C[3].

1.2 Proof Carrying Code

Proof Carrying Code (PCC) is a different approach to certified code, designed by George C. Necula and Peter Lee.[5] The idea behind PCC is that each assembly instruction is associated with a clause in first-order logic that specifies the conditions required for the instruction to be safe, as well as how the state is altered.

Considering the definitions of machine instructions and an idea of what it means for the program to be safe, one can define a set of rules called the Verification Condition generator (VCgen). When the rules are applied to a program, they result in a statement in first-order logic, that is called the Verification Condition (VC) of the program.[1] If the statement is true, then at any point during execution of a program, the program is always in the state where it is safe to execute the next instruction.

At certain points in the program it is hard to determine the required set of conditions needed for VC to be true. For example, functions and loops may need to contain additional conditions than are not obvious from the code. These are accommodated by a special INV instruction that specifies invariants for entry into that block of code. Specifying incorrect invariants can not circumvent the safety of a program, since the invariants only serve as guides, and not as declared truths.

In order for the code to execute natively on a real machine, once the program is checked, the INV instructions are stripped to allow the code to run just like any native binary.

The last part of the PCC program is the guarantee of the truth of the VC. This is a precise, machine-checkable proof of the VC. A trusted proof checker can be used to check the proof. If the proof is valid, then the VC of the program is true, and hence the program can never execute an instruction while in the state that contradicts the safety requirement for that instruction.

The typical work-flow of a PCC-certified program is:

1. Code provider generates a PCC program by compiling from some source language. The resulting program contains INV instructions, as well as other information that will be helpful in generating the proof.

2. Provider runs VCgen to generate the VC (the logical statement about safety of the program).

3. Provider generates the proof of the VC (using the extra information in the assembly). This proof is attached to the program.

4. The user gets the code and the proof.

5. The user regenerates the VC using VCgen. This is done so that the provider could not provide an incorrect VC for the program.

6. User checks the proof of VCgen, which is an easy problem.

7. User is now sure that the program satisfies the safety that is demanded by VCgen, and proceeds to run the program.

2 Translation

2.1 The Starting Point

A precise definition of a TAL program consists of 3 items:

1. **Starting Heap.** A heap is a set of word-sized labels that point to large values that are stored in memory. There are two types of values that can exist in the heap: code/function (making the label a jumpable location), or a tuple. The items in the tuple may contain any word-sized values including labels back into the heap.

2. **Starting Register File.** A register file is an initial set of word-sized values that are assigned to registers before the program is run. The values may include labels into the heap.

3. **Starting Code Block.** This is a block of assembly instructions that is executed upon running the program. These differ from the code blocks in the heap by the fact that there are no labels pointing to the starting code block. This means that the instructions in the starting code block can be executed only once.

The code values in a heap of TAL program look like this:

```
code []{r1:<>,r2:int}
mov r1, r2
halt_int
```

Each code block contains a `code` header which describes the type requirements on arguments. In the example above, the code requires that `r2` (register 2) contains an integer, and that `r1` is a pointer to an empty tuple(unit). The representation of that is not specified, and can be simply treated as any value. The rest of the code value is a list of instructions that is well-typed given that the

$$\{\texttt{r1} :<>, \texttt{r2} : \mathit{int}\}$$

is indeed true.

The mathematical definition of well-typedness for code is specified like this:

$$\Psi; \Delta; \Gamma \vdash_{\texttt{TAL}} \texttt{I}$$

This means that the instructions `I` are well-typed in the context of a heap with a type Ψ, and a register file with type Γ. Δ is a list of type variables that are needed to express polymorphism, and full explanation is beyond the scope of the paper. However, it is handled in the tech report available from the author.

The well-typedness of the code is checked through the use of inductive rules such as the one for move:

$$\frac{\Psi; \Delta; \Gamma \vdash_{\texttt{TAL}} \texttt{v} : \tau \qquad \Psi; \Delta; \Gamma\{\texttt{r}_\texttt{d} : \tau\} \vdash_{\texttt{TAL}} \texttt{I}}{\Psi; \Delta; \Gamma \vdash_{\texttt{TAL}} \texttt{mov}\ \texttt{r}_\texttt{d},\ \texttt{v};\ \texttt{I}} \ \text{(s-mov)}$$

