

Modularly Certified Dynamic Storage Allocation in SCAP

Sen Xiang Yiyun Chen Chunxiao Lin Long Li
Computer Science and Technology Department,
University of Science and Technology of China,
Hefei, Anhui, China
{sengx,cxlin3,liwis}@mail.ustc.edu.cn
yiyun@ustc.edu.cn

Abstract

Critical applications and increasing scale of software has made software assurance a big problem. Currently, programmers can write type-safe codes in typed languages with sound type systems, such as Java, Cyclone, even typed assembly language(TAL). But high assurance does mean not only type safe, but also correctness and security. Since type is not expressive enough, there are still no high assurance software released in typed language, especially operating system and runtime library, which are infrastructure softwares in the computing system. Logic predicates are more expressive than types, thus substituting types with logic predicates as program specification seems a good idea. In this paper, we present a certified dynamic storage allocation library (malloc/free) in SCAP, which is a new MIPS-like assembly language with expressive and power specification structure. And we encode the SCAP language and the certified library in a modern higher-order logic(HOL) proof assistant Coq. In this work, we confirm the expressiveness and modularity of SCAP. So we can expect SCAP to be a expressive target language for some safe high-level languages in the future.

1. Introduction

In the research field of code safety, program verification(or certification with minor difference) is a strict and reliable methodology. Frameworks of program verification mainly include Typed Assembly Language (TAL)[7, 6], Proof Carrying Code(PCC)[8], etc.

TAL programs are statically guaranteed safe through its sound type system, which provides support for enforcing high-level language abstractions, such as closure, tuples, and user-defined abstract data types. The type system ensures that well-typed programs never violate these abstractions. Additionally, the typing constructs admit many

low-level compiler optimization. It can serve as a target language for compilers of high-level languages such as ML. The original TAL assume the compiler would perform a continuation-passing style(CPS) transform and eliminate the control stack by heap-allocating activation records, while most compilers are based on stack allocation. Therefore, a stack-based TAL[6] is proposed. Since types can not express enough program properties, TAL or STAL is not suitable for high assurance programming. Other researches such as dependent type[11] try to add new type constructor in order to enrich type systems, but the added type primitives and typing rules make its soundness proof difficult.

PCC[8] is a general framework pioneered by Necula and Lee, and it provides a mechanism to verify the safety of mobile codes through machine-checkable proof before executing them. The machine-checkable proof is explicitly carried in PCC code package, while TAL implicitly reconstructs the proof through type checking. Existing PCC systems have focused only on programs whose safety proofs can be automatically generated, such as TAL programs. Since types have not the same expressiveness as logic predicates, many low-level system libraries, such as dynamic storage allocation, have not been certified. However, these infrastructure softwares are very important components of software systems worth certification with some costs. Additionally, PCC system has a big Verification Condition Generator(VCGen), which introduces a big trusted computing base(TCB). Foundation PCC [1](FPCC) is proposed to decrease the TCB of PCC systems. FPCC is verification from the smallest possible set of axioms, using the simplest possible verifier and the smallest possible runtime system.

Certifying Assembly Programming[12](CAP) provides a complementary approach for PCC in which general properties and correctness are semi-automatically certified. CAP is a reasoning framework of Hoare-logic[5] style at assembly level, in which a certified dynamic storage allocation library has been built. After that, More CAP extensions for different program features are proposed, such as CMAP[2],

CCAP[13]. But original CAP is based a relative ideal machine model, which means the proofs possibly do not hold in physical machine. More importantly, the simplicity of CAP limits its expressiveness in the context of modularity, which means library routines' specification and proofs can not written independently from higher-level applications. This greatly reduces its applicability in large-scale code certification. In order to solve the problem of modularity, stack-based CAP[4] (SCAP) is proposed, which added function guarantee “g” into its specification structure. More details about SCAP will be introduced in section 2.

Based on SCAP, we make following contributions:

- We formalize the SCAP language in Coq[10], including target machine(mainly operational semantics), SCAP's static semantics, and its soundness theorem , which is the first step of program certification. We also enlarge the instruction set without much efforts, according to the requirements of realistic certifying programming. The description about instructions of the abstract machine and their semantics is incomplete in this paper for the sake of conciseness.
- We port certified dynamic storage allocation(malloc/free) in SCAP and implement a certified link list copy program calling malloc/free routines. It is clean and practical compared with the implementation in old CAP. All routines are independently certified and then linked together. The characteristic of our implementation ensures the possibility of applying this technique into large-scale systems.

In this paper, we first introduce SCAP language in section 2 and its Coq encodings in section 3. then the modular certification of dynamic storage allocation library in section 4. At last, comparison with related work and future work will be presented. In appendix, we give more details about the annotated low-level codes. Complete Coq encodings are now available at [3].

