The Weird Machines in Proof-Carrying Code

Invited paper

IEEE Security and Privacy LangSec Workshop 2014

Julien Vanegue

Bloomberg L.P.
New York, USA.

***Abstract*—We review different attack vectors on Proof-Carrying Code (PCC) related to policy, memory model, machine abstraction, and formal system. We capture the notion of weird machines in PCC to formalize the shadow execution arising in programs when their proofs do not sufficiently capture and disallow the execution of untrusted computations. We suggest a few ideas to improve existing PCC systems so they are more resilient to memory attacks.**

I. Introduction

Proof-Carrying Code (PCC) [Nec97] is a framework in which untrusted programs can be securely executed based on enforcing a contract, 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.

Proof-Carrying code comes into two main flavors: the original Proof-Carrying Code of Necula, and the Foundational Proof-Carrying (FPCC) Code by Appel. While in PCC, it is possible to make use of type rules directly in the axioms of the system (therefore making the system strongly tied to the type system), FPCC forces each type rule to be first defined from ground axioms. As such, new FPCC's type systems can be customized without losing soundness.

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 PCC goal was not to prove 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.

Foundational Proof Carrying Code (FPCC) [App01] was designed in a different approach, where semantic rules for an abstract machine instruction set contain 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 only alues that have changed during the execution of the typed inference rule. As such, FPCC is more resiliant 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 much 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] represent memory at the byte level and models integer signs. It can deal with invalid computations acting on a mix of pointer and integer variables. For example, the pointer type in CompCert cannot be manipulated byte by byte, which blocks attacks based on partial pointer overrides due to 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, it is unclear how to define a program that manipulates such objects in *CompCert*, as this can be the case in common programs.

DOI 10.1109/SPW.2014.37

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 controlled by introducing appropriate representations for resources and instructions and keep soundness when verifying these invariants. 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 introduce potential to perform computations that will discover unintended state space of the program.
Policy modeling can also suffer from limits 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.

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.

This article falls short of defining such new flavor of PCC. Our goal is much simpler: we use the basic rules of Proof-Carrying Code to study formally what weird machines are and where they can hide in program even in the presence of proofs. Our next section introduces basics of proof-carrying code reasoning. We perform a higher level axiomatic reasoning on program traces to characterize how additional state space exploration can be achieved without invalidating the precepts of Proof-Carrying Code.

II. 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 general form:

$$\rho \vDash o : T$$

where ρ 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**:

- While this approach to verification is central to various abstract models in program analysis and certification, it is a widespread misconception that PCC can be used for lightweight program integrity. We explain in this section why using **PCC** for program integrity is insecure even though **PCC** employs a sound proof system to verify mobile proofs.

- We show that this problem is independent of the chosen proof construction, encoding or verification algorithms, neither are the problems specific to particular programs or policies.

- Using abstractions in proofs gives attackers highways for introducing additional malicious program parts whose execution do not invalidate original proofs but allow attackers to perform other unspecified operations as part of 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 the number of times that resources (registers, processor flags, memory cells, etc) are consumed and produced is central to such proofs. Such

immune systems are typically constructed out of linear (or affine) logic or game semantics [HO00]

In other words, any program can satisfy the proof enforced by the property checker, as long as the same values are provably always observed at these program points.

The Global Safety Proof for a program is expressed as follow:

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

which enforces that 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

Formally, the limits of proof-carrying code are illustrated by the creation of another program Π' which also verifies the proof originally made for Π :

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

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. Ideal Proof-Carrying Code

It is possible to define an ideal 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 is a very strong statement which is equivalent to the existence of an isomorphism between low-level programs and their proofs. A weaker safety condition that avoids unwanted computation is similar but allow used resources (such as register names) to be different while allowing the same proof (modulo the same renaming):

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

where Inv_α (resp. $Post_\alpha$) are the original Invariant (resp. Postcondition) of the Safety Proof *SP* after applying the same α-renaming used to obtain P_2 from P_1 . Inversely, a symmetric definition is given for proof-equivalence modulo α-renaming from P_2 to P_1 . This alternate definition can be useful when resources used in proofs are identified by indices, addresses or offsets rather than names.

An obvious limitation to this approach is that any two programs must have strictly different proofs (e.g. 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 ideal 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.

III. Weird Machines

In this section, we approach the definition of **weird machines** (WM) objects [BLP+11] . WM are made of untrusted control flow and instructions which can happen when proof-carrying systems like PCC fail to capture all required and sufficient conditions to characterize what a safe program constitutes when verifying a given specification. The use of the axiomatic semantic ala Hoare [Hoa69] allows us to study the machines independently of the chosen instruction set.

Following Hoare, 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 pre-conditions *P*. Axiomatic semantics of a program is given by applying such rules compositionally based on the semantics of its individual fragments.

A. Weird control-flow

Let $CFG =< \{V\}, \{E\} >$ a control flow graph is 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 Hoare's Axiomatic semantics as interpretation over the Weird Control-Flow Graph (WCFG) :

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

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. Future work

Beyond simple principles described in this document, a more complete detailed review of proof-carrying code security is due. Weird machines are a convenient way of modeling untrusted code execution but a complete definition of weird machines remain 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. Such a complete definition and illustrative example is the subject of future work.

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 can be resolved using program simplification such as dead-code elimination, the latter can be difficult if the intent of the program is to store the results of computations for later reuse (as in dynamic 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

We characterized the concept of *weird* machines in relation to the framework of proof-carrying code. Proof-Carrying code guarantees that a program satisfies safety properties. The introduction of abstraction in such proof systems surrenders the ability to distinguish the presence of unintended program computations when these do not invalidate program proofs. Such abstract traits can be used to perform shadow execution of code that remains uncaptured by the program specification. As such, Proof-Carrying Code should be used in conjunction to systems providing strong integrity, such as these introduced by the usage of cryptographic signatures.

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.

[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.