Below the line is the definition of the safety of any execution starting with the `mov`. Above the line is the requirement for the declaration below the line to be true. Thus in the example

```
code []{r1:<>,r2:int}
mov r1, r2
halt_int
```

the code block is safe if $\texttt{r}_2$ is an integer, and the $\texttt{halt}_{int}$ is well typed in the same initial heap, and with $\texttt{r}_1$ and $\texttt{r}_2$ being integers. The former is true by the requirements on the parameters, and the latter is true since the $\texttt{halt}_{int}$ instruction safety is given by the rule:

$$\frac{\Psi; \Delta; \Gamma \vdash_{\texttt{TAL}} \texttt{r}_1 : \tau}{\Psi; \Delta; \Gamma \vdash_{\texttt{TAL}} \texttt{halt}_\tau} \ (\text{s-halt}_\tau)$$

An additional observation about TAL code is that it is always in a continuation passing style.[4] This means that any `halt` instruction exits the program immediately, and that any jump taken is permanent, i.e. there is no way to get back to that point in the code, except to recursively call the function.

2.2 The Destination

The definition of PCC does not specify anything about initial heap or registers. The PCC program is considered safe when each block of code has a VC that is true. The VC for a block of code is generated based only on the the instructions in that block through a procedure called VCgen. VCgen is a deterministic function from assembly instruction to a logical term, and is completely specified in the PCC tech report.[6] Here is a snippet of the VCgen:

Instruction	VC
$\mathtt{MOV}\ r_s, r_d$	$[r_s/r_d]VC_{i+1}$
$\mathtt{BNE}\ r_s, n$	$(r_s = 0 \supset VC_{i+1}) \wedge$
	$(r_s \neq 0 \supset VC_{i+n+1})$
$\mathtt{INV}\ \mathcal{P}$	$\mathcal{P}$

The above rules state that MOV followed by some code would have a true VC if and only if the operations that follow it have a true VC given that r_d is replaced by r_s. This captures the meaning of the MOV operation, and puts that meaning into the logical definition of truth for a block of code. Similarly, the VC of the "jump if not zero" instruction captures the meaning of checking wether the register is zero, and jumping to the appropriate location in the code afterwards.

The VC of the INV instruction is interesting, as it is completely specified by the program. The invariant is supposed to be both necessary and sufficient for the VC of the operations that follow it to be true. Assuming that the invariants are placed at every entry point, the VC of the entire program is

$$\bigwedge_{i\ INV} (INV_i \supset VC_{i+1})$$

Here is an example of a PCC code block that is similar to the one for TAL:

$$\begin{array}{l}\mathtt{INV}\ \mathtt{r}_2 : \mathtt{int}\\ \mathtt{MOV}\ \mathtt{r}_1,\ \mathtt{r}_2\\ \mathtt{RET}\end{array}$$

Assume that the requirement for exiting the program is $r_1 : int$. Thus the VC of the RET is $r_1 : int$. Notice that the requirement on the entry point of the block is similar to the one in the TAL example. We require that r_2 be an integer, and require nothing of r_1. The VCgen of the code segment above is:

$$r_2 : int \supset [r_2/r_1]r_1 : int$$

which simplifies into

$$r_2 : int \supset r_2 : int$$

which is clearly true. This means that when we jump into the code block with r_2 being an integer, we exit the code block only in the ways that satisfy the invariants of the program. In this case it is the RET invariant of $r_1 : int$.

One extra feature is needed for the PCC that is the destination of a TAL translation: an alloc function. This is achieved by introducing an `ALLOC d, v` trusted assembly instruction that allocates `v` words of heap, and places the address into d. The VCgen for this instruction is an obvious one, which introduces the section of memory as readable and writable in the VCgen of the instructions that follow. In PCC terminology, the rule is:

$$\mathtt{v} > 0 \wedge \forall_{0 \leq \mathtt{i} \leq \mathtt{v}}(\mathtt{v} \oplus \mathtt{i} : \mathbf{rw\_addr})$$

2.3 Translation

The goal of the translation is to take a TAL program that is well typed, and convert it into an equivalent PCC program which has a true VC. There are two things that need to be converted: the values and the instructions. The values need to have a conversion mechanism because certain data might not be represented the same way in PCC as it is in TAL. A good example is a function pointer, something that TAL has, and PCC lacks. As a result, function pointers have to be represented as indexes, which are then compared, and appropriate jumps taken.

Thus there are two functions defined: TV for value translation, which can be seen in figures 2 and 1, and TRANS for instructions translation, defined in figure 3. Since there may be values embedded in the instructions, TRANS will use TV to decode the values.

