Certify Once, Trust Anywhere
Modular Certification of Bytecode Programs for Certified Virtual Machine

Yuan Dong, Kai Ren, Shengyuan Wang, and Suqin Zhang

Tsinghua University, Beijing, China

Abstract. Bytecode (such as JAVA bytecode and .NET CIL) and virtual machine are the key technologies for hardware- and operating system-independence. Although some efforts have been made to build logic system for bytecode programs, modular certification of bytecode still remains challenging. Moreover, certified programs will still get stuck due to the virtual machine faults. To overcome these challenges, this paper presents a logic system to verify bytecode programs with their running environment. We define a Hoare-style logic system which fully supports abstract control stacks and unstructured control flows for modular certification of bytecode programs. We also implemented a certified stack-based virtual machine with simulation relation proof. This logic system guarantees that a certified bytecode program will run on the certified virtual machine without getting stuck unless hardware faults occur. We prove the soundness and demonstrate its power by certifying some example programs with the Coq proof assistant. This work not only provides a solid theoretical foundation for reasoning about bytecode programs, but also gains insight into building proof-preserving compilers.

1 Introduction

Bytecode (such as JAVA bytecode [15] and .NET CIL [7]) and VM (virtual machine) are the key components of the many current web applications.

Major Challenges. Formal reasoning about bytecode programs is required both for trustworthy web applications and for proof-transforming compilers. Java and CIL are already verifiably type safe with the well-defined type system. Clearly, we want to certify more properties such as memory safty and partial correctness. Although some efforts [17, 3, 2] on building logic system for bytecode programs have been made, the task still remains challenging because of the complexity of abstract control stacks and the lack of control flows information. Moreover, all these logic systems do focus on bytecode programs; none of them takes virtual machines into account. Unfortunately, there are lots of bugs in the well tested virtual machine [20]. Thus, even a certified program may get stuck due to the virtual machine faults.

To tackle these challenges, this paper presents a way of building certified virtual machine and an end-to-end certification logic system for bytecode programs. We provide a logic system for modularly verifying bytecode programs, a certified virtual machine for interpreting bytecode programs, and a guarantee that a certified bytecode program will run fine on the certified virtual machine. It is very difficult to build a logic system for certifying bytecode programs as well as a corresponding certified virtual machine. The major points are:

– How can we link certified bytecode programs and certified VM together? An open logic framework was designed to integrate [8] the proof of different logic systems

```
;Method: factorial  |  -{(p0, g0)}   ;entry point, instruction sequence 1
;with while loop    |  0  pushc 1  ;push imm 1  8  pushc 1  ;push imm 1
int factor(){       |  1  pop r    ;r = 1       9  binop_   ;n-1
  r = 1;            |  2  goto 11  ;to the end  10 pop n    ;save var n
  while(n != 0){    |  -{(p3, g3)}   ;start loop -{(p11, g11)}  ;inst seq 3
    r = r*n;        |  3  pushv r  ;push var r  11 pushv n  ;push var n
    n = n-1;        |  4  pushv n  ;push var n  12 pushc 0  ;push imm 0
  }                 |  5  binop*   ;r*n         13 binop#   ;n#0?
}                   |  6  pop r    ;save var r  14 brture 3 ;conditional goto
                    |  7  pushv n  ;push var n  15 ret      ;function ret
```

Fig. 1. Stack-Based Bytecode Program

for the X86 machine. But, it is very difficult to integrate the separated certified program modules of different logic systems for different machines.

– To certify bytecode programs modularly, program logic for the virtual machine is required to support both runtime stacks and unstructured control flows. We should use similar logic systems for both the assembly program and the bytecode program to make it easy to link the proof together. But, is the idea of logic system for assembly code certification applicable to bytecode programs for a virtual machine?

Our Approach. A bytecode program with source code which involving while loop control structure is shown in Figure 1. The contents in the shadow box can be ignored now, which will be discussed in details later. Here we give an informal overview about how to certify this program in our method.

Firstly, we formalize two machines. We present the formal definition of bytecode language which runs on a stack-based virtual machine named BCM (ByteCode Machine). We use the formal definition of the X86 machine mentioned in SCAP paper [9].

Then, two logic systems for these two machines are provided to verify bytecode programs and the virtual machine implementation separately and modularly. CertVM (Certified VM), an implementation of BCM on the X86 machine is constructed. We use SCAP, a simple but flexible Hoare-style logic (see Feng *et al.*, [9]), to certify CertVM modularly. Furthermore, we present a Hoare-style logic CBP(Certifying Bytecode Programs) system for BCM. This logic follows the invariant-based proof technique. We define a program invariant to encode the memory safety property and the partial correctness which we are interested in.

Finally, the most important thing is to put these two logic systems together to guarantee that certified bytecode programs run on the certified virtual machines without getting stuck. The simulation relation proof shows that CertVM implementation is satisfied with BCM operational semantics. This main theorem proves that for each bytecode program that is verified in the CBP logic, one can find an equivalent X86 program which is in a simulation relation with the execution of the bytecode program by the virtual machine. This equivalent program is verifiable in SCAP.

Our Contributions. In general, the most interesting point made by this paper about the improvement over previous work is that of the certified virtual machine CertVM. We present a Hoare-style logic system to support modular verification of bytecode programs with all kinds of stack-based control abstractions and unstructured control flows. Formalizing the memory model of our CertVM, we give a certified virtual machine with machine simulation relation proof. Building upon previous work on verification, we make the following contributions:

$$\begin{array}{rrll rrll}
(\mathit{World}) & \mathbb{W} & ::= & (\mathbb{C},\mathbb{S},\mathbb{K}_c,\mathtt{pc}) & (\mathit{Memory} & \mathbb{H} & ::= & \{\mathtt{k} \leadsto \mathtt{w}\}^* \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \leadsto \mathbb{I}\}^* & (\mathit{EStack} & \mathbb{K} & ::= & \mathsf{nil} \mid \mathtt{w} :: \mathbb{K} \\
(\mathit{State}) & \mathbb{S} & ::= & (\mathbb{H},\mathbb{K}) & (\mathit{Labels}) & \mathtt{f},\mathtt{k} & ::= & n \ (\mathit{nat\ nums}) \\
(\mathit{CStack} & \mathbb{K}_c & ::= & \mathsf{nil} \mid \mathtt{f} :: \mathbb{K}_c & (\mathit{Word}) & \mathtt{w} & ::= & i \ (\mathit{integers}) \\
(\mathit{ProgCnt} & \mathtt{pc} & ::= & n \ (\mathit{nat\ nums}) & (\mathit{OprNum}) & \mathtt{m} & ::= & \{+\ldots/,-\cdots+\} \\
(\mathit{Instr}) & \iota & ::= & \multicolumn{5}{l}{\mathsf{pushc}\ \mathtt{w} \mid \mathsf{pushv}\ \mathtt{k} \mid \mathsf{pop}\ \mathtt{k} \mid \mathsf{binop}\ \mathtt{m} \mid \mathsf{unop}\ \mathtt{m} \mid \mathsf{brtrue}\ \mathtt{f} \mid \mathsf{call}\ \mathtt{f}} \\
(\mathit{Commd}) & \mathtt{c} & ::= & \multicolumn{5}{l}{\iota \mid \mathsf{ret} \mid \mathsf{goto}\ \mathtt{f}} \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= & \multicolumn{5}{l}{\iota;\mathbb{I} \mid \mathsf{ret} \mid \mathsf{goto}\ \mathtt{f}}
\end{array}$$

Fig. 2. Definition of BCM Bytecode Machine

- As far as we know, our work presents the first program logic facility with certified VM for certifying the partial correctness of bytecode programs. Our work is static certifying so there is no additional runtime overhead.
- With the “plus simulation” relation, we prove the semantics preservation property of our virtual machine. Furthermore, VM implementation and simulation relation proof can be developed on any physical machines. As an important advantage, once the properties of a bytecode program are certified, they will be preserved on any certified virtual machine. That’s the reason of “Certify once, trust anywhere”.
- This logic system is, to our best knowledge, the first to extend FPCC(Foundational Proof-Carrying Code) concepts [1] which is useful for machine code certification to mid-level bytecode language. As we know, an interpreter is similar to the code generator of a compiler. So, it is a feasible way to build a logic system for proof and semantics preserving compilation from bytecode to machine code.

This system is fully mechanized. We give the complete soundness proof and a full verification of an example in the Coq proof assistant [6]. The virtual machine CertVM is implemented in X86 assembly language and is certified with SCAP logic system. Furthermore, it is executable in the Bochs simulator [11].

The rest of this paper is organized as follows: we first formalize the bytecode virtual machine BCM, give its operational semantics, and present a Hoare-style logic system CBP for bytecode program certifying(Sec 2). We then give the implementation of CertVM, prove the simulation relation, and put them together to prove the soundness (Sec 3). After that, we show some examples and the implementation with Coq proof assistant tools (Sec 4). Finally we discuss related works and draw a conclusion.

2 CBP Logic for ByteCode Virtual Machine

In this section, we present the definition and the operational semantics of BCM bytecode machine. Then, we give the program logic CBP for certifying bytecode programs.

