On the Certification of Thread Context Switching

Yu Guo, Xin-Yu Jiang, Yi-Yun Chen

Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, P.R.China
Software Security Laboratory, Suzhou Institute for Advanced Studym University of Science and Technology of China, Suzhou 215123, China

E-mail: {guoyu, wewewe}@mail.ustc.edu.cn; yiyun@ustc.edu.cn

Abstract With recent efforts to build foundational certified software systems, two different approaches have been proposed to certify thread context switching. One is to certify both threads and the context switching in a single logical framework, and the other certifies threads and the context switching at different abstraction levels. The former requires heavyweight extensions in the logical framework to support first-class code pointers and recursive specifications. Moreover, the specification for context switching is very complex. The later supports simpler and more natural specifications, but it requires the contexts of threads to be abstracted away completely when threads are certified. As a result, the conventional implementation of context switching used in most systems needs to be revised to make the abstraction work.
In this paper, we extend the second approach to certify the conventional implementation, where the clear abstraction for threads is unavailable since both threads and the context switching hold pointers of thread contexts. To solve this problem, we allow the program specifications for threads to refer to pointers of thread contexts. The thread contexts are treated as opaque structures, whose contents are unspecified and should never be accessed by the code of threads. Therefore, the advantage of avoiding the direct support of first-class code pointers is still preserved in our method. Besides, our new approach is also more lightweight. Instead of using two different logics to certify threads and the context switching, we employ only one program logic with two different specifications for the context switching. One is used to certify the implementation itself, and the more abstract one is used as an interface between threads and the context switching at a higher abstraction level. The consistency between the two specifications are enforced by the global program invariant.

Keywords program verification, context switching, proof-carrying code, program safety

1 Introduction

Thread context switching is an indispensable component of operating system kernel and thread implementations. Verification of its implementation is essential in building foundational certified software systems. However, implementation of context switching requires low-level operations such as manipulation of stack pointers and return code pointers (for each thread). This makes it difficult to accurately specify the code behaviors.

Generally speaking, context switching is the computing process of saving and restoring the state of a processor such that it can be shared by many tasks. Below we show the C interface of the `swapctxt` subroutine, which implements thread context switching.

```
void swapctxt(ucontext *old, ucontext *new)
```

Here `old` points to the structure used to save the current machine context (*i.e.,* values of registers), and `new` points to the new machine context structure to be loaded. Implementation of the subroutine is usually done with assembly language, which directly accesses machine registers. In Figure 1 we show an x86 implementation of the subroutine.

As shown in Figure 2, the arguments `old` and `new` are passed on stack (at `[esp+4]` and `[esp+8]` respectively).

```
swapctxt:
  ;; save old ctxt          ;; load new ctxt
  mov eax, [esp+4]          mov eax, [esp+8]
  mov [eax+0], 0            mov esp, [eax+28]
  mov [eax+4], ebx          mov ebp, [eax+24]
  mov [eax+8], ecx          mov edi, [eax+20]
  mov [eax+12], edx         mov esi, [eax+16]
  mov [eax+16], esi         mov edx, [eax+12]
  mov [eax+20], edi         mov ecx, [eax+8]
  mov [eax+24], ebp         mov ebx, [eax+4]
  mov [eax+28], esp         mov eax, [eax+0]
                            ret
```

Fig. 1. Implementation of Thread Context Switching

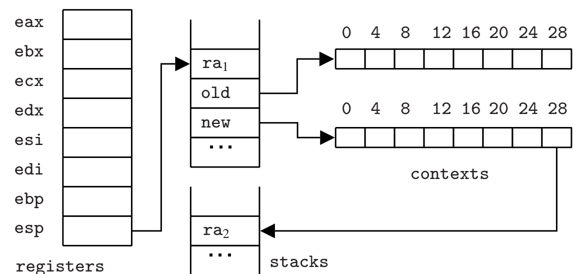

Fig. 2. Thread Context Switching

Supported by the National Natural Science Foundation of China under Grant No.60673126 and No. 90718026, China Postdoctoral Science Foundation (No. 20080430770) and Natural Science Foundation of Jiangsu Province, China (No. BK2008181).

```
0   threadA:              threadB:
1     mov  ebx, buf         mov  ebx, buf
2     mov  eax, 1           mov  eax, 0
3     mov  [ebx], eax       mov  [ebx], eax
4     mov  ebx, thrd        mov  ebx, thrd
5     mov  edx, [ebx+4]     mov  edx, [ebx+4]
6     mov  ecx, [ebx]       mov  ecx, [ebx]
7     mov  [ebx], edx       mov  [ebx], edx
8     mov  [ebx+4], ecx     mov  [ebx+4], ecx
9     push edx              push edx
10    push ecx              push ecx
11    call swapctxt         call swapctxt
12    add  esp, 8           add  esp, 8
13    jmp  threadA          jmp  threadB
```

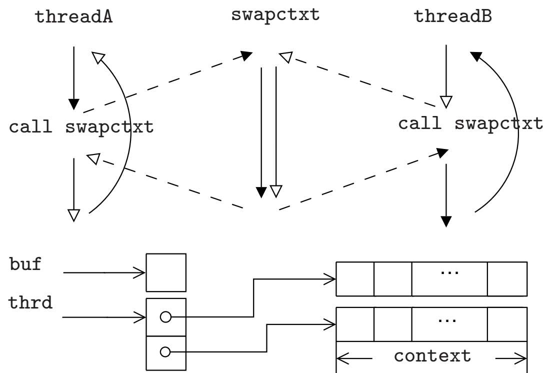

Fig. 3. Two Simple Threads

They point to the execution contexts of the *current* thread and the *target* thread to be switched to, respectively. The function saves all the registers except eax into the old context, and loads the new context into registers. It is important to note that the stack pointer esp is also changed. Therefore, when the function returns, it uses the return address ($\texttt{ra}_2$ shown in Figure 2) stored on the stack of the target thread instead of the current thread.

1.1 Challenges

We explain the challenges of certifying context switching code using a small example with two threads, as shown in Figure 3. Thread A sets a shared memory cell, pointed by buf, to 1 and then switches to thread B, which resets the memory cell to 0 and switches back to thread A. In the shared memory, there is also a simple thread queue pointed to by thrd. It contains two pointers pointing to the machine context structures of A and B, respectively (see Figure 3).

To modularly certify the program, that is, to certify each of the three parts (*i.e.,* thread A, B and swapctxt) without knowing the implementation details of the other two, the following three problems need to be addressed.

Manipulation of return code pointers When function returns, it needs the value stored on top of the stack as return address. The return address of a normal function is the same one passed to it by the caller. However, swapctxt changes the stack pointer. So when called at line 11 of thread A, swapctxt does not return to $\texttt{ra}_1$ shown in Figure 2, which points to the line 12 of thread A. Instead, it returns to $\texttt{ra}_2$, which points to the line 12 of *thread B*. To certify swapctxt, we need to prove that it always uses a valid return address.

Specifying the context switching The smart implementation of swapctxt gives threads the illusion that a function call of swapctxt returns back immediately. However, as shown in Figure 3, before swapctxt finally returns to the calling thread (*e.g.,* thread A), the control has been transferred to a different thread (thread B). This presents issues before us: Should the behavior of the thread B be viewed as part of the swapctxt as well? And how can swapctxt be specified without knowing in advance the thread it may switch to?

Abstraction of thread contexts Thread contexts are used in both threads and the context switching. Threads A and B in Figure 3 should be aware of the contexts and pass their pointers to swapctxt. This makes it difficult to build a proper abstraction for threads where the contexts are just part of the runtime support and should be invisible to programmers. We will explain more about the problem in the next section.

1.2 Previous approaches

Recently, two different approaches have been proposed to certify the context switching, which represent two different methodologies to build foundational certified software systems.

The first approach, proposed by Ni *et al* [1, 2], uses a very general and expressive logical framework to certify all the modules in the system, including threads and the context switching. They address the first problem by proposing a Hoare-style logic with the support of first-class code pointers (a.k.a. embedded code pointers) [3, 4]. By giving proper "types" to code pointers, their logic prevents any misuse of the code pointers. Despite its generality, it requires heavyweight extension of the assertion language to specify code pointers. To address the second problem, Ni *et al* [1, 2] applied polymorphic and recursive specifications to abstract away the data used by threads. As we will explain later in Section 6, this method yields complex program specifications that are hard to understand. Since they try to certify threads and the context switching at the same level and there is no abstraction for threads, the third problem doesn't exist here.

The second approach certifies threads and the context switching at different abstraction levels using different verification systems. At the higher abstraction level for threads, the concrete representation of their execution contexts and return code pointers in memory are abstracted away. They are modeled as mathematical structures accessed only by an abstract context switching operation. Therefore, the return code pointers are not first-class data at the high level. At the low level, we only need to ensure that the context switching saves registers into and loads new values from proper places, and then jumps to a code pointer stored on top of the stack of the target thread. Specification of context switching behavior

can be spared from the threads' point of view because threads are not certified at the low-level. After being certified, code at the two levels can be linked together either using a general foundational logic [5] or using simulation proofs [6] to get a fully certified system. Specifications of the context switching subroutine in this approach are simpler than those by Ni *et al*. They agree with the programmer's intuition naturally. However, to make the abstraction work, code at different abstraction levels must access disjoint parts of the memory. The previous works [7, 8, 6] following the second approach fail to address the third problem. They cannot certify `swapctxt` and the two threads in Figures 1 and 3, because both sides hold pointers of the thread contexts. Their context switching subroutines are implemented differently from `swapctxt`, and their threads cannot access the memory of contexts.

1.3 Our work and contributions

In this paper, we design a *lightweight* verification framework to certify both threads and the context switching. Our work follows the aforementioned second methodology, with extension to certify the `swapctxt` subroutine. In particular, we make the following new contributions in this paper.

- Our framework can certify `swapctxt` without changing its implementation and interface. To address the third problem mentioned in Section 1.1, program assertions for threads are enabled to refer to pointers of thread contexts. However, the contexts are treated as opaque structures whose contents are inaccessible from threads and are left unspecified. Therefore, the only way a thread can use the context pointers is to pass them to `swapctxt`. This effectively prevents updates of contexts from threads. So we can still build abstractions for threads and avoid supporting first-class code pointers.

- To abstract away the implementation details of thread contexts and the context switching, we give two different specifications for `swapctxt` at the thread abstraction level. One is at a lower abstraction level and specifies the concrete behavior of the subroutine. The other is at a higher level and is used as an abstract interface to threads. The gap between the two different specifications are bridged in the global program invariant. Both specifications are simpler and more natural to understand than those by Ni *et al*.