The code translated using TRANS and TV has no guarantee of safety by itself. The way to establish safety is to convert the type information into invariants, such that the VC of the instructions will be true. Recall the rule for the mov instruction in TAL:

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathtt{TAL}} \mathtt{v} : \tau \qquad \Psi;\Delta;\Gamma\{\mathtt{r_d} : \tau\} \vdash_{\mathtt{TAL}} \mathtt{I}}{\Psi;\Delta;\Gamma \vdash_{\mathtt{TAL}} \mathtt{mov}\ \mathtt{r_d},\ \mathtt{v};\ \mathtt{I}}\ \text{(s-mov)}$$

$$\begin{aligned}
&\mathtt{TV_{wval}}(\mathtt{pack}\,[\tau,\mathtt{w}]\ \mathtt{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'\ \mathtt{wval})\\
&\quad\Rightarrow \mathtt{TV_{wval}}(\mathtt{w} : [\tau/\alpha]\tau'\ \mathtt{wval})\\
&\mathtt{TV_{wval}}(\mathtt{l} : \tau\ \mathtt{wval}) \Rightarrow \mathtt{l}\\
&\mathtt{TV_{wval}}(\mathtt{i} : \mathtt{int}\ \mathtt{wval}) \Rightarrow \mathtt{i}
\end{aligned}$$

Figure 1: Word-size Value Conversions

$$\begin{aligned}
&\mathtt{TV}(\mathtt{r} : \tau) \Rightarrow \mathtt{r}\\
&\mathtt{TV}(\mathtt{w} : \tau) \Rightarrow \mathtt{TV_{wval}}(\mathtt{w} : \tau\ \mathtt{wval})\\
&\mathtt{TV}(?\tau : \tau^0) \Rightarrow \mathtt{0}\\
&\mathtt{TV}(\mathtt{w} : \tau^\rho) \Rightarrow \mathtt{TV_{wval}}(\mathtt{w} : \tau\ \mathtt{wval})\\
&\mathtt{TV}(\mathtt{pack}\,[\tau,\mathtt{v}]\ \mathtt{as}\ \exists\alpha.\tau' : \exists\alpha.\tau')\\
&\quad\Rightarrow \mathtt{TV}(\mathtt{v} : [\tau/\alpha]\tau')
\end{aligned}$$

Figure 2: Value Conversion

$$\begin{aligned}
&\mathtt{TRANS}(\mathtt{mov}\ \mathtt{r_d},\ \mathtt{v};\mathtt{I}) \Rightarrow \mathtt{MOV}\ \mathtt{r_d},\ \mathtt{TV}(\mathtt{v});\mathtt{TRANS}(\mathtt{I})\\
&\mathtt{TRANS}(\mathtt{halt}_\tau) \Rightarrow \mathtt{RET}\\
&\mathtt{TRANS}(\mathtt{ld}\ \mathtt{r_d},\ \mathtt{r_s}[\mathtt{i}];\mathtt{I}) \Rightarrow \mathtt{LD}\ \mathtt{r_d},\mathtt{i}(\mathtt{r_s});\mathtt{TRANS}(\mathtt{I})\\
&\mathtt{TRANS}(\mathtt{st}\ \mathtt{r_d}[\mathtt{i}],\ \mathtt{r_s};\mathtt{I}) \Rightarrow \mathtt{ST}\ \mathtt{r_s},\mathtt{i}(\mathtt{r_d});\mathtt{TRANS}(\mathtt{I})\\
&\mathtt{TRANS}(\mathtt{jmp}\ \mathtt{v})\ \text{where}\ \mathtt{v} : \forall[].\Gamma'\ \Rightarrow \mathtt{JMP}\ (\mathtt{TJUMP}(\mathtt{v}))\\
\\
&\mathtt{TRANS}(\mathtt{beq}\ \mathtt{r},\mathtt{v};\mathtt{I}) \Rightarrow\ \mathtt{BEQ}\ \mathtt{r},\mathtt{TJUMP}(\mathtt{v});\mathtt{TRANS}(\mathtt{I})\\
&\mathtt{TRANS}(\mathtt{add}\ \mathtt{r_d},\mathtt{v_1},\mathtt{v_2};\mathtt{I}) \Rightarrow\ \mathtt{ADD}\ \mathtt{r_d},\mathtt{TV}(\mathtt{v_1}),\mathtt{TV}(\mathtt{v_2});\mathtt{TRANS}(\mathtt{I})\\
&\mathtt{TRANS}(\mathtt{mallocr_d}[\tau];\mathtt{I}) \Rightarrow \mathtt{ALLOCr_d},\mathtt{sizof}(\tau);\mathtt{TRANS}(\mathtt{I})\\
&\mathtt{TRANS}(\mathtt{unpack}\,[\mathtt{r_d},\alpha]\ \mathtt{as}\ \mathtt{v};\mathtt{I}) \Rightarrow \mathtt{MOV}\ \mathtt{r_d},\mathtt{TV}(\mathtt{v});\mathtt{TRANS}(\mathtt{I})
\end{aligned}$$

