0.1 Abstract

Hosts of, for instance, mobile code environments, multi-agent systems or runtime environments bring up special requirements when intending to integrate foreign code into their trusted system: External code producers have to verify that their independently developed code is safe to execute in the host system. Solutions based on certifying specific compilers and code producers by means of cryptography do not fully meet these requirements, as even trusted programmers and compilers are possible error causes when they are integrated into the host architecture.

George C. Necula [necula@cs.cmu.edu] presented proof carrying code, a scalable mechanism supplying code producers with the possibility to proof their code to meet predefined rules the code consumer can validate easily using its own safety rules before integrating it.
## Contents

0.1 Abstract ................................................. 1

1 Introduction 3

1.1 What is proof carrying code? ......................... 3

2 The concept of proof carrying code 3

2.1 General process ....................................... 3

2.2 The safety policy ...................................... 4

2.3 Annotating code ....................................... 5

2.4 Computing the verification condition predicate ...... 7

2.5 The safety proof ....................................... 8

3 Extensions implemented in the certifying compiler 9

3.1 Introduction ........................................... 9

3.2 Achievements .......................................... 10

4 Conclusion and annotations 11

4.1 Proof-carrying code for non-designated purposes ... 11

4.2 Conclusion ............................................. 11

5 References ............................................... 13
1 Introduction

This section provides a compact overview of the concepts and achievements of proof carrying code, which are to be explained in the following section.

1.1 What is proof carrying code?

As the term implies, proof carrying code is capable of proving itself for the code consumer. This is achieved by adding controllable check points into the code, which have, at compile time, to be proved as valid and allowed system states.

As most of the proofs are based on type checking, a foreign program is then expressed as relational mapping of first-order logic objects, that are comfortable to use for further proving purposes.

In general, proof carrying code shifts the complex work of proving a code’s correctness to the producer, simplifying the process of validation the code consumer has to perform.

2 The concept of proof carrying code

2.1 General process

- Host and code producer agree on a set of rules (policy) the foreign code has to fulfill
- The code producer writes annotated code, which uses the policy to proof itself compatible to it by generating a verification condition.
- The verification condition is to be encoded by a proof generator using the rules previously established in the policy
- Both the resulting proof and the annotated code are sent to the host
- The processes of verification condition generation is performed exactly as on producer-side, whereas the proof is checked against the policy instead of generated.
- Upon positive validation, the code is integrated into the host environment
2.2 The safety policy

The safety policy is a technical agreement between code consumer (host) and code producer. Any binary code following it is considered safe to be executed on the host system.

The policy is formally declared using first-order logic rules stating allowed operations as well as valid system states. It is illustrated as ‘axioms and rules’ in figure 1. Basically, the policy can be reduced to typing rules. A valid pair of an integer and a float number, for instance, is represented with the formula

\[
\begin{align*}
\text{m} & \vdash e \colon \text{int} \times \text{float} \\
\text{m} & \vdash e \colon \text{addr} \land m \vdash e+4 : \text{addr} \land m \vdash \text{sel}(m,e) : \text{int} \land m \vdash \text{sel}(m,e+4) : \text{float}
\end{align*}
\]

In general, the safety rules are an unambiguous, logical base for all operations and system states that are allowed by the code consumer. That is to say: A system state is only and exactly then considered valid when it can be proven to be correct using the safety rules only.
As pure native machine code does not suffice to proof a system state or operation, modifications to the foreign code are necessary.

2.3 Annotating code

The crucial work in proving code safety is performed in annotating the foreign code. Developers have to proof stable system states on critical program points. Consider the pseudo assembler code in figure 2, that computes the faculty of the topmost stack element.

Critical program points are, in this case, function and loop entry points. These are the most common situations where invariants have to be used in order to proof a safe system state.

According to Floyd-style verification, a precondition for the function has to be declared, and every loop has to contain at least one invariant for each register used in the loop. Figure 3 shows the same function including annotations.

On program and loop entry points, invariants like

INV <register>, <type>

demand a data type for a register that has to be released by being part of REG, the list of modifiable registers.