2.1 Bytecode Machine BCM

BCM Definition. In Figure 2, we show BCM definition. The whole machine configuration is called a “World” ($\mathbb{W}$), and consits of a read-only code heap (C), an updatable state ($\mathbb{S}$), a function call stack ($\mathbb{K}_c$), and a program counter ($\mathtt{pc}$). The code heap is a finite partial mapping from code labels ($\mathtt{f}$) to instruction sequences ($\mathbb{I}$). The state $\mathbb{S}$ contains a memory heap ($\mathbb{H}$) and an evaluation stack ($\mathbb{K}$). The program counter $\mathtt{pc}$ points to the current command in $\mathbb{C}$. The instruction sequence $\mathbb{I}$ is a sequence of sequential instructions ending with jump or return commands. $\mathbb{C}[\mathtt{f}]$ extracts an instruction sequence starting from $\mathtt{f}$ in $\mathbb{C}$, as defined in Figure 3. We use the dot notation to represent a component in a tuple, *e.g.*, $\mathbb{S}.\mathbb{K}$ means the stack in state $\mathbb{S}$. We also use function $\mathtt{top}()$ and

$$\mathbb{C}[\mathtt{f}] \triangleq \begin{cases} \mathtt{c} & \mathtt{c} = \mathbb{C}(\mathtt{f}) \text{ and } \mathtt{c} = \text{goto } \mathtt{f}', \text{ or ret} \\ \iota;\mathbb{I} & \iota = \mathbb{C}(\mathtt{f}) \text{ and } \mathbb{I} = \mathbb{C}[\mathtt{f}+1] \end{cases} \qquad (F\{a \rightsquigarrow b\})(x) \triangleq \begin{cases} b & \text{if } x = a \\ F(x) & \text{otherwise} \end{cases}.$$

$$\mathtt{valid}\mathbb{K}\ \ n\ \mathbb{K}\ \triangleq\ \mathtt{top}(\mathbb{K})\ +\ n \le \mathtt{max}(\mathbb{K}) \qquad \mathtt{valid}\mathbb{K}_c\ n\ \mathbb{K}_c\ \triangleq\ \mathtt{top}(\mathbb{K}_c)\ +\ n \le \mathtt{max}(\mathbb{K}_c)$$

$$\mathtt{validRa}\quad \mathbb{K}_c\ \triangleq\ \exists \mathtt{f}, \exists \mathbb{K}_c{}'.\mathbb{K}_c = \mathtt{f} :: \mathbb{K}_c{}'$$

Fig. 3. Definition of Representations

$\mathtt{NextS}_{(\mathtt{c},\mathtt{pc},\mathbb{K}_c)}\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S} = (\mathbb{H},\mathbb{K})$

if c =	if $\mathtt{Enable}_{(\mathtt{c})}\ \mathbb{K}_c\ \mathbb{S}$ =	then $\mathbb{S}'$ =
pushc w	$\mathtt{valid}\mathbb{K}\ 0\ \mathbb{K}$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K})$
pushv f	$\mathtt{valid}\mathbb{K}\ 0\ \mathbb{K}$ and $\mathbb{H}(\mathtt{f}) = \mathtt{w}$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K})$
pop f	$\mathbb{K} = \mathtt{w} :: \mathbb{K}'$	$(\mathbb{H}\{\mathtt{f} \rightsquigarrow \mathtt{w}\}, \mathbb{K}')$
binop *bop*	$\mathbb{K} = \mathtt{w}_1 :: \mathtt{w}_2 :: \mathbb{K}', \mathtt{w} = bop(\mathtt{w}_1, \mathtt{w}_2)$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K}')$
unop *uop*	$\mathbb{K} = \mathtt{w}_1 :: \mathbb{K}', \mathtt{w} = uop(\mathtt{w}_1)$	$(\mathbb{H}, \mathtt{w} :: \mathbb{K}')$
brtrue f	$\mathbb{K} = \mathtt{w} :: \mathbb{K}', \mathtt{w} =$ True or False	$(\mathbb{H}, \mathbb{K}')$
call f	$\mathtt{valid}\mathbb{K}_c\ 0\ \mathbb{K}_c$	$(\mathbb{H}, \mathbb{K})$
ret	$\mathtt{validRa}\quad \mathbb{K}_c$	$(\mathbb{H}, \mathbb{K})$
...		$(\mathbb{H}, \mathbb{K})$

$\mathtt{NextKc}_{(\mathtt{c},\mathtt{pc},\mathbb{S})}\ \mathbb{K}_c\ \mathbb{K}_c{}'$ where $\mathbb{S} = (\mathbb{H},\mathbb{K})$

if c =	if $\mathtt{Enable}_{(\mathtt{c})}\ \mathbb{K}_c\ \mathbb{S}$ =	then $\mathbb{K}_c{}'$ =
call f	$\mathtt{valid}\mathbb{K}_c\ 0\ \mathbb{K}_c$	$(\mathtt{pc}+1) :: \mathbb{K}_c)$
ret	$\mathtt{validRa}\quad \mathbb{K}_c$	$\mathbb{K}_c{}'$
...	...	$\mathbb{K}_c$

$\mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{K}_c)}\ \mathtt{pc}\ \mathtt{pc}'$ where $\mathbb{S} = (\mathbb{H},\mathbb{K})$

if c =	if $\mathtt{Enable}_{(\mathtt{c})}\ \mathbb{K}_c\ \mathbb{S}$ =	then $\mathtt{pc}'$ =
brtrue f	$\mathbb{K} = \mathtt{w} :: \mathbb{K}', \mathtt{w} =$ True	f
	$\mathbb{K} = \mathtt{w} :: \mathbb{K}', \mathtt{w} =$ False	$\mathtt{pc}+1$
call f	$\mathtt{valid}\mathbb{K}_c\ 0\ \mathbb{K}_c$	f
ret	$\mathtt{validRa}\quad \mathbb{K}_c$	f
goto f		f
...	...	pc+1

$$\frac{\mathtt{c} = \mathbb{C}(\mathtt{pc}) \quad \mathtt{Enable}_{(\mathtt{c})}\ \mathbb{K}_c\ \mathbb{S} \quad \mathtt{NextS}_{(\mathtt{c},\mathtt{pc},\mathbb{K}_c)}\ \mathbb{S}\ \mathbb{S}' \quad \mathtt{NextKc}_{(\mathtt{c},\mathtt{pc},\mathbb{S})}\ \mathbb{K}_c\ \mathbb{K}_c{}' \quad \mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{K}_c)}\ \mathtt{pc}\ \mathtt{pc}'}{(\mathbb{C},\mathbb{S},\mathbb{K}_c,\mathtt{pc}) \longmapsto (\mathbb{C},\mathbb{S}',\mathbb{K}_c{}',\mathtt{pc}')}\ (\mathrm{PC})$$

Fig. 4. operational semantics of *BVM*

$\mathtt{max}()$ to get the current pointers and the upper bounds of $\mathbb{K}$, $\mathbb{K}_c$. Valid $\mathbb{K}$ or $\mathbb{K}_c$ means that current pointer $\mathtt{top}()$ is in domain $[0, \mathtt{max}()]$ and points to some value.

The BCM Operational Semantics. In Figure 4, we also define the machine configuration transition operational semantics of each instruction in a formal way. Here $\mathtt{Enable}(\mathtt{c})\ \mathbb{K}_c\ \mathbb{S}$ gives the weakest condition for instruction c to execute. The relation $\mathtt{NextS}_{(\mathtt{c},\mathtt{pc},\mathbb{K}_c)}$ shows the transition of states by executing c with program counter pc and call stack $\mathbb{K}_c$. While $\mathtt{NextPC}_{(\mathtt{c},\mathbb{S},\mathbb{K}_c)}$ shows how pc changes after c is executed with $\mathbb{S}$ and $\mathbb{K}_c$. $\mathtt{NextKc}_{(\mathtt{c},\mathtt{pc},\mathbb{S})}$ gives the $\mathbb{K}_c$ changes after c execution with program counter pc and $\mathbb{S}$. The semantics of most instructions are straightforward. The execution of programs is modeled as a small-step transition from one world to another. $\mathbb{W} \longmapsto \mathbb{W}'$ is made by executing the instruction pointed to by pc.

Specification Language We use the mechanized *meta-logic* which is implemented in the Coq proof assistant [6] as our specification language. The logic corresponds to a higher-order predicate logic with inductive definitions. To specify a program with code heap $\mathbb{C}$, the programmer must insert specifications s at instruction sequence start points, see Figure 1. As shown in Figure 5, the specification s is a pair $(\mathtt{p}, \mathtt{g})$. The assertion p

$$\begin{array}{rrcl rrcl}
(\mathit{Pred}) & \mathtt{p} & \in & \mathit{CStack} \to \mathit{State} \to \mathit{Prop} & (\mathit{Guarantee}) & \mathtt{g} & \in & \mathit{State} \to \mathit{State} \to \mathit{Prop} \\
(\mathit{Spec}) & \mathtt{s} & ::= & (\mathtt{p},\mathtt{g}) & (\mathit{MPred}) & \mathtt{m} & \in & \mathit{Memory} \to \mathit{Prop} \\
(\mathit{CdHpSpec}) & \Psi & ::= & \{(\mathtt{f}_1,\mathtt{s}_1),\ldots,(\mathtt{f}_n,\mathtt{s}_n)\} & & & &
\end{array}$$

Fig. 5. Specification Constructs for *CBP*

is a predicate over function call stack $\mathbb{K}_c$ and program state $\mathbb{S}$, while guarantee $\mathtt{g}$ is a predicate over two program states. We use $\mathtt{p}$ to specify the precondition over function call stack, memory heap and stack. And use $\mathtt{g}$ to specify the guaranteed behavior from the specified program point to the point when the *current* function returns.

As we can see, the $\mathtt{Enable}(\mathtt{c})$ defined in Figure 4 is a special $\mathtt{p}$. And the $\mathtt{NextS}_{(\mathtt{c},\mathtt{pc})}$ relation is a special form of $\mathtt{g}$ which is over the two adjacent states. We use the predicate $\mathtt{m}$ to specify the memory heap. Specification Ψ for code heap $\mathbb{C}$ associates code labels $\mathtt{f}$ with corresponding $\mathtt{s}$. Note that multiple $\mathtt{s}$ may be associated with the same $\mathtt{f}$, just as a function may have multiple specified interfaces.

2.2 The CBP Program Logic

We use the following judgments to define the inference rules:

$$\begin{array}{ll}
\Psi \vdash \{\mathtt{s}\}\,\mathbb{W} & \text{(well-formed world)} \\
\Psi \vdash \mathbb{C} : \Psi' & \text{(well-formed code heap)} \\
\Psi \vdash \{\mathtt{s}\}\,\mathbb{I} & \text{(well-formed instruction sequence)}
\end{array}$$

Inference rules of the program logic are shown in Figure 6.