- Unlike previous works [7, 8], we use one set of rules to certify both the context switching and threads, instead of using two different program logics. This makes our framework more lightweight since we no longer need the OCAP framework [5] to link code at different abstraction levels.

- Since we can now certify `swapctxt`, we compare our specifications and proofs for `swapctxt` with those by Ni *et al* [1, 2], This is a direct comparison of the two representative methodologies to build foundational certified systems, *i.e.,* the one-logic-for-all approach versus the approach using multiple abstraction levels.

We have formalized the framework in the Coq proof assistant [9], including the machine model, the logic rules and the soundness proofs.

In the rest of this paper, we first introduce the basic settings in Section 2. We then informally describe the basic ideas of our approach in Section 3 and present the formal details of our framework in Section 4. We show how to certify code in our framework, report the verification results, and compare it with the work by Ni *et al* in Section 5. Lastly, we discuss more related work in Section 6 and conclude in Section 7.

2 Basic Settings

To illustrate the work clearly, we first present the basic settings, including meta language, simplified machine model and its operational semantics.

The mechanized meta-language First of all, a mechanized meta-language is required to formalize all the concepts in this paper, such as the machine model, programs, specifications, program logic rules, related theorems and proof. The meta-language we use is calculus of inductive constructions (CiC) [10] and supported by the Coq proof assistant [9], by which we implement all the work presented in this paper.

$$
\begin{array}{rl}
(\textit{Term})\ A,B ::= & \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} \mid X \mid \lambda X\!:\!A.B \mid A\ B \\
& \mid A \rightarrow B \mid \forall X\!:\!A.\ B \mid \exists X\!:\!A.\ B \\
& \mid \textit{inductive def.} \mid \ldots
\end{array}
$$

CiC is a typed lambda calculus, and its syntax, shown above, follows the conventions of common lambda calculi. For example, $A \rightarrow B$ represents function spaces. It also means logical implication when A and B have sort Prop. In addition, Prop is the universe of all propositions, Set is the universe of all data sets, and Type is the (stratified) universe of all terms.

The machine model In this paper, we model the x86 CPU in real-mode, and present the formal definitions of machine model in Figure 4. To better illustrate the main ideas, we omit some physical machine features, such as variable-length instruction encoding, bits-arithmetic, segmentation, *etc*. A machine configuration $\mathbb{M}$ contains a code heap $\mathbb{C}$, a mutable state $\mathbb{S}$, and an instruction pointer `ip`. A code heap $\mathbb{C}$ is a segment of memory mapping addresses to machine instructions. It is read-only and isolated from the mutable data heap $\mathbb{H}$. A machine state $\mathbb{S}$ consists of a general purpose register file, a mutable data heap $\mathbb{H}$ and a flags register $\mathbb{F}$, which only contains the zero flag for simplicity. A label `l` is a memory address which may point to either a data structure (in $\mathbb{H}$) or an instruction (sequence) (in $\mathbb{C}$). For maps, such as $\mathbb{H}$ and $\mathbb{C}$, notations of set operations are overloaded to specify the similar operations. For example, $\uplus$ specifies disjoint union, $\in$ is including, $\setminus$ is excluding, and $\emptyset$ is an empty set. $\mathbb{H}\{\mathtt{l} \rightsquigarrow \mathtt{w}\}$ means the new data heap updated at location `l` with `w`.

The basic machine instruction set covers the common x86 instructions for arithmetics, control transfer, conditional branch and data movement. It is easy to add more instructions to our abstract machine. It is worth noting that in our

machine model, every instruction is assumed one-word-size long.

We define an operation $\mathbb{C}[\mathtt{f}]$ to extract an instruction sequence starting from $\mathtt{f}$ in $\mathbb{C}$.

$$\mathbb{C}[\mathtt{f}] \triangleq \begin{cases} \mathbb{C}(\mathtt{f}) & \text{if } \mathbb{C}(\mathtt{f}) = \mathtt{jmp\ f}, \mathtt{call\ f}, \text{ or } \mathtt{ret} \\ \mathbb{C}(\mathtt{f}); \mathbb{C}[\mathtt{f}+1] & \text{otherwise} \end{cases}$$

Operational semantics The relation $\mathbb{M} \longmapsto \mathbb{M}'$ indicates that the machine configuration $\mathbb{M}$ steps to the machine configuration $\mathbb{M}'$. The relation $\mathbb{M} \longmapsto^k \mathbb{M}'$ means that $\mathbb{M}$ reaches $\mathbb{M}'$ in k steps, and $\longmapsto^*$ is the reflexive and transitive closure of the step relation.

$$\frac{\mathsf{NextS}_{(\mathtt{i},\mathtt{ip})}\ \mathbb{S}\ \mathbb{S}' \qquad \mathsf{NextIP}_{(\mathtt{i},\mathbb{S})}\ \mathtt{ip}\ \mathtt{ip}'}{(\mathbb{C},\mathbb{S},\mathtt{ip}) \longmapsto (\mathbb{C},\mathbb{S}',\mathtt{ip}')} \ (\text{EVAL})$$

The relation NextS specifies state transition of one machine execution step and NextIP the next program execution point. The formal details of NextS and NextIP are presented in Figure 4.

3 Our Approach

As mentioned in Section 1.3, our approach follows the second methodology, which is also used in SCAP [11] and previous work [7, 8]. In SCAP, a code specification consists of a precondition $\mathtt{p}$ and an action $\mathtt{g}$, rather than using the methodology reasoning about first-class code pointers,to support modular certification of function call, especially the last return instruction of a function. The SCAP logic maintains a global invariant to ensure the existence of a well-formed control stack in memory, and thus to guarantee the safety of function returns. Similarly, our certification framework maintains a global invariant to specify well-formed machine context structures in memory. As a result, the program logic only needs to certify that the programs always switch to these well-formed contexts.

We explain the basic ideas about the our framework and related the global invariant in the following Section 3.1, and discuss how we achieve the reasoning about embedded code pointers in Section 3.2.

3.1 Basic ideas

Memory partition To support modular reasoning, the whole memory $\mathbb{H}$ is logically divided into several parts: the private memory of each thread $\mathbb{H}_0,\ldots,\mathbb{H}_n$, the shared memory $\mathbb{H}_s$ and the memory $\mathbb{H}_c$ storing the data of all the machine contexts.

$$\mathbb{H} = (\mathbb{H}_0 \uplus \cdots \uplus \mathbb{H}_n) \uplus \mathbb{H}_s \uplus \mathbb{H}_c$$

As illustrated in Figure 5, we define different memory invariants over these partitions. The private memory of current thread $\mathbb{H}_i$ merged with the shared memory $\mathbb{H}_s$, $\mathbb{H}_i \uplus \mathbb{H}_s$, satisfies the predicate of well-formed thread WFThread(defined in Figure 6). The private memory of each ready-thread satisfies the predicate of well-formed ready thread WFRdyThrd. The memory $\mathbb{H}_c$ is separated from others, so that the machine context data of all threads is hidden from thread code and kept valid.

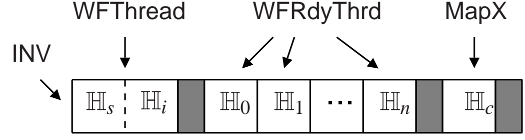

Fig. 5. Context Switching

Two specifications for context switch In our framework, two specifications are given to the context switching, one for certifying the implementation code of context switching and the other for certifying the code of threads, so to realize certification modularity.

Take the implementation of context switching in Section 1 as an example, it is obvious that the code to store the registers of CPU to old context structure and restores the registers from the new context structure won't affect the validity of the new and the old context pointers. A simple specification of context switching (θ^G_{sw}) can be given without any information about the embedded code pointer stored in the new context structure. Therefore, the first-class code pointers problem occurring in certifying the context switching implementation can be avoided and the certification would be largely simplified.

Given the separated thread private memory, the thread-local specification of context switching θ^L_{sw} will be easy to define, specifying that the private memory of a thread will be unchanged during a context switch.

Machine context abstraction The locations of those machine context structures will be used by thread code to save and restore contexts, but the data stored in machine contexts should always be valid. We introduce two new notations, $\mathbb{X}$ to specify the data of all the machine contexts, and X to specify the locations of all the machine contexts. A machine context $\mathtt{mc}$ contains eight general purpose registers in $\mathbb{R}$.

$$\begin{array}{lrcl} (\mathsf{MCtxSet}) & \mathbb{X} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{mc}\}^* \\ (\mathsf{MCtxLabelSet}) & X & ::= & \{\mathtt{l}\}^* \\ (\mathsf{MCtx}) & \mathtt{mc} & ::= & \langle\mathbb{R}\rangle \end{array}$$

We let code specifications be parameterized by X and put these data in shared memory $\mathbb{H}_s$. In this way, even though the thread code can get and pass the locations of context structures to context switching function, they cannot modify the data of machine contexts. Similarly, we introduce a variable t, which indicates the location of the current thread machine context, as a parameter of the specifications, just like X. The predicate MapX (in Section 4.1) is to ensure that $\mathbb{X}$ is consistent with the data in the memory partition $\mathbb{H}_c$

In this way, the specifications in our verification framework will be like:

$$\begin{array}{l} \mathtt{p} \triangleq \lambda t, X, \mathbb{S}.\cdots \\ \mathtt{g} \triangleq \lambda t, X, X', \mathbb{S}, \mathbb{S}'.\cdots \end{array}$$

When certifying, the old context pointer passed to context switching function should be equal to the logical value of t, while the new context pointer should belong to X and be unequal to t.

