Using XCAP to Certify Realistic Systems Code: Machine Context Management

Zhaozhong Ni[1], Dachuan Yu[2], and Zhong Shao[3]

[1] Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A.
zhaozhong.ni@microsoft.com
[2] DoCoMo Communications Laboratories USA, Inc, Palo Alto, CA 94304, U.S.A.
yu@docomolabs-usa.com
[3] Department of Computer Science,Yale University, New Haven, CT 06520, U.S.A.
shao@cs.yale.edu

Abstract. Formal, modular, and mechanized verification of realistic systems code is desirable but challenging. Verification of machine context management (a basis of multi-tasking) is one representative example. With context operations occurring hundreds to thousands of times per second on every computer, their correctness deserves careful examination. Given the small and stable code bases, it is a common misunderstanding that the context management code is suitable for informal scrutiny and testing. Unfortunately, after being extensively studied and used for decades, it still proves to be a common source of bugs and confusion. Yet its verification remains difficult due to the machine-level detail, irregular patterns of control flows, and rich application scenarios.

This paper reports our experience applying XCAP—a recent theoretical verification framework—to certify a realistic x86 implementation of machine context management. XCAP supports expressive and modular logical specifications, but has only previously been applied on simple idealized machine and code. By applying the XCAP theory to an x86 machine model, building libraries of common proof tactics and lemmas, composing specifications for the context data structures and routines, and proving that the code behave accordingly, we achieved the first formal, modular, and mechanized verification of realistic x86 context management code. Our proofs are fully mechanized in the Coq proof assistant. Our certified library code runs on stock hardware and can be linked with other certified systems and application code. Our technique applies to other variants or extensions of context management (*e.g.*, more complex context, different platforms), provides a solid basis for further verification of thread implementation and concurrent programs, and illustrates how to achieve formal, modular, and mechanized verification of realistic systems code.

1 Introduction

Formally establishing safety and correctness properties of realistic systems code in a modular and machine-checkable fashion is a highly desirable but extremely challenging goal. Among various systems code libraries, machine context management, the basis of multi-tasking and an essential feature of modern system software, is one representative example. We use the following 19-line x86 machine-context switching routine (omitting floating-point and special registers) to illustrate our point.

```
swapcontext:
    mov eax, [esp+4]      |  mov eax, [esp+8]
    mov [eax+_eax], 0     |  mov esp, [eax+_esp]
    mov [eax+_ebx], ebx   |  mov ebp, [eax+_ebp]
    mov [eax+_ecx], ecx   |  mov edi, [eax+_edi]
    mov [eax+_edx], edx   |  mov esi, [eax+_esi]
    mov [eax+_esi], esi   |  mov edx, [eax+_edx]
    mov [eax+_edi], edi   |  mov ecx, [eax+_ecx]
    mov [eax+_ebp], ebp   |  mov ebx, [eax+_ebx]
    mov [eax+_esp], esp   |  mov eax, [eax+_eax]
                          |  ret
```

The left half of the code saves the current machine context, whereas the right half loads the new machine context and resumes the program control from there. Although conceptually easy to recognize, context switching is hard to reason about even at the meta-level—it involves program counters, register files, stacks, private and shared heaps, function calls/returns, higher-order control-flows, *etc*.

Because context switching occurs so frequently—hundreds to thousands of times per second on every computer—code sequences similar to the above one deserve the most careful examination. Their safety and correctness are crucial to all multi-tasking software built on top of them. A common misunderstanding is that such code is so stable and small in size that it is more suitable for informal scrutiny and testing. Unfortunately, despite having been extensively studied and used for decades, this code still proves to be a common source of bugs and confusion. A quick web search reveals many reports and discussions of bugs in context management code [8, 20, 11, 7, 17, 19] even today. Thus, it should be a focus of formal methods and mechanized verifications. Yet to the authors' knowledge, there has been no formal proof, either manual or mechanized, of the safety and correctness of code similar to the above one.

Recently, formal studies on systems code for operating system kernels have attracted growing interest. One example is the Singularity project [5], which aims to build a highly reliable OS using a type-safe language (C#) and other techniques. Another example is the Verisoft project [3], which uses computer-aided logical proofs to obtain the correctness of critical systems including OS kernels. Unfortunately, both fall short at the above 19-line code: Singularity trusts unsafe assembly code for doing context switching; Verisoft uses an abstract model of user process that hides the context switching details. Moreover, without a full verification of context management, existing work on concurrent verification is prevented from being integrated with the verification of systems code, as it relies on the correctness of concurrency primitives.

This paper reports our experience on applying XCAP [12], a recent assembly code verification framework, to an x86 machine model, and certifying a realistic machine context implementation with it. XCAP allows expressive and modular logical specifications of safety and correctness, but has been applied previously only on a simple idealized machine model and application code. Starting with the code for context management, we show in this paper how to build the implementation infrastructure and carry out the verification. We first applied the XCAP theory to an x86 machine model, which required some changes to the inference rules for program reasoning. To support practical verification, we built libraries of proof tactics and lemmas. We then specified

the context data structures and routines using the XCAP logic-based specifications, and proved that the code behaves accordingly.

Our library code and proof can be linked with other systems code and application code, as well as their corresponding proofs. The code runs on stock hardware. Our approach is applicable to other variants of context management (such as more complex contexts, different hardware platforms, *etc.*). It provides a solid basis for further verification of concurrent programs. Besides achieving the first formal, modular, and mechanized verification of x86 context management code, our experience also illustrates how verification of realistic systems code can be done in general.

In particular, we want to point out the following features in our case study.

- As shown by *swapcontext*(), our code is representative and realistic, and runs on stock hardware. Its verification does not require a change of programming style or the abandonment of legacy code. There are no performance penalties or compatibility issues.
- Without compromising soundness, our machine model supports realistic features such as variable-length instruction decoding, finite machine-word, word-aligned byte-addressed memory, conditional flags, stack push/pop, and function call/return.
- Our specifications are modular and expressive. For example, the private local data that belongs to a context can be of arbitrary shape and size. The approach is not specialized to a particular kind of multi-tasking implementation, thus our method will likely support other possible usages of machine contexts.
- Everything—the code, machine model, adapted XCAP meta theory (including soundness), proof tactics and lemmas, and code specification and proof—is fully mechanized using the Coq proof assistant, thus leaving a tiny trusted computing base.