Program Invariants. The WLD rule formulates the program invariant enforced by our program logic:

– The code heap $\mathbb{C}$ needs to be well-formed following the CDHP rule.
– The imported interface Ψ is a subset of the exported interface Ψ', therefore $\mathbb{C}$ is self-contained and each imported specification has been certified.
– Current $\mathtt{pc}$ has a specification $\mathtt{s}$ in Ψ, thus the current instruction sequence $\mathbb{C}[\mathtt{pc}]$ is well-formed with respect to $\mathtt{s}$.
– Given exported Ψ', the current state $\mathbb{S}$ satisfies the assertion $\mathtt{s}$.

Program Modules. In the CDHP rule, Ψ contains specifications for external code (imported by the local module $\mathbb{C}$), while Ψ' contains specifications for code blocks in the module $\mathbb{C}$ for other modules. Thus, the *CBP* logic supports *separate verification* of program modules. Modules are modeled as small code heaps which contain at least one code block. The specification of a module contains not only specifications of the code blocks in the current module, but also specifications of external code blocks which will be called by this module. The well-formedness of each individual module is established via the CDHP rule. Then, two non-intersecting well-formed modules can then be linked together via the LINK rule. The WORLD rule requires all modules to be linked into a well-formed global code heap.

Sequential Instructions. Like traditional Hoare-logic [10], our logic also uses the pre- and post-condition as specifications for programs. The SEQ rule is a *schema* for instruction sequences starting with an instruction ι (ι cannot be conditional jump or function call instructions). It says it is safe to execute the instruction sequence $\mathbb{I}$ starting at the code label $\mathtt{pc}$, given the imported interface in Ψ and a precondition $(\mathtt{p},\mathtt{g})$. An intermediate specification $(\mathtt{p}'',\mathtt{g}'')$ with respect to which the remaining instruction sequence is well-formed should be found. It is also used as a post-condition for the current instruction ι. We use $\mathtt{g}_\iota$ to represent the state transition made by the instruction ι, which is defined in Figure 7 and Figure 4. Since $\mathtt{NextS}$ does not depend on the current program counter for these instructions "_" is used to represent arbitrary $\mathtt{pc}$.

$\boxed{\Psi \vdash \{\mathtt{s}\}\, \mathbb{W}}$ ***(Well-formed World)***

$$\frac{\Psi \vdash \mathbb{C} : \Psi' \quad \Psi \subseteq \Psi' \quad \Psi \vdash \{\mathtt{s}\}\, \mathtt{pc} : \mathbb{C}[\mathtt{pc}] \quad \{\mathtt{s}\}\, \Psi'\, \mathbb{S}}{\Psi \vdash \{\mathtt{s}\}\, (\mathbb{C}, \mathbb{S}, \mathtt{pc})} \quad (\textsc{wld})$$

$\boxed{\Psi \vdash \mathbb{C} : \Psi'}$ ***(Well-formed Code Heap)***

$$\frac{\text{for all } (\mathtt{f}, \mathtt{s}) \in \Psi' : \quad \Psi \vdash \{\mathtt{s}\}\, \mathtt{f} : \mathbb{C}[\mathtt{f}]}{\Psi \vdash \mathbb{C} : \Psi'} \quad (\textsc{cdhp})$$

$$\frac{\Psi_1 \vdash \mathbb{C}_1 : \Psi'_1 \quad \Psi_2 \vdash \mathbb{C}_2 : \Psi'_2 \quad \mathbb{C}_1 \# \mathbb{C}_2}{\Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi'_1 \cup \Psi'_2} \quad (\textsc{link})$$

$\boxed{\Psi \vdash \{\mathtt{s}\}\, \mathbb{I}}$ ***(Well-formed Instr. Sequence)***

$$\frac{\iota \notin \{\text{brtrue}, \text{call}\} \quad \Psi \vdash \{(\mathtt{p}'', \mathtt{g}'')\}\, \mathtt{pc}{+}1 : \mathbb{I} \quad \mathtt{p} \Rightarrow \mathtt{g}_\iota \quad (\mathtt{p} \rhd \mathtt{g}_\iota) \Rightarrow \mathtt{p}'' \quad (\mathtt{p} \circ (\mathtt{g}_\iota \circ \mathtt{g}'')) \Rightarrow \mathtt{g}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\, \mathtt{pc} : \iota; \mathbb{I}} \quad (\textsc{seq})$$

$$\frac{\begin{array}{l}(\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \Psi \qquad \Psi \vdash \{(\mathtt{p}'', \mathtt{g}'')\}\, \mathtt{pc}{+}1 : \mathbb{I} \\ (\mathtt{p} \rhd \mathsf{g}_{\mathsf{brT}}) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \circ (\mathsf{g}_{\mathsf{brT}} \circ \mathtt{g}')) \Rightarrow \mathtt{g} \quad (\mathtt{p} \rhd \mathsf{g}_{\mathsf{brF}}) \Rightarrow \mathtt{p}'' \quad (\mathtt{p} \circ (\mathsf{g}_{\mathsf{brF}} \circ \mathtt{g}'')) \Rightarrow \mathtt{g}\end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\, \mathtt{pc} : \text{brtrue } \mathtt{f}'; \mathbb{I}} \quad (\textsc{brture})$$

$$\frac{\begin{array}{l}(\mathtt{pc}{+}1, (\mathtt{p}'', \mathtt{g}'')) \in \Psi \qquad \Psi \vdash \{(\mathtt{p}'', \mathtt{g}'')\}\, \mathtt{pc}{+}1 : \mathbb{I} \\ (\mathtt{p} \rhd \mathsf{g}_{\mathsf{call}}) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \rhd \mathsf{g}_{\mathsf{fun}}) \Rightarrow \mathtt{p}'' \quad (\mathtt{p} \circ (\mathsf{g}_{\mathsf{fun}} \circ \mathtt{g}'')) \Rightarrow \mathtt{g} \quad (\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \Psi \quad \mathsf{g}_{\mathsf{fun}} = ((\mathsf{g}_{\mathsf{call}} \circ \mathtt{g}') \circ \mathsf{g}_{\mathsf{ret}})\end{array}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\, \mathtt{pc} : \text{call } \mathtt{lf}'; \mathbb{I}} \quad (\textsc{call})$$

$$\frac{(\mathtt{p} \circ \mathsf{g}_{\mathsf{ret}}) \Rightarrow \mathtt{g}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\, \mathtt{pc} : \text{ret}} \quad (\textsc{ret})$$

$$\frac{(\mathtt{f}', (\mathtt{p}', \mathtt{g}')) \in \Psi \quad (\mathtt{p} \rhd \mathsf{g}_{\mathsf{goto}}) \Rightarrow \mathtt{p}' \quad (\mathtt{p} \circ (\mathsf{g}_{\mathsf{goto}} \circ \mathtt{g}')) \Rightarrow \mathtt{g}}{\Psi \vdash \{(\mathtt{p}, \mathtt{g})\}\, \mathtt{pc} : \text{goto } \mathtt{f}'} \quad (\textsc{goto})$$

Fig. 6. CBP Inference Rules

$$\begin{array}{lll}
\mathsf{g}_{\mathsf{brT}} \triangleq \lambda \mathbb{S}, \mathbb{S}'. \mathtt{NextS}_{(\text{brture},\_)}\, \mathbb{S}\, \mathbb{S}' & & (\text{where } \mathbb{S}.\mathbb{K} = w :: \mathbb{K}', w = \mathsf{True}) \\
\mathsf{g}_{\mathsf{brF}} \triangleq \lambda \mathbb{S}, \mathbb{S}'. \mathtt{NextS}_{(\text{brture},\_)}\, \mathbb{S}\, \mathbb{S}' & & (\text{where } \mathbb{S}.\mathbb{K} = w :: \mathbb{K}', w = \mathsf{False}) \\
\mathsf{g}_{\mathtt{c}} \triangleq \lambda \mathbb{S}, \mathbb{S}'. \mathtt{NextS}_{(\mathtt{c},\_)}\, \mathbb{S}\, \mathbb{S}' & & (\text{for all other } \mathtt{c})
\end{array}$$

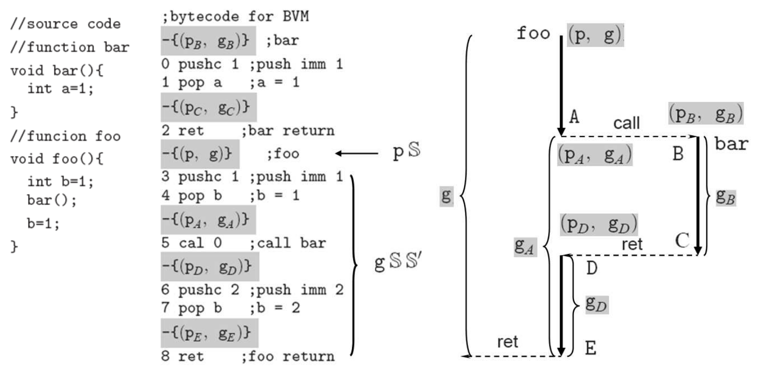
Fig. 7. Local State and Program Point Transitions

The definitions in Figure 8 are used in these rules. The predicate $\mathtt{p} \rhd \mathtt{g}_\iota$ specifies the state resulting from the state transition $\mathtt{g}_\iota$, knowing the initial state satisfies $\mathtt{p}$. The composition of two subsequent transitions $\mathtt{g}$ and $\mathtt{g}'$ is represented as $\mathtt{g} \circ \mathtt{g}'$, and $\mathtt{p} \circ \mathtt{g}$ refines $\mathtt{g}$ with the extra knowledge that the initial state satisfies $\mathtt{p}$. The predicate $\mathtt{p} \Rightarrow \mathtt{g}_\iota$ means that the state transition $\mathtt{g}_\iota$ would not get stuck as long as the starting state satisfies $\mathtt{p}$. The second premise in the SEQ rule means if the current state satisfies $\mathtt{p}$, after state transition $\mathtt{g}_\iota$, the new state satisfies $\mathtt{p}'$. The last premise in the SEQ rule requires the composition of $\mathtt{g}_\iota$ and $\mathtt{g}''$ fulfilling $\mathtt{g}$, knowing the current state satisfies $\mathtt{p}$.