$\mathtt{TJUMP}(\mathtt{v}) \Rightarrow \mathtt{n}$ where $\mathtt{n}$ is the offset to the code that will inspect $\mathtt{TV}(\mathtt{v})$, and will make the jump to the function, whose address is stored in v. Formal description requires more machinery than is developed in this paper. Precise specification is shown in the technical report available from the author.

Figure 3: Instruction Translation

This rule shows how the typing is determined at a precise point in the code. The $\Psi; \Delta; \Gamma$ capture the notion of current state. The top left clause captures the requirements that the value must satisfy, and the top right clause does the recursion, to indicate that the rest of the code is well-typed.

This exact notion can be translated into PCC as proving

$$\frac{\mathtt{TT}(\Delta; \Gamma, \mathtt{TV}(\mathtt{v}) : \tau) \qquad \mathtt{TCTX}(\Psi; \Delta; \Gamma\{\mathtt{r_d} : \tau\}) \supset \mathtt{VCgen}(\mathtt{I})}{\mathtt{TCTX}(\Psi; \Delta; \Gamma) \supset \mathtt{VCgen}(\mathtt{MOV}\ \mathtt{TV}(\mathtt{v}), \mathtt{r_d}; \mathtt{TRANS}(\mathtt{I}))}$$

where TCTX captures the invariants in the current state of heap and registers, and TT captures an invariant of one specific value based on its type.

The idea is to select TCTX and TT such that the PCC analog is provable. The functions that work for the entirety of TAL are shown in the figures 4 and 5. It should not come as a surprise that the TCTX depends on TT, since the register file and the heap consist of values.

2.3.1 Functions

The definitions above have specified how to translate blocks of code given a translation of a context. However this does not cover the entry point. In TAL the "code" statement defines the starting context of the block of code, as well as defines the minimum set of requirements (types of heap and register, i.e. $\Psi; \emptyset; \Gamma$) on the state so that the machine can jump to it. This is exactly the job of the INV instruction in PCC. Thus a simple $\mathtt{TCTX}(\Psi; \emptyset; \Gamma)$ will produce the necessary invariant, as well as the needed starting context for the block of code that follows.

Going back to the example:

```
code []{r1:<>,r2:int}
mov r1, r2
halt_int
```

The translation prescribes that the `code` directive gets translated into

$$\mathtt{INV}\ \mathtt{r}_2 : \mathtt{int}$$

since the empty tuple type <> is an empty conjunction. Thus the PCC example is a translation of a TAL example.

2.3.2 Final Touches

The last few difficulties in translations are the starting and halting. The halt is actually quite easy: simply define the VCgen of RET to be $\mathtt{TT}(\mathtt{r}_1 : \tau)$ where τ is the type of a halt instruction. Obviously this means that all halts must result in the same type, which is not true in TAL. However, this is a very minor point, since we can define the return value to simply hold any type. Since the program stops running after exiting, forgetting the halt type will not affect any other part of the program.

The loader is a much more sensitive issue. The idea is by the time the program loads, the heap and the initial registers have a proper type, and for PCC they must have an invariant which is translation of the types.

An example would be a TAL program that starts with a tuple in the heap, and r_1 having a label that points to the tuple. The precise definition is:

$$(\{l_1 \mapsto < 5, 4 >\}, \{r_1 \mapsto l_1\}, I)$$

This sounds simple enough, but PCC has no direct way of dealing with this. There are many possible solutions to this problem, but the simple one is to simply have PCC load the values into memory and check them for consistency with the invariant.

Since the starting memory is consistent, and every code block has a true VC given the starting memory, the entire program has a true VC. Thus the entire program is safe.

3 Theorems

Just because the translation is well-specified, it does not mean it is correct. To be sure that the translation works we need answers to the following:

1. Does the translation result in PCC programs that have true VCs? This is a consequence of the *preservation of safety* theorem, which states that given a well-typed program in TAL, the translation produces a program with a true VC.