The paper is organized as follows. We start in Section 2 by discussing our machine context implementation. In Section 3 we show how to apply XCAP to an x86 machine model. We then specify and verify the context code in Section 4. Section 5 discusses our Coq implementation. Finally, we compare with related work and conclude in Section 6.

2 Machine Context Management

Context refers to the local (private) data of a computation task. It is a widely used concept in software and programming. It is crucial to multi-tasking, as the latter is eventually carried out by doing a context switch. Depending on its application and abstraction level, a context may contain program counters, register files, stacks, private heap, thread control blocks, process control blocks, *etc*. Common context management operations include context creation, restoring, and switching.

Context data structures. The x86 machine context implementation in this paper is at the same level as those found in typical Windows, Unix, and Linux systems. To simplify the problem and focus on the most critical part, we ignore orthogonal features such as floating point and special registers. The machine context structure mctx_st contains the eight general purpose registers of an x86 processor (including a stack pointer register, which should point to a stack with a valid return address on top). The corresponding pointer type is mctx_t. Below are the definitions in C.

```
typedef struct mctx_st *mctx_t;
struct mctx_st {int eax, int ebx, int ecx, int edx,
                int esi, int edi, int ebp, int esp};
```

Context creation. The *makecontext*() function initializes a new context with its arguments: location of context, new stack pointer, return link, address of target function, and argument for target function. It is basis of and analogous to the creation of a new thread. The new stack pointer points to a stack frame prepared for the target function. Notice here $func$() is a **higher-order function pointer**, and *lnk* is also a **higher-order continuation pointer**. When the newly created context gets first switched to, $func$() will start execution. When it finishes, $func$() should return to *lnk*.

```
void makecontext (mctx_t mctx, char *sp, void *lnk, void *func, void *arg);
  mov eax, [esp+4]    // load address of the context
  mov ecx, [esp+8]    // load stack top pointer for the new stack frame
  mov edx, [esp+20]   // load the function's argument
  mov [ecx-4], edx    // push it onto new stack
  mov edx, [esp+12]   // load the function's return link
  mov [ecx-8], edx    // push it onto new stack
  mov edx, [esp+16]   // load the function address
  mov [ecx-12], edx   // push it as return IP onto new stack
  sub ecx, 12
  mov [eax+_esp], ecx // all useful info for fresh context is on new stack
  ret
```

Context switching. The *swapcontext*() function saves the current old context and loads a new one for further execution. Because the stack pointer is changed to the new context's, instead of returning to its direct caller in the old context, it returns to its previous caller in the new context. (Alternatively, the first time a new context gets switched to, it "returns" to the function pointer $func$(), as supplied upon its creation.) Because its code was presented in the beginning of the paper, we only present its interface below.

```
void swapcontext (mctx_t old, mctx_t new);
```

Context loading. The *loadcontext*() function loads a new context and continues execution from there. It is essentially the second half of *swapcontext*(), with a slight difference in the location of context pointers. Technically speaking, it is not a "function" since it never returns to its caller.

```
void loadcontext (mctx_t mctx);
  mov eax, [esp+8]    // load address of the new context
  mov esp, [eax+_esp] // load the new stack pointer
  mov ebp, [eax+_ebp] // load the new registers
  mov edi, [eax+_edi]
  mov esi, [eax+_esi]
  mov edx, [eax+_edx]
  mov ecx, [eax+_ecx]
  mov ebx, [eax+_ebx]
  mov eax, [eax+_eax]
  ret                 // invoke the new context
```

Why traditional methods are insufficient? One may be tempted to think that traditional type systems or program logics should be sufficient to verify the safety and correctness of this "simple" code. After all, it is only 3 functions and 40 assembly instructions. However, as will become apparent in the detailed discussion in the next few sections, a general verification of this code requires at least the following features: 1) polymorphism over arbitrary shape of program data; 2) explicit multiple stacks with flexible stack handling (such as one context manipulating other contexts' stacks); 3) separation-logic-like strong-update style of memory model (*i.e.*, context data should not be managed by garbage collectors, as is in the real-world scenario); 4) general embedded code pointers (not merely higher-order functions, but also higher-order continuations, *e.g.*, see the discussion of $func$ and lnk for $makecontext()$ earlier); 5) support of partial correctness. Traditional type systems such as Typed Assembly Languages (TAL) [9] typically have problems with (1), (2), (3), and (5). Traditional program logics, including Hoare Logic [4] and Separation Logic [16] often fail on (4), as explained in [12]. The very recent index-based semantic model approach for Foundational Proof-Carrying Code [1] does not support (5). Only with recent hybrid type/logic systems such as XCAP [12], can all of these features be supported simultaneously.

3 Applying XCAP to x86 Machine Model

Our verification is carried out following XCAP [12], a logic-based verification framework that facilitates modular reasoning in the presence of embedded code pointers and other higher-order features. In [12], XCAP is applied to an idealized RISC-like toy machine and simple user-level code. When applying XCAP to x86, we had to make practical adaptations and build useful abstractions, particularly for the machine-level details and the handling of the stack and function calls. In this section we first briefly present our adaptation steps and then show how to carry out the verification in general. Interested readers are referred to [12] and the technical report [15] for more details.

Machine model. We formalized a subset of x86 as the machine model for XCAP, as presented in Fig. 1. The execution environment consists of a memory, a register file of general-purpose registers, a flags register made up of a carry bit and a zero bit, and a program counter. The memory appears as a single continuous address space, where both code and data (static and dynamic) reside in. The data part of the memory, the register file, and the flags register form the machine state. We support common instructions for arithmetic, data movement, comparison, control flow transfer, and stack manipulation, as well as realistic x86 features such as variable-length instruction decoding, finite machine-word, word-aligned byte-addressed memory, conditional flags, stack push/pop, and function call/return. Operational semantics for the x86 machine model can be found in the TR [15] . In the rest of this paper, $\mathbb{I}$ represents a sequence of instructions; and $\texttt{Next}_{\texttt{c}}$ denotes a function that computes the resulting state of executing instruction `c`.