Function Call and Return. Figure 9(b) shows the meaning of the specification $(\mathtt{p}, \mathtt{g})$ for the function `foo` defined in Figure 9(a). Note that $\mathtt{g}$ may cover multiple instruction sequences. If a function has multiple return points, $\mathtt{g}$ governs all the traces from the current program point to any return point. Figure 9(c) illustrates a function call to `bar` (point `B`) from `foo` at point `A` (label $\mathtt{pc} = 5$), with the return address $\mathtt{pc} + 1$ (point `D`). The specification of `bar` is $(\mathtt{p}_B, \mathtt{g}_B)$. Specifications at `A` and `D` are $(\mathtt{p}_A, \mathtt{g}_A)$ and $(\mathtt{p}_D, \mathtt{g}_D)$ respectively, where $\mathtt{g}_A$ governs the code segment `A-E` and $\mathtt{g}_D$ governs `D-E`.

$$\begin{array}{ll}
\mathtt{p} \Rightarrow \mathtt{g} \triangleq \forall \mathbb{S}.\, \mathtt{p}\, \mathbb{S} \rightarrow \exists \mathbb{S}', \mathtt{g}\, \mathbb{S}\, \mathbb{S}' & \mathtt{p} \rhd \mathtt{g} \triangleq \lambda \mathbb{S}.\, \exists \mathbb{S}_0, \mathtt{p}\, \mathbb{S}_0 \wedge \mathtt{g}\, \mathbb{S}_0\, \mathbb{S} \\
\mathtt{g} \circ \mathtt{g}' \triangleq \lambda \mathbb{S}, \mathbb{S}''.\, \exists \mathbb{S}'.\, \mathtt{g}\, \mathbb{S}\, \mathbb{S}' \wedge \mathtt{g}'\, \mathbb{S}'\, \mathbb{S}'' & \mathtt{p} \Rightarrow \mathtt{p}' \triangleq \forall \mathbb{S}.\, \mathtt{p}\, \mathbb{S} \rightarrow \mathtt{p}'\, \mathbb{S} \\
\mathtt{g} \Rightarrow \mathtt{g}' \triangleq \forall \mathbb{S}, \mathbb{S}'.\, \mathtt{g}\, \mathbb{S}\, \mathbb{S}' \rightarrow \mathtt{g}'\, \mathbb{S}\, \mathbb{S}' & \mathtt{p} \circ \mathtt{g} \triangleq \lambda \mathbb{S}, \mathbb{S}'.\, \mathtt{p}\, \mathbb{S} \wedge \mathtt{g}\, \mathbb{S}\, \mathbb{S}'
\end{array}$$

Fig. 8. Connectors for $\mathtt{p}$ and $\mathtt{g}$



```
//source code
//function bar
void bar(){
  int a=1;
}
//funcion foo
void foo(){
  int b=1;
  bar();
  b=1;
}
```

```
;bytecode for BVM
-{(p_B, g_B)}  ;bar
0 pushc 1 ;push imm 1
1 pop a   ;a = 1
-{(p_C, g_C)}
2 ret     ;bar return
-{(p, g)}    ;foo
3 pushc 1 ;push imm 1
4 pop b   ;b = 1
-{(p_A, g_A)}
5 cal 0   ;call bar
-{(p_D, g_D)}
6 pushc 2 ;push imm 2
7 pop b   ;b = 2
-{(p_E, g_E)}
8 ret     ;foo return
```

Fig. 9. The Model for Function Call/Return in CBP

To ensure that the program behaves correctly, we must enforce the following conditions with a special guarantee $\mathsf{g}_{\mathsf{fun}} \triangleq \lambda\mathbb{S},\mathbb{S}''.\exists\mathbb{S}',\exists\mathbb{S}^*,\mathsf{g}_{\mathsf{cal}}\,\mathbb{S}\,\mathbb{S}' \wedge \mathsf{g}_B\,\mathbb{S}'\,\mathbb{S}^* \wedge \mathsf{g}_{\mathsf{ret}}\,\mathbb{S}^*\,\mathbb{S}''$.

– the precondition of `bar` should be satisfied, *i.e.*, $\forall\mathbb{S},\exists\mathbb{S}'.\mathsf{p}_A\,\mathbb{S} \wedge \mathsf{g}_{\mathsf{cal}}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathsf{p}_B\,\mathbb{S}'$;
– after `bar` returns, caller `foo` resumes from `D`, $\forall\mathbb{S},\mathbb{S}''.\mathsf{p}_A\,\mathbb{S} \rightarrow \mathsf{g}_{\mathsf{fun}}\,\mathbb{S}\,\mathbb{S}'' \rightarrow \mathsf{p}_D\,\mathbb{S}''$;
– if the function `bar` and the code segment `D`-`E` satisfy their specifications, the specification for `A`-`E` is satisfied, *i.e.*, $\forall\mathbb{S},\mathbb{S}'',\mathbb{S}'''.\mathsf{p}_A\,\mathbb{S} \rightarrow \mathsf{g}_{\mathsf{fun}}\,\mathbb{S}\,\mathbb{S}'' \rightarrow \mathsf{g}_D\,\mathbb{S}''\,\mathbb{S}''' \rightarrow \mathsf{g}\,\mathbb{S}\,\mathbb{S}'''$.

The RET rule simply requires that the function has finished its guaranteed transition at this point. In this rule, we do not need to know any information about the return address. So it can be used to modularly certify any callee function separately.

Call Stack Invariant. Generalizing the safety requirement, we recursively define the "well-formed function call stack" as follows:

$$\mathsf{WFST}(\mathsf{g},\mathbb{K}_c,\mathbb{S},\Psi) \triangleq \neg\exists\mathbb{S}'.\,\mathsf{g}\,\mathbb{S}\,\mathbb{S}', \quad \text{where } \mathbb{K}_c = \mathsf{nil}.$$
$$\mathsf{WFST}(\mathsf{g},\mathbb{K}_c,\mathbb{S},\Psi) \triangleq \forall\mathbb{S}'.\mathsf{g}\,\mathbb{S}\,\mathbb{S}' \rightarrow \mathsf{p}'\,\mathbb{S}' \wedge \mathsf{WFST}(\mathsf{g}',\mathbb{K}_c{}',\mathbb{S}',\Psi),$$
$$\text{where } \exists\mathtt{f},\exists\mathbb{K}_c{}'.\mathbb{K}_c = \mathtt{f} :: \mathbb{K}_c{}', (\mathsf{p}',\mathsf{g}') = \Psi(\mathtt{f}).$$

When the function call stack is empty, we are in the top function which has no return code pointer,*i.e.*, $\neg\exists\mathbb{S}'.\,\mathsf{g}\,\mathbb{S}\,\mathbb{S}'$. Then the stack invariant at every step of program execution is that, at each program point with (p,g), the program state $\mathbb{S}$ must satisfy p and there exists a well-formed control stack in $\mathbb{S}$. So the stack invariant is:

$$\{(\mathsf{p},\mathsf{g})\}\,\Psi\,\mathbb{S} \triangleq \mathsf{p}\,\mathbb{S} \wedge \mathsf{WFST}(\mathsf{g},\mathbb{K}_c,\mathbb{S},\Psi).$$

Soundness of CBP. The soundness of the program logic is proved following the syntactic approach based on the progress and preservation lemmas. It guarantees that the complete system after linking never gets stuck as long as the initial state satisfies the program invariant defined by the WLD rule. Furthermore, the invariant will be always holding during execution, from which we can derive rich properties of programs.

Lemma 1 (Progress). If $\Psi \vdash \{\mathtt{s}\}\mathbb{W}$, there exists a program $\mathbb{W}'$, such that $\mathbb{W} \longmapsto \mathbb{W}'$.

Lemma 2 (Preservation). If $\Psi \vdash \{\mathtt{s}\}\mathbb{W}$, and $\mathbb{W} \longmapsto \mathbb{W}'$, then there exists $\mathtt{s}'$, $\Psi \vdash \{\mathtt{s}'\}\mathbb{W}'$.

Theorem 3 (CBP Soundness). For all program $\mathbb{W}$, specification Ψ and assertion $\mathtt{s}$. If $\Psi \vdash \{\mathtt{s}\}\mathbb{W}$, then for all natural number n, there exists a program $\mathbb{W}'$ such that $\mathbb{W} \longmapsto^n \mathbb{W}'$.