2. Can the translation handle any TAL program? This is a simple corollary of *preservation of safety*, and that the translation is defined for all possible programs.

3. Does the translated code perform the same computation? The correctness theorem answers that question by showing that the TAL program and a PCC program start in isomorphic states. Furthermore, if TAL program evaluates an operation to get to the next state, then the translated program can take one or more

Figure 4: Definition of TT

$\mathtt{TT}(\alpha, \Delta; \mathtt{tv} : \tau) \Rightarrow \bigvee_{\mathtt{T} \in \alpha} [\mathtt{T}/\alpha]\, [\mathtt{TT}(\Delta, \mathtt{v} : \tau)]$
$\mathtt{TT}(\cdot; \mathtt{tv} : \mathtt{int}) \Rightarrow \mathtt{tv} : \mathtt{int}$
$\mathtt{TT}(\cdot; \mathtt{tv} : \tau^0) \Rightarrow \mathtt{true}$ or perhaps $tv : int$
$\mathtt{TT}(\cdot; \mathtt{tv} : \tau^1) \Rightarrow \mathtt{TT}(\cdot; \mathtt{tv} : \tau)$
$\mathtt{TT}(\cdot; \mathtt{tv} :< \tau_1^{\varphi_1}, \ldots, \tau_i^{\varphi_i}, \ldots, \tau_n^{\varphi_n} >) \Rightarrow \bigwedge_{\mathtt{i}} (\mathtt{tv} \oplus \mathtt{i} : \mathtt{addr} \wedge \mathtt{TT}(\mathtt{sel}(\mathtt{r_m}, \mathtt{tv} \oplus \mathtt{i}) : \tau_i^{\rho_i}))$
$\mathtt{TT}(\cdot; \mathtt{v} : \forall[].\Gamma) \Rightarrow \bigvee_{\mathtt{f}:\forall[].\Gamma} (\mathtt{tv} = \mathtt{f})$
This means that r has the index (pc) of one of the functions that are of this type.
$\mathtt{TT}(\cdot; \mathtt{tv} : \exists\ \alpha.\tau) \Rightarrow \bigvee_{\mathtt{T} \in \alpha} \mathtt{TT}(\cdot; \mathtt{tv} : [\mathtt{T}/\alpha]\, \tau)$

$$\mathtt{TCTX}(\Psi; \Delta; \Gamma) \Rightarrow \bigwedge_{l:\tau \in \Psi} \mathtt{TT}(\emptyset; l : \tau) \wedge \bigwedge_{\mathtt{r}:\tau \in \Gamma} \mathtt{TT}(\Delta; \mathtt{r} : \tau)$$

Figure 5: Definition of TCTX

steps, to end up in a state isomorphic to the state of the TAL program. What this means is that the final state of the TAL program will be isomorphic to the final state of the PCC program, meaning that the programs are equivalent.

Together, these theorems mean that the translation is indeed correct.

4 Conclusion and Future Work

The work resulted in a precise description of the inductive translation from TAL to PCC, along with a proof that such translation is correct, i.e. any valid TAL program results in an equivalent valid PCC program. The next step is to construct an actual implementation of the translation. The translator should convert TALx86 into Touchstone PCC, which are working implementations of the TAL and PCC as described earlier.

Looking ahead, there is an open problem to show the opposite translation (PCC into TAL). This task would require fitting various logic statements into a type system. A possible interesting outgrowth of that work would be an instantaneous type system, which is specific to the program and its safety requirements, generated on the fly, and is accompanied by a proof of soundness. Once the PCC to TAL translation is designed, it would mean that the PCC and TAL are different, yet equivalent approaches to certifying binary code.

References

[1] Floyd, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science* (1967), J. T. Schwartz, Ed., American Mathematical Society, pp. 19–32.

[2] Hamid, N., Shao, Z., Trifonov, V., Monnier, S., and Ni, Z. A syntactic approach to foundational proof carrying-code, 2002.

[3] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. TALx86: A realistic typed assembly language. 1999.

[4] Morrisett, G., Walker, D., Crary, K., and Glew, N. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (1999), 527–568.

[5] Necula, G. C. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)* (Paris, Jan. 1997), pp. 106–119.

[6] Necula, G. C., and Lee, P. Proof-carrying code. Tech. Rep. CMU-CS-96-165, Carnegie Mellon University, November 1996.

[7] Necula, G. C., and Lee, P. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA* (Berkeley, CA, USA, 1996), USENIX, Ed., USENIX, pp. 229–243.