Basic reasoning. The basic idea of XCAP follows Hoare logic—a program point can be associated with an assertion which documents its requirement on the machine state. The readers can assume that an assertion is simply a logical predicate on machine state.

$$
\begin{array}{lll}
(\mathit{Prog})\ \mathbb{P} ::= (\mathbb{S}, pc) & & (\mathit{Word})\ \mathtt{w} ::= i\ \ (\mathit{uint}32) \\
(\mathit{State})\ \mathbb{S} ::= (\mathbb{H}, \mathbb{R}, \mathbb{F}) & & (\mathit{CdLbl})\ \mathtt{f} ::= i\ \ (\mathit{uint}32) \\
(\mathit{Mem})\ \mathbb{H} ::= \{\mathtt{l}_1 \rightsquigarrow \mathtt{w}_1, \ldots, \mathtt{l}_n \rightsquigarrow \mathtt{w}_n\} & & (\mathit{Label})\ \mathtt{l} ::= i\ \ (i\%4 = 0) \\
(\mathit{Rfile})\ \mathbb{R} ::= \{\mathsf{eax} \rightsquigarrow \mathtt{w}_1, \ldots, \mathsf{esp} \rightsquigarrow \mathtt{w}_8\} & & (\mathit{Bool})\ \mathtt{b} ::= tt \mid ff \\
(\mathit{FReg})\ \mathbb{F} ::= \{\mathsf{cf} \rightsquigarrow \mathtt{b}_1, \mathsf{zf} \rightsquigarrow \mathtt{b}_2\} & & (\mathit{Addr})\ \mathtt{d} ::= i \mid \mathtt{r} \pm i \\
(\mathit{Cond})\ cc ::= \mathsf{a} \mid \mathsf{ae} \mid \mathsf{b} \mid \mathsf{be} \mid \mathsf{e} \mid \mathsf{ne} & & (\mathit{Opr})\ \mathtt{o} ::= i \mid \mathtt{r} \\
(\mathit{Reg})\ \mathtt{r} ::= \mathsf{eax} \mid \mathsf{ebx} \mid \mathsf{ecx} \mid \mathsf{edx} \mid \mathsf{esi} \mid \mathsf{edi} \mid \mathsf{ebp} \mid \mathsf{esp} & & \\
(\mathit{Instr})\ \mathtt{c} ::= \mathsf{add}\ \mathtt{r}, \mathtt{o} \mid \mathsf{sub}\ \mathtt{r}, \mathtt{o} \mid \mathsf{mov}\ \mathtt{r}, \mathtt{o} \mid \mathsf{mov}\ \mathtt{r}, [\mathtt{d}] \mid \mathsf{mov}\ [\mathtt{d}], \mathtt{o} \mid \mathsf{cmp}\ \mathtt{r}, \mathtt{o} \mid \mathsf{j}cc\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{o} & & \\
\quad \mid \mathsf{push}\ \mathtt{o} \mid \mathsf{pop}\ \mathtt{r} \mid \mathsf{call}\ \mathtt{o} \mid \mathsf{ret} & &
\end{array}
$$

Fig. 1. Syntax of the x86 machine model

A program can be viewed as a collection of code blocks (instruction sequences) connected together with control-flow instructions such as jmp and call. Every code block has an associated assertion as its precondition, which is put together in a *code heap specification* Ψ (think of this as a header file). The verification is carried out by finding appropriate intermediate assertions for all program points, following inference rules.

To verify that a code block $\mathsf{add}\ \mathtt{r}, \mathtt{o}; \mathbb{I}$ is *well-formed* (*i.e.,* correct with respect to the specification) under pre-condition $\mathtt{a}$, which can be thought of as a predicate of type $\mathit{State} \rightarrow \mathit{Prop}$, we must find an intermediate assertion $\mathtt{a}'$ to serve both as the post-condition of $\mathsf{add}\ \mathtt{r}, \mathtt{o}$ and as the pre-condition of $\mathbb{I}$. More specifically, we must establish: (1) if $\mathtt{a}$ holds on a machine state, then $\mathtt{a}'$ holds on the updated machine state after executing $\mathsf{add}\ \mathtt{r}, \mathtt{o}$; (2) $\mathbb{I}$ is well-formed under pre-condition $\mathtt{a}'$. Below is the corresponding inference rule ($\Rightarrow$ and $\circ$ stand for assertion implication and function composition).

$$
\frac{\mathtt{a} \Rightarrow (\mathtt{a}' \circ \mathtt{Next}_{\mathsf{add}\ \mathtt{r},\mathtt{o}}) \qquad \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{add}\ \mathtt{r}, \mathtt{o}; \mathbb{I}}\ (\textsc{add})
$$

Reasoning about memory. The reasoning on memory (data heap and stack) operations is carried out following separation logic [16]. In particular, the primitives of separation logic are defined as shorthands using the primitives of the underlying assertion logic in XCAP via a shallow embedding. Some representative cases are given as follows.

$$
\begin{array}{rcl}
\mathsf{emp} & \triangleq & \lambda\mathbb{H}.\, \mathbb{H} = \{\} \\
\mathtt{l} \mapsto \mathtt{w} & \triangleq & \lambda\mathbb{H}.\, \mathtt{l} \neq \mathsf{NULL} \wedge \mathbb{H} = \{\mathtt{l} \rightsquigarrow \mathtt{w}\} \\
\mathtt{l} \mapsto \_ & \triangleq & \lambda\mathbb{H}.\, \exists \mathtt{w}.\, (\mathtt{l} \mapsto \mathtt{w}\ \mathbb{H}) \\
\mathtt{a}_1 * \mathtt{a}_2 & \triangleq & \lambda\mathbb{H}.\, \exists \mathbb{H}_1, \mathbb{H}_2.\, \mathbb{H}_1 \uplus \mathbb{H}_2 = \mathbb{H} \wedge \mathtt{a}_1\ \mathbb{H}_1 \wedge \mathtt{a}_2\ \mathbb{H}_2 \\
\mathtt{l} \mapsto \mathtt{w}_1, \ldots, \mathtt{w}_n & \triangleq & \mathtt{l} \mapsto \mathtt{w}_1 * \mathtt{l}{+}4 \mapsto \mathtt{w}_2 * \ldots * \mathtt{l}{+}4(n{-}1) \mapsto \mathtt{w}_n \\
\mathtt{l} \mapsto [n] & \triangleq & \mathtt{l} \mapsto \_, \ldots, \_ \qquad \textit{(the number of \_ is } n/4\textit{)}
\end{array}
$$

Reasoning about control-flow transfer. Direct control transfers are simple—when transferring control to code at label $\mathtt{f}$, one should establish the pre-condition of $\mathtt{f}$ as required by the code heap specification. This is illustrated with the following rule:

$$
\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{f}}\ (\textsc{jmpi})
$$