The locations of machine context structures are stored in the shared memory $\mathbb{H}_s$, so that they may be accessed by

$$
\begin{array}{rlll}
\text{(Machine)} & \mathbb{M} & ::= & (\mathbb{C}, \mathbb{S}, \mathtt{ip}) \\
\text{(CodeHeap)} & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{i}\}^* \\
\text{(State)} & \mathbb{S} & ::= & (\mathbb{H}, \mathbb{R}, \mathbb{F}) \\
\text{(DataHeap)} & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \quad (\mathtt{l} = 4{*}n) \\
\text{(RegFile)} & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
\text{(Register)} & \mathtt{r} & ::= & \mathtt{eax} \mid \mathtt{ecx} \mid \mathtt{edx} \mid \mathtt{ebx} \\
 & & \mid & \mathtt{esp} \mid \mathtt{ebp} \mid \mathtt{esi} \mid \mathtt{edi} \\
\text{(FlagReg)} & \mathbb{F} & ::= & \{\mathsf{zf}\} \\
\text{(ZeroFlag)} & \mathsf{zf} & ::= & \mathsf{zd} \mid \mathsf{ze} \\
\text{(Word)} & \mathtt{w} & ::= & i \ \ (\textit{integers}) \\
\text{(Label)} & \mathtt{f}, \mathtt{l}, \mathtt{ip} & ::= & \mathtt{w} \\
\text{(InstrSeq)} & \mathbb{I} & ::= & \mathtt{i} \mid \mathtt{i}; \mathbb{I} \\
\text{(Instruction)} & \mathtt{i} & ::= & \mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s \mid \mathtt{sub}\ \mathtt{r}_d,\ \mathtt{r}_s \mid \mathtt{cmp}\ \mathtt{r}',\ \mathtt{r} \\
 & & \mid & \mathtt{mov}\ \mathtt{r}_d,\ \mathtt{r}_s \mid \mathtt{mov}\ \mathtt{r}_d,\ \mathtt{w} \\
 & & \mid & \mathtt{mov}\ \mathtt{r}_d,\ [\mathtt{r}_s{+}\mathtt{w}] \mid \mathtt{mov}\ [\mathtt{r}_d{+}\mathtt{w}],\ \mathtt{r}_s \\
 & & \mid & \mathtt{jmp}\ \mathtt{f} \mid \mathtt{call}\ \mathtt{f} \mid \mathtt{ret} \\
 & & \mid & \mathtt{push}\ \mathtt{r}_s \mid \mathtt{push}\ \mathtt{w} \mid \mathtt{pop}\ \mathtt{r}_d
\end{array}
$$

$\mathsf{NextIP}_{(\mathtt{i},(\mathbb{H},\mathbb{R},\{\mathsf{zf}\}))}\ \mathtt{ip}\ \mathtt{ip}'$	
if i =	$\mathtt{ip}' =$
je f	f if zf = ze
je f	ip + 1 if zf = zd
call f	f
jmp f	f
ret	f if $\mathtt{f} = \mathbb{H}(\mathbb{R}(\mathtt{esp}))$
other cases	ip + 1

$\mathsf{NextS}_{(\mathtt{i},\mathtt{ip})}\ \mathbb{S}\ \mathbb{S}'$ where $\mathbb{S} = (\mathbb{H}, \mathbb{R}, \mathbb{F})$	
if i =	$\mathbb{S}' =$
add $\mathtt{r}_d$, $\mathtt{r}_s$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow v\}, \{\mathsf{ze}\})$ if $v = 0$, where $v = \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_d)$ $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow v\}, \{\mathsf{zd}\})$ if $v \neq 0$, where $v = \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_d)$
sub $\mathtt{r}_d$, $\mathtt{r}_s$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow v\}, \{\mathsf{ze}\})$ if $v = 0$, where $v = \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_d)$ $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow v\}, \{\mathsf{zd}\})$ if $v \neq 0$, where $v = \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_d)$
cmp $\mathtt{r}'$, r	$(\mathbb{H}, \mathbb{R}, \{\mathsf{ze}\})$ if $\mathbb{R}(\mathtt{r}) = \mathbb{R}(\mathtt{r}')$ $(\mathbb{H}, \mathbb{R}, \{\mathsf{zd}\})$ if $\mathbb{R}(\mathtt{r}) \neq \mathbb{R}(\mathtt{r}')$
mov $\mathtt{r}_d$, $\mathtt{r}_s$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{F})$
mov $\mathtt{r}_d$, w	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{w}\}, \mathbb{F})$
mov $\mathtt{r}_d$, $[\mathtt{r}_s + \mathtt{w}]$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + \mathtt{w})\}, \mathbb{F})$ if $(\mathbb{R}(\mathtt{r}_s) + \mathtt{w}) \in \mathrm{dom}(\mathbb{H})$
mov $[\mathtt{r}_d + \mathtt{w}]$, $\mathtt{r}_s$	$(\mathbb{H}\{(\mathbb{R}(\mathtt{r}_d) + \mathtt{w}) \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{R}, \mathbb{F})$ if $(\mathbb{R}(\mathtt{r}_d) + \mathtt{w}) \in \mathrm{dom}(\mathbb{H})$
push $\mathtt{r}_s$	$(\mathbb{H}\{\mathbb{R}(\mathtt{esp}) - 4 \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{R}\{\mathtt{esp} \rightsquigarrow \mathbb{R}(\mathtt{esp}) - 4\}, \mathbb{F})$ if $\mathbb{R}(\mathtt{esp}) - 4 \in \mathrm{dom}(\mathbb{H})$
push w	$(\mathbb{H}\{\mathbb{R}(\mathtt{esp}) - 4 \rightsquigarrow \mathtt{w}\}, \mathbb{R}\{\mathtt{esp} \rightsquigarrow \mathbb{R}(\mathtt{esp}) - 4\}, \mathbb{F})$ if $\mathbb{R}(\mathtt{esp}) - 4 \in \mathrm{dom}(\mathbb{H})$
pop $\mathtt{r}_d$	$(\mathbb{H}, \mathbb{R}\{\mathtt{esp} \rightsquigarrow \mathbb{R}(\mathtt{esp}) + 4, \mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{esp}))\}, \mathbb{F})$ if $\mathbb{R}(\mathtt{esp}) \in \mathrm{dom}(\mathbb{H})$
call f	$(\mathbb{H}\{(\mathbb{R}(\mathtt{esp})) \rightsquigarrow (\mathtt{ip} + 1)\}, \mathbb{R}\{\mathtt{esp} \rightsquigarrow (\mathbb{R}(\mathtt{esp}) - 4)\}, \mathbb{F})$
ret	$(\mathbb{H}, \mathbb{R}\{\mathtt{esp} \rightsquigarrow (\mathbb{R}(\mathtt{esp}) + 4)\}, \mathbb{F})$ if $\mathbb{R}(\mathtt{esp}) \in \mathrm{dom}(\mathbb{H})$
other cases	$\mathbb{S}$

Fig. 4. The simplified x86 machine model

threads. To ensure these locations are consistent with the structures in $\mathbb{H}_c$, the invariant of shared memory INV is parameterized by t and X. Thus, only part information of threads are exposed to each others via the shared memory.

Tagged specifications The specifications are parameterized by three arguments, t, X and $\mathbb{S}$. The specifications for thread code will be passed different arguments from those for context switching. Since θ^G specifies the global semantics of the context switching, $\mathbb{S}$ passed to the code specification of context switching is the global machine state while t and X can be any values. On the other hand, code of threads only manipulates partial memory, thus $\mathbb{S}$ passed to the code specification of threads is a partial state configuration. We distinguish these two kinds of specifications by tags. The global specification θ^G_{sw} and thread-local specification θ^L_{sw} are tagged by G and L, respectively.

Tagging specifications with the global and local tags allows us to present the key ideas of our paper without using the OCAP framework. We use a single set of inference rules for reasoning both thread code and context code.

3.2 Discussion

To ensure the validity of embedded code pointers stored in machine context structures, we firstly disallow the modification of them via memory partition. Then we abstract machine context locations as X and t and pass them to thread specifications. Our certification framework keeps a global invariant which states that all the embedded code pointers in the machine context structures are valid. Thus, we can certify the safety of a context switch operation once we know the switch destination belongs to X but t.

Meanwhile, the semantics of context switching are specified by two different specifications, both of which have nothing to do with the validity of embedded code pointers in machine context structures. Therefore, our approach avoids reasoning about first-class code pointers and results in a more lightweight certification framework than Ni's work. Our certification framework still supports modularity of certification. A more detailed comparison and some experimental results will be given in Section 5.2.

c

4 Formal Verification Framework

In this section, we present the formal details of our verification framework, covering specification language, inference rules of the program logic, and the soundness proof of the whole framework.

4.1 Specification

Specifications describe the predicted behaviors of the code. To certify code, programmers need to give a set of specifications Ψ, which is a finite map from code labels $\mathtt{f}$ to code specifications θ. In our framework, a code specification θ is a triple, consisting of a precondition $\mathtt{p}$, an action $\mathtt{g}$ and a tag $\mathtt{b}$. The precondition $\mathtt{p}$ and action $\mathtt{g}$ are like the notations used

in original SCAP [11] but they have two extra arguments, t and X, as we explained in Section 3.

$$
\begin{array}{rrcl}
\text{(SpecSet)} & \Psi & ::= & \{\mathtt{l} \rightsquigarrow \theta\}^* \\
\text{(Spec)} & \theta & ::= & (\mathtt{p},\mathtt{g},\mathtt{b}) \\
\text{(Pred)} & \mathtt{p} & : & \mathsf{Label} \to \mathsf{MCtxLabelSet} \to \mathsf{State} \to \mathsf{Prop} \\
\text{(Grt)} & \mathtt{g} & : & \mathsf{Label} \to \mathsf{MCtxLabelSet} \to \mathsf{MCtxLabelSet} \\
 & & & \quad \to \mathsf{State} \to \mathsf{State} \to \mathsf{Prop} \\
\text{(Tag)} & \mathtt{b} & ::= & G \mid L
\end{array}
$$

In a code specification θ, the precondition $\mathtt{p}$ specifies the precondition before machine executes an instruction. The action $\mathtt{g}$ is a state transition relation, which specifies the actions performed by some instructions. Also, the action $\mathtt{g}$ can be viewed as a predicate composed of a precondition and a postcondition, and relating the machine state of current point and the machine state of return point. In other words, $\mathtt{g}$ specifies the remaining actions of the current function before return. We use a specification tag $\mathtt{b}$ to distinguish the code of threads (L) and context switching implementation (G).

