Formal Verification of MILS Partition Scheduling Module Using Layered Methods

Yang Gao, Xia Yang, Wensheng Guo, and Xiutai Lu

***Abstract*—MILS partition scheduling module ensures isolation of data between different domains completely by enforcing secure strategies. Although small in size, it involves complicated data structures and algorithms that make monolithic verification of the scheduling module difficult using traditional verification logic (e.g., separation logic). In this paper, we simplify the verification task by dividing data representation and data operation into different layers and then to link them together by composing a series of abstraction layers. The layered method also supports function calls from higher implementation layers into lower abstraction layers, allowing us to ignore implementation details in the lower implementation layers. Using this methodology, we have verified a realistic MILS partition scheduling module that can schedule operating systems (Ubuntu 14.04, VxWorks 6.8 and RTEMS 11.0) located in different domains. The entire verification has been mechanized in the Coq Proof Assistant.**

***Index Terms*—MILS, separation kernel, formal methods, layered methodology.**

I. Introduction

MILS (Multiple Independent Levels of Security and Safety) architecture is based on the idea of separation [1], building multiple separate domains with different security levels on the same hardware platform. As a critical module, the MILS scheduling module enforces secure strategies, which ensures data are completely isolated between domains [2], [3]. For example, when a VCPU is ready to switch to another partition, it must clear the contents of the previous partition to ensure its partition security. In order to ensure data isolation properties of the MILS architecture, it is important to verify the implementation of its schedule module. Incorrect implementation of a schedule function can invalidate essential isolation properties and even crash the entire system.

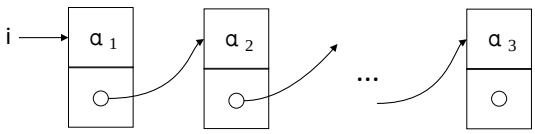


Fig. 1. Single linked list structure.

However, the verification of the schedule module of foundation software system is a difficult task because it is usually written in a low-level language that makes use of linked list with multi pointers, and it is usually not written with verification in mind. For example, to formal specify a simple single linked list as shown in Fig. 1, we need to write down the following specifications and notations using traditional separation logic [4] as shown in Fig. 2.

Although the traditional separation logic has capability of representing simple linked lists intuitively, it is not suitable for specifying and reasoning more complex linked lists as its data representation and data operation are mixed in a same layer. However, for performance considerations, the queue in the MILS scheduling module is usually implemented by a complex multi-pointer domain linked list. If the traditional separation logic method is used to model the linked list, the reasoning process is extremely complicated and difficult, which may cause the entire verification work to fail. A promising approach to the above problems is to build multiple layers with more abstract computation models as smaller abstractions tend to be easier to prove and maintain, while larger abstractions can be still achieved by composing the smaller ones. Unfortunately, creating abstract models and linking across them is seen as ad-hoc and tedious additional work in the traditional separation logic community.

formal definition of the linked list(α,i):

$$list \in i \stackrel{def}{=} emp \wedge i = nil$$

$$list(a \cdot \alpha)\, i \stackrel{def}{=} \exists j.\, i \mapsto a, j * list\ \alpha\ j$$

default properties:

$$list\ a\ (i,j) \Leftrightarrow i \mapsto a, j;$$

$$list\ \alpha \cdot \beta(i,k) \Leftrightarrow \exists j.\, list\ \alpha\ (i,j) * list\ \beta\ (j,k)$$

$$list\ \alpha \cdot \beta(i,k) \Leftrightarrow \exists j.\, list\ \alpha\ (i,j) * j \mapsto b,k$$

$$list\ \alpha\ i \Leftrightarrow list\ \alpha\ (i,nill)$$

$$list(nil,nil) \Leftrightarrow emp$$

notations:

α,β,…

--indicates linked list sequence.

α·β

--indicates that the linked list sequence α is followed by the sequence β.

list α(i,j)

--indicates that the first node of the linked list sequence α is pointed by i, and the last node is pointed to by j.

p1 * p2

--indicates that the heap can split into two disjoint parts such that p1 holds for one part and p2 holds for the other.

Fig. 2. Separation specifications and notations of a single linked list.

In this paper we show how to reduce the effort required to define specifications and linking, so that complex code

Manuscript received February 9, 2021; revised July 23, 2021.

Yang Gao, Xia Yang, Wensheng Guo, and Xiutai Lu are with School of Information and Software Engineering University of Electronic Science and Technology of China, Chengdu, China (e-mail: gyang@std.uestc.edu.cn, xyang@uestc.edu.cn, gws@uestc.edu.cn, 1158829283@qq.com) .

doi: 10.18178/ijfcc.2021.10.4.578

verification using layered methods becomes an effective approach. More precisely, our paper makes the following contributions:

1) We present a layered methodology with separation logic for quickly defining multiple abstract models and their verification layers.
2) We show how our methodology can be used to define and link different abstraction and implementation layers.
3) We show how to hierarchize MILS scheduling modules and define abstract models for each layer of the scheduling modules.