A distinguishing feature of XCAP compared to other program logics is the special support for embedded code pointers. It introduces a syntactic construct $\mathsf{cptr}(\mathtt{f},\mathtt{a})$ to serve as an assertion of "$\mathtt{f}$ points to code with pre-condition $\mathtt{a}$". For an indirect jump, the verification only uses the information enclosed in the cptr assertion. Modularity is achieved by not referring to the actual target address or the local code heap specification.

$$\frac{\mathtt{a} \Rightarrow (\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}'\ (\mathbb{H},\mathbb{R}) \ \wedge\ \mathsf{cptr}(\mathbb{R}(\mathtt{r}),\mathtt{a}'))}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmpr})$$

The above rule claims that it is safe to jump to register $\mathtt{r}$ under assertion $\mathtt{a}$, if $\mathtt{r}$ contains a code pointer with a pre-condition $\mathtt{a}'$ that is weaker than $\mathtt{a}$. Interested readers are referred to [12] for more details on cptr and the new assertion language, *PropX*. For this paper, please keep in mind that assertions are actually of type $State \rightarrow PropX$.

The function call and return instructions are supported as in the following new rules:

$$\frac{\mathtt{a} \Rightarrow (\Psi(\mathtt{f}) \circ \mathtt{Next}_{\mathsf{push}\ \mathtt{f}_{ret}}) \qquad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{call}\ \mathtt{f}; [\mathtt{f}_{ret}]}\ (\textsc{calli})$$

$$\frac{\mathtt{a} \Rightarrow \lambda(\mathbb{H},\mathbb{R},\mathbb{F}).\, \mathsf{cptr}(\mathbb{H}(\mathbb{R}(\mathsf{esp})),\mathtt{a}') \qquad \mathtt{a} \Rightarrow (\mathtt{a}' \circ \mathtt{Next}_{\mathsf{pop}})}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{ret}}\ (\textsc{ret})$$

A call instruction pushes a return address onto the stack and transfers control to the target code. In our actual implementation, the return address is calculated from the *pc*. To avoid obfuscating the presentation, we use an explicit $[\mathtt{f}_{ret}]$ in the above Rule CALLI. This rule says, if $\mathtt{a}$ holds on the current state, then $\Psi(\mathtt{f})$ holds on the updated state after executing the stack push. The rule RET says, the top of the stack is a code pointer with pre-condition $\mathtt{a}'$ and, if $\mathtt{a}$ holds on the current state, $\mathtt{a}'$ holds on the updated state after executing the stack pop (of the return address). It is worth noting that Rule CALLI does not enforce the validity of the return address. This allows some "fake" function calls that never return, a pattern indeed used in $loadcontext()$.

Stack and calling convention. The support for call and return instructions is one of the many adaptations made for x86. Besides specializing XCAP for the machine model, we also built key abstractions to help manage the complexity of the reasoning, such as the handling of the stack and calling convention. The calling convention is illustrated in Fig. 2. It is convenient to build a specification template reflecting this convention.

For a function with n arguments $a_1 \ldots a_n$, we write its specification (*i.e.,* a pre-condition in the form of an assertion) as:

$$\mathsf{Fn}\ \ a_1,\ldots,a_n\ \{\mathsf{Aux}: x_1,\ldots,x_m;\ \mathsf{Local}: [fs];\ \mathsf{Pre}: \mathtt{a}_{pre};\ \mathsf{Post}: \mathtt{a}_{post}\}$$

The intention of this macro is that $x_1,\ldots,x_m$ are "auxiliary variables" commonly used in Hoare-logic style reasoning, fs is the size of required free space on the stack, and $\mathtt{a}_{pre}$ and $\mathtt{a}_{post}$ are the pre- and post-conditions of the function. This macro is defined as:

$$\begin{aligned}\exists a_1,\ldots,a_n,cs,sp,ss,ret,x_1,\ldots,x_m,\mathtt{a}_{prv}.\ & \\ & \mathsf{reg}(cs,sp)\ \wedge\ ss \geq fs \\ & \wedge\ \mathsf{stack}(sp,ss,ret,a_1,\ldots,a_n) \quad * \ \mathtt{a}_{prv} * \mathtt{a}_{pre} \\ \wedge\ \mathsf{cptr}(ret,\ \exists retv.\ & \mathsf{reg}(cs,sp{+}4) \wedge \mathsf{eax} = retv \\ & \wedge\ \mathsf{stack}(sp{+}4,ss{+}4,a_1,\ldots,a_n) * \mathtt{a}_{prv} * \mathtt{a}_{post})\end{aligned}$$

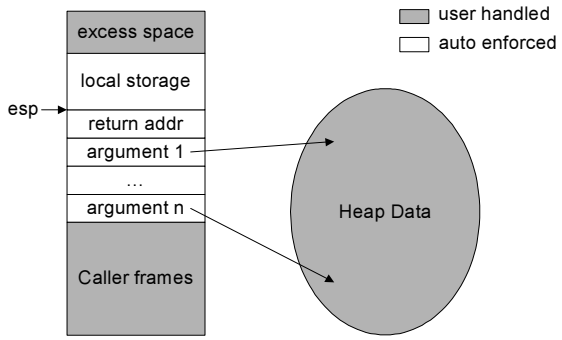

Fig. 2. Function calling convention

The first line of this definition quantifies over the values of (1) function arguments $a_1, \ldots, a_n$, (2) callee-save registers *cs* (a 4-tuple), (3) the stack pointer *sp*, (4) the size of available space on stack *ss*, (5) the return address *ret*, (6) auxiliary variables $x_1, \ldots, x_m$, and (7) some hidden private data expressed as the predicate a_{prv}.

The second line relates the register file with the callee-save values *cs* (4-tuple) and the stack pointer *sp* and makes sure that there is enough space available on the stack.

$$\mathsf{reg}(ebx, esi, edi, ebp, esp) \triangleq \mathsf{ebx} = ebx \wedge \mathsf{esi} = esi \wedge \mathsf{edi} = edi \wedge \mathsf{ebp} = ebp \wedge \mathsf{esp} = esp$$

The third line describes (1) the stack frame upon entering the function using the macro below, (2) the private data hidden from the function, and (3) the user customized pre-condition a_{pre}, which does not directly talk about register files and the current stack frame, because they are already handled by the calling convention.

$$\mathsf{stack}(sp, ss, w_1, \ldots, w_n) \triangleq sp - ss \mapsto [ss] * sp \mapsto w_1, \ldots, w_n.$$

The last two lines of the Fn definition specify the return address *ret* as an embedded code pointer using cptr. When a function returns, the callee-save registers, stack frame, and private data must all be preserved, and the post-condition a_{post} must be established. Note that (1) eax may contain a return value *retv*, and (2) the return instruction automatically increases the stack pointer by 4.