2. The SCAP Language

2.1. Syntax

SCAP is a MIPS-like assembly language with powerful specification structures, which can express accurate program properties. Figure 1 shows the syntax of SCAP language. A SCAP program is of type \mathbb{P} , which consists of a read-only code heap(\mathbb{C}), an updatable abstract storage state(\mathbb{S}), and an entry instruction sequence(\mathbb{I}). \mathbb{C} is a finite partial mapping from code labels to \mathbb{I} . \mathbb{S} contains a register file (\mathbb{R}) and a data heap (\mathbb{H}). \mathbb{I} is a code block ended with some jump instruction. In this abstract machine, \mathbb{I} is

involved instead of a program counter for simplicity. SCAP program specification (code heap specification) is of type Ψ , which is a map from code block label to its pre-assertion p and guarantee g . p specifies the state assertion that must be satisfied when control reaches the code block, while g specifies the assertion on current state and the state at current function return point. Using g , SCAP can do modular function call reasoning.

(Program)	\mathbb{P}	::=	$(\mathbb{C}, \mathbb{S}, \mathbb{I})$
(CodeHeap)	\mathbb{C}	::=	$\{f \rightsquigarrow \mathbb{I}\}^*$
(State)	\mathbb{S}	::=	(\mathbb{H}, \mathbb{R})
(Heap)	\mathbb{H}	::=	$\{l \rightsquigarrow w\}^*$
(RegFile)	\mathbb{R}	::=	$\{r \rightsquigarrow w\}^*$
(Registers)	r	::=	$\{r_k\}_{k \in \{0 \dots 31\}}$
(Nat)	w, f, l	::=	$i \text{ (nat nums)}$
(InstrSeq)	\mathbb{I}	::=	$c; \mathbb{I} j \ f jal \ f, f_{ret}$ $j \ r \ r_s$
(Command)	c	::=	$addu \ r_d, r_s, r_t$ $addiu \ r_d, r_s, i$ $subu \ r_d, r_s, r_t$ $beq \ r_s, r_t, f \dots$ $lw \ r_t, i(r_s)$ $sw \ r_t, i(r_s)$
(Assertion)	p, q	\in	$State \rightarrow Prop$
(Guarantee)	g	\in	$State \rightarrow State \rightarrow Prop$
(CdSpec)	θ	::=	(p, g)
(CdHpSpec)	Ψ	::=	$\{f \rightsquigarrow \theta\}^*$

Figure 1. SCAP Syntax

SCAP Programmers can use 32 registers. According to the MIPS register usage convention, these registers have following aliases:

zero	r_0	always zero
$v_0 - v_1$	$r_2 - r_3$	return values
$a_0 - a_3$	$r_4 - r_7$	arguments
$t_0 - t_9$	$r_8 - r_{15}, r_{24} - r_{25}$	caller-saves
$s_0 - s_7$	$r_{16} - r_{23}$	callee-saves
sp	r_{29}	stack pointer
fp	r_{30}	frame pointer
ra	r_{31}	return address

2.2. Operational Semantics

The execution of program in the abstract machine is modeled as a small-step transition from one program to another, like $\mathbb{P} \mapsto \mathbb{P}'$. Figure 2 shows the transition function. In this figure, $\mathbb{R}\{- \rightsquigarrow -\}$ and $\mathbb{H}\{- \rightsquigarrow -\}$ mean the update of corresponding register and heap slot. The operational semantics of most instructions are the same with corresponding MIPS instructions.

2.3. The Static Semantics

The static semantic of SCAP concerns the well-formedness judgements at four level: Well-formed program

if $\mathbb{I} =$	then $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \mapsto$
j f	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(f))$ when $f \in \text{dom}(\mathbb{C})$
jal f, f_{ret}	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}\{r_{31} \rightsquigarrow f_{ret}\}), \mathbb{C}(f))$ when $f \in \text{dom}(\mathbb{C})$
jr r_s	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathbb{R}(r_s)))$ when $\mathbb{R}(r_s) \in \text{dom}(\mathbb{C})$
beq $r_s, r_t, f; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ when $\mathbb{R}(r_s) \neq \mathbb{R}(r_t)$ $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(f))$ when $\mathbb{R}(r_s) = \mathbb{R}(r_t)$
c; \mathbb{I}'	$(\mathbb{C}, \text{Next}_c(\mathbb{H}, \mathbb{R}), \mathbb{I}')$

where

if c =	then $\text{Next}_c(\mathbb{H}, \mathbb{R}) =$
addu r_d, r_s, r_t	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + \mathbb{R}(r_t)\})$
addiu r_d, r_s, i	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + i\})$
subu r_d, r_s, r_t	$(\mathbb{H}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) - \mathbb{R}(r_t)\})$
lw $r_t, i(r_s)$	$(\mathbb{H}, \mathbb{R}\{r_t \rightsquigarrow \mathbb{H}(\mathbb{R}(r_s) + i)\})$ when $\mathbb{R}(r_s + i) \in \text{dom}(\mathbb{H})$
sw $r_t, i(r_t)$	$(\mathbb{H}\{\mathbb{R}(r_s) + i \rightsquigarrow \mathbb{R}(r_t)\}, \mathbb{R})$

