Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 19–34
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Bytecode Analysis for Proof Carrying Code

Martin Wildmoser and Amine Chaieb and Tobias Nipkow

Institut für Informatik,
Technische Universität München

Abstract

Out of annotated programs proof carrying code systems construct and prove verification conditions that guarantee a given safety policy. The annotations may come from various program analyzers and must not be trusted as they need to be verified. A generic verification condition generator can be utilized such that a combination of annotations is verified incrementally. New annotations may be verified by using previously verified ones as trusted facts. We show how results from a trusted type analyzer may be combined with untrusted interval analysis to automatically verify that bytecode programs do not overflow. All trusted components are formalized and verified in Isabelle/HOL.

Keywords: Proof carrying code, bytecode, safety policy, type analyzer, Isabelle/HOL

1 Introduction

Proof carrying code (PCC) enables a code producer to convince a code consumer that a program is safe to execute. It associates to the program a machine-checkable proof of its safety. The code consumer checks this proof against a formula that guarantees safety of the program at hand. This formula is constructed individually for each program by a verification condition generator (VCG). In the earlier PCC systems [10] the VCG is a highly engineered component that tries to produce proof obligations that are easily prove- and checkable. This comes at the price of having a complex component that is difficult to understand and trust. A bug in the VCG can lead to provable verification conditions for unsafe programs. Foundational proof carrying code [3] remedies this by requiring the verification condition to directly express the desired safety property in terms of the machine semantics. A VCG in form

doi:10.1016/j.entcs.2005.02.040

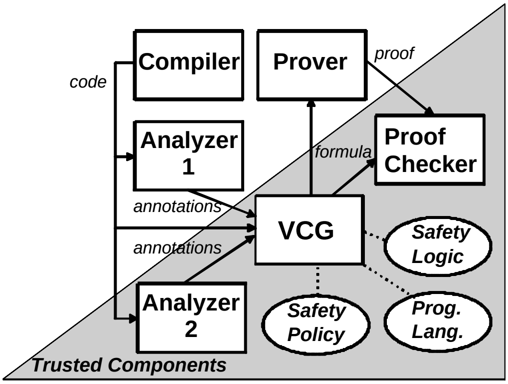

Fig. 1. PCC system architecture

of a proof tactic may still be applied. In this case all the formula transformations it does are inside the logic and must be verified. This typically leads to big proof objects. In addition verification conditions directly expressing safety require higher order logics, which hampers automatic proof generation. For some safety policies, such as type safety, correct instruction decoding, program counter in range etc., this is yet feasible as proofs can be extracted from sophisticated type systems [9]. A promising compromise between purely foundational and classical PCC is the Open Verifier [6] architecture. While verifying a powerful VCG as in [10] is a daunting task, it is possible to use a minimal VCG as trusted core. It delegates more complicated transformations to untrusted extensions that are obliged to certify their work. In [17] we follow a similar idea, except that in our case extensions are verified and thus trusted. We formalize a parameterized framework for a generic VCG in the theorem prover Isabelle/HOL. The framework defines signatures and specifies requirements for functions used as parameters by the generic VCG. By defining these parameters to a particular programming language, safety policy and safety logic one obtains an instantiated VCG. If one verifies that the instantiated parameters meet the framework's requirements the correctness and soundness proof for the generic VCG automatically carry over to the instantiated one. In this paper we will instantiate a VCG that checks for arithmetic overflow in Jinja bytecode programs. Jinja [8] is a downsized version of Java featuring object creation, inheritance, dynamic method calls and exception handling. As safety logic we use a first order expression language [16] deeply embedded into Isabelle/HOL.

A general problem in program verification is to find invariants for loops. Our VCG assumes that these are given in form of annotations in the code. It is the task of the code producer to find them. For many safety properties inductive invariants can be generated automatically using static program analysis. The main contribution of this paper is to show how annotations can be gained from interval analysis and type inferences done by a Java bytecode verifier. In our previous papers annotations had to be added manually by the code producer. Since one has to verify annotations as part of the verification condition the analyzers must not be trusted. Any external tool whose results can be transformed to annotations in our first order expression language may be used. However, sometimes one knows that some analyzer can be trusted. For example in the Jinja formalization we have a completely verified bytecode verifier. Given a bytecode program it infers the types of registers and stack elements for all program positions. In this paper we will show how one can instantiate the generic VCG such that it combines trusted and untrusted information. Our aim are verification conditions that oblige us to verify untrusted parts and allow us to assume trusted facts. We call such verification conditions incremental as they only have some value, if their assumptions hold for a particular program. To check this, one has several possibilities. First, the assumptions may arise from a trusted and efficient analyzer. In this case the client can run this analyzer and hand the results over to the VCG as trusted parts. Second, we may use an analyzer that emits a certificate for each result. This is generally just another instance of PCC and could be handled by giving the analyzer's results directly to the VCG as untrusted facts. Instead of having one big VCG that combines the results of various untrusted sources, we can employ multiple ones. In this case we can prove the results of different analyzers in isolation.