4 Formal Verification of Machine Context Management

What is a machine context? Although the C specification in Section 2 appears to indicate that a context is merely eight words, the actual assumptions underlying it are rather complex. As illustrated in Figure 3, the eight words represent a return value *retv*, six registers referred to collectively as *cs*, and a stack pointer *sp*. *sp* points to a return address *ret* found on top of a stack frame. The saved registers may point to some private data. There may also be some environment data shared with external code. Eventually, a context is consumed when being invoked. A correct invocation of jumping to the return address *ret* requires (1) the saved register contents be restored into the register file, (2) the stack and private data be preserved, and (3) the shared environment be available.

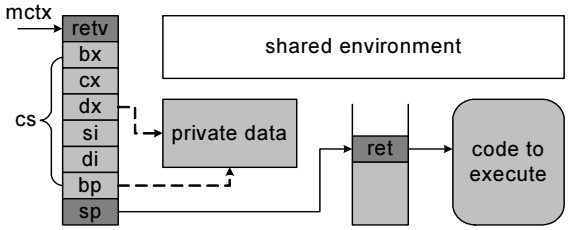

Fig. 3. Machine context

All these requirements make it challenging to specify the invariants on contexts. Because of the expressiveness of XCAP, we can define a heap predicate $\mathsf{mctx\_t}(\mathsf{a}_{env}, mctx)$ for the context data structure, parametric to the environment described as a_{env},

$$\exists retv, cs, sp, ret, \mathsf{a}_{prv}. \qquad mctx \mapsto retv, cs, sp \ * \ sp \mapsto ret \ * \ \mathsf{a}_{prv}$$
$$\wedge\ \mathsf{cptr}(ret, \mathsf{reg}_6(cs, sp{+}4)\ \wedge\ \mathsf{eax} = retv\ \wedge\ \mathsf{a}_{env}\ *\ mctx \mapsto retv, cs, sp\ *\ sp \mapsto ret\ *\ \mathsf{a}_{prv})$$

where a_{prv} describes the private data, and reg_6 is defined similarly to reg with 6-tuple.

Context switching. The *swapcontext*() function take two pointers (one to the old context and another to the new context) and performs three tasks: saves registers to the old context, loads registers from the new context, and transfers control to the new context. From the implementation, *swapcontext*() gets called by one client and "returns" to another. However, this is entirely transparent to the clients—when *swapcontext*() returns, the stack and private data of that client are kept intact.

We present the specification and proof outline of *swapcontext*() in Fig. 4. It uses a macro Fn_6, a variant of Fn obtained by replacing reg with reg_6 and changing *cs* to refer to 6 registers. Fn_6 automatically manages the preservation of the stack and private data. In addition, the pre-condition specifies three pieces of memory: (1) the old context pointed to by *old*—at the beginning of the routine it is simply 32 bytes of memory available for use, (2) the shared data a_{env}, and (3) the new context pointed to by *new*.

The new context is specified with the help of the macro mctx_t. The environment parameter of this macro consists of two parts: the shared data a_{env} and another mctx_t macro, describing the old context. This is because the old context will be properly set up by the routine before switching to the new one. One tricky point is that the old context will be expecting an (existentially quantified) **new shared environment** a_{newenv}. Although one may expect the new environment to be simply the old one, a_{env}, together with the new context at *new*, this may not necessarily be the case. For instance, the new context may be dead already and de-allocated when the old context regains control.

The post-condition of *swapcontext*() is relatively simple: the space for the old context will still be available, together with the new shared data a_{newenv}.

An interesting proof step in Fig. 4 is the one after the old context is packed but before the new one is unpacked. At that point, there is no direct notion of stack or function. The relevant machine state essentially comprises of two contexts and one environment:

$$\mathsf{eax} = new \wedge\ \mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}\ * \mathsf{mctx\_t}(\mathsf{mctx\_t}(\mathsf{a}_{newenv}, old) * \mathsf{a}_{env}, new)$$

Intuitively, it should be safe to load the new context from eax and then switch to it.