3 Proof of ByteCode Virtual Machine

In this section, we present how to certify the implementation of a virtual machine CertVM for BC/0. We first give the formal definition of x86 real-mode machine where CertVM runs on. Then we introduce the program logic for this machine. Finally, we show the design, implementation and formal proof of CertVM.

$$\begin{array}{rlll rlll}
(\textit{World}) & \mathbb{W} & ::= & (\mathbb{C},\mathbb{S},\mathtt{pc}) & (\textit{Labels}) & l,\mathtt{f},\mathtt{pc} & ::= & n\ (\textit{nat nums})\\
(\textit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* & (\textit{Flags}) & \mathtt{zf} & ::= & \mathtt{b}\\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{H},\mathbb{R},\mathtt{zf}) & (\textit{Bit}) & \mathtt{b} & ::= & 0 \mid 1\\
(\textit{Memory}) & \mathbb{H} & ::= & \{l \rightsquigarrow \mathtt{w}\}^* & (\textit{Word}) & \mathtt{w} & ::= & i\ (\textit{integers})\\
(\textit{RegFile}) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* & (\textit{WordReg}) & \mathtt{r} & ::= & \mathtt{r}_{AX} \mid \mathtt{r}_{BX} \mid \mathtt{r}_{CX} \mid \mathtt{r}_{DX}
\end{array}$$

$$\begin{array}{rll}
(\textit{Instr}) & \iota ::= & \mathsf{je}\ \mathtt{f} \mid \mathsf{movw}\ \mathtt{w},\mathtt{r} \mid \mathsf{movw}\ \mathtt{r}_s,\mathtt{r}_d \mid \mathsf{movw}\ w(\mathtt{r}_s),\mathtt{r}_d \mid \mathsf{movw}\ \mathtt{r}_s,w(\mathtt{r}_d) \mid \mathsf{addw}\ \mathtt{w},\mathtt{r} \mid \mathsf{subw}\ \mathtt{w},\mathtt{r} \mid \mathsf{cmpw}\ \mathtt{w},\mathtt{r}\\
(\textit{Commd}) & \mathtt{c} ::= & \iota \mid \mathsf{jmp}\ \mathtt{f} \mid \mathsf{jmpw}\ \mathtt{r}\\
(\textit{InstrSeq}) & \mathbb{I} ::= & \iota;\mathbb{I} \mid \mathsf{jmp}\ \mathtt{f} \mid \mathsf{jmpw}\ \mathtt{r}
\end{array}$$

Fig. 10. Definition of x86 Machine

$\mathtt{NextS}_{\mathtt{c},\mathtt{pc}}\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S} = (\mathbb{H},\mathbb{R},\mathtt{zf})$

if c =	then $\mathbb{S}' =$
movw $\mathtt{w},\mathtt{r}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow\mathtt{w}\},\mathtt{zf})$
movw $\mathtt{r}_s,\mathtt{r}_d$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d\rightsquigarrow\mathbb{R}(\mathtt{r}_s)\},\mathtt{zf})$
movw $\mathtt{r}_s,w(\mathtt{r}_d)$	$(\mathbb{H}\{l\rightsquigarrow\mathbb{R}(\mathtt{r}_s)\},\mathbb{R},\mathtt{zf})$, if $l = \mathbb{R}(\mathtt{r}_d)+w$ and $l \in dom(\mathbb{H})$
movw $w(\mathtt{r}_s),\mathtt{r}_d$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}_d\rightsquigarrow\mathbb{R}(l)\},\mathtt{zf})$, if $l = \mathbb{R}(\mathtt{r}_s)+w$ and $l \in dom(\mathbb{H})$
addw $\mathtt{w},\mathtt{r}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow(\mathbb{R}(\mathtt{r})+\mathtt{w})\},\mathtt{zf})$
subw $\mathtt{w},\mathtt{r}$	$(\mathbb{H},\mathbb{R}\{\mathtt{r}\rightsquigarrow(\mathbb{R}(\mathtt{r})-\mathtt{w})\},\mathtt{zf})$
cmpw $\mathtt{w},\mathtt{r}$	$(\mathbb{H},\mathbb{R},\mathtt{b})$, $\mathtt{b}=0$, if $\mathtt{w}=\mathbb{R}(\mathtt{r})$; $\mathtt{b}=1$, else
...	$(\mathbb{H},\mathbb{R},\mathtt{zf})$

$\mathtt{NextPC}_{(\mathtt{c},\mathbb{S})}\ \mathtt{pc}\ \mathtt{pc}'$ where $\mathbb{S} = (\mathbb{H},\mathbb{R},\mathtt{zf})$

if c =	then $\mathtt{pc}' =$
je $\mathtt{f}$	$\mathtt{f}$ if $\mathtt{zf}=0$; $\mathtt{pc}+1$ others
jmp $\mathtt{f}$	$\mathtt{f}$
jmpw $\mathtt{r}$	$\mathtt{f}$ if $\mathtt{f}=\mathbb{R}(\mathtt{r})$
...	pc+1

$$\frac{\mathtt{c}=\mathbb{C}(\mathtt{pc}) \quad \mathtt{NextS}_{(\mathtt{c},\mathtt{pc})}\ \mathbb{S}\ \mathbb{S}' \quad \mathtt{NextPC}_{(\mathtt{c},\mathbb{S})}\ \mathtt{pc}\ \mathtt{pc}'}{(\mathbb{C},\mathbb{S},\mathtt{pc}) \longmapsto (\mathbb{C},\mathbb{S}',\mathtt{pc}')}\ (\textsc{pc})$$

Fig. 11. Operational semantics of x86 machine

3.1 x86 Machine and SCAP Program Logic

X86 machine is defined in Figure 10. And Figure 11 shows its operational semantics. This is a simplified version which includes only four general purpose registers. We use SCAP [9] logic system to verify the CertVM implementation. The inference rules of SCAP program logic are given in Figure 12. The soundness proof of SCAP is carried out based on the progress and preservation lemmas which are similar to that of CBP.

3.2 The Design of CertVM

CertVM is implemented in real-mode x86 assembly language, and it is executable in the Bochs simulator. The current implementation of CertVM mainly includes the loader and the interpreter. Other advanced features such as garbage collection and just in-time compilation are not included yet.

The memory space of CertVM consists of four major parts: code heap ($\mathbb{C}$), memory heap ($\mathbb{H}$), evaluation stack ($\mathbb{K}$) and function call stack ($\mathbb{K}_c$). All of them are located in x86 machine's memory heap ($\mathbb{H}_{\mathtt{X}}$) as arrays. In the following analysis, function $\mathtt{base}()$ and $\mathtt{max}()$ are used to get the base address and the maximum length of $\mathbb{C}$, $\mathbb{H}$, $\mathbb{K}$ and $\mathbb{K}_c$. And $\mathtt{top}()$ denotes the top pointer of stack $\mathbb{K}$ and $\mathbb{K}_c$.

A loader is designed to launch the bytecode program. It loads bytecode programs into $\mathbb{C}$, initializes the memory heap $\mathbb{H}$ and the evaluation stacks $\mathbb{K}$ (setting the stack pointer $\mathtt{sp}$ to zero). For the top level function, the bottom cell of function call stack $\mathbb{K}_c$

$\boxed{\Psi \vdash \{s\}\,\mathbb{W}}$ ***(Well-formed World)***

$$\frac{\Psi \vdash \mathbb{C}:\Psi' \quad \Psi \subseteq \Psi' \quad \Psi \vdash \{s\}\,\mathtt{pc}:\ \mathbb{C}[\mathtt{pc}] \quad \{s\}\,\Psi'\,\mathbb{S}}{\Psi \vdash \{s\}\,(\mathbb{C},\mathbb{S},\mathtt{pc})} \quad (\textsc{WLD})$$

$\boxed{\Psi \vdash \mathbb{C}:\Psi'}$ ***(Well-formed Code Heap)***

$$\frac{\text{for all } (\mathtt{f},s) \in \Psi' : \quad \Psi \vdash \{s\}\,\mathtt{f}:\ \mathbb{C}[\mathtt{f}]}{\Psi \vdash \mathbb{C}:\Psi'} \quad (\textsc{CDHP})$$

$$\frac{\Psi_1 \vdash \mathbb{C}_1:\Psi_1' \quad \Psi_2 \vdash \mathbb{C}_2:\Psi_2' \quad \mathbb{C}_1 \# \mathbb{C}_2}{\Psi_1 \cup \Psi_2 \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1' \cup \Psi_2'} \quad (\textsc{LINK})$$

$\boxed{\Psi \vdash \{s\}\,\mathbb{I}}$ ***(Well-formed Instr. Sequence)***

$$\frac{\iota \notin \{\mathsf{je}\ \} \quad \Psi \vdash \{(p'',g'')\}\,\mathtt{pc}{+}1:\ \mathbb{I} \quad p \Rightarrow g_\iota \quad (p \triangleright g_\iota) \Rightarrow p'' \quad (p \circ (g_\iota \circ g'')) \Rightarrow g}{\Psi \vdash \{(p,g)\}\,\mathtt{pc}:\ \iota;\mathbb{I}} \quad (\textsc{SEQ})$$

$$\frac{\begin{array}{c}(\mathtt{f}', (p',g')) \in \Psi \qquad \Psi \vdash \{(p'',g'')\}\,\mathtt{pc}{+}1:\ \mathbb{I} \\ (p \triangleright g_{\mathsf{jeT}}) \Rightarrow p' \quad (p \circ (g_{\mathsf{jeT}} \circ g')) \Rightarrow g \quad (p \triangleright g_{\mathsf{jeF}}) \Rightarrow p'' \quad (p \circ (g_{\mathsf{jeF}} \circ g'')) \Rightarrow g\end{array}}{\Psi \vdash \{(p,g)\}\,\mathtt{pc}:\ \mathsf{je}\ \mathtt{f}';\mathbb{I}} \quad (\textsc{JE})$$

$$\frac{(\mathtt{f}', (p',g')) \in \Psi \quad (p \triangleright g_{\mathsf{jmp}}) \Rightarrow p' \quad (p \circ (g_{\mathsf{jmp}} \circ g')) \Rightarrow g}{\Psi \vdash \{(p,g)\}\,\mathtt{pc}:\ \mathsf{jmp}\ \mathtt{f}'} \quad (\textsc{JMP})$$

$$\frac{(\mathbb{R}(\mathtt{r}), (p',g')) \in \Psi \quad (p \triangleright g_{\mathsf{jmpw}}) \Rightarrow p' \quad (p \circ (g_{\mathsf{jmpw}} \circ g')) \Rightarrow g}{\Psi \vdash \{(p,g)\}\,\mathtt{pc}:\ \mathsf{jmpw}\ \mathtt{r}} \quad (\textsc{JMPW})$$

Fig. 12. SCAP Inference Rules for x86 Machine

```
#Source code of CertVM                 | 12  addw %ax, %bx   # 2 word long
-{(p_fetch, g_fetch)}    #entry point    | 13  movw (%bx), %ax #get entry point
1  fetch:                #bytecode fetch | 14  jmpw *%ax       #jump to code entry
2   movw (pc), %ax       #bytecode pc    | -{(p_goto, g_goto)}    #instr. sequence 2
3   cmpw $0xFFFF,%ax     #compare ra     | 15 goto:
4   je   fetch           #loop forever   | 16  movw (pc), %ax  # code point
5  decode:                               | 17  movw $code, %bx # code base
6   movw $code, %bx      #code base      | 18  addw %ax, %bx   # current base
7   addw %ax, %bx        #current code   | 19  movw 2(%bx),%cx # fetch i.a
8   movw (%bx), %ax      #fetch i.f      | 20  addw %cx, %cx   # 4 bytes instr.
9  dispatch:                             | 21  addw %cx, %cx   #
10  movw $table, %bx     #dispatch table | 22  movw %cx, (pc)  # target address
11  addw %ax, %bx        #offset of code | 23  jmp  fetch
```

Fig. 13. Fragment of CertVM Implementation

is set to -1 (0xFFFF) and the stack pointer `csp` points to the second cell. And CertVM's `pc` is set to the entry point of the loaded bytecode program. After all the initializations, it is ready to execute the bytecode program.