2 PCC Framework

Our generic VCG constructs formulas out of annotated control flow graphs. The control flow of a program Π is determined by a parameter function *succsF* which yields for each position p a list of successor positions paired with branch conditions. The intuition is that $(p',B) \in \mathit{set}\ (\mathit{succsF}\ \Pi\ p)$ means that p' is directly accessible from p in situations where B holds. Formally, we write $\Pi,(p,m) \models B$ to say that a state (p,m), where p is the program counter and m the memory, satisfies some formula B. The safety policy is expected in form of a function *sF* from programs and positions to formulas, such that $\mathit{sF}\ \Pi\ p$ expresses the safety condition we expect for program Π when we reach p at runtime. Analogously we write $\mathit{aF}\ \Pi\ p$ for the annotation at position p. For each node p with successor p' with branch condition B the VCG constructs this proof obligation:

$$(\mathit{sF}\ \Pi\ p \mathbin{\llcorner\!\wedge\!\lrcorner} \mathit{aF}\ \Pi\ p \mathbin{\llcorner\!\wedge\!\lrcorner} B) \mathbin{\llcorner\!\Rightarrow\!\lrcorner} \mathit{wpF}\ \Pi\ p\ p'\ (\mathit{sF}\ \Pi\ p' \mathbin{\llcorner\!\wedge\!\lrcorner} \mathit{aF}\ \Pi\ p')$$

The safety formula $\mathit{sF}\ \Pi\ p$ conjoined with annotation and branch condition must imply the (weakest) precondition for the transition from p to p' and a postcondition consisting of the annotation and safety formula at p'. Note that the marked connectives $\llcorner\!\wedge\!\lrcorner$ and $\llcorner\!\Rightarrow\!\lrcorner$ are part of the safety logic, whereas $\wedge$ and $\longrightarrow$ are Isabelle/HOL's logical connectives. In addition to these proof obligations the VCG emits a safety constraint for initial states. States that satisfy the formula *initF* Π must satisfy the safety formula and annotation of the initial position *ipc* Π.

$$\mathit{initF}\ \Pi\ \llcorner\!\Rightarrow\!\lrcorner\ (\mathit{sF}\ \Pi\ (\mathit{ipc}\ \Pi)\ \llcorner\!\wedge\!\lrcorner\ \mathit{aF}\ \Pi\ (\mathit{ipc}\ \Pi))$$

Our framework requires that *initF* Π is instantiated such that it covers all initial states *initS* Π of a program Π. Note that *initF* Π yields a formula in the safety logic and may abstract properties of concrete initial states modeled by *initS*. The letters F and S in function names indicate whether they belong to the formula or the semantical level. Apart from *initS* we use a state transition relation *effS* Π to model the semantics of a program. We say a program is safe with respect to some safety policy *sF*, i.e. *isSafe sF* Π, if and only if all reachable states satisfy the safety policy.

$$\mathit{isSafe}\ \mathit{sF}\ \Pi = \mathit{inv}\ \Pi\ \mathit{sF}$$
where $\mathit{inv}\ \Pi\ I = \forall\,(p,m) \in \mathit{ReachableS}\ \Pi.\ \Pi,(p,m) \models\ I\ \Pi\ p$
$$\mathit{ReachableS}\ \Pi = \{(p,m) \mid \exists\,(p_0,m_0) \in \mathit{initS}\ \Pi.\ ((p_0,m_0),(p,m)) \in (\mathit{effS}\ \Pi)^*\ \}$$

The correctness theorem we have proven in Isabelle/HOL guarantees that a program Π is safe and has valid annotations if one can prove the generic verification condition for it.

theorem *correct-vcg*:
$(\mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF} \wedge \mathit{wf}\ \Pi$
$\wedge\ \Pi \vdash \mathit{vcg}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF}\ \Pi)$
$\longrightarrow (\mathit{inv}\ \Pi\ \mathit{sF} \wedge\ \mathit{inv}\ \Pi\ \mathit{aF})$

The predicate *VCGReqs* ... *wf initF succsF wpF aF sF* indicates that the parameter functions meet the requirements our framework demands. With ... we indicate that some parameters, such as $\llcorner\!\wedge\!\lrcorner$, $\llcorner\!\Rightarrow\!\lrcorner$, *initS* and *effS* are suppressed for better readability. With *vcg initF succsF sF aF* we denote an instance of the generic VCG to particular setting of parameters. The wellformedness predicate *wf* is meant to check simple syntactic constraints on programs that help to simplify the definition of other parameter functions. The PCC framework only demands that it checks for sufficient annotations in programs. Our generic VCG requires at least one annotation per cycle in the control flow graph. In this case it constructs formulas for non annotated positions by pulling back the annotations of successor positions using *wpF*. Details can be found in [17]. Additional checks can be added to *wf*, but one has to keep in mind that a code consumer is supposed to evaluate *wf* efficiently. We do not go into the details of the requirements *VCGReqs* poses onto the parameters. These can be found in [17] and in the Isabelle theories [2]. Most requirements are easy to prove for a particular instantiation. The hardest part is usually to verify that *succsF* and *wpF* are an abstraction of the concrete semantics defined by *effS* and *initS*. That is, if the semantics *effS* Π allows a transition from (p,m) to (p',m') then p' must be among the successors statically predicted by *succsF* Π p and the corresponding branch condition must hold for (p,m). In addition if (p,m) satisfies the precondition for some postcondition Q, i.e. $\Pi,(p,m) \models \mathit{wpF}\ \Pi\ p\ p'\ Q$, then Q must be satisfied by (p',m'). If the resulting VCG shall be complete, that is safe programs yield provable verification conditions, stronger requirements are necessary. The *wpF* operator must then yield weakest(!) preconditions and *succsF* must give exact control flow information. In our instantiation to Jinja, we have proven both correctness and completeness of the VCG.

