# The Weird Machines in Proof-Carrying Code

IEEE SPW LangSec 2014

Invited paper

Julien Vanegue

Bloomberg L.P.
New York, USA.

***Abstract*—A grand challenge in computer programming is the formal verification of dynamic properties of programs execution. Verification frameworks such as Proof-Carrying Code (PCC) enforce policies and memory models to prove that programs cannot go wrong. Due to the use of automated deduction techniques on models with machine abstraction, low-level details of the program execution can be changed without invalidating the formal proof. We capture the notion of *Weird Machines* in PCC to formalize the unspecified execution in programs whose proofs do not sufficiently enforce key properties to disallow execution of untrusted computations. We discuss ideas to improve existing verification systems so they are more resilient to weird machines.**

## I. Introduction

Proof-Carrying Code (PCC) [Nec97] and other computational verification systems [XL12] are frameworks in which untrusted programs can be verified to be safely executed according to the type and inference rules used to enforce formal contracts, checked either before (or during) program execution. In the last few decades, a great amount of effort has been dedicated to verify the safety of critical programs from embedded real-time systems to common software part of major operating systems [KEH+09]

Proof-Carrying code comes into two main flavors: the original Proof-Carrying Code [Nec97], and the Foundational Proof-Carrying code (FPCC) [App01]. It is expected in such systems to make use of type rules either directly in the axioms of the system (therefore making the system strongly tied to the type system), although FPCC [App01] forces each type rule to be first defined from ground logical axioms before they can be used in proofs. A tempting assumption for PCC is to use it as an integrity system where the program is ensured to satisfy its specification as long as its proof is independently verified by the executing system. However, the original goal of PCC does the capture the intent that the system *only* executes what the specification enforces, and nothing else. This is shown in the set of original PCC articles where invariants are checked at specific program points by introducing a new *virtual* instruction **INV** whose parameter is an assertion that must be verified by the program in that context so that execution can continue. PCC does not capture whether the program also executed additional instructions that were not accounted for in the specification. We say that PCC is vulnerable to **weird machines**, computational artifacts (some say *gadgets*) where additional code execution can happen in proved programs and will escape the program specification.

The original definition of Foundational Proof Carrying Code (FPCC) [App01] is defined by semantic rules for an abstract machine instruction set, and contains additional conditions capturing that parts of the machine contexts (memory, registers, etc) are not affected by instructions. This is characterized in FPCC by using memory contexts to track values that have changed during the execution of the typed inference rule. As such, FPCC is more resilient to *weird machines* than PCC. Additionally, FPCC suggests the use of **type-preserving compilers**, such as the one in CompCert [Ler06], where proofs in source code can translate to proofs on machine code. This is to protect against invalidating the safety invariants in an untrusted or incorrect compiler. Such end-to-end certification systems offer fewer opportunities to go wrong than traditional PCC-style systems.

Nonetheless, attacks on FPCC can happen when the memory model, the machine abstraction, or the policy itself are incomplete or incorrect. In attacks against memory model, an attacker takes advantage of the fact that real machine operations are not captured in the PCC machine semantics. For example, the order of bits in the encoding of data types (such as integers, pointers and bit vectors) is specified by the memory model. The CompCert memory model [XL12] represents memory at the byte level and models the sign of integers. It can deal with invalid computations acting on a mix of pointer and integer variables. For example, single bytes of pointer typed values in CompCert cannot be manipulated directly, thus capturing errors due to partial pointer overrides happening during memory corruption issues. On the other hand, no bit field algebra is available besides the one allowing conversion from integer to float (double) type such as in the presence of union types in the C language. As such, the model fails to capture cases where bit field of 31 bits are cast

from/to 32 bits integers. Since variable-length bit fields are not supported in the memory model, and it is unclear how to define a program that manipulates such objects in *CompCert*, as this can be the case in common programs.

The machine abstraction is used to simplify the real machine and forget details that are not important to perform proofs. If the (F)PCC framework is driven by a fixed set of invariants, then such loss of precision can be avoided by introducing appropriate representations for resources and instructions and keep soundness when verifying the invariants of the contract. However, when faced to an attacker with the ability to inject code in the program, the proof system can be built around specific properties. Therefore, any used abstraction is the opportunity for an attacker to introduce uncaptured computations or side effects that are not accounted for in the proof. As such, failure to capture some of the real machine specification introduces potential to perform computations that will discover unintended state space of the program.