Every bytecode instruction is simulated by a sequence of x86 assembly instructions. The simulation of a bytecode instruction consists of four phases: instruction fetching, decoding, dispatching and interpreting. Figure 13 shows the assembly instruction sequence which simulate bytecode instruction goto as an example. The instruction sequence of bytecode fetching, decoding and dispatching is shared by all BC/0 instructions. Thus the simulation of every bytecode start at the label `fetch`. After fetching and decoding a bytecode, CertVM will jump to the unique entry point for each bytecode. For bytecode goto, the entry point is the label "goto" of line 15 in Figure 13. The entry points for all BC/0 instructions are stored in a bytecode instruction dispatching table. Data structure "table" of line 9 in Figure 13 is the dispatching table of CertVM. It should be preserved during virtual machine execution.

$$\frac{\mathbb{W}_{\mathtt{x}}.\mathbb{C}_{\mathtt{x}} = \mathbb{C}_{\mathtt{VM}} \quad \mathbb{W}_{\mathtt{x}}.\mathtt{pc}_{\mathtt{x}} = \mathtt{fetch} \quad \alpha(\mathbb{W},\ \mathbb{W}_{\mathtt{x}}.\mathbb{H}_{\mathtt{x}})}{\mathbb{W}\ \sim\ \mathbb{W}_{\mathtt{x}}}$$

Fig. 14. Simulation Relation

$$\frac{\begin{array}{c}\mathbb{H}_{\mathtt{x}} = \mathbb{H}_{\mathtt{xc}} \uplus \mathbb{H}_{\mathtt{xh}} \uplus \mathbb{H}_{\mathtt{xk}} \uplus \mathbb{H}_{\mathtt{xkc}} \uplus \mathbb{H}_{\mathtt{xp}} \uplus \mathbb{H}_{\mathtt{xo}} \\ \mathbb{H}_{\mathtt{xc}}(\mathtt{f} \times 4) = \mathbb{C}(\mathtt{f}),\ \forall \mathtt{f} \in [0, \mathtt{max}(\mathbb{C})] \quad \mathbb{H}_{\mathtt{xh}}(l \times 2) = \mathbb{H}(l),\ \forall l \in [0, \mathtt{max}(\mathbb{H})] \\ \mathbb{H}_{\mathtt{xk}}(l \times 2) = \mathbb{K}(l),\ \forall l \in [0, \mathtt{max}(\mathbb{K})] \quad \mathbb{H}_{\mathtt{xkc}}(l \times 2) = \mathbb{K}_c(l),\ \forall l \in [0, \mathtt{max}(\mathbb{K}_c)] \\ \mathbb{H}_{\mathtt{xp}}(0) = \mathtt{pc} \quad \mathbb{H}_{\mathtt{xp}}(2) = \mathtt{top}(\mathbb{K}) \quad \mathbb{H}_{\mathtt{xp}}(4) = \mathtt{top}(\mathbb{K}_c)\end{array}}{\alpha(\mathbb{W},\ \mathbb{W}_{\mathtt{x}}.\mathbb{H}_{\mathtt{x}})}$$

Fig. 15. The Memory Map Relation

3.3 Proof of the Correctness of CertVM

Simulation Relation. To execute bytecode programs correctly, the x86 simulation program should maintain an invariant for the interpretation of every bytecode instruction. This invariant, called ”simulation relation”, is defined as a relation between the bytecode machine world $\mathbb{W} = (\mathbb{C}, (\mathbb{H}, \mathbb{K}), \mathbb{K}_c, \mathtt{pc})$ and x86 machine world $\mathbb{W}_{\mathtt{x}} = (\mathbb{C}_{\mathtt{x}}, (\mathbb{H}_{\mathtt{x}}, \mathbb{R}, \mathtt{zf}), \mathtt{pc}_{\mathtt{x}})$. This relation should be maintained when the simulation program executes to the fetching phase. This relation is shown in in Figure 14, which indicates:

– the code heap of x86 world should be the code of CertVM,
– current program counter of x86 world points to `fetch`,
– the bytecode machine world $\mathbb{W}_{\mathtt{x}}$ is mapped to x86 machine memory heap $\mathbb{H}_{\mathtt{x}}$, following the memory relation α,
– there is no constrain for register file and flag.

We define certified virtual machine $\mathsf{WFVM}(\mathbb{W}, \mathbb{W}_{\mathtt{x}})$ for all bytecode program $\mathbb{W}$ and X86 program $\mathbb{W}_{\mathtt{x}}$ as a virtual machine with this simulation relation:

Definition 4 (Well-Formed VM). For all bytecode program $\mathbb{W}, \mathbb{W}'$ and x86 program $\mathbb{W}_{\mathtt{x}}$, if $\mathbb{W} \sim \mathbb{W}_{\mathtt{x}}$ and $\mathbb{W} \longmapsto \mathbb{W}'$, there exists a x86 program $\mathbb{W}'_{\mathtt{x}}$ such that $\mathbb{W}' \sim \mathbb{W}'_{\mathtt{x}}$ and $\mathbb{W}_{\mathtt{x}} \longmapsto^{+} \mathbb{W}'_{\mathtt{x}}$.

We use “Plus” simulation relation to describe the bytecode interpretation of VM. This relation shows that CertVM implementation is satisfied with BCM operational semantics. Once we carry out the simulation relation proof, we get a certified virtual machine.

The Memory Relation. As mentioned before, the code heap, memory heap, evaluation stack and function call stack of bytecode machine are all stored as arrays in x86 machine memory heap. These arrays are denoted as $\mathbb{H}_{\mathtt{xc}}, \mathbb{H}_{\mathtt{xh}}, \mathbb{H}_{\mathtt{xk}}$ and $\mathbb{H}_{\mathtt{xkc}}$ respectively. In addition, $\mathbb{H}_{\mathtt{xp}}$ denotes the memory chunk that stores the value of $\mathtt{pc}, \mathtt{sp}$ and $\mathtt{csp}$, and $\mathbb{H}_{\mathtt{xo}}$ denotes the free memory space.

The exact configuration of this memory heap partition is defined as follows:

$$\begin{array}{ll}\mathbb{H}_{\mathtt{xc}} & \triangleq \mathbb{H}_{\mathtt{x}}[\mathsf{base}(\mathbb{H}_{\mathtt{xc}}), (\mathsf{base}(\mathbb{H}_{\mathtt{xc}}) + \mathtt{max}(\mathbb{C}) \times 4)] \\ \mathbb{H}_{\mathtt{xh}} & \triangleq \mathbb{H}_{\mathtt{x}}[\mathsf{base}(\mathbb{H}_{\mathtt{xh}}), (\mathsf{base}(\mathbb{H}_{\mathtt{xh}}) + \mathtt{max}(\mathbb{H}) \times 2)] \\ \mathbb{H}_{\mathtt{xk}} & \triangleq \mathbb{H}_{\mathtt{x}}[\mathsf{base}(\mathbb{H}_{\mathtt{xk}}), (\mathsf{base}(\mathbb{H}_{\mathtt{xk}}) + \mathtt{max}(\mathbb{K}) \times 2)] \\ \mathbb{H}_{\mathtt{xkc}} & \triangleq \mathbb{H}_{\mathtt{x}}[\mathsf{base}(\mathbb{H}_{\mathtt{xkc}}), (\mathsf{base}(\mathbb{H}_{\mathtt{xkc}}) + \mathtt{max}(\mathbb{K}_c) \times 2)] \\ \mathbb{H}_{\mathtt{xp}} & \triangleq \mathbb{H}_{\mathtt{x}}[\mathsf{base}(\mathbb{H}_{\mathtt{xp}}), (\mathsf{base}(\mathbb{H}_{\mathtt{xkc}}) + 6)]\end{array}$$

Note that every bytecode instruction is 4 bytes long, and so it occupies $\mathtt{max}(\mathbb{C}) \times 4$ cells in $\mathbb{H}_{\mathtt{x}}$. And every item of $\mathbb{K}$ and $\mathbb{K}_c$ is only 2 bytes long. Therefore, the map relation between x86 machine memory heap and bytecode world is defined in Figure 15.