3 Incremental Verification

Once the requirements are proven for a particular setting of parameters, some of these parameters can be replaced by extended versions, without having to redo the proofs. A safety policy *sF* may be replaced by another safety policy *sF'* as no requirements depend on it.

lemma $\mathit{replace}{-}\mathit{sF}$:
$\mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF}$
$\longrightarrow\ \mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}'\ \mathit{aF}$

In addition one can always replace the wellformedness checker *wf* with a stronger version *wf'*.

lemma $\mathit{upgrade}{-}\mathit{wf}$:
$(\forall\ \Pi.\ \mathit{wf}'\ \Pi \longrightarrow \mathit{wf}\ \Pi) \wedge \mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF}$
$\longrightarrow\ \mathit{VCGReqs}\ \ldots\ \mathit{wf}'\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF}$

Invariants can be used to upgrade successor functions. For this purpose we use a higher order functor *upg*, which upgrades a given successor function *succsF* by conjoining an invariant I with its branch conditions.

$\mathit{succsF}\ \Pi\ p = [(p_1,B_1),\ldots,(p_k,B_k)] \longrightarrow$
$(\mathit{upg}\ I\ \mathit{succsF})\ p = [(p_1,B_1\ \llcorner\!\bigwedge\!\lrcorner\ I\ p),\ \ldots,\ (p_k,B_k\ \llcorner\!\bigwedge\!\lrcorner\ I\ p)]$

lemma $\mathit{upgrade}{-}\mathit{succsF}$:
$((\mathit{wf}\ \Pi \longrightarrow \mathit{inv}\ \Pi\ I) \wedge\ \mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF})$
$\longrightarrow\ \mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ (\mathit{upg}\ I\ \mathit{succsF})\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF}$

Now imagine we have two analyzers for some programming language and we want to use both to verify some safety policy sF. Let $aF_1\ \Pi$ and $aF_2\ \Pi$ be the annotations we gain from both analyzers.

A simple solution is to instantiate a PCC system with the conjunction of both annotation sets. We take $aF_{12} = \lambda\ \Pi\ p.\ aF_1\ \Pi\ p \llcorner\bigwedge\lrcorner\ aF_2\ \Pi\ p$ and prove the framework's requirements $\mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ aF_{12}$. Then the code producer is obliged to give us a proof of $\Pi \vdash \mathit{vcg}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ \mathit{sF}\ aF_{12}\ \Pi$, which would guarantee $\mathit{inv}\ \Pi\ aF_{12}$ as well as $\mathit{inv}\ \Pi\ sF$. This approach is acceptable if neither analyzer is trustworthy. Otherwise, we repeat work as both annotation sets have to be proven.

Alternatively, we can instantiate two VCGs and do the verification incrementally. First we instantiate a VCG for a trivial safety policy that just enforces that the annotations $aF_1\ \Pi$ hold. We show the requirements $\mathit{VCGReqs}\ \ldots\ \mathit{wf}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ (\lambda\ \Pi\ p.\ \llcorner \mathit{True} \lrcorner)\ aF_1$ **(1)** and ask the producer to prove $\Pi \vdash \mathit{vcg}\ \mathit{initF}\ \mathit{succsF}\ \mathit{wpF}\ (\lambda\ \Pi\ p.\ \llcorner \mathit{True} \lrcorner)\ aF_1\ \Pi$ **(2)**. For wellformed programs this gives us $\mathit{inv}\ \Pi\ aF_1$, and enables us to apply lemmas $\mathit{upgrade}{-}\mathit{wf}$ and $\mathit{upgrade}{-}\mathit{succsF}$ to obtain another instantiation where $\mathit{VCGReqs}\ \mathit{wf}'\ \mathit{initF}\ (\mathit{upg}\ aF_1\ \mathit{succsF})\ \mathit{wpF}\ \mathit{sF}\ aF_2$ **(3)** holds. With $\mathit{wf}'\ \Pi$ we employ a stronger wellformedness checker. It guarantees $\mathit{wf}\ \Pi$ and ensures $\mathit{inv}\ \Pi\ aF_1$ by checking a proof for **(2)**. If wf' accepts the program the producer is asked to send a proof for $\Pi \vdash \mathit{vcg}\ \mathit{initF}\ (\mathit{upg}\ aF_1\ \mathit{succsF})\ \mathit{wpF}\ \mathit{sF}\ aF_2\ \Pi$ **(4)**, which guarantees the safety policy we are actually interested in, i.e. $\mathit{inv}\ \Pi\ sF$, as well as $\mathit{inv}\ \Pi\ aF_2$. This approach is superior to the first, if the second proof can make use of $aF_1\ \Pi$, which now only appears on the left hand side of implications.

If $aF_1\ \Pi$ comes from a trusted source, we have a third alternative. In this case we get $\mathit{inv}\ \Pi\ aF_1$ either by assumption or because we have a correctness proof of the analyzer, that is a proof of $\forall\ \Pi.\ \mathit{wf}'\ \Pi \longrightarrow \mathit{inv}\ \Pi\ aF$. In both cases the code producer can skip **(2)** and only has to prove **(4)**. In this paper we will apply the second option for the Jinja bytecode verifier, which is proven correct in [8]. The types it infers for registers and stack elements can be trusted and must not be verified again inside verification conditions. Other example for trusted annotations are system invariants. These are properties maintained by the Jinja Virtual Machine for any wellformed program. For example, the Jinja virtual machine preallocates objects for system exceptions, e.g. *OutOfMemory*, and always keeps these objects at specific positions in the heap. This can be expressed as a safety logic formula and yields a system invariant.