The rest of this paper is organized as follows. In section 2, we introduce the scheduling module of MILS. In section 3, we propose a proof plan for the MILS scheduling module. In section 4, we take the function of the scheduling module as an example to introduce the proof method and proof process. In section 5, we explain related work and summary this paper.

II. The Overview of MILS Scheduling Module

A. MILS Scheduling Module

```
void rtsc_vcpu_insert(const struct scheduler *ops, struct vcpu *vc):
Feature: Insert a VCPU in the ready queue.
Parameters: ops: Scheduler structure variable, which stores the ready queue
            vc: VCPU to be inserted into the ready queue
Return：None

void rtsc_vcpu_remove(const struct scheduler *ops, struct vcpu *vc):
Feature: Remove a VCPU from the ready queue
Parameters: ops: Scheduler structure variable, which stores the ready queue
            vc: VCPU to be removed
Return：None

struct task_slice rtsc_schedule(const struct scheduler *ops, s_time_t now,
                                bool_t    tasklet_work_scheduled):
Feature: The core scheduling function, select a VCPU to run when scheduling is triggerd
Parameters: ops: Scheduler structure variable
            now: Time to trigger the schedule
            tasklet_work_scheduled: Task scheduling status
            Return：Return to the next time slice description(the run time, next VCPU, etc)

void rtsc_vcpu_sleep(const struct scheduler *ops, struct vcpu *vc):
Feature: When the VCPU is sleeping, remove the VCPU from the ready queue.
Parameters: ops: Scheduler structure variable
            vc: VCPU to sleep
Return：None

void rtsc_vcpu_wake(const struct scheduler *ops, struct vcpu *vc):
Feature: When waking up a VCPU, insert the VCPU into the ready queue
Parameters: ops: Scheduler structure variable
            vc: VCPU being awakened
Return：None

void rtsc_context_saved(const struct scheduler *ops, struct vcpu *vc):
Feature: When switching context at the end of scheduling, insert the last VCPU that was swapped out
into the ready queue
Parameters: ops: Scheduler structure variable
            vc: The last VCPU
Return：None
```

Fig. 3. Analysis of key functions of MILS scheduling module.

In the trusted separation kernel of the MILS architecture, the purpose of the scheduler is to select the most suitable VCPU from the VCPU's ready queue according to a certain scheduling algorithm to occupy PCPU. This paper uses a scheduling algorithm based on fixed priority [5]. This algorithm can ensure that real-time tasks in the strong real-time domain can be completed on time. Its key functions are shown in Fig. 3.

. Our proof of the scheduling module is the proof of these key functions

In the next section, we will use our verification method to propose a verification plan for these functions.

III. Overview and Plan for Verification

Through the analysis of these functions in Fig. 3 and the source code, we divide the MILS scheduling module into six abstract models (Fig. 4). And the different models are connected to each other through function call relationships. From bottom to top:

Abstract model of VCPU, DOMAIN and the linked list: This model introduces read and write operations of VCPU-related data (vcpu_get/set), read and write operations of DOMAIN-related data, read and write operations of linked list node.

Abstract model of linked list add/delete/initialize operation: This model introduces fixed-point insertion and deletion operations to the linked list and initialization operation for the linked list (list_add/del_tail/init).

Abstract model of linked list initialization and deletion function: This model introduces linked list deletion and initialization operation (list_del_init).

Abstract model of interrupt queue insertion function, pick/delete suitable VCPU function: This model introduces the insertion operation of the interrupt queue (irqq_insert), the picking and deleting VCPUs operations from the ready queue (vcpu_pick/rm_q).

Abstract model of ready queue related operations: This model introduces ready queue insertion and update operations. (runq_insert/update).

Abstract model of VCPU-related operation and scheduling function : This model introduces the operations of inserting (rtsc_vcpu_insert), deleting (rtsc_vcpu_remove) VCPU in the ready queue, the operation of selecting VCPU when triggering (rtsc_schedule), the operation of deleting VCPU when it sleeps (rtsc_vcpu_sleep), the operation of inserting VCPU to the ready queue when it is awakened (rtsc_vcpu_wake), the operation of switching context (rtsc_context_saved). After we decompose the MILS scheduling module, we can separate and verify the implementation of the upper-layer algorithm and the lower-layer complex data structure. The advantage of this is that after we complete the verification of the lower-layer complex data structure, the proof of the upper-layer model only needs to verify the function call relationship. We don't need to prove the complex data again. This improves verification efficiency and saves verification time.