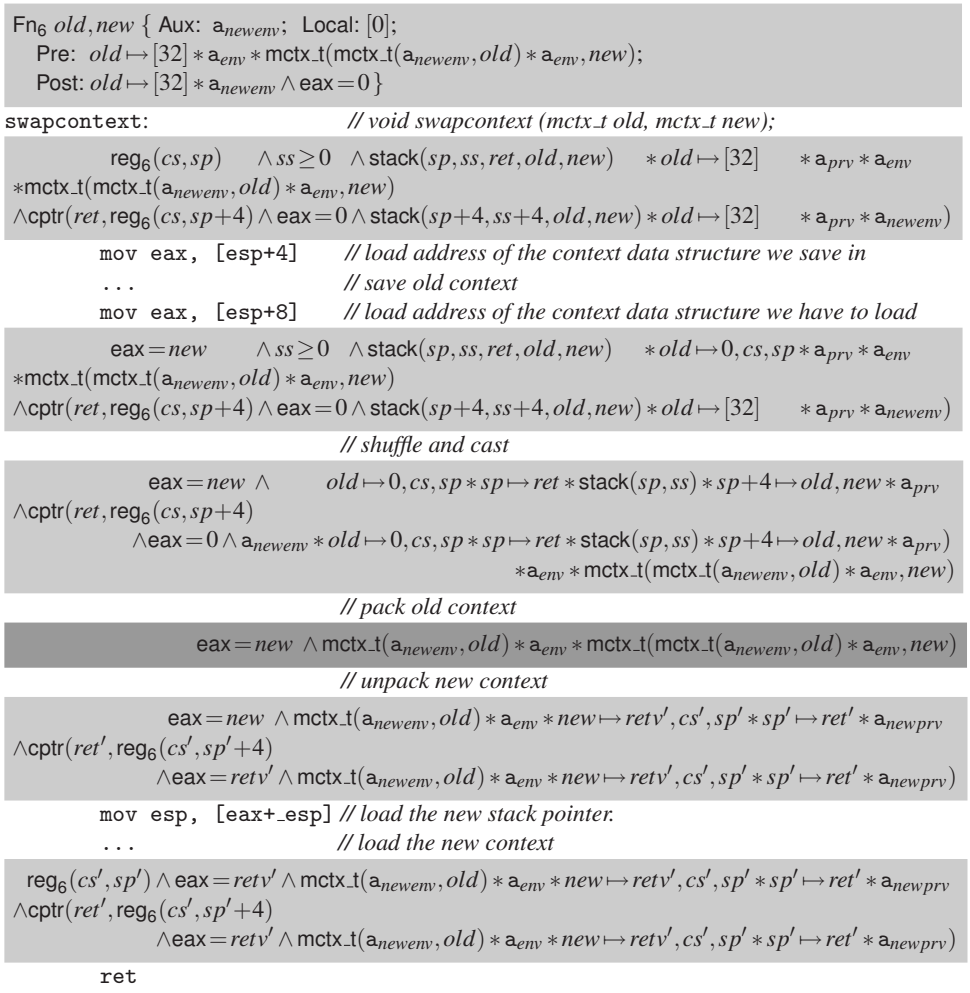

Fig. 4. Verification of machine context switching

Context creation. We present the specification and proof outline of *makecontext*() in Figure 5. The intermediate assertions are also organized using Fn. For conciseness, we omitted common parts of these macros, thus emphasizing only the changeable parts.

The pre-condition of the routine specifies (1) an empty context at *mctx*, (2) a stack *nsp* with available space of size *nss*, (3) some private data of the target context (potentially accessible from the argument *arg* of the function $func()$ of the target context, (4) a link (return address) *lnk* to be used when the target context finishes execution, and (5) a function pointer $func()$ for the code to be executed in the target context. It also specifies the exact requirements on the code at pointers *lnk* and $func$: (1) a_{ret} occurs both in the post-condition of $func()$ and in the pre-condition of *lnk*, indicating that the code at the return address *lnk* may expect some results from the context function $func()$; (2)

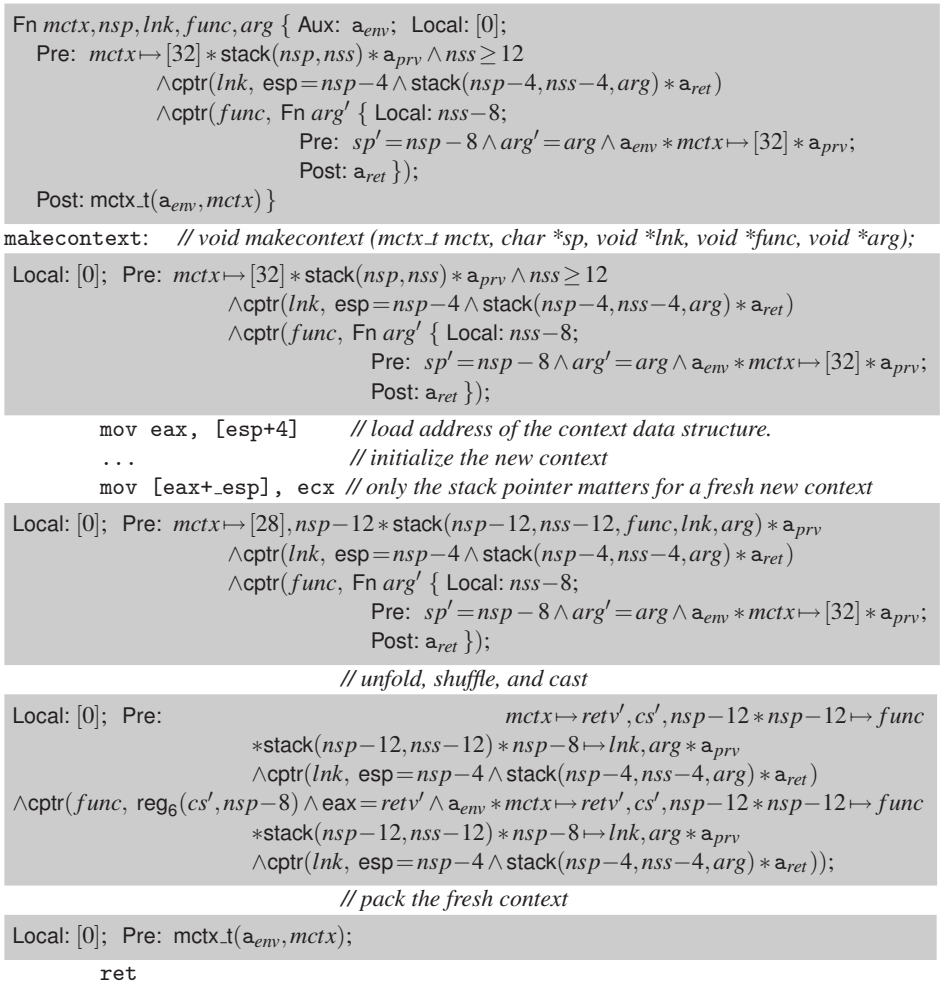

Fig. 5. Verification of machine context creation

the stack should be properly maintained upon returning to *lnk*. The post-condition of the routine simply states that, when *makecontext*() returns, *mctx* will point to a proper context which expects a shared environment of a_{env}.

Our interface of *makecontext*() is faithful to the Unix/Linux implementations. The heavy usage of function and continuation pointers shows how crucial XCAP's support of embedded code pointers is. As the proof shows, the most complex step is on transforming code pointers' pre-conditions and packing all the resources into a context.

Context loading. *loadcontext*() essentially performs the second half of the task of *swapcontext*(), with some differences in the stack layout. Although we refer to it as a "function", it actually never returns and does not require the stack top to contain a valid return address. We present the verification of *loadcontext*() in Figure 6.