4 PCC for Jinja Bytecode

Jinja bytecode is a down-sized version of Java bytecode and covers essential object oriented features: object creation, inheritance, dynamic method calls and exceptions. In this paper, we only use the basic arithmetic and move instructions. Our intervall analysis is not (yet) powerful enough to handle object references and method calls. Programs with these features can be verified in our PCC system, but require manual annotations. Fig. 2 shows a method that adds all numbers from 1 to n. The two safety checks prevent arithmetic overflow and enable our interval analyzer to generate a linear inductive invariant.

Each method is executed in its own frame, which contains a program counter, registers and an operand stack. Apart from that Jinja states contain a heap mapping references to objects. Program positions are triples (C,M,pc), where *pc* is the number of the current instruction in method *M* of class *C*. Registers and stack contain booleans, e.g. *Bool True*, integers, e.g. *Intg 5*, references, e.g. *Addr 3*, null pointers, e.g. *Null* or dummy elements, e.g. *Unit*.

datatype *val* = *Bool bool* | *Intg int* | *Addr addr* | *Null* | *Unit*

Each value has a type associated with it:

datatype *ty* = *Boolean* | *Integer* | *Class cname* | *NT* | *Void*

Jinja programs are started in position $(Start,main,0)$, with one method frame that contains an empty operand stack and uninitialized registers. The semantics of a Jinja program Π is defined by *initS* Π, the set of its intial states, and *effS* Π, a relation mapping each state to its direct successors. We skip the formal definitions, as the instructions used here are simple enough: `Push` *val* pushes a constant value onto the operand stack. `Store` *n* removes the topmost stack value and stores it in register *n*. `Load` *n* copies the content of register *n* onto the stack. `IfIntLeq` *t* jumps *t* instructions forward (backward if *t* is negative) if the upper stack value is less than or equal to the lower one and erases both. `IAdd` pops the two upper values and replaces them with their sum. Finally, `Goto` *t* jumps *t* instructions forward and `Return` terminates a method handing the topmost stack element over to the caller. To write annotations and to specify a safety policy we have to instantiate a safety logic. An easy way to do so would be to use the logic of the theorem prover Isabelle/HOL as safety logic by defining formulas semantically as predicates on Jinja states. We have used this shallow embedding technique in other instantiations [15], but decided to use a deeply embedded first order expression language here. A shallow embedding eases the definition and verification of a *wpF* function significantly, but makes the definition of a precise control flow function

```
    Load n                          push input
    Push Intg 65536                 push bound
    IfIntLeq 20                     safety check
    Push Intg 0                     initialize Rg k
    Store k
    Push Intg 0                     initialize Rg r
    Store r
lp: Load k
    Load n
    IfIntLeq 13                     if n ≤ k goto ex;
    Load r                          else {
    Push Intg 2147385346
    IfIntLeq 10                     safety check
    Load k
    Push Intg 1
    IAdd
    Store k                         k := k+1
    Load k
    Load r
    IAdd
    Store r                         r := r+k
    Goto −14                        goto lp: }
ex: Load r
    Return
```

Fig. 2. Example Program

harder. In the deep embedding *succsF* can extract information, such as types or jump targets, from structural analysis of annotations. This helps to limit the branches *succsF* yields for dynamic method invocations and exceptions. Moreover, the limitation to first order simplifies the proofs of the resulting verification conditions and allows us to use other tools than Isabelle/HOL for this purpose. The expression language we use is designed to adequately model Jinja states and can express weakest preconditions for all Jinja instructions [16]. In this paper we only need a small fragment of it:

$$\textbf{datatype}\ expr\ =\ Cn\ val\ \mid\ Rg\ nat\ \mid\ St\ nat\ \mid\ Ty\ expr\ ty\ \mid\ expr\ op\ expr\ \mid\ \ldots$$
$$op\ =\ \llcorner{+}\lrcorner \mid \llcorner{-}\lrcorner \mid \llcorner{*}\lrcorner \mid \llcorner{=}\lrcorner \mid \llcorner{\leq}\lrcorner \mid \llcorner{\bigwedge}\lrcorner \mid \llcorner{\Rightarrow}\lrcorner$$

We have expressions for constants, register and stack access, type checking as well as arithmetic and logical operations. When evaluated on a Jinja state all these expressions yield a Jinja value. The formal definition of $eval{::}prog \Rightarrow state \Rightarrow expr \Rightarrow val$ is in [16]. Here it is only important to know that expressions are considered valid formulas if they evaluate to *Bool True*.

$$\Pi{,}s \models F \equiv (eval\ \Pi\ s\ F = Bool\ True)$$

To define the provability judgment $\vdash$ one usually gives a calculus of introduction and elimination rules. However, if we want to do the proofs in Isabelle/HOL we can also define provability semantically.