After we decompose the MILS scheduling module, we can separate and verify the implementation of the upper-layer algorithm and the lower-layer complex data structure. The advantage of this is that after we complete the verification of the lower-layer complex data structure, the proof of the upper-layer model only needs to verify the function call relationship. We don't need to prove the complex data again. This improves verification efficiency and saves verification time.

In the next section, we illustrate our entire verification process in detail. Our verification method for this function can be applied to the verification of the entire scheduling

module, and even the verification of the entire MILS. It is universal.

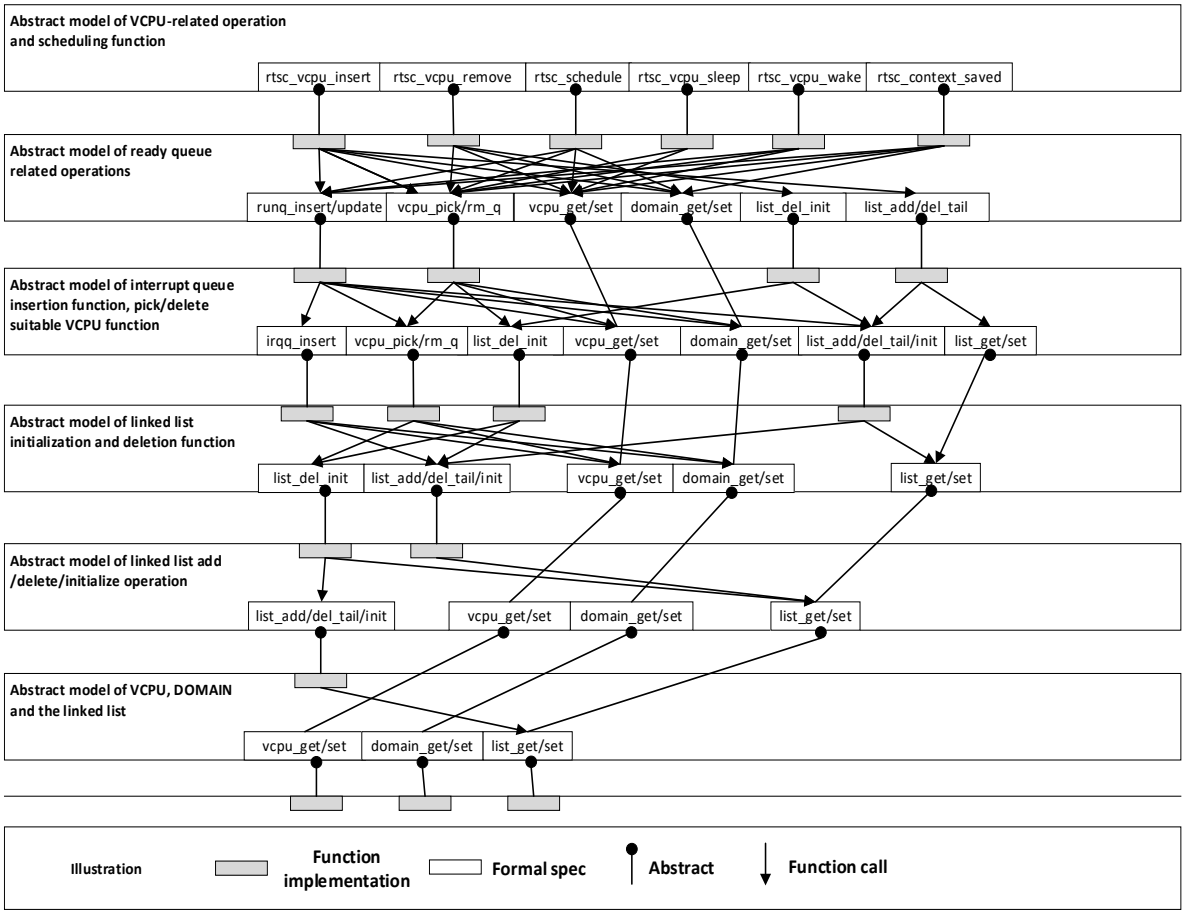


Fig. 4. Overall verification plan for MILS scheduling module abstract.