Separation logic notations We define some notations commonly seen in separation logic [4, 12]. These notations are formalized as shallow embedding in the meta-language CiC.

$$
\begin{array}{rcl}
\mathbb{H} \Vdash A & \triangleq & A\,\mathbb{H} \\
\mathsf{Top} & \triangleq & \lambda\mathbb{H}.\,\mathsf{True} \\
!P & \triangleq & \lambda\mathbb{H}.\,\mathbb{H}=\emptyset \wedge P \\
\mathtt{l} \mapsto \mathtt{w} & \triangleq & \lambda\mathbb{H}.\,\mathtt{l} \neq \mathsf{NULL} \wedge l=4n \wedge \mathbb{H}=\{\mathtt{l}\rightsquigarrow\mathtt{w}\} \\
\mathtt{l} \mapsto _ & \triangleq & \lambda\mathbb{H}.\,\exists\mathtt{w}.(\mathbb{H} \Vdash \mathtt{l} \mapsto \mathtt{w}) \\
A_1 * A_2 & \triangleq & \lambda\mathbb{H}.\,\exists\mathbb{H}_1,\mathbb{H}_2.\,\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H} \wedge (\mathbb{H}_1 \Vdash A_1) \wedge (\mathbb{H}_2 \Vdash A_2) \\
\mathtt{l} \hookrightarrow \mathtt{w} & \triangleq & \mathtt{l} \mapsto \mathtt{w} * \mathsf{Top} \\
A_1 -\!\!* A_2 & \triangleq & \lambda\mathbb{H}_2.\,\forall\mathbb{H}_1,\mathbb{H}.\,\mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H} \wedge (\mathbb{H}_1 \Vdash A_1) \to (\mathbb{H} \Vdash A_2) \\
\mathtt{l} \mapsto \mathtt{w}_0,\ldots,\mathtt{w}_n & \triangleq & \mathtt{l} \mapsto \mathtt{w}_1 * \mathtt{l}{+}4 \mapsto \mathtt{w}_2 * \ldots * \mathtt{l}{+}4n \mapsto \mathtt{w}_n
\end{array}
$$

We use $\mathbb{H} \Vdash A$ if the heap predicate A is valid with $\mathbb{H}$. $A_1 * A_2$ asserts the heap $\mathbb{H}_1 \uplus \mathbb{H}_2$ in which $\mathbb{H}_1 \Vdash A_1$ and $\mathbb{H}_2 \Vdash A_2$ hold. Top is valid with any heap. $\mathtt{l} \mapsto \mathtt{w}$ asserts a heap with only one memory cell, at address $\mathtt{l}$ with content $\mathtt{w}$.

A machine context structure is 8-word-size and contains eight general purpose registers.

$$
\begin{array}{rcl}
l \overset{\mathsf{mctx}}{\longmapsto} \langle\mathbb{R}\rangle & \triangleq & \lambda\mathbb{H}.\,\mathbb{H} \Vdash l \mapsto \mathbb{R}(\mathtt{eax}),\mathbb{R}(\mathtt{ebx}),\ldots,\mathbb{R}(\mathtt{esp}) \\
\mathsf{MapX}(\mathbb{H}_c,\mathbb{X}) & \triangleq & \mathbb{X} = \{l_0 \rightsquigarrow \mathtt{mc}_0,\ldots,l_n \rightsquigarrow \mathtt{mc}_n\} \\
 & & \wedge\, \mathbb{H}_c \Vdash l_0 \overset{\mathsf{mctx}}{\longmapsto} \mathtt{mc}_0 * \cdots * l_n \overset{\mathsf{mctx}}{\longmapsto} \mathtt{mc}_n
\end{array}
$$

The predicate MapX relates an abstract machine context set $\mathbb{X}$ with a block of memory $\mathbb{H}_c$. $\mathbb{X}$ can be regarded as the data dumped from $\mathbb{H}_c$.

Invariant of shared memory The invariant INV is a predicate over shared memory, but it is an abstract variable in our framework and can be instantiated to any *precise* memory predicate. The definition of precise memory predicates is given below:

$$
\begin{array}{rcl}
\mathsf{INV} & : & \mathsf{Label} \to \mathsf{MctxLabelSet} \to \mathsf{DataHeap} \to \mathsf{Prop} \\
\mathsf{Precise}(A) & \triangleq & \forall\mathbb{H}_1,\mathbb{H}_2,\mathbb{H}.\,\mathbb{H}_1 \subseteq \mathbb{H} \wedge \mathbb{H}_2 \subseteq \mathbb{H} \\
 & & \wedge (\mathbb{H}_1 \Vdash A) \wedge (\mathbb{H}_2 \Vdash A) \to \mathbb{H}_1 = \mathbb{H}_2
\end{array}
$$

Note that INV is parameterized by t and X. Thus, the locations of all machine contexts can be stored in the shared memory and visited by threads.

Specifications of the context switch Two different specifications are given for context switching, as shown below: θ^G_{sw} and θ^L_{sw}.

$$
\begin{array}{rcl}
\theta^G_{\mathsf{sw}} & \triangleq & (\mathtt{p}^G_{\mathsf{sw}},\mathtt{g}^G_{\mathsf{sw}},G) \\
\mathtt{p}^G_{\mathsf{sw}} & \triangleq & \lambda t,X,(\mathbb{H},\mathbb{R},_).\,\exists\mathbb{R}',old,new. \\
 & & \mathbb{H} \Vdash old \overset{\mathsf{mctx}}{\longmapsto} \langle_\rangle * new \overset{\mathsf{mctx}}{\longmapsto} \langle\mathbb{R}'\rangle * \mathsf{Top} \\
 & & \quad * \mathbb{R}(\mathtt{esp}) \hookrightarrow _,old,new * \mathbb{R}'(\mathtt{esp}) \hookrightarrow _ \\
\mathtt{g}^G_{\mathsf{sw}} & \triangleq & \lambda t,X,X',(\mathbb{H},\mathbb{R},_),(\mathbb{H}',\mathbb{R}',_).\,\exists old,new,ip,ip'. \\
 & & \wedge \left\{ \begin{array}{l} old \overset{\mathsf{mctx}}{\longmapsto} \langle_\rangle * new \overset{\mathsf{mctx}}{\longmapsto} \langle\mathbb{R}'\rangle \\ \quad * \mathbb{R}(\mathtt{esp}) \mapsto ip,old,new * \mathbb{R}'(\mathtt{esp}) \mapsto ip' \\ old \overset{\mathsf{mctx}}{\longmapsto} \langle\mathbb{R}\{\mathtt{eax} \rightsquigarrow 0\}\rangle * new \overset{\mathsf{mctx}}{\longmapsto} \langle\mathbb{R}'\rangle \\ \quad * \mathbb{R}(\mathtt{esp}) \mapsto ip,old,new * \mathbb{R}'(\mathtt{esp}) \mapsto ip' \end{array} \right\} \mathbb{H}\;\mathbb{H}'
\end{array}
$$

The thread-local specification θ^G_{sw}, defined as $(\mathtt{p}^G_{\mathsf{sw}},\mathtt{g}^G_{\mathsf{sw}},G)$, specifies the global semantics of the context switching and is tagged by G. The precondition $\mathtt{p}^G_{\mathsf{sw}}$ requires two machine context structures pointed by *old* and *new*, which are stored on stack. The notation _ stands for the value that we don't care about, and actually, it is an existentially quantified value. The action of the context switching $\mathtt{g}^G_{\mathsf{sw}}$ requires that: the register file of post-state $\mathbb{R}'$ be equal to the values saved in the context structure in the pre-state; the return address stored on the stack of post-state be exactly the value saved in the context structure in pre-state; the register file $\mathbb{R}$ and the return address $\mathtt{ip}$ of pre-state be saved to context structures in post-state.

The memory relation of the form $\left\{ \begin{array}{l} \mathtt{p}_1 \\ \mathtt{p}_2 \end{array} \right\}$ is defined as below:

$$
\left\{ \begin{array}{l} \mathtt{p}_1 \\ \mathtt{p}_2 \end{array} \right\} \triangleq \lambda\mathbb{H}_1,\mathbb{H}_2.\,\exists\mathbb{H}'_1,\mathbb{H}'_2,\mathbb{H}'. \\
(\mathtt{p}_1\;\mathbb{H}'_1) \wedge (\mathtt{p}_2\;\mathbb{H}'_2) \wedge (\mathbb{H}'_1 \uplus \mathbb{H}' = \mathbb{H}_1) \wedge (\mathbb{H}'_2 \uplus \mathbb{H}' = \mathbb{H}_2)
$$

It means that the memory block $\mathbb{H}_1$ satisfies $\mathtt{p}_1$ and is transformed to the block satisfying $\mathtt{p}_2$, while the rest part of memory block is unchanged.

$$
\begin{array}{rcl}
\theta^L_{\mathsf{sw}} & \triangleq & (\mathtt{p}^L_{\mathsf{sw}},\mathtt{g}^L_{\mathsf{sw}},L) \\
\mathtt{p}^L_{\mathsf{sw}} & \triangleq & \lambda t,X,(\mathbb{H},\mathbb{R},_).\,\exists old,new.\,old=t \wedge new \neq t \wedge new \in X \\
 & & \wedge\,\mathbb{H} \Vdash \mathsf{INV}\;t\;X * \mathbb{R}(\mathtt{esp}) \hookrightarrow _,old,new \\
\mathtt{g}^L_{\mathsf{sw}} & \triangleq & \lambda t,X,X',(\mathbb{H},\mathbb{R},_),(\mathbb{H}',\mathbb{R}',_).\,\exists old,new,ip. \\
 & & \wedge\,\mathbb{R}\{\mathtt{eax} \rightsquigarrow 0\} = \mathbb{R}'\{\mathtt{eax} \rightsquigarrow 0\} \\
 & & \wedge \left\{ \begin{array}{l} \mathsf{INV}\;new\;X * \mathbb{R}(\mathtt{esp}) \mapsto ip,old,new \\ \mathsf{INV}\;old\;X' * \mathbb{R}'(\mathtt{esp}) \mapsto ip,old,new \end{array} \right\} \mathbb{H}\;\mathbb{H}' \wedge \mathcal{R}\;t\;X\;X' \\
\mathcal{R} & \triangleq & \lambda t,X,X'.\,X=X'
\end{array}
$$

The thread-local specification of context switching, θ^L_{sw}, is used to certify the code of threads and tagged by L. The precondition $\mathtt{p}^L_{\mathsf{sw}}$ requires that: the shared memory satisfy the invariant $\mathsf{INV}\;t\;X$; there be three values on the top of stack, return address *ip*, context structure pointers *old* and *new*; *old* be equal to t; and the value in *new* belong to X but be not equal to t. The action $\mathtt{g}^L_{\mathsf{sw}}$ requires that: shared memory change from INV *new X* to INV *old X*; the registers file be unchanged across context switching; the values on stack be unchanged; and machine context labels satisfy the relation $\mathcal{R}$. In this paper, since we only consider the context switching, the machine context label set X will be unchanged always. But it is easy to enrich our framework with context loading, context making and context saving support by specifying the machine context changing in the relation $\mathcal{R}$.

As explained in Section 3, these two specifications for the context switching play important roles in our verification framework and make it possible to certify context switching in a modular and lightweight way.

4.2 Inference rules

A set of inference rules is used to prove the judgements for well-formed machines, code heaps, and instruction sequences. The rules are presented in Figure 6.