$$\Pi \vdash F \equiv \forall\ s{\in}\ ReachableS\ \Pi.\ \Pi{,}s \models F$$

We decided for this option, as it trivializes the correctness and completeness requirement for the safety logic. Formulas are considered provable if we can prove in Isabelle/HOL that they hold for all reachable states. Using these expressions, we instantiate a safety policy that prohibits arithmetic overflow. We simply check that each addition yields a result that is below *MAXINT*, which is 2147483647 for Java.

$$\mathit{safeF}\ \Pi\ p = \mathit{if}\ (\mathit{cmd}\ \Pi\ p = \mathit{IAdd})\ \mathit{then}\ \mathit{St}\ \mathit{0} \llcorner\!+\!\lrcorner\ \mathit{St}\ \mathit{1} \llcorner\!\leqslant\!\lrcorner\ \mathit{MAXINT}\ \mathit{else}\ \llcorner \mathit{True} \lrcorner$$

The instantiation work lies in defining the successor function *succsF* and a weakest precondition operator *wpF* and proving that they meet the requirements the framework poses on them. The first order expression language is designed to be expressive enough to define a complete *wpF*, that is a function that exactly reflects the semantics *effS* at the level of formulas. The *wpF* operation is defined via substitutions on formulas [16]. We do not show its definition as the principle it follows is straightforward. Whenever an instruction changes the value of some subexpression, we substitute it with an expression that compensates this effect. For example in case of *IAdd* we substitute each occurrence of *St 0* in the postcondition with $\mathit{St}\ \mathit{0} \llcorner\!+\!\lrcorner\ \mathit{St}\ \mathit{1}$. All the requirements for *wpF* follow easily from this lemma we have been able to prove:

$$\mathit{wf}\ \Pi \wedge (p,m) \in \mathit{ReachableS}\ \Pi \wedge ((p,m),(p',m')) \in (\mathit{effS}\ \Pi)$$
$$\longrightarrow \mathit{eval}\ \Pi\ (p,m)\ (\mathit{wpF}\ \Pi\ p\ p'\ Q) = \mathit{eval}\ \Pi\ (p',m')\ Q$$

It says that evaluating the postcondition in the successor state amounts to evaluating the weakest precondition in the current state.

5 Program analysis

Since we are interested in supporting verification in different safety policies we use a general purpose static analyzer. It is an implementation of a generic abstract interpretation framework as presented in [7]. It transforms programs into a control flow graph representation, where the edge labels reflect the original commands and conditions . These labels have to be accurately interpreted by each abstract domain implementation and give raise to the abstract semantics in form of abstract transfer functions of type $D \rightarrow D$, D is an abstract domain. Replacing the labels in the control flow graph by the corresponding abstract transfer functions leads to the representation of the abstract semantics equation system. Any adequate iterative solver can solve it and provide a post-fixpoint as a safe approximation for it, provided the convergence is guaranteed.

Load i	$\mathsf{st}\ 0 := \mathsf{rg}\ i$
Push v	$\mathsf{st}\ 0 := v$
IfFalse n	$\mathsf{st}\ 0 = \mathbf{ff}$; $\mathsf{st}\ 0 = \mathbf{tt}$
Pop	*Nop ”pop”*

Fig. 3. Encoding bytecode instructions in labels

5.1 Adjustment of the analyzer to bytecode

The main challenge in analyzing bytecode programs, is that the instructions have implicit parameters given by the functionality for the virtual machine. Furthermore stack manipulations blur the dependency of the effects in terms of the registers. In contrast to other methods [13] we do not translate the bytecode program into an intermediate language, but implement domains that encode an abstraction of the virtual machine. This decision was due to our bad experience with an intermediate language. The only drawback we have to put up with is that the resulting implementations are only useful for bytecode-like languages.

The first step we have to mention is the encoding of the bytecode instructions into the labeling language. Some encodings are given in Fig.3. Any abstract domain implementation has to interpret the labels accurately in the sense of this transformation. For example when interpreting Push v, encoded by $\mathsf{st}\ 0 := v$, the abstract transfer function should take into account that the abstraction of the stack has to mimic the behavior of the stack on the virtual machine. A label $x := e$ can encode any assignment of the expression e to the variable x. But interpreting an assignment may differ from domain to an other. For example interpreting $\mathsf{st}\ 0 := v$ for a bytecode language dose not simply consist in storing the value v in the variable $\mathsf{st}\ 0$ but should rather reflect the fact that the stack elements move down. The main reason for this is that we leave the labeling language common to all domain implementations and use it to encode different imperative or bytecode languages. For readability we note variables and expressions in a near notation to the JVM, where $\mathsf{st}\ i$ and $\mathsf{rg}\ j$ refer to the i^{th} stack position and to the j^{th} register. Conditional jumps must have two successors, where the condition evaluates to $\mathbf{tt}$ or to $\mathbf{ff}$. This is in Fig.3 reflected by the two labels assigned to the successor edges in the case of IfFalse n.

$$e_s ::= x \in \mathit{regs} | v \in \mathbb{V} | e_s \ \mathit{bin} \ e_s | \mathit{un} \ e_s$$

Fig. 4. Stack expressions

$\mathsf{rg}\ i := \mathsf{st}\ 0$	$\lambda(a, e :: es).(a[i \mapsto \mathit{eval}\ a\ e], \mathit{map}(\lambda e.e[a[i]/\mathsf{rg}\ i])\ es)$
$\mathsf{st}\ 0 := v$	$\lambda(a, es).(a, e_v :: es)$
$\mathsf{st}\ 0 := \mathsf{st}\ 0 + \mathsf{st}\ 1$	$\lambda(a, e_1 :: e_2 :: es).(a, (e_1 + e_2) :: es)$