IV. Methodologies and Process

Fig. 5 is the framework diagram of the verification method. The program is written and implemented in C language. All spec definitions, functional correctness proofs, and spec consistency proofs are completed in the Coq which is an auxiliary theorem tool. The specific verification process is divided into three parts:

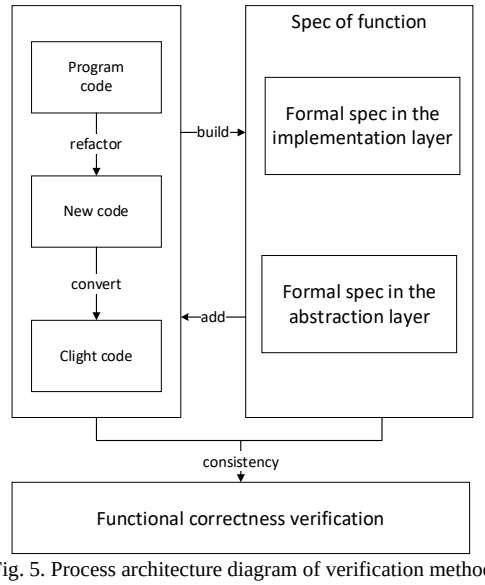


Fig. 5. Process architecture diagram of verification method.

The first part is the refactoring of the program code and convert the code to Clight [6] code.

The second part is to describe the specification of the the program's implementation layer based on the separated logic according to the converted Clight code, and use functional programming to construct the specification of the program's abstract layer.

The third part is to add the described function specification to the Clight code, and verify the consistency between the functional abstraction layer and the functional implementation layer with the corresponding Clight code.

Next, we use the function "list_add_tail" in Fig. 6 to illustrate the entire verification process of this method.

A. The Overview of Program

Function "list_add_tail" inserts a new node at the previous node position of the linked list's node "head". The insert operation is done in function "__list_add".

```
struct list_head
{
  struct list_head *next, *prev;
};
static inline void __list_add(struct list_head *new, struct list_head *prev, struct list_head *next)
{
 next->prev =new;
 new->next = next;
 new->prev = prev;
 prev ->next = new;
}
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
 __list_add(new,head->prev,head);
}
```

Fig. 6. Program source code.

B. Refactor the Source Code (Source Code Normalization)

For some pointers, structures and other data structures in the function, especially pointer nesting, structure nesting, they will be converted to the corresponding nested structure when the source code is converted. When we describe the spec of the nested structure of these functions, it will appear that the described specification may be too long and complicated. As a result, the deduction strategy cannot be identified in the actual deduction proof, which makes the final proof impossible to complete. In order to avoid this problem, we separate data structures (pointers, structures, etc.) as an independent function to prove separately. In the final function, the complex operation is just to call the independent function, that is, when proving the final function, you do not need to prove the complex operation, only need to prove the calling relationship between functions, thus simplifying the proof process.

Fig. 7 is the program code after refactoring. Pointer operations are all encapsulated in functions. The parent function's verification only needs to call the encapsulation functions' proof result After encapsulation functions are proved.

```
struct list_head
{
   struct list_head *next, *prev;
};
static list_head* get_list_head_prev(struct list_head *head)
{
  return head->prev;
}
void set_list_head_next(struct list_head *head, struct list_head *t)
{
  head->next = t;
}
void set_list_head_prev(struct list_head *head, struct list_head *t)
{
 head->prev = t;
}
void __list_add(struct list_head *new, struct list_head *prev, struct list_head *next)
{
  set_list_head_prev(next,new);
  set_list_head_next(new,next);
  set_list_head_prev(new,prev);
  set_list_head_next(prev,new);
}
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
   struct list_head *p;
   p = get_list_head_prev(head);
   __list_add(new,p,head);
}
```

Fig. 7. Program code after refactoring.

C. Generate the Clight Code

We use Clightgen tool to convert C code to Clight code. Clightgen is a tool of compiler CompCert [7]. CompCert is a formalized compiler for the C programming language. The Clight code after conversion by Clightgen can truly describe C pointer and can eliminate some difficult-to-model semantics in C language. In addition, Clightgen can factor out function calls and assignments from inside subexpressions (by moving them into their own assignment statements), can Factor && and || operators into if statements (to capture short circuiting behavior). And when the -normalize flag is used, Clightgen can factor each memory dereference into a top level expression, i.e. "x=a[b[i]];" becomes "t=b[i];x=a[t];".

Clight code can be recognized by the Coq auxiliary theorem prover, so all our specification definitions, functional correctness proofs, and spec consistency proofs can be completed in Coq. It ensures that the semantic gap between different formal specs is minimized.