A central trait of architecture in both **PCC** and **FPCC** resides in the underlying formal system used to verify logic formulae encoding the desired invariants of programs. In PCC, a subset of first order predicate logic as well as application-dependent predicates are predominating. In FPCC, Church's Higher Order Logic (HOL) is used, giving proofs the ability to reason on function types (as well as record types). None of these logic take resource into account, and it is possible to define proofs in multiple ways depending on what order of application is chosen on hypothesis (in lambda calculus jargon: multiple evaluation strategies can be chosen to reduce the proof term down its normal form). For example, proving the type of a record $r : A \times B$ can be proved first by proving $\pi_1(r) : A$ then $\pi_2(r) : B$, or by first proving $\pi_2(r) : B$ then $\pi_1(r) : A$. The order of evaluation is not specified by the formal logic. Moreover, there can be unused hypothesis, or hypothesis can be used multiple times. This introduces an opportunity for attackers to perform hypothesis reuse and compute additional operations without invalidating proofs. Other systems based on linear logic [Gir87] attempted controlling the resource management aspect of such proofs directly in the logic [PP99], though no complete theory or implementation of linear proof carrying code has been established as of today.

Additional policies can be used to enforce structural constraints on programs, and can also incur unwanted behavior when data and code can be intermixed (sometimes on purpose to support self-modifying code). Such policies are dangerous not only when code can be rewritten but also when data can be executed. This gives a full cycle of code generation primitives for an attacker to fool the security system. Therefore, we discourage the allowance of such primitives when real program safety is expected.

This article discusses the need for verification systems aimed at understanding attacker potential and minimizing opportunities of unspecified behavior in verified programs. Though the case is made using the example of Proof-Carrying Code, any formal verification system introducing abstraction in proofs is exposed to weird machines and other uncaptured computations.

## II. On Limits of Proof-Carrying Code

**Proof-Carrying Code** (or **PCC**) is a framework for the verification of safety policies applied to mobile, untrusted programs. **PCC** relies upon the fact that, while the construction of a proof is a complex task involving the code compiler and a Verification Condition generator (**VCGen**), verification of proofs is easy enough given the proof and the program.

Mechanisms of **PCC** are captured using type rules of the simple form:

$$\rho \vDash o : T$$

where $\rho$ is the register state containing the values of registers $r_0, r_1, r_2, ..., r_n$, and $o$ is a program object of type **T** down to individual expressions and (constant) variables. Type rule derivations employed to represent the program constitute the proof that the program executes accordingly to its type specification. Types can be used to prove that an address is valid, read-only, or that a result register holds the expected value at chosen program points given certain inputs of the verification procedure. As such, proof-carrying code in its simple form corresponds to program property checking, where particular constraints are expected to be true and consistent at a given program point (for example, at the precondition of a particular API, or at the header of a loop, etc).

We make the following remarks about **PCC** and related systems:

- We warn of a potential misconception that PCC could be used for lightweight program integrity. We explain why using **PCC** for program integrity is insecure even though **PCC** employs a sound proof system to verify mobile proofs.
- We claim that the problem of unspecified computations is independent of the chosen proof construction, encoding or verification algorithms
- We note that the described problem is not specific to either particular programs or policies.
- We argue that the use of abstractions in proofs gives up potential for attackers to introduce additional malicious program parts whose execution do not invalidate original proofs but still perform other unspecified operations together with the normal proved program behavior.

Only specific families of proof systems taking resources into accounts have the ability to express proofs in a way that can avoid unwanted computations. In particular, the ability to control precise resource creation and consumption is central to desired security proofs. For example, such systems could be tentatively constructed from linear and affine logic [Gir87] or game semanatics [HO00] where hypotheses contexts precisely track the number of available resource instances.