Fig. 5. Some transfer functions for :=

5.2 Interval analysis for the bytecode

We implemented interval analysis as a result of lifting the lattice of intervals, a special case of some values lattice $\mathbb{V}$, to the lattice $\mathcal{V} \to \mathbb{V}$, where $\mathcal{V}$ represents the set of variables we are interested in. This transformation is done by a functor which has minimal assumptions on the implementation of the lattice of values, whence the techniques we present are applicable to all such liftings. Lattices for values have to approximate values of the programming language by abstract values ($\mathbb{V}$). This is ensured by a function $\alpha : \mathit{val} \to \mathbb{V}$. They also have to provide an arithmetic and a logic in order to reflect the operations $+, *, -, \leq, >, =,$ etc . . . . For intervals we use a standard arithmetic, widening and narrowing as described in [11]. We abstract numerical constants c by the interval $[c, c]$. We used the intervals $[0, 0], [1, 1]$ and $[0, 1]$ to represent **ff**, **tt**, and the unknown truth value.

Dealing with the stack

The set of variables $\mathcal{V}$ we are interested in is composed by the registers *regs* and the stack elements *svars* i.e. $\mathcal{V} = \mathit{regs} \uplus \mathit{svars}$. The key to a precise bytecode analysis, is to keep track of what the values on the stack stand for. This is of great use in the exploitation of the branching conditions, since these are usually expressed in terms of stack elements. For this purpose we associate an expression to each stack position, which is a safe description of the associated value in terms of the registers. The syntax of stack expressions is given in Fig.4. Since the shape of the stack changes along the program, we represent it by a list. Hence the lattice of our insensitive liftings, and thus also interval analysis, is $\mathcal{LV} = (\mathit{regs} \to \mathbb{V}) \times (\mathit{sexpr\ list})$.

Transfer functions

The transfer functions are strict in the sense if an expression evaluates to $\bot_{\mathbb{V}}$, the next abstract state is unreachable and hence will be $\bot_{\mathcal{LV}}$. Since we analyze

```
$bcond\ (\mathsf{st}\ 0 = \mathbf{ff})\ (a, e :: es)\ =$
case $e$ of
  $\mathsf{rg}\ i\ bin\ \mathsf{rg}\ j \Rightarrow$
  let
    $v_i = a[i] \sqcap_{\mathbb{V}}\ varncond\ bin\ a[j]$
    $v_j = a[j] \sqcap_{\mathbb{V}}\ varncond\ \overline{bin}\ a[i]$
  in $(a[i \mapsto v_i; j \mapsto v_j], es)$
  $\mathsf{rg}\ i\ bin\ e' \Rightarrow (a[i \mapsto\ varncond\ bin\ (eval\ a\ e')], es)$
  $\neg e' \Rightarrow\ bcond\ (\mathsf{st}\ 0 = \mathbf{tt})(a, e' :: es)$
  $\_ \Rightarrow (a, es)$
```

Fig. 6. Treatment of one case of branching conditions

only well formed programs, the existence and the type correctness of the stack elements at different program positions is guaranteed by the bytecode verifier. Some of the transfer functions for the “assignments” are listed in Fig. 5. The interesting case is when the label corresponds to a Store i instruction. The associated abstract value to register i will change and hence for consistency all stack expressions referring to register i have to be updated. Replacing every occurrence of $\mathsf{rg}\ i$ in the expressions on the stack by the actual value v_i of register i updates these accurately. In the second line of Fig. 5 we associate to a value in the labeling language v a *sexpr* expression $e_v = \alpha(v)$. With $eval\ a\ e$ we perform a bottom-up evaluation of the expression e in the abstract state a by simply looking up variables in a, abstracting constants by α, and otherwise recurcively applying the operations defined by $\mathbb{V}$. The transfer functions for the branching conditions use the expressions on the stack in order to restrict the next abstract state. We consider the case where the value on top of the stack is supposed to be false in the next states. If the approximation of this value is $\alpha(\mathbf{tt})$, then the next state will be unreachable, otherwise we restrict the next state according to the function *bcond* shown in Fig.6. The interesting case is when the boolean condition is of the form $x\ bin\ b$, where $x \in regs$. In this case the lifting functor updates the abstract value v_x associated to x by $v_x \sqcap_{\mathbb{V}} varncond\ bin\ v_b$, where v_b is the abstract value the expression b evaluates to, and *varncond* is a function specific to each domain of values and gives a best way to restrict an abstract value given a binary operator and the value of the right hand side, where the binary condition is supposed not to hold. Domains for values have also to provide a function *varcond* for the analogous scenario, but where the binary condition is supposed to hold. $\overline{bin}$ is a binary operator such that $x\ bin\ y \Leftrightarrow y\overline{bin}\ x$ holds. The other cases are similar.

Finally, every domain implementation has to make a function *toAnnotation* available, which transforms a domain element to a formula in the safety logic used for annotations. It implements the concretization function γ.