Auxiliary definitions The following auxiliary definitions are used in inference rules. The notation $\widehat{\mathbb{X}}$ is a short synonym for the domain set of $\mathbb{X}$. The action g_{fun} is the requirement of the stack balance during function calls. The notation g_{id} specifies the identity state transition. We wrap the state transition function $\mathsf{NextS}_{(\mathtt{i},\mathtt{ip})}$ as an action $g_{(\mathtt{i},\mathtt{ip})}$. The notation $g_{\{\mathsf{zf}\}}$ is an action that specifies an identity state transition with the flag zf. The relation retaddr takes two arguments, a label f and a state $\mathbb{S}$, and requires that the value on the top of stack $\mathbb{H}(\mathbb{R}(\mathtt{esp}))$ is equal to the label f.

$$
\begin{array}{lll}
\widehat{\mathbb{X}} & \triangleq & \mathrm{dom}(\mathbb{X}) \\
g_{\mathsf{fun}} & \triangleq & \lambda t,X,X',\mathbb{S},\mathbb{S}'.\,\mathbb{R}(\mathtt{esp}) = \mathbb{R}'(\mathtt{esp}) \\
 & & \quad \wedge \mathbb{H}(\mathbb{R}(\mathtt{esp})) = \mathbb{H}'(\mathbb{R}'(\mathtt{esp})) \\
g_{\mathsf{id}} & \triangleq & \lambda t,X,X',\mathbb{S},\mathbb{S}'.\,X = X' \wedge \mathbb{S} = \mathbb{S}' \\
g_{(\mathtt{i},\mathtt{ip})} & \triangleq & \lambda t,X,X',\mathbb{S},\mathbb{S}'.\,\mathsf{NextS}_{(\mathtt{i},\mathtt{ip})}\,\mathbb{S}\,\mathbb{S}' \wedge X = X' \\
g_{\{\mathsf{zf}\}} & \triangleq & \lambda t,X,X',\mathbb{S},\mathbb{S}'.\,\mathbb{S}' = (\mathbb{S}.\mathbb{H},\mathbb{S}.\mathbb{R},\{\mathsf{zf}\}) \wedge X = X' \\
\mathsf{retaddr} & \triangleq & \lambda(\mathtt{f},\mathbb{S}).\,\exists(\mathbb{H},\mathbb{R},\mathbb{F}).\,\mathbb{S} = (\mathbb{H},\mathbb{R},\mathbb{F}) \wedge \mathbb{H}(\mathbb{R}(\mathtt{esp})) = \mathtt{f}
\end{array}
$$

Combination of p and g Some combination operations over predicates p and g are defined to simplify the inference rules. The term $\mathsf{enable}(\mathtt{p},\mathtt{g})$ means that the precondition p implies the precondition within g. The combination of $\mathtt{p} \triangleright \mathtt{g}$ is a predicate which specifies the post-state described by g. The term $\mathtt{g} \circ \mathtt{g}'$ is a composed action. The term $\mathtt{p} \circ \mathtt{g}$ is the action whose precondition is strengthened by p. The term $\mathtt{p} \Rightarrow \mathtt{p}'$ means that the precondition p implies p′. The term $\mathtt{g} \Rightarrow \mathtt{g}'$ means that the action g implies g′.

$$
\begin{array}{lll}
\mathsf{enable}(\mathtt{p},\mathtt{g}) & \triangleq & \forall t,X,\mathbb{S}.\,\mathtt{p}\;t\;X\;\mathbb{S} \rightarrow \exists X',\mathbb{S}'.\,\mathtt{g}\;t\;X\;X'\;\mathbb{S}\;\mathbb{S}' \\
\mathtt{p} \triangleright \mathtt{g} & \triangleq & \lambda t,X,\mathbb{S}.\,\exists X_0,\mathbb{S}_0.\,\mathtt{p}\;t\;X_0\;\mathbb{S}_0 \wedge \mathtt{g}\;t\;X_0\;X\;\mathbb{S}_0\;\mathbb{S} \\
\mathtt{g} \circ \mathtt{g}' & \triangleq & \lambda t,X,X'',\mathbb{S},\mathbb{S}''.\,\exists X',\mathbb{S}'.\,\mathtt{g}\;t\;X\;X'\;\mathbb{S}\;\mathbb{S}' \\
 & & \quad \wedge \mathtt{g}'\;t\;X'\;X''\;\mathbb{S}'\;\mathbb{S}'' \\
\mathtt{p} \circ \mathtt{g} & \triangleq & \lambda t,X,X',\mathbb{S},\mathbb{S}'.\,\mathtt{p}\;t\;X\;\mathbb{S} \wedge \mathtt{g}\;t\;X\;X'\;\mathbb{S}\;\mathbb{S}' \\
\mathtt{p} \Rightarrow \mathtt{p}' & \triangleq & \forall t,X,\mathbb{S}.\,\mathtt{p}\;t\;X\;\mathbb{S} \rightarrow \mathtt{p}'\;t\;X\;\mathbb{S} \\
\mathtt{g} \Rightarrow \mathtt{g}' & \triangleq & \forall t,X,X',\mathbb{S},\mathbb{S}'.\,\mathtt{g}\;t\;X\;X'\;\mathbb{S}\;\mathbb{S}' \rightarrow \mathtt{g}'\;t\;X\;X'\;\mathbb{S}\;\mathbb{S}'
\end{array}
$$

Well-formed machine A machine configuration is well-formed if there exists a specification θ such that : the imported specifications set Ψ is the subset of the exported specifications set Ψ' by code $\mathbb{C}$; the code heap satisfies the exported specifications set Ψ' with the imported specifications set Ψ; the current state of machine $\mathbb{S}$ is well-formed with respect to the specification θ and the imported Ψ; and the current instruction sequence $\mathbb{C}[\mathtt{ip}]$, from the label ip, is well-formed with respect to θ.

Well-formed code heap A code heap $\mathbb{C}$ is well-formed with an exported specifications set Ψ' and an imported specifications set Ψ, if for every code label f in the domain set of Ψ', the instruction sequence $\mathbb{C}[\mathtt{f}]$ is well-formed with respect to $\Psi'(\mathtt{f})$. Moreover, the global specification θ_{sw}^{G} should be exported in Ψ'.

Well-formed state There exists two different definitions for the well-formedness of global machine state according to the specification tags. As described before, tags classify the code of threads and context switching, which manipulate the different parts of memory.

When the tag is G, WFState specifies the conditions of a well-formed machine state when machine runs in context switch code: $\mathbb{S}$ satisfies the precondition p; before context switching function returns, there is always a return address f′ on the stack; there is a specification $\Psi(\mathtt{f}')$, which is satisfied by the machine state after machine returns from context switching.

The WFState, tagged by L, specifies the conditions of a well-formed global machine state when machine runs in the code of threads:

1. the whole memory of the state is divided into several blocks, the private memory of every threads $\mathbb{H}_{l_0}$ to $\mathbb{H}_{l_n}$, the memory containing data of contexts $\mathbb{H}_c$, and the shared memory $\mathbb{H}_s$;
2. there exists an abstract machine context set $\mathbb{X}$, which satisfies the relation MapX with $\mathbb{H}_c$;
3. there exists a t, which belongs to the domain set of $\mathbb{X}$ and indicates the location of the machine context of the current thread;
4. the local state of current thread, which consists of $\mathbb{H}_t \uplus \mathbb{H}_s$, current register file $\mathbb{R}$ and current flag register $\mathbb{F}$, satisfies the well-formed thread predicate WFThread;
5. for every machine context label in $\widehat{\mathbb{X}}$ except for t, Ψ, $\widehat{\mathbb{X}}$ and the state $(\mathbb{H}_l,\mathbb{R}_l,_)$ satisfy the well-formed ready-thread predicate WFRdyThrd.

Well-formed thread The predicate WFThread specifies that the state $\mathbb{S}$ of the current running thread t satisfies the specification θ: the tag in specification is L; t, machine context label set X and state $\mathbb{S}$ satisfy the precondition p; and they also satisfy the guarantee g via WFStack predicate.

Well-formed ready-thread The predicate WFRdyThrd states that a ready thread is well-formed if :

1. there is a return address $\mathtt{ip}_l$ stored on the top of thread stack;
2. the specification $\Psi(\mathtt{ip}_l)$ is tagged by L;
3. $\Psi(\mathtt{ip}_l)$ and the state $(\mathbb{H},\mathbb{R},\mathbb{F})$ after the ret instruction satisfies WFThread if the private memory is merged with the shared memory specified by INV t X';
4. the current X and the X' in future (when the ready thread is activated) satisfy the relation $\mathcal{R}$.

$$\frac{\Psi \subseteq \Psi' \quad \Psi \vdash \mathbb{C} : \Psi' \quad \mathsf{WFState}(\Psi, \mathbb{S}, \theta) \quad \Psi, \mathtt{ip} \vdash \{\theta\}\mathbb{C}[\mathtt{ip}]}{\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathtt{ip})} \ (\textsc{mach})$$

$$\frac{\Psi(\mathtt{f}') = (\mathtt{p}', \mathtt{g}', \mathtt{b}) \quad \Psi, \mathtt{f}+1 \vdash \{(\mathtt{p}'', \mathtt{g}'', \mathtt{b})\}\mathbb{I} \quad \mathtt{p} \triangleright \mathtt{g}_{\{\mathtt{ze}\}} \Rightarrow \mathtt{p}' \quad \mathtt{p} \circ (\mathtt{g}_{\{\mathtt{ze}\}} \circ \mathtt{g}') \Rightarrow \mathtt{g} \quad \mathtt{p} \triangleright \mathtt{g}_{\{\mathtt{zd}\}} \Rightarrow \mathtt{p}'' \quad \mathtt{p} \circ (\mathtt{g}_{\{\mathtt{zd}\}} \circ \mathtt{g}'') \Rightarrow \mathtt{g}}{\Psi, \mathtt{f} \vdash \{(\mathtt{p}, \mathtt{g}, \mathtt{b})\}\mathtt{je}\ \mathtt{f}'; \mathbb{I}} \ (\textsc{je})$$

$$\frac{\Psi'(\mathtt{swapctxt}) = \theta^{G}_{\mathsf{sw}} \quad \forall\, \mathtt{f} \in \mathrm{dom}(\Psi').\ \ \Psi, \mathtt{f} \vdash \{\Psi'(\mathtt{f})\}\mathbb{C}[\mathtt{f}]}{\Psi \vdash \mathbb{C} : \Psi'} \ (\textsc{cdhp})$$

