The Weird Machines in Proof-Carrying Code

IEEE Security and Privacy LangSec Workshop

Julien Vanegue

Bloomberg L.P.
New York, USA.

May 18, 2014

Proof-Carrying Code

- PCC: bundle a proof with a program so the proof can be checked once the program is taken from an untrusted source
- Ex: a web site, an untrusted compiler...
- Type rules of the form: $\rho \vDash o : T$
- ρ is the register state containing the values of registers $r_0, r_1, r_2, \ldots, r_n$
- o is a program object of type **T** down to individual expressions and (constant) variables.
- FPCC: Foundational Proof-Carrying Code - PCC proving everything from the ground up.
- FPCC first prove ground type rules in logic, then use proved type rules in proofs.

PCC example

```
XOR r2, r2 // r2 = 0
INC r2 // r2 = 1
MOV r1, 4 // r2 = 1 and r1 = 4
ADD r1, r2 // r2 = 1 and r1 = 5
MUL r1, r1 // r2 = 1 and r1 = 25
INV (r1 == 25)
DIV r1, 5 // r2 = 1 and r1 = 5
RET r1
INV (r1 == 5)
```

PCC Global Safety Predicate

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

First issues with PCC

- Checks are performed locally using a virtual **INV** instruction (invariant)
- Very much like a partial observational equivalence
- **No guarantees on the order or nature of intermediate computations**
- $r : A \times B$ can be proved first by proving $\pi_1(r) : A$ then $\pi_2(r) : B$, or by first proving $\pi_2(r) : B$ then $\pi_1(r) : A$.

Design issue with PCC

- While PCC is based on sound proof systems, it was made to verify discrete proofs of programs
- Note: FPCC is stricter on that matter though real-world implementations
- **PCC was not created to reason about attacker capabilities**
- Consequence: attacker an execute unspecified operations as long as the proof remains valid

Side remark : Proof aliasing Problem

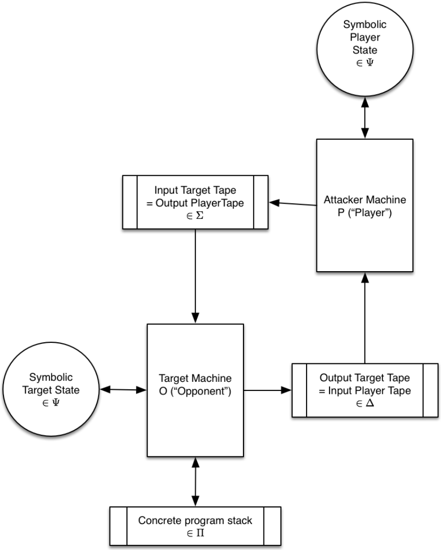
$$\exists \Pi' : SP(\Pi', Inv, Post)$$

$$SP(\Pi, Inv, Post) \quad \overset{\displaystyle \Pi \equiv \Pi'}{\triangleq} \quad \Longleftrightarrow \quad SP(\Pi', Inv, Post)$$

Weird Machines

$WM =< P, O, \Sigma, \Pi, \Delta, \Psi >$ where:

- P is the player e.g. the attacker machine.
- O is the opponent/environment modelling the program under attack.
- Σ is the concrete language of the attacker output tape (which coincides with the program input tape).
- Π is the concrete language of the program stack.
- Δ is the concrete language of the program output tape (which coincides with the attacker input tape).
- Ψ is the symbolic language modelling the player and opponent heap states.


Symbolic Player State ∈ Ψ
Input Target Tape = Output PlayerTape ∈ Σ
Attacker Machine P (“Player”)
Symbolic Target State ∈ Ψ
Target Machine O (“Opponent”)
Output Target Tape = Input Player Tape ∈ Δ
Concrete program stack ∈ Π

Properties of Weird Machines

- 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.
- The WM is a hybrid concrete / symbolic abstract machine (*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**

Concrete example: TLS heartbeat vulnerability

Require: **sock** : Valid network socket
Ensure: **True** on success, **False** on failure

```
char buff[MAX_SIZE]
int readlen = recv(sock, buff, MAX_SIZE);
if (readlen ≤ 0) return False;
rec_t *hdr = (rec_t *) buff;
char *out = malloc(sizeof(rec_t) + hdr->len);
if (NULL == out) return (false);
memcpy(out, buff + sizeof(rec_t), hdr->len);
out->len = hdr->len;
send(sock, out, hdr->len + sizeof(rec_t));
free(out);
return True
```

Target semantics : assignment

$$\dfrac{\dfrac{\Psi \;\vDash\; \Psi', \Psi'' \qquad a \in \Psi' \qquad a \notin \Psi''}{a = b}}{\Psi \;\vDash\; \Psi' \qquad a = b}$$

Target semantics : function call

$$\frac{\dfrac{\Pi = \Pi' :: p_n :: (...) :: p_1}{l1 : f(p_1, ..., p_n) \quad l2 :}}{\Pi = \Pi' :: l2 \qquad \$pc = addr(f)}$$

Target semantics : return

$$\frac{\dfrac{\Pi = \Pi' :: l2}{return \quad v}}{\Pi = \Pi' :: v \qquad \$pc = l2}$$

Target semantics : send

$$\dfrac{\dfrac{\Pi = \Pi' :: a :: b :: c}{ret = send(a, b, c)}}{\Pi = \Pi' :: ret \qquad \Delta = \Delta' :: b1 :: b2 :: (...) :: b_{ret}}$$

Target semantics : receive

$$\frac{\dfrac{\Sigma = \Sigma' :: v_1 :: v_2 :: (...) :: v_c \qquad \Pi = \Pi' :: a :: b :: c}{ret = recv(a, b, c)}}{\Pi = \Pi' :: ret \qquad ret \leq c \qquad \forall\delta \in [0, ret[:\ b[\delta] = v_\delta}$$

Target semantics : malloc

$$\cfrac{\cfrac{\exists p \forall \delta \in [0, sz[: \mathbf{M}(p+\delta) \wedge \neg \mathbf{A}(p+\delta)}{p = malloc(sz)}}{\forall \delta \in [0, sz[: \mathbf{A}(v+\delta) \wedge v = p}$$

Target semantics : free

$$\frac{\dfrac{\forall \delta \in [0, sz[: A(v+\delta) \wedge v = p}{free(p)}}{\exists p \forall \delta \in [0, sz[: \mathbf{M}(p+\delta) \wedge \neg \mathbf{A}(p+\delta)}$$

Target semantics : memcpy

$$\frac{\dfrac{R(\&size) \qquad \forall \delta_1 \in [0, sz[:\ \mathbf{A}(v1 + \delta_1) \wedge \mathbf{W}(v1 + \delta_1) \qquad \forall \delta_2 \in [0, sz[:\ \mathbf{A}(v2 + \delta_2) \wedge \mathbf{R}(v2 + \delta_2)}{\mathit{memcpy}(v1, v2, sz)}}{\forall \delta_3 \in [0, sz[:\ v1[\delta_3] = v2[\delta_3]}$$

Discussion: Ideal Proof-Carrying code?

Global condition: $\exists! p$ such that $SP(p, Inv, Post)$

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

Conclusion

- Proof-Carrying Code is a useful system to prove safety properties of programs
- Yet it should not be confused with an integrity system
- Moreover, PCC is not suited to reason about attacker capabilities
- We give the very first formal definition of Weird Machines as a hybrid concrete symbolic machine
- More examples should be developed (memory corruptions, use-after-free, etc) to refine formalism