Our goal is to illustrate that rogue programs can satisfy legitimate proofs as long as the same properties are provably observed at predefined chosen program points. The Global Safety Proof for a program is expressed as follow in PCC:

$$SP(\Pi, Inv, Post) = \forall r_k : \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

where the verification condition, constructed from the verified program since the beginning of the $i^{th}$ procedure segment, implies that the enforced invariant is true at the end of the procedure segment $i+1$ .

### A. The Proof aliasing problem

The limits of proof-carrying code are illustrated by the creation of another program $\Pi'$ which also verifies the proof originally made for $\Pi$ :

$$\exists \Pi' : SP(\Pi', Inv, Post)$$

We call this phenomenon the *program proof aliasing* or *PPA* problem. The PPA problem has macro-level consequences for the whole program proof as expressed in PCC since there is now an equivalence relation such as:

$$\Pi \equiv \Pi' \\ \triangleq \\ SP(\Pi, Inv, Post) \iff SP(\Pi', Inv, Post)$$

Two programs are proof-aliased when one satisfy the proof if and only if the other does.

### B. Perfect Proof-Carrying Code

One may want to define a perfect version of PCC where the PPA problem does not arise. We call this version $PCC_{\equiv_\alpha}$ to distinguish it from PCC as originally defined. The absence of proof aliasing for programs leads to defining the *strongest formulation* for the safety condition that avoids unwanted computations. Such formalization states that there is a unique program satisfying a given safety proof:

$$\exists ! p \text{ such that } SP(p, Inv, Post)$$

Under this definition, one cannot construct a proof that is applicable to two programs. This very strong statement is equivalent to the existence of an isomorphism between low-level programs and their proofs. Such definition does not leave any room for small optimization or other changes that are otherwise harmless for the program or its proof. A weaker perfect safety condition, easier to employ but still avoiding unwanted computation, would allow used resources (such as memory or register instances) to be different while allowing the same proof (modulo renaming):

$$\Pi_1 \equiv_\alpha \Pi_2 \\ \triangleq \\ SP(\Pi_1, Inv, Post) \iff SP(\Pi_2, Inv_\alpha, Post_\alpha)$$

where $Inv_\alpha$ (resp. $Post_\alpha$ ) are the original invariants (respectively post-condition) in the safety proof *SP* after applying the same $\alpha$-renaming used to obtain $P_2$ from $P_1$ . Proof-equivalence modulo $\alpha$-renaming from $P_2$ to $P_1$ can be expressed similarly. This alternate definition can be useful when resources used in proofs are identified by indexes, addresses or offsets rather than names.

A limitation to this approach is that any two structurally different programs (like two program with different control flows) must have strictly different proofs (and different proof trees) even when these programs are observationally equivalent. This captures the intuition that the amount of resources needed to satisfy a specification should be minimal and well identified for each program pretending to satisfy it. Unlike such perfect system, a realistic system should aim at finding equivalence classes of programs where the same proof is acceptable for two different elements as long as certain properties of interest are guaranteed without loss of precision.

## III. The Nature of Untrusted Computations

An original approach to modeling attacker potential is by formally studying the **weird machines** (WM) [BLP+11] intending to describe the security exploits slipping through verified programs. For simplicity, the use of the axiomatic semantic ala Hoare [Hoa69] allows us to study the machines independently of the chosen instruction set.

### A. Weird control-flow

Let $CFG = < \{V\}, \{E\} >$ be a usual definition of a control flow graph made of a set of vertices and edges (with the edge set $E : V \to V$) .
$V_i \in V = (i_1, i_2, ..., i_n)$ is a vertice in the CFG such as a basic block made of a list of instructions).

We define a family of projections $\Pi_j : V \to V$ such that $\Pi_j(V_i) = V_{i'} = \{i_j\}$ a singleton obtained by unitary projection on the list of instructions of the basic block.

Let $\Pi_S(V_S) : V \to V$ such that $V'_S = \{i_{j \in S}\}$ $= \{V_{j1}, V_{j2}, ..., V_{jn}\}$ with $\{j_1, ..., j_n\} \in S$ such that $j_1 < ... < j_n$ .

be a partition obtained by bigger (union of) projections. The sequence of instructions obtained by union of projections is guaranteed to be in order, but does not have to be contiguous over V.