$$\frac{\Psi' = \Psi\{\mathtt{swapctxt} \rightsquigarrow \theta^{L}_{\mathsf{sw}}\} \quad \Psi'(\mathtt{f}') = (\mathtt{p}', \mathtt{g}', L) \quad \mathsf{enable}(\mathtt{p}, \mathtt{g}_{(\mathtt{call},\mathtt{f})}) \quad \Psi(\mathtt{f}+1) = (\mathtt{p}'', \mathtt{g}'', L) \quad \mathtt{p} \triangleright \mathtt{g}_{(\mathtt{call},\mathtt{f})} \Rightarrow \mathtt{p}' \quad \mathtt{p} \triangleright (\mathtt{g}_{(\mathtt{call},\mathtt{f})} \circ \mathtt{g}' \circ \mathtt{g}_{(\mathtt{ret},_)}) \Rightarrow \mathtt{p}'' \quad \mathtt{p}' \triangleright \mathtt{g}' \Rightarrow \mathtt{g}_{\mathsf{fun}} \quad \mathtt{p} \circ (\mathtt{g}_{(\mathtt{call},\mathtt{f})} \circ \mathtt{g}' \circ \mathtt{g}_{(\mathtt{ret},_)} \circ \mathtt{g}'') \Rightarrow \mathtt{g}}{\Psi, \mathtt{f} \vdash \{(\mathtt{p}, \mathtt{g}, L)\}\mathtt{call}\ \mathtt{f}'} \ (\textsc{call})$$

$$\frac{\mathtt{i} \notin \{\mathtt{call}, \mathtt{jmp}, \mathtt{jne}, \mathtt{ret}\} \quad \Psi, \mathtt{f}+1 \vdash \{(\mathtt{p}', \mathtt{g}', \mathtt{b})\}\mathbb{I} \quad \mathsf{enable}(\mathtt{p}, \mathtt{g}_{(\mathtt{i},\mathtt{f})}) \quad \mathtt{p} \triangleright \mathtt{g}_{(\mathtt{i},\mathtt{f})} \Rightarrow \mathtt{p}' \quad \mathtt{p} \circ (\mathtt{g}_{(\mathtt{i},\mathtt{f})} \circ \mathtt{g}') \Rightarrow \mathtt{g}}{\Psi, \mathtt{f} \vdash \{(\mathtt{p}, \mathtt{g}, \mathtt{b})\}\mathtt{i}; \mathbb{I}} \ (\textsc{seq})$$

$$\frac{\Psi(\mathtt{f}') = (\mathtt{p}', \mathtt{g}', \mathtt{b}) \quad \mathtt{p} \Rightarrow \mathtt{p}' \quad \mathtt{p} \circ \mathtt{g}' \Rightarrow \mathtt{g}}{\Psi, \mathtt{f} \vdash \{(\mathtt{p}, \mathtt{g}, \mathtt{b})\}\mathtt{jmp}\ \mathtt{f}'} \ (\textsc{jmp}) \qquad \frac{\mathtt{p} \circ \mathtt{g}_{\mathsf{id}} \Rightarrow \mathtt{g}}{\Psi, \mathtt{f} \vdash \{(\mathtt{p}, \mathtt{g}, \mathtt{b})\}\mathtt{ret}} \ (\textsc{ret})$$

$$\begin{array}{lll}
\mathsf{WFState}(\Psi, \mathbb{S}, (\mathtt{p}, \mathtt{g}, G)) & \triangleq & \forall t, X, \mathbb{S}', \mathbb{S}''.\, \mathtt{p}\ t\ X\ \mathbb{S} \wedge (\mathtt{g}\ t\ X\ X\ \mathbb{S}\ \mathbb{S}' \wedge \mathsf{NextS}_{(\mathtt{ret},_)}(\mathbb{S}', \mathbb{S}'') \\
& & \quad \rightarrow \exists \mathtt{f}'.\, \mathsf{retaddr}(\mathtt{f}', \mathbb{S}') \wedge \Psi(\mathtt{f}') = (\mathtt{p}', \mathtt{g}', L) \wedge \mathsf{WFState}(\Psi, \mathbb{S}'', (\mathtt{p}', \mathtt{g}', L)) \\
\mathsf{WFState}(\Psi, \mathbb{S}, (\mathtt{p}, \mathtt{g}, L)) & \triangleq & \exists (\mathbb{H}, \mathbb{R}, \mathbb{F}) = \mathbb{S}.\, \exists \mathbb{H}_{l_0}, \ldots, \mathbb{H}_{l_n}, \mathbb{H}_s, \mathbb{H}_c.\, \mathbb{H} = \mathbb{H}_{l_0} \uplus \cdots \uplus \mathbb{H}_{l_n} \uplus \mathbb{H}_s \uplus \mathbb{H}_c \\
& & \quad \wedge \exists t, \mathbb{X}.\, \mathsf{MapX}(\mathbb{H}_c, \mathbb{X}) \wedge t \in \widehat{\mathbb{X}} \wedge \mathsf{WFThread}(\Psi, t, \widehat{\mathbb{X}}, (\mathbb{H}_t \uplus \mathbb{H}_s, \mathbb{R}, \mathbb{F}), (\mathtt{p}, \mathtt{g}, L)) \\
& & \quad\quad \wedge (\forall l \in \widehat{\mathbb{X}} \setminus \{t\}.\, \exists \mathbb{R}_l.\, \mathbb{X}(l) = \langle \mathbb{R}_l \rangle \wedge \mathsf{WFRdyThrd}(\Psi, \widehat{\mathbb{X}}, (\mathbb{H}_l, \mathbb{R}_l, _))) \\
\mathsf{WFThread}(\Psi, t, X, \mathbb{S}, (\mathtt{p}, \mathtt{g}, L)) & \triangleq & \mathtt{p}\ t\ X\ \mathbb{S} \wedge \exists n.\, \mathsf{WFStack}(n, \Psi, t, X, \mathbb{S}, \mathtt{g}) \\
\mathsf{WFRdyThrd}(\Psi, X, \mathbb{S}_l) & \triangleq & \forall X', (\mathbb{H}, \mathbb{R}, \mathbb{F}).\, \mathcal{R}\ l\ X\ X' \wedge \mathsf{NextS}_{(\mathtt{ret},_)}\ \mathbb{S}_l\ (\mathbb{H}, \mathbb{R}, \mathbb{F}) \\
& & \quad \rightarrow \exists \mathtt{ip}_l.\, \mathsf{retaddr}(\mathtt{ip}_l, \mathbb{S}_l) \wedge \Psi(\mathtt{ip}_l) = (_, _, L) \\
& & \quad\quad \wedge \mathbb{H} \Vdash (\mathsf{INV}\ l\ X') \mathrel{-\!\!*} \lambda \mathbb{H}'.\, \mathsf{WFThread}(\Psi, t, X', (\mathbb{H}', \mathbb{R}, \mathbb{F}), \Psi(\mathtt{ip}_l)) \\
\mathsf{WFStack}(0, \Psi, t, X, \mathbb{S}, \mathtt{g}) & \triangleq & \neg \exists \mathbb{X}', \mathbb{S}'.\, \mathtt{g}\ t\ X\ X'\ \mathbb{S}\ \mathbb{S}' \\
\mathsf{WFStack}(n+1, \Psi, t, X, \mathbb{S}, \mathtt{g}) & \triangleq & \exists \mathtt{f}' \in \Psi.\, \Psi(\mathtt{f}') = (\mathtt{p}', \mathtt{g}', L) \\
& & \quad \wedge \forall \mathtt{ip}', X', \mathbb{S}', \mathbb{S}''.\, \mathtt{g}\ t\ X\ X'\ \mathbb{S}\ \mathbb{S}' \wedge \mathsf{NextS}_{(\mathtt{ret},\mathtt{ip}')}\ \mathbb{S}'\ \mathbb{S}'' \\
& & \quad\quad \rightarrow \mathsf{retaddr}(\mathtt{f}', \mathbb{S}') \wedge \mathtt{p}'\ t\ X'\ \mathbb{S}'' \wedge \mathsf{WFStack}(n, \Psi, t, X', \mathbb{S}'', \mathtt{g}')
\end{array}$$

Fig. 6. Inference Rules

Well-formed control stack The predicate WFStack is to specify the well-formed chain of return addresses recursively. If the first argument, the depth of the control stack, is zero, there will be no return address at the bottom of stack, that is, the machine cannot execute return instruction when the control stack of thread is empty. If the depth is $n+1$, the well-formed control stack requires that

(1) the post-state $\mathbb{S}'$ of g contain a return address $\mathtt{f}'$;

(2) $\mathbb{S}'$ step to any state $\mathbb{S}''$ by `ret` instruction;

(3) for $\mathtt{f}'$, there be a specification in Ψ, $(\mathtt{p}', \mathtt{g}', L)$;

(4) $\mathbb{S}''$ satisfy $\mathtt{p}'$;

(3) and the rest of control stack in $\mathbb{S}''$ be well-formed with respect to $\mathtt{g}'$.

Well-formed instruction sequence The remaining rules are used to reason about an instruction sequence from the label f under a specification θ. The rule (SEQ) specifies a well-formed instruction sequence led by the sequent instructions, including arithmetic, data movement and stack instructions.

The premises of (SEQ) include that: (1) the rest instruction sequence $\mathbb{I}$ is well-formed under a specification $(\mathtt{p}', \mathtt{g}', \mathtt{b})$; (2) the precondition p implies the action of instruction i; (3) the precondition p bound with instruction action implies $\mathtt{p}'$; (4) the action $\mathtt{g}'$, which is bound with instruction action and strengthened by p, implies g.

The rule (JMP) is straightforward, and it specifies the conditions of well-formed jump instruction: the precondition p implies the precondition of jump destination $\mathtt{p}'$ and the action of jump destination $\mathtt{g}'$ implies g. The rule (JE) is like (JMP) and also straightforward.

The rule (RET) says if the precondition p implies the action (g), then the return instruction is well-formed. This rule checks whether a function action is finished eventually.

The most interesting rule is (CALL), which specifies the conditions of the well-formed instruction sequence led by call instruction and most of these conditions are defined as combinations of p and g. Note that the specification tag L in the conclusion indicates that only threads can call functions. To support certifying context switching, the specification of called function belongs to Ψ is merged with $(\mathtt{swapctxt}, \theta^{L}_{\mathsf{sw}})$.

4.3 Soundness proof

The soundness proof of our framework with respect to the machine operational semantics is proved by the approach of proving type soundness. Following the progress and preservation lemmas, we can guarantee that certified program in our framework is safe to execute.