6 Bytecode Verification

The Jinja Bytecode Verifier infers the types for register and operand stack values. Encoded into our safety logic these facts become expressions of the form $\mathit{Ty}\ (\mathit{Rg}\ 0)\ (\mathit{Class}\ \mathit{Start})\ \mathbin{\llcorner\!\bigwedge\!\lrcorner}\ \ldots\ \mathbin{\llcorner\!\bigwedge\!\lrcorner}\ \mathit{Ty}\ (\mathit{St}\ 0)\ \mathit{Integer}\ \mathbin{\llcorner\!\bigwedge\!\lrcorner}\ \ldots$. Using the type safety proof of [8] we can easily derive the following theorem, which allows us to trust the Bytecode Verifier's result.

$$\mathit{wtP}\ (\mathit{bcv}\ \Pi)\ \Pi \longrightarrow \mathit{inv}\ \Pi\ (\mathit{conv\text{-}Ty}\ (\mathit{bcv}\ \Pi))$$

When the Jinja type checker *wtP* accepts the types $\mathit{bcv}\ \Pi$, inferred by the bytecode verifier, as a welltyping for a program Π, then we can convert these types to formulas that hold at runtime. Following the schema from §3 we upgrade *succsF* with these type constraints and strengthen the wellformedness checker with $\mathit{wtP}\ (\mathit{bcv}\ \Pi)$. That is, we verify $\mathit{VCGReqs}\ \ldots\ (\lambda\ \Pi.\ \mathit{wf}\ \Pi \wedge \mathit{wtP}\ (\mathit{bcv}\ \Pi))\ \mathit{initF}\ (\mathit{upg}\ (\lambda\ \Pi.\ \mathit{conv\text{-}Ty}\ (\mathit{bcv}\ \Pi))\ \mathit{succsF})\ \mathit{wpF}\ \mathit{sF}\ \mathit{aF}$, which gives us a correct VCG for Jinja.

7 System Overview

7.1 Generating Verification Conditions

Fig. 7 shows the process of synthesizing the verification condition, which is common to both consumer and producer. Java classfiles are converted into Jinja ML representation by J2ML, a tool we implemented using Perl and Jissa[1]. Jissa decodes binary classfiles to a textual assembly notation, which Perl then transforms to a ML representation of Jinja bytecode. In ML we have the interval analyzer as well as executable versions for the VCG and the BCV available. The latter two are automatically generated from their Isabelle/HOL formalizations, which we have verified. First, we run the interval analysis to obtain invariants for loops. These are then given to the VCG together with the program code. We have instantiated the VCG with a successor function that internally uses the Bytecode Verifier to construct branch conditions with type information for registers and stack. These are helpful for proof construction as many of our Jinja specific proof rules have type constraints. To construct proofs we consider two different techniques: First one can employ a theorem prover. Although other first order provers with proof objects and sufficient support for arithmetics could be employed, we rather tend to use Isabelle/HOL for this purpose as well. It may not be the best prover as it is built for interactive verification, but it enables us to define the provability judgment semantically. The second technique is to use a proof producing program analysis.

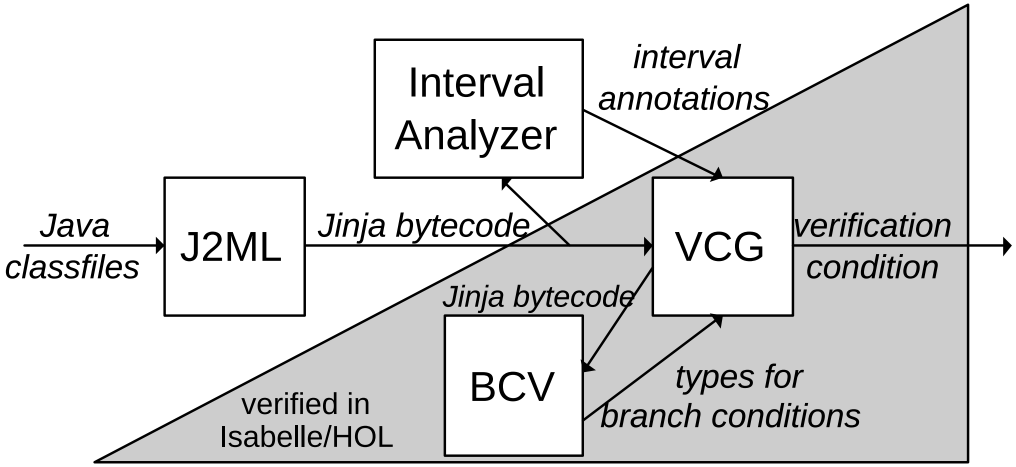

Fig. 7. Generation of Verification Conditions for Jinja

7.2 Proof Construction with Isabelle

For the entry point of the loop (*lp*:) in Fig. 2 the interval analysis gives us this expression:

$$\mathit{Cn}\ (\mathit{Intg}\ 0) \llcorner\leqslant\lrcorner \mathit{Rg}\ r \llcorner\wedge\lrcorner \mathit{Rg}\ r \llcorner\leqslant\lrcorner \mathit{Cn}\ (\mathit{Intg}\ 2147385346) \llcorner\wedge\lrcorner \mathit{Cn}\ (\mathit{Intg}\ 0) \llcorner\leqslant\lrcorner \mathit{Rg}\ k \llcorner\wedge\lrcorner \mathit{Rg}\ k \llcorner\leqslant\lrcorner \mathit{Cn}\ (\mathit{Intg}\ 65535) \llcorner\wedge\lrcorner \mathit{Rg}\ n \llcorner\leqslant\lrcorner \mathit{Cn}\ (\mathit{Intg}\ 65535)$$