Figure 2. SCAP Operational Semantics

gram, well-formed control stack (predicate WFST inductively defined by WFST-0 and WFST-N), well-formed code heap and well-formed instruction sequence, as figure 3 shows. These rules are all hoare logic style, which means the precondition of a instruction must imply postcondition. Top part of these rules related with g are for the preservation of stack well-formedness preservation after one step execution. Since we are not discussing the internals of SCAP in this paper, interested readers can get more details in [4, 3].

3. SCAP encoding in Coq

Our work use Coq to develop SCAP programs and their well-formedness proof, so at first SCAP must be encoded in Coq. File `scap.v` in [3] encodes SCAP syntax and operational semantics, while `scap_rules.v` encodes its static semantics (inference rules) and soundness proof in `scap_sound.v`. In this section, main syntax objects' Coq encodings of SCAP are given.

The SCAP program type is defined in Coq as `world`:

Notation `world := (state × (codeheap × iseq)) % type`, in which `state`, `codeheap`, `iseq` are defined as following:

Notation `state := (heap × rfile) % type`.

Inductive `codeheap : Set :=`

| `emptyCH : codeheap`
| `consCH : lab → iseq → codeheap → codeheap`.

Inductive `rfile : Set :=`

| `emptyR : rfile`
| `consR : rfile → reg → nat → rfile`.

Definition `heap : Set := addr → hvopt`.

Inductive `iseq : Set :=`

| `seq : iota → iseq → iseq`
| `j : lab → iseq`
| `jal : lab → lab → iseq`
| `jr : reg → iseq`.

$\Psi; (p, g) \vdash \mathbb{P}$	(Well-formed Program)
$\Psi \vdash \mathbb{C} : \Psi \quad pS \quad \exists n, \text{WFST}(n, gS, \Psi) \quad \Psi; (p, g) \vdash \mathbb{I}$	(PROG)
$\Psi; (p, g) \vdash (\mathbb{C}, S, \mathbb{I})$	
$\text{WFST}(n, q, \Psi)$	(Well-formed Control stack)
$\forall S, qS \rightarrow S.\mathbb{R}(ra) = \text{NULL}$	(WFST-0)
$\text{WFST}(0, q, \Psi)$	
$\forall S, qS \rightarrow (S.\mathbb{R}(ra) \in \text{dom}(\Psi) \wedge p'S$ $\wedge \text{WFST}(n-1, q'S, \Psi) \quad \text{where } (p', q') = \Psi(S.\mathbb{R}(ra))$	(WFST-N)
$\text{WFST}(n, q, \Psi)$	
$\Psi \vdash \mathbb{C} : \Psi$	(Well-formed Code Heap)
$\Psi; \Psi(f) \vdash \mathbb{C}(f) \quad \forall f \in \text{dom}(\mathbb{C})$	(LINK)
$\Psi \vdash \mathbb{C} : \Psi$	
$\Psi; (p, g) \vdash \mathbb{I}$	(Well-formed Instruction Sequence)
$c \in \{\text{addu}, \text{addiu}, \text{lw}, \text{subu}, \text{sw}\}$ $\Psi; (p', g') \vdash \mathbb{I}' \quad \forall S.pS \rightarrow p'(\text{Next}_c(S))$ $\forall S, S'.pS \rightarrow g'(\text{Next}_c(S))S' \rightarrow gSS'$	(CMD)
$\Psi; (p, g) \vdash c; \mathbb{I}$	
$(p', g') = \Psi(f)$ $\forall S.S.\mathbb{R}(r_s) = S.\mathbb{R}(r_t) \rightarrow pS \rightarrow p'S$ $\forall S, S'.S.\mathbb{R}(r_s) = S.\mathbb{R}(r_t) \rightarrow pS \rightarrow g'SS' \rightarrow gSS'$ $\forall S.S.\mathbb{R}(r_s) \neq S.\mathbb{R}(r_t) \rightarrow pS \rightarrow p'S$ $\forall S, S'.S.\mathbb{R}(r_s) \neq S.\mathbb{R}(r_t) \rightarrow pS \rightarrow g'SS' \rightarrow gSS'$ $\Psi; (p'', g'') \vdash \mathbb{I}'$	(BRANCH)
$\Psi; (p, g) \vdash \text{beq } r_s, r_t, f; \mathbb{I}'$	
$(p', g') = \Psi(f)$ $\forall S.pS \rightarrow p'S \quad \forall S, S'.pS \rightarrow g'SS' \rightarrow gSS'$	(JUMP)
$\Psi; (p, g) \vdash j \ f$	
$(p', g') = \Psi(f) \quad (p'', g'') = \Psi(f_{ret})$ $\forall \mathbb{H}, \mathbb{R}.p(\mathbb{H}, \mathbb{R}) \rightarrow p'(\mathbb{H}, \mathbb{R}\{r_{31} \rightsquigarrow f_{ret}\})$ $\forall \mathbb{H}, \mathbb{R}, S'.p(\mathbb{H}, \mathbb{R}) \rightarrow g'(\mathbb{H}, \mathbb{R}\{r_{31} \rightsquigarrow f_{ret}\})S' \rightarrow p''S'$ $\forall \mathbb{H}, \mathbb{R}, S', S''.p(\mathbb{H}, \mathbb{R}) \rightarrow$ $g'(\mathbb{H}, \mathbb{R}\{r_{31} \rightsquigarrow f_{ret}\})S' \rightarrow g''S'S'' \rightarrow g(\mathbb{H}, \mathbb{R})S''$ $\forall S, S'.g'SS' \rightarrow S.\mathbb{R}(r_{31}) = S'.\mathbb{R}(r_{31})$	(CALL)
$\Psi; (p, g) \vdash \text{jal } f, f_{ret}$	
$\forall S.pS \rightarrow (gSS \wedge S.\mathbb{R}(r_{31}) \neq \text{NULL})$	(RET)
$\Psi; (p, g) \vdash \text{jr } ra$	