Safety Safety of the machine configuration means that the execution of the machine will not go stuck; (*ii*) the machine

state always satisfies the corresponding specification. The safety can be defined formally: if a machine configuration $\mathbb{M}$ steps to $\mathbb{M}'$, the machine configuration $\mathbb{M}'$ can at least execute one step to $\mathbb{M}''$.

$$\mathsf{Safety}(\mathbb{M}) \triangleq \forall n, \mathbb{M}' . \mathbb{M} \longmapsto^n \mathbb{M}' \rightarrow \exists \mathbb{M}'' . \mathbb{M}' \longmapsto \mathbb{M}''$$

Since the proof is tedious and long, we skip omit the proof details here. The complete proof is fully mechanized in the proof assistant Coq, and interested readers may download it from the web site [13] to check it by Coq.

Lemma 1 (Consistency of `swapctxt` Specifications).
For any specifications set Ψ and state $\mathbb{S}$, if $\mathbb{S}$ is well-formed under the thread-local specification θ^L_{sw}, then $\mathbb{S}$ is also well-formed under the global specification of context switching θ^G_{sw}.

We can prove that a well-formed machine can execute a call instruction:

Lemma 2 (Call-Progress).
For any specifications set Ψ, Ψ', code heap $\mathbb{C}$, machine state $\mathbb{S}$, instruction pointer $\mathtt{ip}$, if the machine configuration is well-formed, $\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathtt{ip})$, and the $\mathtt{ip}$ points to a $\mathtt{call\ f}$ instruction, then there exists a new state $\mathbb{S}'$ and a new $\mathtt{ip}'$ such that the machine can step to a new machine configuration $(\mathbb{C}, \mathbb{S}, \mathtt{ip}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathtt{ip}')$.

Lemma 3 (Call-Preservation).
For any specifications set Ψ, Ψ', code heap $\mathbb{C}$, machine state $\mathbb{S}$, instruction pointer $\mathtt{ip}$, if the machine configuration $(\mathbb{C}, \mathbb{S}, \mathtt{ip})$ is well-formed, $\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathtt{ip})$, and the machine steps to a new configuration $(\mathbb{C}, \mathbb{S}, \mathtt{ip}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathtt{ip}')$ by a $\mathtt{call}$ instruction, then the new configuration is well-formed: $\Psi \vdash (\mathbb{C}, \mathbb{S}', \mathtt{ip}')$.

Lemma 4 (Progress).
For any specifications set Ψ, code heap $\mathbb{C}$, machine state $\mathbb{S}$, instruction pointer $\mathtt{ip}$, if the machine configuration is well-formed, $\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathtt{ip})$, and the $\mathtt{ip}$ points to $\mathtt{call}$ instruction, then there exists a new state $\mathbb{S}'$ and a new $\mathtt{ip}'$ such that the machine can step to a new machine configuration $(\mathbb{C}, \mathbb{S}, \mathtt{ip}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathtt{ip}')$.

Lemma 5 (Preservation).
For any specifications set Ψ, code heap $\mathbb{C}$, machine state $\mathbb{S}$, instruction pointer $\mathtt{ip}$, if the machine configuration is well-formed, $\Psi \vdash (\mathbb{C}, \mathbb{S}, \mathtt{ip})$, and the machine step to a new configuration $(\mathbb{C}, \mathbb{S}, \mathtt{ip}) \longmapsto (\mathbb{C}, \mathbb{S}', \mathtt{ip}')$ by $\mathtt{call}$ instruction, then the new configuration is well-formed: $\Psi \vdash (\mathbb{C}, \mathbb{S}', \mathtt{ip}')$.

Theorem 1 (Safety).
For any machine configuration $\mathbb{M}$ is well-formed under specification set Ψ, $\Psi \vdash \mathbb{M}$, then it will be safe, $\mathsf{Safety}(\mathbb{M})$.

5 Verification Results

In this section, we demonstrate how to certify the code in Section 1.1 modularly. The code consists of three parts, thread A, thread B, and the context switching implementation.

5.1 Certifying context switching and threads code

It is easy to certify that the context switching implementation satisfies its global specification θ^G_{sw}. We use $\mathbb{C}_{\mathsf{sw}}$ to specify the code heap containing the context switching implementation. By the inference rules presented in Section 4.2, we can have the following judgement.

$$\emptyset \vdash \mathbb{C}_{\mathsf{sw}} : \{\mathtt{swapctxt} \leadsto \theta^G_{\mathsf{sw}}\}$$

Then we give a memory invariant to specify the shared memory.

$$\begin{array}{lll} \mathsf{INV} & \triangleq & \lambda t, X, \mathbb{H}. \exists l_r, v. l_r \in X \wedge l_r \neq t \\ & & \wedge \mathbb{H} \Vdash \mathtt{thrd} \mapsto t, l_r * \mathtt{buf} \mapsto v * !(v{=}0 \vee v{=}1) \end{array}$$

It contains:

- the pointer to the context structure of the current thread;
- the pointer to the context structure of the other thread;
- the buffer of data.

The invariant satisfies the precise requirement.

Lemma 6 (INV-Precise).
For any context label t and label set X, the invariant $\mathsf{INV}\ t\ X$ is precise, that is, $\mathsf{Precise}(\mathsf{INV}\ t\ X)$.

In the following, we use instruction sequence rules to prove the well-formedness of code of two threads (A and B), according to the specifications in Figure 7: $\mathbb{C}_A$ and $\mathbb{C}_B$ are to specify the code heaps of thread A and B, respectively. Ψ_A and Ψ_B are to specify the specifications of thread A and B:

$$\begin{array}{lll} \Psi_A & \triangleq & \{\mathtt{threadA} \leadsto (\mathrm{p}_A, \mathrm{g}_A, L), \mathtt{lab1} \leadsto (\mathrm{p}_{A3}, \mathrm{g}_{A3}, L)\} \\ \Psi_B & \triangleq & \{\mathtt{threadB} \leadsto (\mathrm{p}_B, \mathrm{g}_B, L), \mathtt{lab2} \leadsto (\mathrm{p}_{B3}, \mathrm{g}_{B3}, L)\} \end{array}$$

By the inference rules in Section 4.2, we can prove that the code heaps of thread A and B are well-formed:

$$\Psi_A \vdash \mathbb{C}_A : \Psi_A \qquad \Psi_B \vdash \mathbb{C}_B : \Psi_B$$

By the rule (CDHP), we can prove that the complete code is well-formed:

$$\Psi_A \uplus \Psi_B \vdash \mathbb{C}_A \uplus \mathbb{C}_B \uplus \mathbb{C}_{\mathsf{sw}} : \Psi_A \uplus \Psi_B \uplus \{\mathtt{swapctxt} \leadsto \theta^G_{\mathsf{sw}}\}$$

We give an initial machine state $\mathbb{S}_0$, which is well-formed with respect to the specification of entry point, $\Psi_A(\mathtt{threadA})$:

$$\Psi_A \uplus \Psi_B \vdash \mathbb{S}_0 : (\mathrm{p}_A, \mathrm{g}_A, L)$$

Then the following code safety theorem holds.

Theorem 2 (Code-Safety).
$\mathsf{Safety}((\mathbb{C}_A \uplus \mathbb{C}_B \uplus \mathbb{C}_{sw}, \mathbb{S}_0, \mathtt{threadA}))$

5.2 Comparison with Ni's work

Ni *et al* applied the theory of XCAP [3] to certify context switching. To support certifying first-class code pointers, a specification language PropX is introduced to support the predicate cptr, which can relate the jump destination `f` and its specification p, cptr(f,p). It's worth noting that the predicate cptr cannot be defined straightforward, and much research [14, 3, 15] has been devoted to defining it. The language PropX is deeply embedded inside the meta-language,

$p_A \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . t \in X \wedge (\mathbb{H} \Vdash \mathsf{INV}\ \ t\ \ X * \mathbb{R}(\mathtt{esp}) - 12 \hookrightarrow _, _, _)$
$g_A \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
threadA:
     mov   ebx, buf
     mov   eax, 1
     mov   [ebx], eax
```

$p_{A1} \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . \exists l_r . t \in X \wedge l_r \neq t \wedge l_r \in X \wedge (\mathbb{H} \Vdash \mathtt{thrd} \mapsto t, l_r * \mathtt{buf} \mapsto 1 * \mathbb{R}(\mathtt{esp}) - 12 \hookrightarrow _, _, _)$
$g_{A1} \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
     mov   ebx, thrd
     mov   edx, [ebx+4]
     mov   ecx, [ebx]
     mov   [ebx], edx
     mov   [ebx+4], ecx
     push  edx
     push  ecx
```

$p_{A2} \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . \exists old, new . t \in X \wedge old = t \wedge new \neq t \wedge new \in X \wedge (\mathbb{H} \Vdash \mathsf{INV}\ \ (new)\ \ X * \mathbb{R}(\mathtt{esp}) - 4 \hookrightarrow _, old, new)$
$g_{A2} \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
     call  swapctxt
lab1:
```

$p_{A3} \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . t \in X \wedge (\mathbb{H} \Vdash \mathsf{INV}\ \ t\ \ X * \mathbb{R}(\mathtt{esp}) - 4 \hookrightarrow _, _, _)$
$g_{A3} \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
     add   esp, 8
     jmp   threadA
```

$p_B \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . t \in X \wedge (\mathbb{H} \Vdash \mathsf{INV}\ \ t\ \ X * \mathbb{R}(\mathtt{esp}) - 12 \hookrightarrow _, _, _)$
$g_B \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
threadB:
     mov   ebx, buf
     mov   eax, 0
     mov   [ebx], eax
```

$p_{B1} \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . \exists l_r . t \in X \wedge l_r \neq t \wedge l_r \in X \wedge (\mathbb{H} \Vdash \mathtt{thrd} \mapsto t, l_r * \mathtt{buf} \mapsto 1 * \mathbb{R}(\mathtt{esp}) - 12 \hookrightarrow _, _, _)$
$g_{B1} \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
     mov   ebx, thrd
     mov   edx, [ebx+4]
     mov   ecx, [ebx]
     mov   [ebx], edx
     mov   [ebx+4], ecx
     push  edx
     push  ecx
```