D. Function Specification

We use the refactored code in Fig. 7 as an example to illustrate the difference between formal spec at different abstraction levels. The code include the definition of the linked list's head structure "list_head" and the implementation of five functions "get_list_head_prev", "set_list_head_next", "set_list_head_next", "__list_add", "list_add_tail". The first three functions are used to read and write data in the linked list, and the fourth function is used to implement key node insertion operations.

```
Inductive list_head_abs : Type :=
 |List_head (next:Z) (prev:Z).

Definition list_head_pool := ZMap.t list_head_abs.
Definition get_list_prev_abs(al:Z)(lpool:list_head_pool):list_head_pool * Z:=
 match ZMap.get al lpool with
  |List_head _ aprev =>(lpool,aprev)
 end.

Definition set_list_head_next_abs(lpool:list_head_pool)(ah:Z)(nv:Z):
list_head_pool:=
  match ZMap.get ah lpool with
    |List_head next prev=>ZMap.set ah (List_head nv prev) lpool end.

Definition set_list_head_prev_abs(lpool:list_head_pool)(ah:Z)(pv:Z):
list_head_pool:=
  match ZMap.get ah lpool with
    |List_head next prev=>ZMap.set ah (List_head next pv) lpool end.

Definition list_add_tail_abs(new:Z)(head:Z)(lpool:list_head_pool):
list_head_pool:=
   let p:=snd (get_list_prev_abs head lpool) in
    list_add_abs lpool new p head.
```

Fig. 8. Formal spec in the abstraction layer for the insertion of the linked list node.

1) Formal spec in the abstraction layer. "List_head_abs" is the formal spec of the linked list node struct. "Z" is a built-in type of Coq that describes integers in mathematics. "list_head_pool" defines a set of "list_head" types. "get_list_prev" reads the node of the linked list, so the post-state of the list set remains unchanged after executing this function. "set_list_head_next" and "set_list_head_prev" can set the list, the post-state of the linked list set is set to the correct goal state after they are executed. As shown in Fig. 8, in "set_list_head_next_abs" and "set_list_head_prev_abs, lpool" uses "ZMap.set" operator in Coq to update their linked list set's value.
2) Formal spec in the implementation layer. linked list node that we define describes the program logic of the function in a high level of abstraction. It is conducive to the reasoning and validation of program logic, but the structure used is too abstract to establish relation with C structure, which increases the difficulty of consistency verification between formal specs and program source code. Therefore, we use VST validation tool [8] to build the formal spec of functional implementation layer and prove the correctness of program function, consistency of spec between the implementation layer and the abstract layer based on this spec. Finally, we derive the consistency of spec between the abstraction layer and the

source implementation. VST is a C language verifiable tool based on separation logic, we use it to formalize API function specs, including the data structure of API operations, preconditions of API functions (assumptions for input parameters and global variables), postconditions of API functions (the updated guarantees for return values and global variables). To formalize a C language function using VST, you need to follow the following code as Fig. 9. "PROP(P)LOCAL(Q)SEP(R)" represents the pre- and post-conditions of spec. "P" is a list of proposition, "Q" is a list of bound local global variables, "R" is the list of predicates of separate logic, "WITH" is used to describe the logical variable "v" , which is an abstract mathematical variable that can be referenced by pre- and post-conditions. Formal spec of three functions in the implementation describes actions for the linked list in a way that is very close to C code in Fig. 10 and Fig. 11. Fig. 10 shows an action for getting the linked list node, It is consistent with the execution of the function "get_list_head_prev" in Fig. 7. Fig. 11 shows an action for setting the linked list node, it is consistent with the execution of the function "set_list_head_next" in Fig. 7. The spec of the second function "set_list_head_prev" is similar to "set_list_head_next", so we won't describe it here.

"SEP" uses separation logic assertions "field_at Tsh t_struct_list_head [] (Vundef,Vundef)" to describe the contents of the structure "list_head". Function "__list_add" and "list_add_tail" implement their functions by calling intermediate functions, so when we prove that these intermediate functions are correct, function "__list_add" and "list_add_tail" are also correct. The spec in implementation layer of these two functons does not need to define predicate logic. So, we won't introduce them here.

```
Definition func_spec:=
 DECLARE _func  WITH v
   PRE [params]   PROP(P) LOCAL(Q) SEP(R)
   POST [returns]  PROP(P) LOCAL(Q) SEP(R)
```

Fig. 9. The VST spec writing format.

```
Definition get_list_head_prev_spec:=
 DECLARE _get_list_head_prev
  WITH ch:val, cp:val
   PRE [_head OF (tptr t_struct_list_head)]
    PROP () LOCAL (temp _head ch)
    SEP(field_at Tsh t_struct_list_head [] (Vundef,cp) ch)
   POST [tptr t_struct_list_head]
    PROP ()  LOCAL (temp ret_temp cp)
    SEP(field_at Tsh t_struct_list_head [] (Vundef,cp) ch).
```

Fig. 10. Formal spec the implementation layer for getting the linked list node.