Some examples of projections on S are *V*-suffixes, or more general sub-sequences. Such sub-sequences can be contiguous or non-contiguous. Projections can be *V*-suffixes as in the case where new basic blocks are created from the end of existing ones by skipping instructions at the beginning of blocks. They can also be sub-sequences of basic blocks in which not all instructions are executed in the block, but where executed instructions are guaranteed to be found one after the other. We call contiguous sub-sequences of *V* the result of these projections. These can arise if an exception is triggered and not all instructions in the basic block are executed. We also distinguish non-contiguous sub-sequences of V where executed instructions are not guaranteed to be contiguous in the address space of the program. This is the case for architectures with conditional instructions whose execution depends on some internal state of the processor such as status flags, content of translation look-aside buffers used in linear to physical address translation, or other state that may or may not be directly accessible to the program.

We define the *Weird Control-Flow Graph* (WCFG) as:

$$WCFG = \{CFG\} \quad \bigcup \quad \{\langle V', E'\rangle\}$$

such that $E' = W \rightarrow V'$ with $W \in V(CFG)$ and $V' \notin V(CFG)$ . By definition of the WCFG, $E' \notin E$ . Note that a WCFG cannot exist if $E = E'$ since any extra state would not be reachable on the WCFG. Therefore, $E'_{\#} >> E_{\#}$

### B. Weird computations

Weird computations can be defined using axiomatic semantics of Hoare as interpretation over the Weird Control-Flow Graph (WCFG). We note $\{P\}c\{Q\}$ to express the partial verification conditions when a code fragment *c* terminates with given postconditions *Q* when provided with initial preconditions *P*. Axiomatic semantics of a program is given by applying composition rules on the semantics of its individual fragments.

$$\begin{aligned}&\{Pre\} < V > \{Post\}\\ &= \{Pre\} < i_1; i_2; ...; i_n > \{Post\}\\ &= \{Pre\} < i_1 > \{Post_1\} ... < i_n > \{Post_n\}\end{aligned}$$

Each *Pre* and *Post* are invariants conditions (first order logic formulae) locally verified at each state of the execution.

We can express, very much like the tape of a Turing machine, the values in *vs* satisfying the Invariant *Inv*, where **VS** is a value store and **vs** are the values in the store.

$$vs = (d_1, d_2, ..., d_n) : VS \vDash Inv$$

Value stores can represent registers and memory cells. While it is easier to reason about states using invariants at the abstract level, values allow to map invariant to concrete execution states of the program, may they be legitimate (expected) states or unexpected and unspecified weird states. Depending on the invariant, there may be a single, multiple, or no valuation satisfying it. The axiomatic semantics on vertices of the WCFG can be seen as:

$$\begin{aligned}&\{Pre\} < V' > \{Post\}\\ &= \{Pre\}\Pi_S(< i_1; i_2; ...; i_n >)\{Post'\}\\ &= \{Pre\} < i_\alpha > \{Post'\} < i_\beta > \{Post_\beta\} < ... > \{Post_\omega\}\end{aligned}$$

where $\{\alpha, \beta, ..., \omega\} \in S$ such that $\alpha < \beta < ... < \omega$ .

Note: $\{Post^\omega\}$ is a **weird state** $\leftrightarrow$ $\exists \Pi_S$ such that $\{Post^\omega\} \nsubseteq \{Post\}$ .

### C. Weird executions

We now abstract the executable code to focus on the sequence of states produced by executing this code.

A weird sequence $s \in S = Post^\alpha, Post^\beta, ..., Post^\omega$ is a sequence of invariants verified by executing paths on the WCFG. We can represent the weird sequence using valuations satisfying all the intermediate invariants, rather than the invariant themselves:

$$\begin{aligned}vs^\alpha &\vDash Post^\alpha\\ vs^\beta &\vDash Post^\beta\\ &...\\ vs^\omega &\vDash Post^\omega\end{aligned}$$