$p_{B2} \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . \exists old, new . t \in X \wedge old = t \wedge new \neq t \wedge new \in X \wedge (\mathbb{H} \Vdash \mathsf{INV}\ \ (new)\ \ X * \mathbb{R}(\mathtt{esp}) - 4 \hookrightarrow _, old, new)$
$g_{B2} \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
     call  swapctxt
lab2:
```

$p_{B3} \triangleq \lambda t, X, (\mathbb{H}, \mathbb{R}, \mathbb{F}) . t \in X \wedge (\mathbb{H} \Vdash \mathsf{INV}\ \ t\ \ X * \mathbb{R}(\mathtt{esp}) - 4 \hookrightarrow _, _, _)$
$g_{B3} \triangleq \lambda t, X, X', (\mathbb{H}, \mathbb{R}, \mathbb{F}), (\mathbb{H}', \mathbb{R}', \mathbb{F}') . \mathsf{False}$

```
     add   esp, 8
     jmp   threadB
```

Fig. 7. Specifications of Threads

and then cptr is a primitive logical operator. The validity of $\mathsf{cptr}(\mathtt{f}, \mathtt{p})$ is guaranteed by the consistency property of the language PropX and the soundness of the program logic XCAP.

To support modular reasoning, *impredicative polymorphic predicates* are supported by the language PropX to write for the context switching a modular specification, which unifies the global semantics and thread local semantics:

$$\left\{ \begin{array}{l} \exists[a_{env}, a_{newenv}, a_{prvold}] . \mathsf{m}_{old} * a_{prvold} * a_{env} \\ \quad * \exists[a_{prvnew}] . \mathsf{m}_{new} * a_{prvnew} \\ \qquad \wedge \mathsf{cptr}(ecp_{new}, a_{env} * \mathsf{m}^R_{new} * a_{prvnew} \\ \qquad\quad * \exists[a_{prvold}] . \mathsf{m}'_{old} * a_{prvold} \\ \qquad\qquad \wedge \mathsf{cptr}(ecp'_{old}, \mathsf{m}'^R_{old} * a_{prvold} * a_{newenv})) \\ \quad \wedge \mathsf{cptr}(ret, \mathsf{m}''_{old} * a_{prvold} * a_{newenv}) \end{array} \right\}$$

```
swapctxt:
     mov   eax, [esp+4]
     ...
     ret
```

$\{\perp\}$

In the precondition of context switching code, m_{old} and m_{new} specify the machine context structures pointed by old and new arguments before context switching. m'_{old} specifies the machine context structure pointed by *old* when context switching is finished, while m''_{old} specifies the old machine context structure after the program return to the same thread which called context switching. m^R means that the values stored in current registers are equal to the values stored in the described structure. The definitions of these memory predicates are omitted here and it will not impede the comprehension of readers. ecp_{new} and ecp'_{old} refer to the two embedded code pointers stored in context structures after context switch. a_{env} asserts the memory of environment except for the old and new context structures. a_{newenv} specifies the memory of environment except for the old context structure after the context switch back to the original thread. a_{prvold} and a_{prvnew} specify the private memory of thread, e.g. thread stack. We may notice that the environment memory is unknown when certifying the implementation code of context switching alone. Actually, these environment predicates and private memory predicates are polymorphic variables, which can be instantiated to any concrete predicate when we certify threads.

The global semantics of context switching can be derived from the specification above. For example, the fact, a_{env} and a_{prvnew} are preserved inside and outside $\mathsf{cptr}(ecp_{new}, \ldots)$, indicates the implementation code of context switching doesn't modify these two blocks of memory. The thread-local semantics of context switching are also implied. For example, a_{prvold} is preserved both inside and outside of $\mathsf{cptr}(ret, \ldots)$.

Moreover, *recursive predicates* have to be supported to write specifications of threads, because several threads may have mutually recursive expectation of each other.

$$\{\mathsf{m}_{old} * \mathsf{m}_{prvold} * \mathsf{m}^{\mu}_{new}\}\ \mathtt{call\ swapctxt}\ \{\mathsf{m}'_{old} * \mathsf{m}_{prvold} * \mathsf{m}^{\mu}_{new}\}$$

where m^{μ}_{new} is defined recursively:

$$\begin{array}{rl} \mathsf{m}^{\mu}_{new} \triangleq & \mu\alpha . \mathsf{m}_{new} * \exists[a_{prvnew}] . a_{prvnew} \\ & \quad \wedge \mathsf{cptr}(ecp_{new}, \mathsf{m}^R_{new} * a_{prvnew} \\ & \qquad * \exists[a_{prvold}] . \mathsf{m}_{prvold} * \mathsf{m}'_{old} \\ & \qquad\quad \wedge \mathsf{cptr}(ecp'_{old}, \mathsf{m}'^R_{old} * \mathsf{m}_{prvold} * \alpha)) \end{array}$$

Their method using cptr is powerful enough to reason about any code with arbitrary control flow transfers, but it's too heavyweight to reason about multi-threaded program with context switching:

- The idea of supporting first-class code pointers (cptr) is too low-level to certify multi-threaded code, that is,

	Ni's framework	Our framework
Spec. Lang.	4, 500 loc.	0 loc.
Prog. Logic	130 loc.	440 loc.
Soundness	620 loc.	3,000 loc.
swapctxt	4,200 loc.	1,100 loc.
User Threads	N/A	900 loc.
Compilation Time	12 min.	1.5 min.

Fig. 8. Comparation of Coq Implementation

in a *continuation-passing style*. It will lead to complicated specifications, which often involve many nested cptr predicates and recursive memory predicates.

- Since the language PropX is deeply embedded in the meta-langauge, the assistant proving tools provided by meta-language are useless in the PropX level. The impredicative polymorphic and recursive variables lead to higher proving costs.

An experimental statistics and comparison with Ni's Coq implementation are shown in Figure 8. We can see that the XCAP program logic as well as its soundness proof is very simple, less than 1k loc., but the language PropX and its consistency proof (cut elimination) are heavyweight. While in our framework, the complicated parts of framework are shifted to the program logic and its soundness proof, more than 3k loc. It's more straightforward and natural to certify the code of context switching implementation and threads in our framework, because of the more clear and comprehensible semantics of context switching. The safety proof size (line of code) of context switching implementation is only a quarter of that in Ni's framework. The proof of threads is smaller too (less than 1k loc). Our Coq code includes an auxiliary library about 9,500 loc., which is general and don't reflect the proving complexity. The compilation time of our Coq code is only one eighth of that of Ni's code.

6 More Related Work

Feng *et al* proposed the CAP-CR program logic [11], which supports reasoning about coroutines modularly. Compared to our framework, their code specifications of threads are in the forms of (p, g, g_t, g_r). The precondition p and the guarantee g are similar to ours. The extra g_t is used to specify the action of the code segment from the current point to the next switch point and g_r is used to specify the remaining state transition between the return point and the next switch point.

Gargano *et al* showed a framework CVM [16] to build verified kernels in the Verisoft project. CVM is a computational model for concurrent user processes interacting with a micro-kernel. Starostin and Tsyban presented a formal approach [6] to pervasive reasoning about context switch interleaved between user processes and a C-programmed kernel. Their context switch code and proof are integrated in a framework for building verified kernels (CVM) [17]. Their framework keeps a global invariant, *weak consistency*, to verify context switching in the low-level, while user threads are verified in the C language level. There is no memory partition in their global invariant since their framework supports virtual memory. Their approach has a few resemblances to previous work [8], and more precisely, the context data is totally hidden from user programs. Therefore, the context switching implementation and the threads in this paper cannot be verified in their framework.

The L4.verified project [18, 19] aims to providing a proof of the functional correctness of the seL4 microkernel with respect to a high level, formal description of its expected behaviour. VFiasco [20] project is attempting to verify the Fiaso kernel, a variant of L4 directly on the C++ level. In the kernel of either seL4 or Fiaso, because the contexts of user processes are saved prior to entering the kernel, and are restored at the termination of a call to the kernel mode, their context switching model is also different from ours. Thus their verification methods cannot be applied to the verfication of our context switching implementation.

The method of reasoning shared memory in our framework comes from the ownership-transfer semantics [21], which is inspired by concurrent separation logic [12].

7 Conclusion and Future Work

We extended previous work and proposed a lightweight verification framework for certifying low-level context switching in a modular way. By abstraction of context data, both the context switching implementation and threads can be verified in our framework naturally, without regards to the problems of first-class pointers.

Three other machine context operations aren't supported in our framework, such as context loading, context storing and context making. The first two operations are easy and straightforward to support in our framework. As for context making, we believe that it will be not difficult to support if our framework supports dynamic memory allocation. This is one of our future work.

Another issue not addressed in our framework is the interrupts of processors. It's interesting to combine the work of certifying interrupts and our work in this paper to certify the preemptive implementation of threads or OS kernels. Also, it is interesting to extend our framework to support code running over multi-processors, which is commonly seen in the modern operating system kernels.

Our long-term goal is to certify realistic operating system kernels, hypervisors and other modern system software. The work presented in this paper provides a foundation for reasoning about code of concurrency base and makes a solid step toward our goal.

References

[1] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In *Proc. 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07)*, volume 4732 of *Lecture Notes in*

Computer Science, pages 189–206. Springer-Verlag, September 2007.

[2] Zhaozhong Ni. *Modular Machine Code Verification*. PhD thesis, Yale University, 2007.

[3] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Proceedings of 33nd ACM Symposium on Principles of Programming Languages*, pages 320–333, January 2006.

[4] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[5] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM Workshop on Types in Lang. Design and Impl.*, January 2007.

[6] Artem Starostin and Alexandra Tsyban. Verified process-context switch for C-programmed kernels. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, Experiments Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*, pages 240–254, Toronto, Canada, October 2008. Springer.

[7] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proc. Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, pages 54–69, October 2008.

[8] Yu Guo, Xinyu Jiang, Yiyun Chen, and Chunxiao Lin. A certified thread library for multithreaded user programs. In *Proceedings of 1st IEEE IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 127–136, Shanghai, China, Jun 2007. IEEE Computer Society.

[9] Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.1, 2006.

[10] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*, 1993.

[11] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl.*, June 2006.

[12] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67. Springer, September 2004.

[13] Authors. On the certification of thread context switching. `http://sites.google.com/site/lambda/switch/`, March 2009.

[14] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[15] Hongxu Cai. *Logic-based Verification of General Machine Code*. PhD thesis, Tsinghua University, 2008.

[16] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[17] Tom In der Rieden and Alexandra Tsyban. CVM – A verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification (SSV 2008)*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 151–168. Elsevier Science B.V., 2008.

[18] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[19] Gerwin Klein, Michael Norrish, Kevin Elphinstone, and Gernot Heiser. Verifying a high-performance micro-kernel. In *7th Annual High-Confidence Software and Systems Conference*, May 2007.

[20] Michael Hohmuth and Hendrik Tews. The vfiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005.

[21] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, June 2008.