Figure 3. SCAP Static Semantics

Type `state` is a tuple of register file type `rfile` and heap type `heap`; Type `codeheap` is inductively defined by two constructor `emptyCH` and `consCH`, which builds a new codeheap from a label, label-corresponding instruction sequence, and a old smaller codeheap; Similarly, `rfile` is defined by constructors `rfile` and `consR`; `heap` is a function type mapping addresses into heap value; Instruction sequence type are defined as `iseq`, which is an operation instructions(`iota`) list ended with `j`, `jal`, `jr`.

The type of assertion `p` is defined as a predicate type `assert` over `state`.

Notation `assert := (state → Prop)`.

And type of guarantee `g` is defined as a predicate type action over two `state` parameters:

Notation `action := (state → state → Prop)`.

The code specification type θ and code heap specification map type Ψ are defined as:

Inductive `codespec : Type :=`

`cdspec : assert → action → codespec.`

Inductive `codeheapTy : Type :=`

`| emptyCT : codeheapTy`

`| consCT : lab → codespec → codeheapTy → codeheapTy.`

The operational semantics are encoded in Coq as a step predicates which is of type `world → world → Prop` If two program `p1`, `p2` of type `world` has relation `step` (`step p1 p2` is a true proposition), `p1` can be transformed into `p2` with a step of execution. Predicates `step` are defined as:

Inductive `step : world → world → Prop :=`

`| step-j :`

$\forall (f : lab) (h : heap) (rf : rfile) (ch : codeheap)$

$(is : iseq),$

`lookupCH ch f is →`

`step ((h, rf), (ch, j f)) ((h, rf), (ch, is))`

`| ...`

`| step_iota :`

$\forall (s s' : state) (ch : codeheap) (i : iota) (is :$

$iseq),$

`next is s' →`

`step (s, (ch, seq i is)) (s', (ch, is)).`

where predicate `next` is the coq encoding of $Next_c$ function. If $Next_c(S)$ equals to S' , it can be asserted that `c`, S , S' have relation `next`, which is defined as:

Inductive `next : iota → state → state → Prop :=`

`| next_addu :`

$\forall (rd rs rt : reg) (h : heap) (rf : rfile),$

`next (addu rd rs rt)`

(h, rf)

$(h, updateR rf rd ((lookupR rf rs) + (lookupR rf rt)))$

`| ...`

`.`

In file `scap_rules.v` and `lib_lemma.v`, the well-formedness judgements and inference rules are encoded. First of all, the

program well-formedness judgement $\Psi; (p, g) \vdash \mathbb{P}$ is defined by predicates `WF_WLD`, while well-formed code heap as `WF_CH`, well-formed instruction sequence as `Infer`, and well-formed control stack as `WF_ST`. One great feature of SCAP is that its instruction sequence well-formedness inference rules can be proved as lemmas in a more general framework[4], thus, these rules are all proved formally in Coq, such as rule `CALL` is encoded as lemma `WF_jal` in `lib_lemma.v`.