In order to reach one such weird state, we define a distance function : $\delta : S \times S \rightarrow \mathbb{N}$ and we say that a weird sequence converges if:

$$\delta(Post^\alpha, F) > \delta(Post^\beta, F) > ... > \delta(Post^\omega, F)$$

e.g. the distance between the current weird state and the desired final weird state **F** keeps diminishing.

ex: $\delta(S_1, S_2) = \sum_i d_i \in S_1 \equiv d_i \in S_2$

Final states can be chosen depending on the desired end state for an attacker. For example, a final state can be defined as a state where the values of specific registers is controlled (such as the instruction pointer register). Sometimes, an attacker will choose desired final states that do not necessarily involve full untrusted execution, such as these allowing to read or write specific variables. For example, one may want to read credential information from a program, or force the program to accept a successful authentication even though no valid credentials have been entered.

The distance metric between two states can be defined (without loss of generality) as the number of equivalent values in the value stores representing each state. State equivalence can then be defined as:

$$S_1 \equiv S_2 \qquad \Longleftrightarrow \qquad \forall d_i \in S_{1,2} : d_{i1} = d_{i2}$$

If $\delta \longrightarrow 0$ , the weird sequence is said to converge. Otherwise, the sequence diverges (that is, it comes back to a non-weird state, or may simply diverge in the weird state space if not enough computational power is given to reach desired final weird states)

## IV. Computational Models of Trust

Objects like weird machines are convenient to model untrusted code execution but a complete definition of weird machines remains to be given. A weird machine should be defined in terms of push-down automata to accurately represent the stack-based control mechanisms used in common security exploits. For example, overwriting a return address or an exception handler to redirect control flow can be modeled as a reachability problem on a push-down automata. Moreover, features of transducers, in particular the ability to reason on program output, is necessary to develop compositions of traces where a first execution is used to obtain some information about the program (such as variable values or internal address space information) and a subsequent trace is used to perform operation based on this guessed information. For example, an information disclosure vulnerability may be used to guess the location of existing legitimate instructions that will be later be executed to perform new operations. This corresponds to reordering valid computations within a program to reach new states.

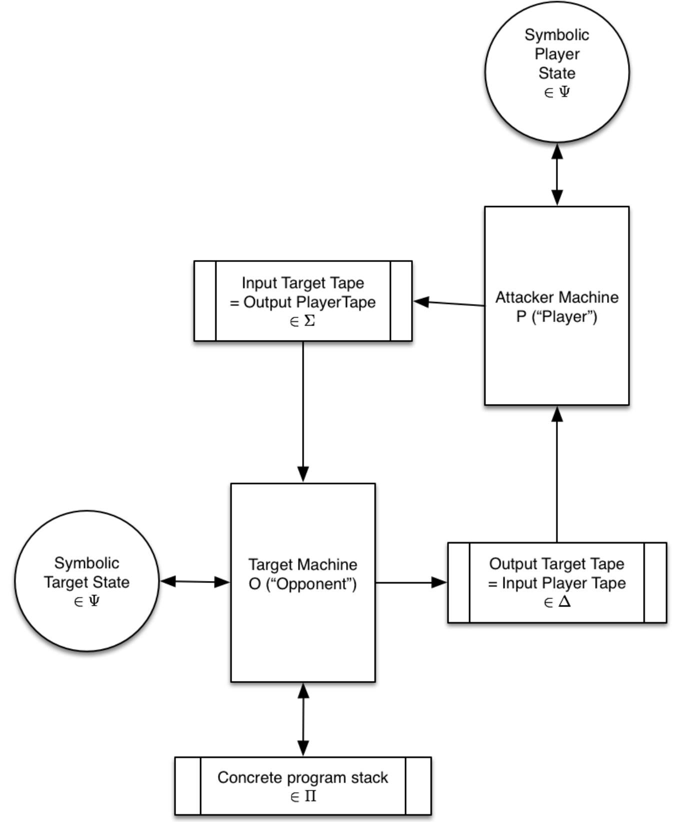
A Weird Machine $WM = < P, O, \Sigma, \Pi, \Delta, \Psi >$ can be used where:

- $P$ is the player e.g. the attacker machine.
- $O$ is the opponent/environment modeling the program under attack.
- $\Sigma$ is the concrete language of the attacker output tape (which coincides with the program input tape).
- $\Pi$ is the concrete language of the program stack.
- $\Delta$ is the concrete language of the program output tape (which coincides with the attacker input tape).
- $\Psi$ is the symbolic language modeling the player and opponent heap states.

in which the attacker transition function $\delta(O, \Delta, \Psi) \in \Sigma$ and the target program transition function $\phi(P, \Sigma, \Pi, \Psi) \in \Delta$ are used to make progress on the WM.