3) Consistency between the abstraction layer and the implementation layer. Since our description of the state predicates in the spec of the implementation layer is consistent with the linked list data structure in Fig. 7, there is no data abstraction relationship between the implementation layer spec and the source code. Therefore, the consistency proof between the implementation layer spec and the code implementation only requires the functional correctness proof. But for the consistency proof between the implementation layer and the abstraction layer, we need to define the relation between the implementation layer and the abstraction layer. We use "lpool" to indicate the linked list which is formalized in the abstraction layer. We use "ch" to indicate the linked list which is formalized in the implementation layer. And the relation between the above two should satisfy: for any address "i", if address "ch" of the linked list node in the implementation layer is evaluated at "i", and the value of this node is "(cn,cp)", the subscript of linked list node in the abstraction layer is evaluated at "i", the value of this node is "List_head an ap", Then, the value of an should be equal to "cn", and the value of ap should be equal to "cp". This relation can be described in formal language as Fig. 12. We can add the above relation to the "SEP"'s predicate logic of the function to be proved, when we do the proof, we will prove the above consistent relation. If there is no problem with the proof, it can be proved that the abstraction layer and the implementation layer satisfy the above relation, and they are consistent.

```
Definition set_list_head_next_spec:=
 DECLARE _set_list_head_next
  WITH cl:val, tv:Z
   PRE [_head OF(tptr  t_struct_list_head), _t OF (tptr t_struct_list_head)]
    PROP () LOCAL(temp _head ch;temp _t (Vint (Int.repr tv)))
    SEP(field_at Tsh t_struct_list_head [] (Vundef,Vundef) ch)
  POST[tvoid]
    PROP() LOCAL()
    SEP(field_at Tsh t_struct_list_head [] ((Vint (Int.repr tv)),Vundef) ch).

Definition set_list_head_prev_spec:=
 DECLARE _set_list_head_prev
  WITH ch:val, tv:Z
  PRE [_head OF(tptr  t_struct_list_head), _t OF (tptr t_struct_list_head)]
  PROP () LOCAL(temp _head ch;temp _t (Vint (Int.repr tv)))
  SEP(field_at Tsh t_struct_list_head [] (Vundef,Vundef) ch)
  POST[tvoid]
  PROP() LOCAL()
  SEP(field_at Tsh t_struct_list_head [] (Vundef,(Vint (Int.repr tv))) ch).
```

Fig. 11. Formal spec in the implementation layer for setting the linked list node.

$$\forall i,\quad Vint\,(Int.repr\ i) = ch,$$
$$field\_at\ Tsh\ t\_struct\_list\_head\ [\,]\ (cn, cp)\ ch\ ,$$
$$ZMap.get\ i\ lpool = List\_head\ an\ ap,$$
$$then\ \exists an, ap, cn, cp,$$
$$Vint(Int.repr\ an) = cn, Vint(Int.repr\ ap) = c$$

Fig. 12. the consistency relationship between the implementation layer and the abstraction layer.

E. Functional Correctness Verification

We use the VST's proof tactic to help us prove the theorem, and we need to follow the spec in Fig. 13.

The predicate "semax_body" states the Hoare tripe of the function body, Δ |—{P}C{Q}. P and Q are taken from the spec which we define in function implementation layer. C is the body of function, and the type-context Δ is calculated from the global type-context overlaid with the parameter- and

local-types of the function.

When we derive the theorem through the tactic "start_function". our proof goal will be transformed into the Hoare triple. Then we will use some proof tactics that VST provides to prove the Hoare triple. If it proves successful, it shows that the function has functional correctness. That is, this function is safe. Next, we take the function "set_list_head_prev" as an example to illustrate the whole proof process. Fig.14 is this function's proof code.

```
Definition Vprog : varspecs. mk_varspecs prog. Defined.
Definition Gprog : funspecs :=  ltac:(with_library prog (func1_spec;func2_spec;…]).

Lemma body_func_spec: semax_body Vprog Gprog f_func fcunc_spec.
Proof.
  start_function.
 [Proof_Tactics].
Qed.
```

Fig. 13. proof spec in VST.

```
Lemma body_set_list_head_prev: semax_body Vprog Gprog
f_set_list_head_prev set_list_head_prev_spec.
Proof.
  start_function.
  Intros cp;Intros anext;Intros ap.
  forward. forward.
  Exists (Vint (Int.repr tv)).
  Exists anext;Exists tv.
  unfold set_list_head_prev.
  inversion H2.
  entailer!. rewrite ZMap.gss. auto.
Qed.
```

Fig. 14. "set_list_head_prev" proof code.