The total SCAP language encodings in Coq has a size about 3000 lines, which include 400 lines some number utility functions and their properties proof, 700 lines data heap utility functions, predicates and their properties proof. The rest lines are all directly SCAP-related.

4. Modularly Certified Dynamic Storage Allocation

4.1. The Algorithm

In order to compare our implementation with old CAP[12], we don't modify the algorithm too much. However, in our implementation for the library routines `malloc` and `free`, we no longer assume an infinite heap, that is, `malloc` may fail.

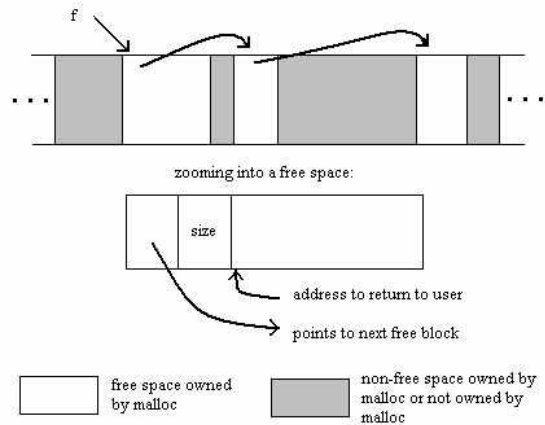


Figure 4. Free Block and Free List

Figure 4 and figure 5 shows the data structures and pseudo codes of `malloc/free`. Each memory block has a head with fields `next` which points to next block, and `size` which specifies the total size of the block. A free block linked list whose head pointer is stored in a global variable `f` is maintained, and the blocks in the list are sorted in order of increasing address. Freeing one block would possibly trigger block coalescing, while a request for memory would possibly trigger block splitting. When requesting a block with size `s`, `malloc` routine will search the free block linked

list and find the first block with bigger size than s , then split it and return the allocated block to user. When freeing a block, free routine will first search the proper position (increasing address) in the free block linked list for the block, and coalesce with its free neighbours, then insert it into the list.

```

void free(void* ptr) {
  hp=ptr-sizeof(chunk);
  prev=0; p=f;
  for (;p!=0;prev=p,p=p->next)
    if (hp<p) {
      if (hp+hp->size==p) { //join high
        hp->size+=p->size;
        hp->next=p->next;
      } else hp->next=p; //link high
      if (prev!=0)
        if (prev+prev->size==hp) { //join low
          prev->size+=hp->size;
          prev->next=hp->next;
        } else prev->next=hp; //link low
      else f=hp;
      return;
    }
  hp->next=nil; //the last block
  if (prev!=0)
    if (prev+prev->size==hp) { //join low
      prev->size+=hp->size;
      prev->next=hp->next;
    } else prev->next=hp; //link low
  else f=hp;
  return;
}

void* malloc(int size) {
  asize=size+sizeof(chunk);
  prev=0, p=f;
  for (;p!=0;prev=p,p=p->next)
    if (p->size>asize+sizeof(chunk)) {
      p->size-=asize; //split
      p+=p->size; p->size=asize;
      return p+sizeof(chunk);
    } else if (p->size>=asize) { //fit
      if (prev==0) f=p->next;
      else prev->next=p->next;
      return p+sizeof(chunk);
    }
  return NULL;
}

```

Figure 5. malloc/free

4.2. Specification Annotation

The biggest challenge in SCAP programming (the same as other language-based program verification approach) is how to annotated the low-level codes with accurate specification— (p, g) in SCAP. It can be easily noticed that p and g are higher-order logic (HOL) predicates (Prop is the type of HOL propositions) on state. Thus the question becomes how to write HOL predicates that must be satisfied

at the program point. It is easy to specify the assertions about registers file, such as r_1 must contain value 12, while it is difficult to specify the assertions on data heap, because there are many dynamic data structures composed by pointers. In this work, Separation Logic [9] is chosen to encode the formal abstraction of these data structures. Before going further, it is necessary to clarify a little about '*' connective in separation logic. The meaning of '*' and related functions and predicates are defined as below:

$$\begin{aligned}
\mathbb{H}_1 \cdot \mathbb{H}_2 &\triangleq \lambda i. \text{if } i \in \text{dom}(\mathbb{H}_1) \text{ then } \mathbb{H}_1 \ i \ \text{else } \mathbb{H}_2 \ i \\
\mathbb{H}_1 \perp \mathbb{H}_2 &\triangleq \forall i, i \notin \text{dom}(\mathbb{H}_1) \vee i \notin \text{dom}(\mathbb{H}_2) \\
\mathbb{H}_1 \equiv \mathbb{H}_2 &\triangleq \forall i, \mathbb{H}_1(i) = \mathbb{H}_2(i) \\
p_1 * p_2 &\triangleq \lambda S. \exists \mathbb{H}_1, \mathbb{H}_2. \mathbb{H}_1 \perp \mathbb{H}_2 \wedge \mathbb{H}_1 \cdot \mathbb{H}_2 \equiv S.H \\
&\quad \wedge p_1(\mathbb{H}_1, S.R) \wedge p_2(\mathbb{H}_2, S.R)
\end{aligned}$$