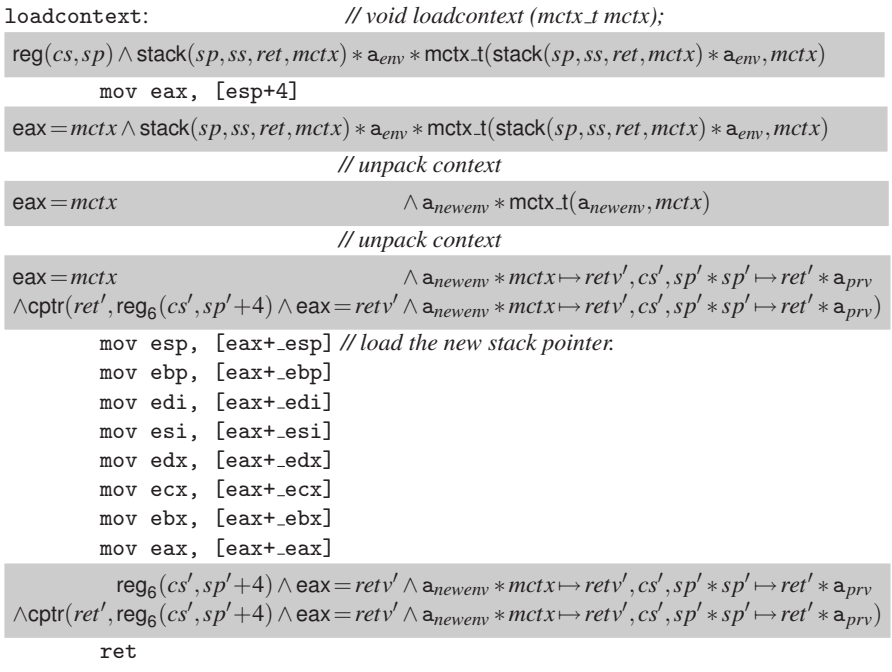

Fig. 6. Verification of machine context loading

5 The Coq Implementation

We have mechanized everything in this paper using the Coq proof assistant [18]. By everything we mean the machine model, XCAP and its meta theory (including soundness proof), separation logic, and specification and proof of the context implementation.

Proof details. Our Coq implementation (downloadable at [14]) consists of approximately 17,000 lines of Coq code. Figure 7 presents a break down of individual components in the implementation. The full compilation of our implementation in Coq (proof-script checking and proof-binary generation) takes about 52 minutes under Windows Vista on an Intel Core 2 Duo T7200 2GHz CPU with 4MB L2 cache and 2GB memory.

The proof for lemmas and the context code is typically on implications from one assertion to another, either on the same machine state (casting), or on two consecutive states before and after the execution of an instruction. Proof steps mostly adjust the shape of the assertions to match, split implications into smaller ones, and track the changed parts down to the basic data units. Coq's proof searching ability is useful, but often too weak for the reasoning about memory and resources.

Coq has many built-in proof tactics for intuitive handling of proofs of *Prop* sort, such as *intro, split, left, right, exists, cut, auto, etc.* Following them, we defined a similar set of proof tactics for *PropX*, such as *introx, splitx, leftx, rightx, existsx, cutx, autox, etc.* Each of these new *PropX* tactics can be used similarly to the *Prop* ones, which increases the productivity when working with *PropX* for experienced Coq users.

Implementation component	Line of code	Compile time
Safe extension to Coq on extensional equality on functions	9 Loc	1s
Math lemmas	384 Loc	1s
Set level mapping	220 Loc	1s
Type level mapping	218 Loc	1s
Type list	103 Loc	1s
Target machine (TM)	441 Loc	10s
PropX meta theory (independent of XCAP meta theory)	4,504 Loc	6m
PropX lemmas (independent of XCAP meta theory)	258 Loc	3s
Separation-logic style memory assertions	74 Loc	1s
Common data-structure related lemmas (independent of XCAP meta theory)	1,171 Loc	10s
XCAP (x86 version)	755 Loc	3s
XCAP common definitions and lemmas	297 Loc	3s
Machine context module: code, data structure, and specification	144 Loc	1s
Proof for loadcontext() function	753 Loc	15s
Proof for swapcontext() function	4,917 Loc	3m
Proof for makecontext() function	2,372 Loc	40m
Total (on Intel Core 2 Duo T7200 2GHz / 4MB L2 / 2GB Ram)	16,620 Loc	52m

Fig. 7. Break-down of Coq implementation

To make *PropX* formulas more readable, we defined some pretty-printing notations. For example, one can directly write the following formula:

```
All x, <<even x>> .\/ Ex y, z. <<x = y + z>> ./\ <<prime y /\ prime z>>
```

and write a function specification in the same style as used in our paper presentation:

```
Fn ... {Aux : ...; Local: [...]; Pre: ... ; Post: ...}
```

Development time. The development of the proof took about three person-months. There are several reasons for the large proof size and development time. First of all, this is the first time we did this kind of realistic proof, so a lot of infrastructure code and experience needed to be developed and learned. For example, about half of the 17,000 lines of code is independent of machine context verification; such code can be reused for other verification purposes. The first procedure we certified, *swapcontext*(), took one person-month and 5,000 lines of code. We believe these numbers would be at least halved if we did it again. Second, there is much redundancy due to the relatively low level of proof reuse. The third reason is the complexity of the machine model, which needs features such as finite integers that are not supported well in Coq. Nevertheless, the biggest reason, we believe, is the complexity of reasoning about the actual code.

As an example, the machine context data type in the previous section is implemented as in Figure 8, which is far more complex than what its meta presentation looks like. Part of the complexity is due to the de Bruijn representation of impredicative quantifiers. Further improved pretty printing notations can help simplify the presentation. Still, there is the inherent complexity surrounding a machine context data structure that can not be omitted or avoided by any formal reasoning about it.

```
Definition mctx_t L aenv mctx : Heap -> PropX L := fun H =>
Ex retv, bx, cx, dx, si, di, bp, sp, ret.
 star (ptolist _ mctx (retv::bx::cx::dx::si::di::bp::sp::nil))
(star (pto _ sp ret)
      (fun H => extv _ _ (eq_rect _ _
        (Lift _ _ (var t0 _ H)
     ./\ codeptr _ ret
          (fun S' => match S' with ((H',R'),F') in
             reg6 _ bx cx dx si di bp (sp+4) R' ./\ << R' eax = retv>>
         ./\ star (fun H => Shift _ _
                              (eq_rect _ _ (aenv H) _ (app_nil_eq _)) _)
            (star (ptolist _ mctx (retv::bx::cx::dx::si::di::bp::sp::nil))
            (star (pto _ sp ret)
                  (fun H => Lift _ _ (var t0 _ H)))) H'
           end))                    _ (nil_app_eq _))))                H.
```

Fig. 8. Coq encoding of mctx_t

Discussion on the proof size. The large size of the proof naturally raises practicality concerns. However, when making comparisons with other verification methods, there are several aspects of our method that need to be taken into consideration.