It is important to note that invariants can be set at any point in the program,
ENTRY: pop r0 % get function parameter  
mov r0, r2 % r2 = r0

% Function preconditions
INV int, r0 % r0: int ?  
INV int, r2 % r2: int ?  
REG r0, r2 % only allow access on r0, r2  
jge r0, 2, LOOP % r0 >= 2 ?  
mov r0, 1 % fac(0) = fac(1) = 1  
jmp EXIT

LOOP:
% Loop-Invariant
INV int, r0 % r0: int ?  
INV int, r2 % r2: int ?

jle r0, 1, EXIT  
sub r0, 1, r0 % r0--  
mul r0, r2, r2 % r2 *= r0  
jmp LOOP

EXIT:
INV int, r2 % post condition  
push r2 % push back result  
RET

Figure 3: The native machine code including annotations
just like assertions in unit test. Preconditions and loop-invariants represent
the minimum requirements only. A good example is validating variables
retrieved from exception-handled user inputs.

2.4 Computing the verification condition predicate

Using the annotations in the foreign code, it is now possible to compute a
predicate that ensures compliance to the safety policy.

As the verification generation process is based on Floyd-style verification, all
function interfaces (i.e. pre- and postconditions) and invariants have to be
known. The basic principle of VC generation is assigning predicates from the
safety rules to each program instruction.

Any program represented in machine code can be transformed into a vec-
tor, whose indices list all instructions in the same order they appear in the
program. This vector is formally declared as $\Pi$, $\Pi_i$ being the instruction at
program point $i$, i.e. a native machine code or an invariant.

This vector is walked through iteratively by the VC generator. When en-
countering native machine code, it assigns a predefined predicate from the
known set of safety rules. In case of $\Pi_i$ being an invariant, it uses the safety
rules to proof this invariant logically:

The instruction $\text{sub}$ from the prior example would be converted into the
rule:

$$\Pi_i = \text{SUB } r_a, op, r_b \rightarrow VC_i = [r_a - op/r_b]VC_{i+1}$$

The system state resulting from the operation is used as precondition for the
next condition $VC_{i+1}$ consequently. A $\text{JLE}$-instruction is treated likewise,
but takes both cases (jump or not) into consideration:

$$\Pi_i = \text{JLE } r_a, op, n \rightarrow VC_i = (r_a = 0 \implies VC_{i+n+1}) \land (r_a > 0 \implies VC_{i+1})$$

If it is possible to proof an invariant by exclusive usage of rules from the
safety policy, it is considered a safe program state. The resulting verification
condition is a transitive hull from function entry (precondition) to exit point
(postcondition), as one verification predicate $VC_i$ is based on the predicate
$VC_{i-1}$:

$$\text{PRE } = VC_0 \iff VC_1 \iff \cdots \iff VC_{i-1} \iff VC_i \iff \text{POST}$$
Other than the postcondition, the precondition is an index in the vector of invariants $VC$. In conclusion, the verification condition is defined by

$$VC(\Pi, Inv, Post) = \forall r_i \bigwedge_{i \in Inv} Inv_i \mapsto VC_{i+1}$$

(The invariants for every system state, represented by $r_i$, are a true precondition for the next system state.)

The valid verification condition is a predicate that is sufficient to prove the foreign code’s compliance to the safety policy. As illustrated in figure 1, it is computed by the code producer to prove safety, as well as by the code consumer to validate it.