Function '.' concatenates two disjoint heaps. Predicate ' \perp ' describes the disjunction relation between two heaps. Predicate ' \equiv ' describes the equivalence relation between two heaps. The '*' connective provides a mechanism of describing the heap state for each disjoint part.

Now we move on to define assertions for all possible heap states that will occur in malloc/free routines as shown in Figure 6.

$$\begin{aligned}
\text{emp} &\triangleq \lambda S. \text{dom}(S.H) = \phi \\
\text{SigH } p \ v &\triangleq (p \rightsquigarrow v) \\
\text{Tuple } p \ s &\triangleq (p, \dots, p + s - 1 \rightsquigarrow -, \dots, -) \\
\text{Stk } p \ q &\triangleq (p, p - 1, \dots, q + 1 \rightsquigarrow -, \dots, -) \\
\text{MBlk } p \ q \ s &\triangleq p > 2 \wedge s > 2 \\
&\quad \wedge (p - 2 \rightsquigarrow q) * (p - 1 \rightsquigarrow s) \\
&\quad * (p, \dots, p + s - 3 \rightsquigarrow -, \dots, -) \\
\text{MBlkLst } 0 \ p \ q &\triangleq \text{emp} \wedge (p = q) \\
\text{MBlkLst } n+1 \ p \ q &\triangleq \exists p'. (\text{MBlk } p+2 \ p') \\
&\quad * (\text{MBlkLst } n \ p' \ q) \\
&\quad \wedge (p < p' \vee p' = 0) \\
\text{EndL } hd \ p \ q &\triangleq (p = 0 \rightarrow \text{MBlkLst } 0 \ hd \ q) \wedge \\
&\quad (p \neq 0 \rightarrow \exists n. (\text{MBlkLst } n \ hd \ p) \\
&\quad \quad * (\text{MBlk } p+2 \ q \ -) \\
&\quad \quad \wedge (p < q \vee q = 0)) \\
\text{MidL } hd \ p \ q &\triangleq \exists n. (\text{EndL } hd \ p \ q) \\
&\quad * (\text{MBlkLst } n \ q \ 0) \\
\text{Good } hd &\triangleq \exists n. (\text{MBlkLst } n \ hd \ 0)
\end{aligned}$$

Figure 6. Heap Assertions

The first assertion emp holds over empty heap. ($\text{SigH } p \ v$) holds over a singleton heap with value v in slot p . ($\text{Tuple } p \ s$) asserts the heap contains a continuous memory area of size s , starting from address p . ($\text{Stk } p \ q$) models the stack, which is implemented as a continuous memory area from p to q . The assertions below Stk are designed to catch the invariants of the free memory block list. ($\text{MBlk } p \ q \ s$) asserts the memory at address p is preceded by a pair of words, first contains q , a pointer to another block and the second contains the size s of the mem-

ory block itself. $(\text{MBlkLst } n \ p \ q)$ models an address-ordered list of blocks with length n in which p is the starting pointer and q is the ending pointer. It is inductively defined in order to be encoded in Coq. $(\text{EndL } hd \ p \ q)$ defines a list segment with a particular ending block pointed by p , whose forward pointer is q . $(\text{MidL } hd \ p \ q)$ asserts a list, in middle of which a block pointed by p exists with forward pointer q .

Now we can define the pre-assertion p of each instruction sequence of malloc/free. For example, considering the prolog instruction sequence of `free` (complete low-level codes and part of specifications are in appendix):

```

prolog : (p0, g0)
         subiu sp, sp, 4
         sw fp, 3(sp)
         addiu fp, sp, 4
         sw s0, -1(fp)
         sw s1, -2(fp)
         sw s2, -3(fp)
         j entry

```

The assertion p_0 over the initial state requires :

- sp and fp should be able to construct a stack frame;
- Value of sp should greater than 4, otherwise the stack will overflow;
- The return address should not be 0, otherwise it can't return;
- The heap should contain global variable f , a free block list pointed by f , an obsolete memory block pointed by argument $a0$, a stack and other arbitrary part since we never care about it.