After tactic "start_function" is executed, the proof goal becomes as Fig. 15.

```
Espec : OracleKind
ch : val
tv : Z
lpool : list_head_pool
ah : Z
Delta_specs : PTree.t funspec
Delta := abbreviate : tycontext
H : 0 <= tv <= Int.max_unsigned
POSTCONDITION := abbreviate : ret_assert
MORE_COMMANDS := abbreviate : statement
_____________________________________(1/1)
semax Delta
  (PROP ( )
   LOCAL (temp _head ch; temp _t (Vint (Int.repr tv)))
   SEP (EX cp : val,
       (EX anext : Z,
        (EX ap : Z,
         !! (0 <= ap <= Int.max_unsigned /\
             Vint (Int.repr ap) = cp /\
             List_head anext ap = ZMap.get ah lpool) &&
         field_at Tsh t_struct_list_head [] (Vundef, cp) ch))))
  ((_head -> _prev) = _t;
   MORE_COMMANDS) POSTCONDITION
```

Fig. 15. the proof goal after excuting "start_function."

Below the line we have one proof goal: the Hoare triple of the function body. The command P is the "PROP()LOCAL()SEP()" clause of the preconditions in "set_list_head_prev_spec". The contents of "PROP()" clause has been mentioned above the line as a hypothesis. We do Hoare logic proof by forward symbolic execution. We follow the C code execution order, statement by statement to prove. So the command C shows only one function statement "_head->_prev=_t". The remaining function statements are hidden in "MORE_COMMANDS".

For many kinds of statements (assignment, return, break, continue), we derive them by the forward tactic, which applies a strongest-postcondition style of proof rule to derive Q.

After we execute the forward more time, the proof target becomes as Fig. 16.

The proof goal becomes an entailment in separation logic, P→Q. VST use "ENTAIL Δ,P⊢ Q" for this entailment, and provide the "entailer!" tactic to derive the entailment.

And in the end, the successful derivation of all the proof goals means that the proof of our function is completed, the function is functionally correct and the function spec is consistent.

```
Espec : OracleKind
ch : val
tv : Z
lpool : list_head_pool
ah : Z
Delta_specs : PTree.t funspec
H : 0 <= tv <= Int.max_unsigned
anext, ap : Z
H0 : 0 <= ap <= Int.max_unsigned
H2 : List_head anext ap = ZMap.get ah lpool
Delta : tycontext
PNch : is_pointer_or_null ch
H3 : is_pointer_or_null (Vint (Int.repr tv))
H4 : field_compatible t_struct_list_head [] ch
H5 : value_fits t_struct_list_head (Vundef, Vint (Int.repr tv))
_____________________________________(1/1)
field_at Tsh t_struct_list_head [] (Vundef, Vint (Int.repr tv)) ch
|-- EX cp : val,
   (EX anext0 : Z,
    (EX ap0 : Z,
     !! (0 <= ap0 <= Int.max_unsigned /\
         Vint (Int.repr ap0) = cp /\
         List_head anext0 ap0 = ZMap.get ah (set_list_head_prev lpool ah tv))
&&
     field_at Tsh t_struct_list_head [] (Vundef, Vint (Int.repr tv)) ch))
```

Fig. 16. the intermediate proof goal after executing the second forward tactic.

V. Related Work and Conclusion

The team of the Department of Computer Science at the University of Tokyo verified the memory management provided by the Topsy Operations Department [9]. The memory management module of the Topsy kernel uses the heap mode to provide basic dynamic memory allocation functions: memory initialization, memory allocation, memory release, and organizes the free memory pool through the block linked list method. They use Coq as a formal tool to define the implementation of the memory allocation algorithm, describe algorithm assertions and specifications through separation logic, and finally prove the correctness of the code interactively in Coq. However, the memory allocation algorithm of the Topsy operating system is too simple. Its verification is limited to verification at the code level, and it is not verified for higher levels. Its verification method is not universal.

The Australian NICTA laboratory initiated the functional correctness verification of the SeL4 (secure embedded L4) [10]. They innovatively adopted a functional language Haskell as an intermediate verification to quickly implement the system prototype, which on the one hand can make the

system prototype more convenient to transform into the theorem prover, on the other hand, the formal verification results can be quickly fed back to the system designer through the Haskell prototype [11]. But the work of Sel4 doesn't perform layered verification of the system, the verification degree is low, and the verification efficiency is low. Klein introduced the final verification results in the report [12]. The SeL4 kernel contains more than 8,700 lines of C code, and more than 200,000 lines of Isabelle/HOL code that has been formally defined and proven. This work takes a total of 20 man-years of work.

The Flint team at Yale University verified the functional correctness [13] and security properties [14] of the CertiKOS kernel. The CompCert compiler we use is the basis of this project. The project adopts a layered verification method to layer the functional structure of CertiKOS, and each layer is verified independently, which greatly improves the verification efficiency. At the beginning of the design, CertiKOS followed the code structure with simple structure, clear hierarchy, and low coupling between different modules. Therefore, based on the above verification method, the functional correctness verification of more than 3000 lines of C code and assembly code only took 1 person-year. Compared with the SeL4 project, its verification efficiency is improved several times. However, the code of the CertiKOS project does not support pointer linked lists, and for complex data structures such as pointer linked lists, this verification method is not applicable.