### 2.5 The safety proof

The finally produced safety proof is a derivation of the previously computed verification condition. In last consequence, it is but an encoding for the predicates in vector $VC$, that consists of objects in the Edinburgh Logical Framework (LF). To verify a proof, the safety rules are used as logical axioms to derive every predicate in the proof. As the safety rules are encoded in LF too, a proof checker is easy to implement, as it is based on logical typechecking, where linear algorithms are known for. Fast and easy proof checking is of significance, as - like generating the verification condition - it is also to be performed by the code consumer. As a consequence, all costly operations in the process of proving are executed by the code producer. This is suggestive as producers may implement and use any auxiliary tools to simplify their tasks, while code consumers cannot optimise proof validation by any means.
Extensions implemented in the certifying compiler

3.1 Introduction

The purely manual process of annotating code represents a weak point in the usage of proof carrying code. As George Necula implied, a certifying compiler that automates this task at compile time is cogitable. Necula soon proved this statement by taking part in the development of a certifying compiler for Java (C. Colby, P. Lee, G. Necula et al.) that generates annotated native code from class files containing non-modified java bytecode.

Figure 4 illustrates the resulting difference in the process of proof generation: Instead of adding annotations by hand, the compiler itself generates annotated machine code, that still can be optimised manually - which is, unfortunately, necessary sometimes.

Proof Carrying Code 9
Failed to prove _fac_
At 0x1234
  Under assumptions
  [ ..., 
    (type r0 int) 
    (nonnull r2)]
Could not prove
  ( saferd4( push ( mul ( sub ( r0, 1) r2))))
  Could not prove

Figure 5: Possible output from a failed verification condition generation

3.2 Achievements

The certifying compiler covers significant but, of course, limited fields of Java. The most extraordinary extensions to the typing rules already shown are float numbers and exceptions. Threads are a complex concept that is not solved in the certifying compiler.

Proving safety of exceptions, or otherwise originated control paths, has been solved by verifying the state of exception stacks.

Apart from automated code annotation, the most significant fact pointed out by the compiler developers is debugging code with aid of output generated by a failed proof. The verification condition generator establishes assumptions on the current system state a function modifies, and fails to derive them in case of a typing error. This logical chain, that finally evaluates to 'false', is valuable information to track error causes with a minimum of costly manual analysis.

Figure 5 is the possible result of a failed verification condition predicate.

The failed proof derivation is now visible as logical chain of predicates that allows far more intuitive understanding of unsafe system states than e.g. breakpoint debugging. In the example, an address has not been safe to read, as a push operation on a multiplication result (...) failed.
4 Conclusion and annotations

4.1 Proof-carrying code for non-designated purposes

It is possible, if not foreseeable that, in future progress, the process of proving will become even more natural in the field of software engineering. It could even become sensible to integrate proof-carrying code into standard debugging processes, as failed verification conditions provide sufficient hints to localise many complex syntactical and especially typing-related error causes. It is possible, for example, to establish a set of rules for each library in an application, and thus enforcing coding guidelines. As annotated by George C. Necula, it is left to the code producers to develop respective auxiliary tools.

An approach not annotated by Necula is integrating whole functions into the safety policy. As all primitive operations, such as add, load, jump etc. are encoded as logical objects, every function using them can be stated as logical object itself. That is to say: Trusted functions (like fac in figure 3) can be integrated into the safety policy by encoding them as rule, like:

\[
\begin{align*}
\text{pop } r_d & \mapsto \text{mov } r_a, r_b \mapsto \text{jge } r_a, 2, L1 \mapsto [\ldots] \\
\text{fac } r_a
\end{align*}
\]

As it is evident intuitively, enforcing sequential (not simultaneous, i.e. \( \mapsto \) in place of \( \land \)) validity of rules is not covered by definitions and presented processes of proof-carrying code. The procedure of stating functions as a logical object is already solved in VC generation, however, and adding them to the safety policy - that makes use of the same logical framework - appears feasible.

It is of course not possible to enforce usage of already trusted functions, but software engineers would use them unsolicited and thankfully in order to save development time.

4.2 Conclusion

This paper presented proof carrying code, a mechanism for proving safe behaviour of programs, and a certifying compiler automating the process of proof generation.

The concepts presented Before being able to integrate proof carrying code into all-day development process, a certifying compiler covering the whole
respective programming language has to exist, as hand-optimised machine code is sensible for the most critical components only. As feasibility has been demonstrated with the certifying compiler for Java, it is left to universities and the market to transform the existing approaches into usable development tools.
5 References