In order to express these requirements, following abbreviation are used:

$$\begin{aligned} \text{Frm}(s) &\triangleq [\text{fp}] = [\text{sp}] + s \wedge [\text{fp}] \leq \text{B} \wedge [\text{sp}] > 0 \\ \text{ArbH} &\triangleq \lambda \mathbb{S}. \text{True} \end{aligned}$$

where $[r]$ means the value r contains in current register file. In our stack layout convention, we assume the program loader arranges the memory from B to 1 as the stack, therefore, $\text{Frm}(s)$ asserts fp, sp satisfy the requirements of a stack frame with size s . Obviously, $\text{Frm}(s)$ specifies the first requirement. ArbH asserts arbitrary heap. So the third requirement is specified as $p_{0h}(\text{ArbH})$ where p_{0h} defines as:

$$p_{0h}(p) \triangleq \exists pqs. (\text{SigH } f \ p) * (\text{MBlk } [a0] \ q \ s) * (\text{Good } p) * (\text{Stk } \text{B } 0) * p$$

So the pre-assertion p_0 is :

$$p_0 \equiv \exists s. \text{Frm}(s) \wedge [\text{sp}] > 4 \wedge [ra] \neq 0 \wedge p_{0h}(\text{ArbH})$$

We have specified p_0 , then what should g_0 look like? The action g_0 of prolog in `free` requires that in the storage states of current position and function exit point:

- Callee-saves registers $s0 - s7$, caller-saves registers $t3 - t9, ra, fp, sp$ should have the same values;
- The callers stack frame from current $[sp]$ to stack bottom B , should never be changed;
- The post heap should contain global variable f , a free block list pointed by f , a stack, and other part reserved from current heap. It must be guaranteed that the reserved heap assertion must be a pure heap assertion which means it do nothing with (insensitive) registers.

Similarly, we need following abbreviation:

$$\begin{aligned} [r] &\equiv \mathbb{R}(r) & [r]' &\equiv \mathbb{R}'(r) \\ [l] &\equiv \mathbb{H}(l) & [l]' &\equiv \mathbb{H}'(l) \end{aligned}$$

$$\begin{aligned} \text{Rid}(rs) &\triangleq \forall r \in rs, [r] = [r]' \\ \text{Res}(r, i) &\triangleq [r]' = [[fp] - i] \\ \text{Hid}(ls) &\triangleq \forall l \in ls, [l] = [l]' \\ \text{Hch}(p, q) &\triangleq \lambda \mathbb{S}. \lambda \mathbb{S}'. p \mathbb{S} \rightarrow q \mathbb{S}' \\ \text{RIns}(p) &\triangleq \forall \mathbb{H}, \mathbb{R}, \mathbb{R}'. p(\mathbb{H}, \mathbb{R}) \rightarrow p(\mathbb{H}, \mathbb{R}') \\ \text{NoP} &\triangleq \lambda \mathbb{S}. \text{True} \\ \text{NoG} &\triangleq \lambda \mathbb{S}. \lambda \mathbb{S}'. \text{True} \end{aligned}$$

$\text{Rid}(rs)$ asserts the registers in the set rs have the same value in two states. $\text{Res}(r, i)$ asserts the value of r in the post-state is restored from the stack frame. $\text{Hid}(ls)$ asserts all slots in ls will not be changed. $\text{Hch}(p, q)$ asserts if the pre-state of heap satisfies p , then post-state satisfies q . $\text{RIns}(p)$ asserts p is a pure heap assertion. NoP and NoG mean no requirements, no guarantees.

Therefore, g_0 of prolog in `free` can be specified as following:

$$g_0 \equiv \text{Rid}(\{s0 - s7, t3 - t9, ra, fp, sp\}) \wedge \text{Hid}(\{[sp] - \text{B}\}) \wedge \forall p. \text{RIns}(p) \rightarrow \text{Hch}(p_{0h}(p), q_h(p))$$

where the heap's post assertion q_h is defined as:

$$q_h(p) \triangleq \exists p. (\text{SigH } f \ p) * (\text{Good } p) * (\text{Stk } \text{B } 0) * p$$

Until now, specification annotation in SCAP has been explained. It is clear that all specifications of library routines are independent from higher-level application which calls these routines.

4.3. The proof of malloc/free

After specification annotation, the code heap specification Ψ has constructed. Well-formedness of every instruction sequence in the code heap must be proved, and after that the routines is certified. For the prolog sequence (named *C_free_prolog*) of `free` routine, following theorem should be proved:

Theorem 1. (WF_free_prolog) $\Psi; (p_0, g_0) \vdash C_free_prolog$

Proof. It can be proved by applying inference rules of corresponding instruction c . While applying the rules, the programmer needs to find the intermediate specification (p', g') , which serves both as the postcondition for c , and as the precondition for the remaining instruction sequence. This finding job is not difficult because a instruction only change (p, g) slightly at most times. \square