The traditional VST verification method supports separation logic and can be used to verify pointer programs, but this method does not separate data representation from operation. For more complex data structures, there will be problems such as complex specifications and high inference difficulty.

Our verification method, on the one hand, is based on the idea of layered verification, which can improve the efficiency of code verification. We use our method to verify function "list_add_tail". Its refactored C code (including data structure) has 32 lines. The Coq code we use is only 320 lines in total, and the Coq code that requires manual handwriting is only 190 lines. On the other hand, our method can verify the data structure of the pointer linked list, and uses a method of separating data representation and operation, which greatly simplifies the difficulty of verification.

We take the function "list_add_tail" in the MILS scheduling module as an example to show a proof method. We first separate the complex structure of the function, use independent functions to encapsulate the complex structure, use the Clightgen tool of CompCert to convert the code into Clight code, then build this function's spec of the abstraction layer and implementation layer and finally prove the consistency of the function's spec of the abstraction layer and the implementation layer, and proves the functional correctness of this function. This method can simplify the proof of complex data structures, which not only be applied to the proof of the doubly linked list in this example, but also the proofs of other complex data structures.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Yang Gao, conceptualization, methodology development, data curation, software programming, validation, writing–original draft.

Xia Yang, supervision, project administration, writing–review and editing

Wensheng Guo, Supervision, writing–review and editing.

Xiutai Lu, investigation, resources.

All authors had approved the final version.

Acknowledgment

The authors thank the anonymous reviewers for their valuable help and useful comments. Our colleagues also kindly provided their helpful insights while the trusted operating system architecture was being certified.

References

[1] J. M. Rushby, "Design and verification of secure systems," *ACM SIGOPS Operating Systems Review*, 1981, vo. 15, no. 5, pp. 12-21.

[2] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *Proc. 2008 Euromicro Conference on Real-Time Systems*, 2008, pp. 181-190.

[3] A Guasque, P. Balbastre, and A. Crespo, "Real-time hierarchical systems with arbitrary scheduling at global level," *Journal of Systems and Software*, 2016, vol. 119, pp. 70-86.

[4] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2002, pp. 55-74.

[5] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proc. 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, 2005, vol. 10, p. 398.

[6] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language," *Journal of Automated Reasoning*, 2009, vol. 43, no. 3, pp. 263-288.

[7] R. Krebbers, X. Leroy, and F. Wiedijk, "Formal C semantics: CompCert and the C standard," in *Proc. International Conference on Interactive Theorem Proving*, Springer, Cham, 2014, pp. 543-548.

[8] Q. Cao, L. Beringer, S. Gruetter *et al.*, "VST-Floyd: A separation logic tool to verify correctness of C programs," *Journal of Automated Reasoning*, 2018, vol. 61, no. 1-4, pp. 367-422.

[9] G. Fankhauser, C. Conrad, E. Zitzler, and B. Plattner, "Topsy - A Teachable Operating System," 2000.

[10] J. Liedtke, "On micro-kernel construction," *ACM SIGOPS Operating Systems Review*, 1995, vol. 29, no. 5, pp. 237-250.

[11] G. Heiser, K. Elphinstone, I. Kuz *et al.*, "Towards trustworthy computing systems: Taking microkernels to the next level," *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, no. 4, pp. 3-11.

[12] G. Klein, K. Elphinstone, G. Heiser *et al*., "L4: Formal verification of an OS kernel," in *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 207-220.

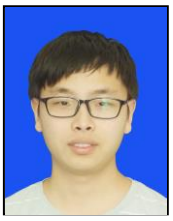
[13] R. Gu, "An extensible architecture for building certified sequential and concurrent OS kernels," Yale University, 2016.

[14] M. Liu, L. Rieg, Z. Shao *et al*., "Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation," in *Proc. ACM on Programming Languages*, 2019.

Yang Gao is a post graduate of School of Information and Software Engineering, Electronic Science and Technology of China. His current research is focused on the embedded operating systems and formal verification.

Xia Yang is an associate professor of School of Information and Software Engineering, Electronic Science and Technology of China. Her research interests include Software formalization theory and methods, blockchain and its formal verification, high-confidence embedded operating system, embedded virtualization technology.

Wensheng Guo is an associate professor of School of Information and Software Engineering, Electronic Science and Technology of China. His research interests include machine learning, image recognition, computer vision.



Xiutai Lu is a post graduate of Electronic Science and Technology of China, Chengdu, China. His current research areas include embedded operating systems and Machine learning.