Proof by Simulation. From CertVM implementation, we know that the entry point of every bytecode is label `fetch`. To prove its correctness, we only have to show ”simulation relation” is achieved when CertVM jumps to `fetch`, and the bytecode world defined in this relation has its successive state. Thus, we use the specification language of SCAP to describe this simulation relation.

$$\begin{array}{llll}
\mathtt{p}_{pre} \triangleq (\mathtt{valid}\mathbb{K}\ 2\ \mathbb{K}) \wedge (\mathtt{valid}\mathbb{K}_c\ 0\ \mathbb{K}_c) \wedge (\mathtt{validRa}\quad \mathbb{K}_c) & & & \\
\mathtt{p}_0 \triangleq \mathtt{p}_{pre} \wedge ((r \leadsto \_) * (\exists i >= 0,\ n \mapsto i)), & & \mathtt{g}_0 \triangleq \mathtt{p}_0 \rightarrow (\mathbb{H}'(\mathsf{r}) = \mathbb{H}(\mathsf{n})!) & \\
\mathtt{p}_3 \triangleq \mathtt{p}_{pre} \wedge ((\mathbb{H}(r) >= 1) \wedge (\mathbb{H}(n) >= 0)), & & \mathtt{g}_3 \triangleq \mathtt{p}_3 \rightarrow (\mathbb{H}'(\mathsf{r}) = \mathbb{H}(\mathsf{r}) * \mathbb{H}(\mathsf{n})!) & \\
\mathtt{p}_{11} \triangleq \mathtt{p}_3, & & \mathtt{g}_{11} \triangleq \mathtt{g}_3 &
\end{array}$$

Fig. 16. Specifications: While Loop Example

Suppose the specification at `fetch` is $(\mathtt{p_C}, \mathtt{g_C})$:

$$\mathtt{p_C} \triangleq \lambda \mathbb{S}_\mathtt{X}, \exists \mathbb{W}. \alpha(\mathbb{W}, \mathbb{S}_\mathtt{X}.\mathbb{H}_\mathtt{X}) \wedge Enable(c, \mathbb{S}, \mathbb{K}_c) \text{ where } \mathbb{W} = (\mathbb{C}, \mathbb{S}, \mathbb{K}_c, \mathtt{pc}) \wedge c = \mathbb{C}(pc)$$
$$\mathtt{g_C} \triangleq \lambda \mathbb{S}'_\mathtt{X}, \exists \mathbb{W}, \mathbb{W}'. \mathbb{W} \rightarrow \mathbb{W}' \wedge \alpha(\mathbb{W}, \mathbb{S}_\mathtt{X}.\mathbb{H}_\mathtt{X}) \wedge \alpha(\mathbb{W}', \mathbb{S}'_\mathtt{X}.\mathbb{H}_\mathtt{X})$$

$\mathtt{p_C}$ means that before executing `fetch`, the x86 machine world should maintain the simulation relation with a bytecode world that can transit to its next step. And $\mathtt{g_C}$ ensures that after interpreting a bytecode instruction, the simulation relation is still held. Thus, we only need to use the inference rules of SCAP to prove that:

$$\Psi_\mathtt{XC} \vdash \{[\![(\mathtt{p_C},\ \mathtt{g_C})]\!]\}\,\mathtt{fetch}:\ \mathbb{C}_\mathtt{VM}[\mathtt{fetch}].$$

where $\Psi_\mathtt{XC} = \{(\mathtt{fetch}, [\![(\mathtt{p_C}, \mathtt{g_C})]\!]), (\mathtt{f}, [\![(\mathtt{p_C}, \mathtt{g_C})]\!])\}$, `f` is the entry point in dispatch table for bytecode `c`. By the well-formedness of code heap module $\mathbb{C}_\mathtt{VM}[\mathtt{fetch}]$, we can conclude that the CertVM is a well-formed virtual machine.

Theorem 5 (Soundness Theorem). For all bytecode program $\mathbb{W}$ and x86 program $\mathbb{W}_\mathtt{X}$, if $\Psi \vdash \{\mathtt{s}\}\mathbb{W}$ and $\mathsf{WFVM}(\mathbb{W}, \mathbb{W}_\mathtt{X})$, there exists specification $\Psi_\mathtt{X}$ and assertion $\mathtt{s_X}$ such that $\Psi_\mathtt{X} \vdash \{\mathtt{s_X}\}\mathbb{W}_\mathtt{X}$.

Well-formed bytecode program guarantees that every instruction can be executed, while the CBP inference rules guarantee that the properties are still held in the new program state. With WFVM, we know that for every bytecode execution step, there is a well-formed x86 code heap. The SCAP logic guarantees that all well-formed x86 code heaps be linked into one single well-formed global one.

4 Example and Implementation

A factorial function implemented with while loop and non-local variables and its caller are shown in this section to demonstrate the particular features of our logic, and to show how to write specification and how to prove bytecode programs with CBP. Actually, the only work a programmer needs to do is to prove bytecode programs with CBP. Then this logic system guarantees that a well-formed bytecode program will runs on CertVM without getting stuck provided the x86 machine works.

4.1 Modular Certification: Factorial Function

Get Instruction Sequences. Factorial function source code and the bytecode program with its specifications for BCM are shown in Figure 1 (Section 1). Finding the instruction sequence is the first step to certify a program. From the definition in Figure 2, we know that an instruction sequence is a set of instructions ending with unconditional jump jmp or function return ret. Thus, it can be seen that there are three instruction sequences in while loop program. The instructions with labels 0∼2 form the first instruction block. And the second one is the instructions with label from 3 to 10. And the last one is the block of remain instructions.

Write Specification for Instruction Sequences. Then the programmer needs to give

```
//function caller  | -{(p16, g16)}    ;spec for caller
void caller(){     | 16 pushc 3       ;push imm 3
  int n=3;         | 17 pop n         ;n = 3
  call factor;     | 18 call 0        ;call factor()
}                  | -{(p19, g19)}    ;spec for return point
                   | 19 ret           ;caller return
```

$$
\begin{array}{llll}
\mathrm{p}'_{pre} \triangleq & (\mathtt{valid}\mathbb{K}\ 2\ \mathbb{K}) \wedge (\mathtt{valid}\mathbb{K}_c\ 1\ \mathbb{K}_c) \wedge (\mathtt{validRa}\ \ \mathbb{K}_c) & & \\
\mathrm{p}_{16} \triangleq & \mathrm{p}'_{pre} \wedge ((r \rightsquigarrow \_) * (n \rightsquigarrow \_)), & \mathrm{g}_{16} \triangleq & \mathrm{p}_{16} \rightarrow (\mathbb{H}'(\mathsf{r}) = 3!) \\
\mathrm{p}_{19} \triangleq & \mathrm{p}'_{pre} \wedge ((r \mapsto 3!) * n \mapsto 0)), & \mathrm{g}_{19} \triangleq & \mathrm{p}_{19} \rightarrow (\mathbb{H}'(\mathsf{r}) = 3!) \\
\mathrm{p}_{0} \triangleq & ?, & \mathrm{g}_{0} \triangleq & ?
\end{array}
$$

Fig. 17. Caller of Factorial Function

code heap specification Ψ, which is a finite mapping from code labels $\mathtt{f}$ to code specifications $\mathtt{s}$ which is a pair (p, g). CBP specifications for code heap are embedded in the code, enclosed by $\mathtt{-}\{\}$ in shadow box. Specifications of this example are given in Figure 16. To simplify our presentation, we write the predicate p in the form of a proposition with free variables referring to components of the state $\mathbb{S}$.

Following the inference rules, the code specifications should be given for these points: the head of a instruction sequence, the target labels of function call instruction call and jump instructions (including goto and brture), and the function call return address which is just after call instruction call.

The specification of the this procedure is given as $(\mathrm{p}_0, \mathrm{g}_0)$. From p_0, we know that the values of variables r and n stored in memory heap are inside the proper scope. The guarantee g_0 specifies the behavior of the function: the non-local variables r and n fulfill $(\mathbb{H}'(\mathsf{r}) = \mathbb{H}(\mathsf{n})!)$.

$(\mathrm{p}_3, \mathrm{g}_3)$ is the assertion for while loop body. The pre-condition p_3 means that the values of variables r and n are still inside the proper scope. The guarantee g_3 says that the result which is stored in memory heap must fulfill the loop fixpoint. The specification $(\mathrm{p}_{11}, \mathrm{g}_{11})$ at the begin point of this while loop is equal to $(\mathrm{p}_3, \mathrm{g}_3)$.

Certify and Link Them Together. To check the well-formedness of an instruction sequence beginning with ι, a programmer should apply the appropriate inference rules and find intermediate assertions such as $(\mathrm{p}', \mathrm{g}')$, which serves both as the post-condition for ι and as the pre-condition for the remaining instruction sequence.

After that, a programmer is also required to establish the well-formedness of each individual module via the CDHP rule. Two non-intersecting well-formed code heaps can then be linked together via the LINK rule. The WLD rule requires that all code heaps be linked into one single well-formed global one.

Support Modular Certification. All the code specifications Ψ used in CBP rules are the *local* specifications for the current module. Thus, CBP supports modular reasoning about function call/return in the sense that caller and callee can be in different modules and be certified separately. When specifying the callee procedure, we do not need any knowledge about the return address in its pre-condition. The RET rule for the instruction “ret” does not have any constraints on the return address.

4.2 Modular Certification: Caller of Factorial Function

Source code and bytecode program with specification of the caller for the while loop factorial example are shown in Figure 17.

This function just initializes the variables n, and then calls function factor. The specification at the entry point is $(\mathrm{p}_{16}, \mathrm{g}_{16})$. The pre-condition p_{16} simply says that the mem-

Component Name	Number of lines
Basic Utility Definitions & Lemmas	2,367
BCM Machine & Operational Semantics	3,285
CBP Rules & Soundness	1,032
X86 Machine & SCAP logic	2,710
CertVM Memory Layout & Proof	15,429
Bytecode Examples Source Code Spec. & Proof	1,469
Total	26,292

Fig. 18. The Verified Package in Coq

ory cells for variables n and r are there for this function to run. The guarantee g_{16} specifies the behavior of the caller procedure: the result r in memory heap is the factorial of 3. The specification of the return point is (p_{19}, g_{19}). p_{19} means that the memory cells for variables n and r are still there. The guarantee g_{19} is just the same as g_{16}.

From CAL inference rule, we know that the specification of the callee's entry point should be added. The specification (p_0, g_0) in Figure 16 can be used. Furthermore, the specification of function entry point defines its interface. A Caller can invoke any callees which share the same interface.

4.3 Implementation with Coq

Our logic system presented in this paper has been applied to bytecode programs for our verified stack-based virtual machine. We have formalized BCM, its operational semantics, and the program logic CBP. We have also formalized a X86 machine, its operational semantics, and the SCAP program logic for it in the Coq proof assistant. With SCAP logic, we proved the simulation relations of our virtual machine CertVM.