4.4. Linking with Certified Library

How can applications be linked with the certified library? Obviously, programmers don't want to rebuild the proof of routines. In this subsection, a linked list copy program calling malloc and free are certified to show the modularity of SCAP. The pseudo code of copy program are given in figure 7. It clones a new linked list from the list pointed by src, and releases the old list.

```
list* copy(list *src){
  tgt=prev=NULL;
  while (src<>NULL){
    p=malloc(2);
    if(!p) return tgt;
    p->data=src->data, p->next=src->next;
    old=src, src=src->next, free(old);
    if(!prev){tgt=p, prev=p}
    else{prev->next=p, prev=p}
  }
  return tgt;
}
```

Figure 7. Copy Program

The copy program uses the data structure of single linked list (Slist), whose formalization is shown in figure 8. $(\text{Pair } p \ x \ q)$ defines the node of linked list, which is a memory block with size bigger than four(unfortunately the library implementation details can't be hidden). $(\text{Slist } \alpha \ p \ q)$ defines a list segment from p to q representing the sequence α .

$$\begin{aligned} \text{Pair } p \ x \ q &\triangleq \lambda s. \exists q', s. \mathbb{H}(p) = x \wedge \mathbb{H}(p+1) = q \\ &\quad \wedge (\text{MBlk } p \ q' \ s) \wedge s \geq 4 \\ \text{Slist } \epsilon \ p \ q &\triangleq \text{emp} \wedge p = q \\ \text{Slist } (x \cdot \alpha) \ p \ q &\triangleq \exists p'. (\text{Pair } p \ x \ p') * (\text{Slist } \alpha \ p' \ q) \end{aligned}$$

Figure 8. Node and Slist

In order to using LINK rule, programmers must first prove all instruction sequences in copy are well-formed. It is similar as the proof of malloc/free, but still has some differences, that is, the proving of function call must using rule CALL, which must use the prolog specification of callee. After the copy code heap is certified, (1)the total code heap include copy program and malloc/free can be

proved well-formed using LINK rules. Because the depth of control stack at copy's entry is 0, (2)the control stack is well-formed according to rule WFST-0. From (1),(2), and rule PROG, it is easy to prove that the program(copy linked with malloc/free) is well-formed.

5. Related Work

The old CAP[12] system uses whole-program reasoning to deal with function call, which sacrifices the advantages of modular reasoning. The whole-program reasoning means the library routines' specification relying on the caller (such as copy program), and the proof of routines must be rebuild when linking. Therefore, it does not support modular certification. These disadvantages are unbearable for a CAP programmer, although CAP designed the library specs and their well-formedness theorems to be templates which can be instantiated according to the user programs. But in SCAP, things are changed by adding g in specification structure and maintaining stack invariants[4]. As we see in section 4, all routines are annotated and proved independently in SCAP.

6. Conclusion and Future Work

We have encoded the SCAP abstract machine, the verification framework, and its soundness proof in Coq. In order to show that SCAP has practical significance in program certification, we port a certified dynamic storage allocation library from old CAP[12] to SCAP in three weeks. The cost of semi-automatically certification is not very high, and we think it is worthwhile especially for softwares of high-assurance. Also we build a coq module for separation logic, which makes proof compact and concise.

Future Work As we mention in 4.4, the implementation details of library are not hidden, such as MBlk predicate appears in Pair predicate. It means the rebuilding of proof and re-annotation of copy program if library algorithm is changed while the interface is the same. It is a bad news for SCAP programmer. Therefore, we take the exploration of real modularity as future work.

Acknowledgments

We thank anonymous referees for suggestions and comments on this paper. This research is based on work supported by NSF of China(under Grant No.60473068) and Intel CRC(Beijing). We would like to thank Professor Zhong Shao for his guidance and support, and thank Xinyu Feng in Yale for his keen help. We are also grateful to Pierre Castéran for his patient answer about our questions on Coq.

References

- [1] Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, pages 247–258, 2001.
- [2] Xinyu Feng and Zhong Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 International Conference on Functional Programming (ICFP'05)*, September 2005.
- [3] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *Technical Report YALEU/DCS/TR-1336 and Coq Implementation, Dept. of Computer Science, Yale University*, November 2005.
- [4] 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 SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, page to appear, New York, NY, USA, June 2006. ACM Press.
- [5] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [6] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. volume 1473, pages 28–52, Kyoto, Japan, March 1998.
- [7] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
- [8] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, January 1997.
- [9] 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.
- [10] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004.
- [11] Hongwei Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15:91–131, October 2002.
- [12] Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for pcc: dynamic storage allocation. *Sci. Comput. Program.*, 50(1-3):101–127, 2004.
- [13] Dachuan Yu and Zhong Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 International Conference on Functional Programming (ICFP'04)*, September 2004.