For comparison with abstraction-based verification methods, we believe that while they provide insights on the invariants of the algorithms and high-level programs, for low-level systems code such as context management, working at an abstract level will not have an actual advantage. As explained in the beginning of this paper, it is the level it works at and the complexity of this code that are most interesting.

For comparison with analysis- and test-based verification methods, the important question to ask is what kind of guarantee one can deliver. Many of these tools can automatically find bugs, but only in a "best-effort" fashion—false negatives are expected. In our case, we want to completely exclude certain categories of bugs, as guaranteed by the language-based approach in general.

For comparison with other language-based methods, such as type systems, the important question to ask is what kind of code is supported. There have been efforts to use higher-level safe languages for systems programming, all with trade-offs in efficiency and/or compatibility, some even with significant changes in programming style. In our case, we require no change to the existing systems programming style and code base, and thus we expect no performance or compatibility issues. In general, we believe that systems and application code requires different level of safety guarantees, thus their verification will naturally result in different levels of productivity. For example, it is feasible to automatically generate proofs for TAL programs in XCAP [13] and link them with the certified context routines in this paper following methods in [13] or [2].

6 Related Work and Conclusion

Hoare-logic style systems have been widely used in program verification. They support formal reasoning using assertions and inference rules based on expressive program logics. Nonetheless, there has been no work in the literature on the logical specification

and verification of machine context management. The challenge there, based on our experience, largely lies in working with embedded code pointers. For example, among the efforts applying Hoare-logic-based verification to machine code, an earlier work, Yu *et al* [21], is able to do mechanized verification of MC68020 object code. However, as explained in [12], they cannot support embedded code pointers and they encountered the problem of "functional parameters". Recently, Myreen *et al* [10] support verification of realistic ARM code, but support of embedded code pointers is missing.

Separation logic [16, 6] provides a helpful abstraction for working with memory. Our verification makes use of a shallow embedding of separation logic in the assertion language to reason about the machine state.

The Singularity OS [5] uses a mixture of C# and assembly code to implement threading. Whereas most parts of the code enjoy type safety, context switching is written in unsafe assembly code. Furthermore, threading and context data structures are specified using managed data pointers, which belong to the *weak update* memory model (ML-like reference types also belong to it). It is significantly different from the separation-logic style unmanaged *strong update* memory model used in this paper, as well as in Unix/Windows kernel. One disadvantage for using *weak update* for certified systems programming is that its memory management relies on garbage collection, which may not be appropriate for some systems programs.

The verification of context switching in this paper addresses the actual implementation details, including those hidden behind the implementation of continuations and storage management. The code we verified is at the same level as the actual executable running on a real machine. The trusted computing base is extremely small. Besides the obvious progress/preservation property, because our verification is based on logical assertions, it conveys a more expressive safety policy: the run-time states of the machine should satisfy the corresponding assertions as written in the specification.

Conclusion. We have presented the first formal, modular, and mechanized verification of machine context management code and showed how to build a verification infrastructure for certifying realistic low-level systems code. In particular, we applied the XCAP framework to an x86 machine model, developed new proof-tactics and lemma libraries, and used it to certify the code and data structures in an x86 machine-context implementation. Everything has been mechanized in the Coq proof assistant. Our approach is applicable to other variants of context, provides a solid basis for further verification of thread implementations, and illustrates the formal, modular, and mechanized verification of realistic systems code in general.

By following better engineering practices and building smarter domain-specific proof-searching tools, especially those for separation logic and program logics, we believe that the automation in systems code specification and verification can be gradually increased to a level where it is practical to apply to the building of provably correct software.

Acknowledgment. We thank David Tarditi and anonymous referees for suggestions and comments on the draft of this paper. A large part of this research was done when the first author was at Yale University and supported in part by gifts from Intel and Microsoft, and NSF grant CCR-0524545. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. A. W. Appel, P.-A. Mellies, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th ACM Symp. on Principles of Prog. Lang.*, Jan. 2007.
2. X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. Workshop on Types in Language Design and Implementation*, Jan. 2007.
3. M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher-Order Logics*, pages 2–16. Springer-Verlag, 2005.
4. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.
5. G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, Oct. 2005.
6. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 14–26, 2001.
7. Jabber software development list. Jabberd crash in swapcontext() via _mio_raw_connect(). `http://mailman.jabber.org/pipermail/jdev/2001-March/005655.html`, 2001.
8. Libc for alpha systems mailing list. {make,set,swap}context broken on powerpc32. `http://www.archivesat.com/Libc_for_alpha_systems/thread2267226.htm`, 2006.
9. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, Jan. 1998.
10. M. O. Myreen and M. J. C. Gordon. Hoare logic for realistically modelled machine code. In *Proc. 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2007.
11. NetBSD port-amd64 mailing list. swapcontext(3) does not work? `http://mail-index.netbsd.org/port-amd64/2004/11/30/0003.html`, 2004.
12. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symp. on Principles of Prog. Lang.*, Jan. 2006.
13. Z. Ni and Z. Shao. A translation from typed assembly languages to certified assembly programming. `flint.cs.yale.edu/flint/publications/talcap.html`, Oct. 2006.
14. Z. Ni, D. Yu, and Z. Shao. Coq code for using xcap to certify realistic systems code: Machine context management. `http://flint.cs.yale.edu/flint/publications/mctx.html`, Mar. 2007.
15. Z. Ni, D. Yu, and Z. Shao. Technical report for using xcap to certify realistic systems code: machine context management. `http://flint.cs.yale.edu/flint/publications/mctx.html`, June 2007.
16. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. 17th Symp. on Logic in Computer Science*, 2002.
17. SecurityTracker.com. Solaris 10 x64 kernel setcontext() bug lets local users deny service. `http://securitytracker.com/alerts/2006/Feb/1015557.html`, 2006.
18. The Coq Development Team. The Coq proof assistant reference manual (v8.0), 2004.
19. The glibc-bugs mailing list. [bug libc/357] new: getcontext() on ppc32 destroys saved parameter 1 in caller's frame. `http://sourceware.org/ml/glibc-bugs/2004-08/msg00201.html`, 2004.
20. The glibc-bugs mailing list. [bug libc/612] new: makecontext broken on powerpc-linux. `http://sources.redhat.com/ml/glibc-bugs/2004-12/msg00102.html`, 2004.
21. Y. Yu. *Automated Proofs of Object Code For A Widely Used Microprocessor*. PhD thesis, The University of Texas at Austin, 1992.