If we use this annotation, we obtain a verification condition that is automatically provable by Isabelle's simplifier. The simplifier automatically translates expressions in our first order language into HOL propositions using rewriting rules for *eval*. For example, here is a rule that turns $\llcorner+\lrcorner$ into Isabelle's $+$.

$$\mathit{eval}\ \Pi\ s\ (\mathit{Ty}\ e\ \mathit{Integer}) = \mathit{Bool\ True} \wedge \mathit{eval}\ \Pi\ s\ (\mathit{Ty}\ e'\ \mathit{Integer}) = \mathit{Bool\ True} \longrightarrow \mathit{eval}\ \Pi\ s\ (e \llcorner+\lrcorner e') = \mathit{the\text{-}Intg}\ (\mathit{eval}\ \Pi\ s\ e) + \mathit{the\text{-}Intg}\ (\mathit{eval}\ \Pi\ s\ e')$$

Note that this rule has type constraints as side conditions. These can automatically be discharged with the type information put into branch conditions by the BCV. After simplification the resulting HOL formulas can be given to Isabelle's decision procedures. For the example above linear arithmetics suffices. If we remove the inner safety check from our example program, interval analysis cannot find an inductive invariant any more, because it does not know the relationship between $\mathit{Rg}\ k$ and $\mathit{Rg}\ n$, i.e. $2 \llcorner*\lrcorner \mathit{Rg}\ n = (\mathit{Rg}\ k) \llcorner*\lrcorner (\mathit{Rg}\ k \llcorner+\lrcorner \mathit{Cn}\ \mathit{Intg}\ 1)$. However, if we manually add this relationship as invariant, the program can be verified automatically, but now requires a fragment of bounded arithmetics to find an upper bound for multiplication expressions.

7.3 Proof producing program analysis

After the analyzer reaches a post-fixpoint, it should return a proof that the found annotations are correct, i.e. form valid Hoare triples. This would avoid reverifying the annotations as in the former case, which needs the presence of special decision procedures, e.g. bounded arithmetic if the programming language admits multiplication. We developed this technique independently from [12] and have almost finished implementing it for interval analysis. More details in [5]. This technique is attractive due to the short proofs we get. Intuitively, since the analyzer infers these invariants, their correctness proofs will rely on a small set of theorems, which express that the analyzer manipulates domain elements correctly. Proof producing analysis gives raise to sound yet incomplete efficient proof-producing decision procedures, which may be helpful to verify the analyzed programs.

7.4 Proof Checking

Isabelle can record proof objects in LF style [4]. These can be transmitted to the consumer side. There Isabelle's proof checker finds out whether the received proof fits the verification condition, which is computed as Fig. 7 shows. The proof checker is a relatively small and simple ML program, and can thus be trusted without a machine checked proof.

8 Conclusion

Our generic VCG can be instantiated such that it makes use of trusted and untrusted analysis results. It can be instrumented such that only the first need to be verified. In this paper we employ an untrusted interval analyzer and a trusted type analyzer (BCV) to verify automatically that Jinja programs do not overflow. This is a major improvement to our previous work [17] [15] [16], where we annotated programs manually. Our formalization of Jinja and PCC is about 51k lines of Isabelle/HOL. It demonstrates that it is feasible to have a fully verified trused code base for a PCC system. In the future we try to extend on this work in various angles. More advanced program analysis techniques are planned to be implemented. Jinja's bytecode is going to be extended towards the Verificard bytecode [14], which now almost covers the entire Java bytecode. A downside of our generic verification conditions is that they contain a lot of redundant and irrelevant information. We plan to improve this by using optimizing postprocessing functions. In the Open Verifier [6], untrusted extensions are used to abstract the exact results of the core verifier. This is analogous, except that we rather try to employ verified optimizers.

References

[1] Jissa website, http://www.quiss.org/jissa/. 2000.

[2] VeryPCC project website in Munich, http://isabelle.in.tum.de/verypcc/, 2003.

[3] A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, June 2001.

[4] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lect. Notes in Comp. Sci.*, pages 38–52. Springer-Verlag, 2000.

[5] A. Chaieb. Proof-producing program analysis. Technical report, TU, München, Dec. 2004.

[6] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. The open verifier framework for foundational verifiers. In *In Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'05).* ACM SIGPLAN Notices, 2005.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[8] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Research report, National ICT Australia, Sydney, 2004.

[9] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 85–97. ACM Press, 1998.

[10] G. C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, 1998.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[12] S. Seo, H. Yang, and K. Yi. Automatic construction of hoare proofs from abstract interpretation results. In *The First Asian Symposium on Programming Languages and Systems, LNCS Vol. 2895*, pages 230–245, Beijing, 2003. Springer.

[13] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[14] Verificard project website in Munich, http://isabelle.in.tum.de/verificard/, 2002.

[15] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2004).* Springer Verlag, 2004. 16 pages.

[16] M. Wildmoser and T. Nipkow. Asserting bytecode safety. *Proceedings of the 15th European Symposium on Programming (ESOP05)*, 2005. to appear.

[17] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Proc. 3rd IFIP Int. Conf. Theoretical Computer Science (TCS 2004)*, 2004.