Such machines have remarkable properties:

- The WM is a combination of two transducers (a.k.a. input/output automata) where the input tape of one transducer corresponds to the output tape of the second. As the attacker can feed input to the program, the attacker machine is represented as the player while the program is the opponent. Execution on such machine is a game between player and opponent, unlike usual computational models where the program under execution is central and attacker is not formally modeled.




- The WM is a hybrid concrete / symbolic abstract machine. Concrete representation is retained to model input and output behavior as well as the stack behavior given the central role played by the stack when storing control records. In the other hand, maintaining all possible concrete heap configurations for a program is intractable. Instead, we choose to represent heap state symbolically using a formal logic language (such as, and without loss of generality, first order logic). As such, WM are *concrete symbolic abstract machine* or *concolic machine*.
- The WM construct is generic and parameterized with a target language semantics **S** (interpretation rules of the target program) and a background predicate **BP** (the set of symbolic assumptions used for heap state management)

Unlike other intermediate forms like *Boogie*, a weird machine representation should maintain a concrete program stack as to retain machine-level encoding for scoped execution and continuations, so that common attack patterns like *return oriented programming* can be modeled. Following patterns of reactive programming using the two transducers will allow for input/output characterization of attacks where information disclosure or multiple interaction exploits are performed on the system. In such weird machines, attacker knowledge can be represented by what hypothesis are known by the player and how these can be used to guide further target program analysis and execution. It is important to split the target program knowledge base and the attacker knowledge base since multiple security protections aims at

hiding internal program state from attackers. Therefore, not all symbolic target program state is known by the attacker and splitting environment is meant to represent this constraint.

A more ambitious problem is to define a version of proof-carrying code that is restricted enough to avoid weird machines but relaxed enough to allow equivalence classes between programs, so that some legit modifications of the program (like optimizing transformation) may be performed without invalidating the proofs. Such system could possibly use principles of linear logic under the hood, so that resources are precisely accounted for. For example, it should be forbidden for programs to compute intermediate results that are not reused, or that are reused multiple times. While the former problem can be approach using program simplification such as dead-code elimination, the latter can be difficult if the program intent is to store results of computations for later reuse as in dynamic or *divide and conquer* programming. It becomes necessary to measure operations performed on these value stores to ensure that only intended code gets executed and no extra computational power is given to attackers.

## V. Conclusion

The *Weird Machines* in verification systems such as Proof-Carrying code take advantage of equivalence classes introduced by abstractions in program proofs. Proof-Carrying code and related systems guarantee that programs satisfy safety properties but does not guarantee the absence of other side effects that may not invalidate the main safety proofs. Abstraction in proof systems may surrender the ability to distinguish unintended program computations from intended ones, and established proofs may not guarantee that the system is free from other contingent behavior remaining uncaptured by the safety conditions. We therefore warn that such verification systems should carefully be considered when computational integrity properties are meant to be preserved.

## References


[App01] Andrew W. Appel. Foundational proof-carrying code. 2001.

[BLP+11] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit programming: from buffer overflows to weird machines and theory of computation.; login, 2011.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987.

[HO00] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for pcf: I, ii, and {III}. *Information and Computation*, 163(2):285 – 408, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. pages 207–220, 2009.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *SIGPLAN Not.*, 41(1):42–54, January 2006.

[Nec97] George C. Necula. Proof-carrying code. 1997.

[PP99] Mark Plesko and Frank Pfenning. A formalization of the proof-carrying code architecture in a linear logical framework. 1999.

[XL12] Sandrine Blazy Gordon Stewart Xavier Leroy, Andrew Appel. The compcert memory model, version 2. *INRIA Research Report 7987*, 2012.