The syntax of our machine (both the bytecode machine and x86 machine), is encoded in Coq using inductive definitions. Operational semantics of the machine and all the inference rules of program logic are defined as inductive relations. The soundness of the framework itself is formalized and certified in Coq following the syntactic approach.

These examples are usually implemented directly in bytecode and are hard to certify using the existing approaches. Manually optimized bytecode or code generated by optimizing compilers can also be certified using our systems. The proof is also formalized and implemented in Coq and is machine-checkable.

The Coq implementation has taken several months per person, out of which a significant amount of efforts have been put on the implementation of basic facilities, including lemmas and tactics for partial mappings and Separation Logic assertions. These common facilities are independent of the task of certifying examples. The implementation of CBP logic system includes around 3200 lines of Coq encoding of BCM and its operational semantics, 1000 lines encoding of CBP rules and the soundness proof. We have written more than 15 thousand lines of Coq tactics to certify CertVM with SCAP logic. We also have written about 1500 lines of Coq tactics to certify practical bytecode examples, including the while-loop and function call/return.

It is found in our experience that human smartness still plays an important role to come up with proper program specifications, and the difficulty depends on the property one is interested in and the subtlety of the algorithms itself. Given proper specifications, proof construction of bytecode is mostly routine work. Some premises of inference rules can be automatically derived after defining lemmas for common instructions.

Compared the experiences in CBP with that in SCAP, we found that the code size ratios of bytecode programs to proofs and assembly code to proofs looks almost the same. While bytecode is a fairly compact format compared to native code. Most JVM instructions use only 1 or 2 bytes. Moreover, they are sophisticated instructions that cannot be translated into a single native processor instruction as a rule. In fact, our CertVM expand code size by the factor of 15, while most Java compilers expand code size by a factor of 5 to 10 [23]. With our logic, we only write proof for bytecode programs rather than write proof for the corresponding assembly code directly. So the workload will be greatly reduced by a factor of 5 to 10. That will be a significant improvement for fully certified subroutines with machine checkable proofs.

Extensions and Future Work. The support of object-oriented features such as objects, references, methods, and inheritance are important and useful. Extension of the program logic to support exception handling is straightforward and interesting work. Following the similar idea of function call/return, reasoning about exceptions is not much different from reasoning about functions. Our logic system does not support concurrency yet. There are a number of subtle problems even in the well-used bytecode programs such as JDK synchronized classes [19]. It is actually an easy work to extend the machine to support concurrency. But it is difficult to define a simple logic system to modularly certify concurrent bytecode programs. We will try it in the near future.

On the certified virtual machine, there are also some interesting extensions. Verification of the useful features such as memory management, just in-time compilation, garbage collectors will lead to some exciting challenges.

5 Related Works and Conclusion

Logic for Bytecode and Virtual Machine. Quigley [17] has demonstrated that it is possible to define a Hoare-style logic for bytecode programs to prove the program containing loops. A program logic [2] which combines Hoare triples for methods with instruction specifications is presented for a JAVA-like bytecode language by Bannwart and Müller. Their logic supports lots of object-oriented features such as objects, references, methods, and inheritance. Benton [3] proposed a typed, compositional logic for a stack-based abstract machine to verify bytecode programs which are written in an imperative subset of .NET CIL.

But, all these work only considered logic system for bytecode programs. None of them took the virtual machine into account. Linking certified bytecode programs with certified VM is very difficult. An open logic framework was designed to integrate [8] the proof of different logic systems for the X86 machine only. In this paper, we integrate the separated proof modules of different logic systems for different machine by simulation relation proof. To our best knowledge, our logic system is the first facility to link certified virtual machine with modularly certified bytecode programs.

Reasoning about Control Stacks. Reasoning about control stacks is extremely difficult for low-level code programs. STAL and its variations [22] can only treat return code pointers as first-class code pointers and stacks as "closures". Tan and Appel [21] use the implicit finite unions structure to study the low-level language. As a result, they arrived at continuation-style Hoare logic explainable by indexed model, with a rather convoluted interpretation of Hoare triples involving explicit fixpoint approximations. Saabas and Uustalu [18] introduced a compositional natural semantics and Hoare logic based on the implicit finite unions structure for a simple low-level language with expressions. Ni and Shao's work [16] combines the syntactic approach used in type systems with logic systems to support code pointer specification. Following the producer/consumer model, Feng *etc.* [9] proposed SCAP to modularly certify assembly code with stack-based control abstractions. Benton's typed, compositional logic for bytecode programs uses a higher-level abstract machine with separate data stack and control stack.

We build a Hoare-style logic system to certify bytecode programs which run on verified virtual machine. As the examples shown, program with complex control stack operations be certified within our logic. Our BCM is a higher-level machine with a dedicated function call stack. It looks like Benton's abstract machine. While our logic system CBP is established following SCAP's producer/consumer stack model. This idea brings much convenience to the integration of SCAP and CBP proof.

Certified Compiler and Interpreter. Large efforts have been made on building reliable compiler and interpreter with formal methods. League *etc.* built a type preserving JAVA compiler [12] and Chen *etc.* developed a type preserving optimizing compiler for MSIL [4]. Chlipala presented a certified compiler from the simply-typed lambda calculus to assembly language [5]. C0 compiler [13], a compiler from C subset language C0 to the DLX machine language, is formal specified and proved in Isabelle/HOL. The realistic and verified Compcert compiler [14], is developed and verified in Coq.

But all these work only focus on semantics preserving without well-formed properties of source programs. With the formalization and the certification of the simulation relation, our work gives a logic system to link the verified bytecode programs with the verified execution environment. It guarantees that a certified bytecode program runs on certified virtual machine will never get stuck as long as hardware works. It's an end-to-end solution and can be considered as a proof and semantics preserving compiler.

Conclusion. This paper presents a logic system to verify bytecode programs as well as a corresponding certified stack-based virtual machine. This paper defines a Hoare-style logic system for modularly specifying bytecode programs with complex stack-based control abstractions and unstructured control flows. The execution of bytecode programs is formalized via a design of an appropriate virtual machine, which is implemented in X86 assembly. The implementation of the virtual machine is verified in a previously published SCAP logic. We proved that for each bytecode program that is verified in the CBP logic, an equivalent X86 program which is in a simulation relation can be found. This equivalent program is well-formed in SCAP logic.

This approach might be powerful and simple enough to become usable in practice. To certify a bytecode program, a programmer's task is only required to find the specification and establish the well-formedness of individual bytecode module. This logic system guarantees that a certified bytecode program will run on the certified VM without getting stuck unless hardware faults occur. By our experiment, proving in CBP instead of directly in SCAP is expected to reduce the workload by a factor of 5-10.

Our work provides a logic system for reasoning about bytecode programs for stack-based virtual machine and makes an advance toward building a proof-transforming compilation environment. We believe this work may serve as a solid theoretical foundation to understand and reason about the popular and complex web applications.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE Computer Society, June 2001.

[2] F. Bannwart and P. Müller. A program logic for bytecode. In *Proceedings of Bytecode05, Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[3] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *In Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, pages 364–380. Springer-Verlag, 2005.

[4] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *Prog. Lang. Design and Impl. (PLDI'08)*, pages 183–192, New York, NY, USA, 2008. ACM.

[5] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Prog. Lang. Design and Impl. (PLDI'07)*, pages 54–65, New York, NY, USA, 2007. ACM.

[6] Coq Development Team. The Coq proof assistant reference manual. Version 8.2, 2008.

[7] ECMA. *Standard ECMA-335 Common Language Infrastructure*. 2006.

[8] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 Workshop on Types in Lang. Design and Impl.*, pages 67–78, 2007.

[9] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Prog. Lang. Design and Impl. (PLDI'06)*, pages 401–414, New York, NY, USA, June 2006. ACM Press.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 26(1):53–56, Oct. 1969.

[11] Kevin Lawton and Bryce Denney and N. David Guarneri and Volker Ruppert and Christophe Bothamy. Bochs user manual. http://bochs.sourceforge.net/, 2008.

[12] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving java compiler. In *In: Proc. Int'l. Conf. on Compiler Construction. (2003)*, pages 106–120. Springer, 2003.

[13] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a c0 compiler: Code generation and implementation correctnes. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 2–12, Washington, DC, USA, 2005. IEEE Computer Society.

[14] X. Leroy. A formally verified compiler back-end. *draft*, 2008. `http://pauillac.inria.fr/∼xleroy/publi/compcert-backend.pdf`.

[15] T. Lindholm and F. Yellin. The java virtual machine specification (second edition), 1999.

[16] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL'06*, pages 320–333, 2006.

[17] C. L. Quigley. A programming logic for java bytecode programs. In *Proc. of 16th Int. Conf. on Theorem Proving in Higher-Order Logics*, pages 41–54. Springer-Verlag, 2003.

[18] A. Saabas and T. Uustalu. Compositional type systems for stack-based low-level languages. In *Proc. of 12th Computing, Australasian Theory Symp.,*, pages 27–39. Australian, 2006.

[19] K. Sen. Race directed randomized dynamic analysis of concurrent programs. In *Proc. 2008 ACM Conf. on Prog. Lang. Design and Impl.*, pages 11–21. ACM Press, June 2008.

[20] Sun Microsystem. Top25 bugs. http://bugs.sun.com/bugdatabase/top25_bugs.do/, 2009.

[21] G. Tan and A. W. Appel. A compositional logic for control flow. In *VMCAI'06*, volume 3855 of *LNCS*, pages 80–94. Springer, 2006.

[22] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *Types in Lang. Design and Impl. (TLDI'03)*, pages 109–122, 2003.

[23] M. Weiss, F. de Ferrire, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler. Turboj, a java bytecode-to-native compiler. In *Proc. LCTES98*, volume 1474 of *LNCS*, pages 119–130. Springer-Verlag, 1998.