Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Verification of Operating Systems for Internet of Things in Smart Cities from the Assembly Perspective using Isabelle/HOL

ZHENJIANG QIAN[1], WEI LIU[2], YIYANG YAO[3]
[1]School of Computer Science and Engineering, Changshu Institute of Technology, Suzhou, 215500, China
[2]NARI Group Corporation (State Grid Electric Power Research Institute), Nanjing 210003, China
[3]State Grid Zhejiang Electric Power Co., Ltd., Hangzhou 310007, China

Corresponding author: Zhenjiang Qian (e-mail: qianzj@cslg.edu.cn).

This work was supported by the Natural Science Foundation of Jiangsu Province under grant No. BK20191475, the fifth phase of "333 Project" scientific research funding project of Jiangsu Province in China under grant No. BRA2020306, the Qing Lan Project of Jiangsu Province in China under grant No. 2019, Science and Technology Project of State Grid Corporation in 2019 (Research on Lightweight Active Immune Technology for Power Monitoring System).

ABSTRACT Formal verification can mathematically prove whether a software satisfies the requirements described in its design. In traditional software development, even if the software systems, especially the operating system for Internet of Things in smart cities, passes the verification test, it is difficult to explain its correctness. The implementation of formal verification after the design and development process proves that the software system meets the expected requirements. This will become a trend for future software development. In this paper, we model the system state of the X86 architecture on the assembly layer, and use a micro operating system prototype for Internet of Things in smart cities as an example to explain the proposed method that can verify operating systems for Internet of Things in smart cities on the assembly layer. The verification result shows that this method is feasible.

INDEX TERMS Assembly layer, Formal verification, Internet of Things, Smart cities, Isabelle/HOL, Operating system.

I. INTRODUCTION

IN an ideal system environment, all systems are strictly designed, developed, and verified with strict formal tools, to ensure that all systems are sound and complete. However, due to the complexity, formal methods are really hard to be used in traditional software development.

For the implementation of a software system, its vulnerabilities are first determined through numerous software tests. Even if no vulnerabilities are found, it cannot be proved that the implementation of the software achieves the expected results.

Owing to the complexity of huge systems, such as an operating system (OS) for Internet of Things (IoT) in smart cities, it is difficult to elaborate their correctness and completeness. Formal verification is a method used to mathematically and strictly verify whether a software system has been implemented as per its design. Even if a software system is not designed and developed by formal methods, we can verify it via modeling and description in logic systems.

This paper illustrates our method of modeling an OS for IoT in smart cities with the X86 architecture on the assembly layer. We considered a self-implemented micro OS prototype called MOS as an example to demonstrate the feasibility of our method.

II. RELATED WORK

A. FORMAL METHODS IN IOT

There are several security issues and challenges in IoT . For example, vehicular network security proves to be a hot topic [1]. What's more, various researchers have presented systems to help vehicle to make safe decisions [2] and provided trust systems for managing vehicle-to-everything communications [3].

As a method to verify the security of a software system, formal methods [4] were first proposed in the 1950s. However, the formal verification of OSs began in the 1970s and formal methods was widely used to verify the safety requirements of communication-based train control (CBTC) systems [5].

In 1990s, researchers started to model railway interlocking systems through model checking techniques [6]. With the development of technology, distributed railway control systems have been developed and verified [7]. During the past few years, researchers have dedicated to simplifying the verification of CBTC systems [8].

B. THE YALE FLINT GROUP

The Yale FLINT Group aimed to develop a novel and practical programming architecture to build large-scale verified system software. The members of the group have made substantial contributions in the field of formal verification, such as presenting a new framework for certified binaries [9], verifying the concurrent OS kernel [10], abstracting preemptive schedulers [11], presenting a language for certified software [12], and verifying distributed systems [13].

C. THE TRUSTWORTHY SYSTEMS GROUP

The Trustworthy Systems Group belongs to the Data61 department of the Commonwealth Scientific and Industrial Research Organisation; it uses a rigorous formal approach to develop trusted software systems. The group has made significant contributions in OSs, formal methods, and programming languages.

The L4.Verified project [14] and the seL4 project [15] were initiated by the original National ICT Australia. Among them, seL4 is the world's most advanced and reliable OS microkernel. Its correctness is verified by the L4.Verified project, and its relevant verification was completed in 2009 [16].

For convenience and reliability while verifying software, a verified compiler named CakeML was presented for a high-level functional language. Additionally, CakeML has verified generational garbage collector [17]. It was used to verify efficient function calls in 2017 [18] and certificate checker in 2018 [19].

A functional language called Cogent [20] was presented to reduce the cost of code for verifying systems. It was used to verify code for file system implementations [21] and property-based testing [22].

To help build microkernel-based embedded systems software [23], CAmkES [24] was presented, which is a platform abstracting low-level mechanisms.

With the development of concurrency and distributed systems, formal models need to be extended. For example, we need to ensure liveness properties [25], and security properties of a compiler [26].

Nowadays, the group is still committed to verified software, such as designing and implementing time protection [27] [28], abstracting the memory management unit [29], and examining the evolution of the L4 kernels [30].

III. MODEL OF SYSTEM STATE

A. DATA LENGTH

In the X86 architecture, the data lengths of the operands are mainly 32, 16, and 8 bits. The aliases for these data lengths are defined as follows.

```
type_synonym u32 = “32 word"
type_synonym u16 = “16 word"
type_synonym u8 = “8 word"
```

We use the keyword “type_synonym” to define aliases. For example, “u32” is an alias for “32 word.” Here, “word” denotes the data type provided in the Isabelle/HOL library; “1 word” represents 1-byte data and “32 word” represents 32-byte data.

B. REGISTER

The model of the register is shown below.

```
record x86_register = eax :: u32
                      edx :: u32
                      ecx :: u32
                      ebx :: u32
                      ebp :: u32
                      esi :: u32
                      edi :: u32
                      esp :: u32
                      cs :: u16
                      ss :: u16
                      ds :: u16
                      es :: u16
                      fs :: u16
                      gs :: u16
                      eflags :: u32
                      eip :: u32
                      eiz :: u32
```

The keyword “record” is used to define the structure type. The register structure contains variables such as eax, edx, and ecx. The type of each variable is the data type after the symbol “::”. For example, eax is a 32-byte variable that represents the EAX register. Especially, the eiz variable represents the EIZ pseudo register, whose value is always zero.

C. MEMORY

We consider memory as a mapping from the addresses to the 8-byte data. In particular, to simulate the execution of instructions, it is necessary to know the length of each instruction. The memory for storing instructions is modeled separately. The memory model can be represented as follows.

```
type_synonym x86_memory = “nat ⇒ u8”
type_synonym x86_instrmem = “nat ⇒ (nat ×
x86_instruction)”
```

Here, “nat” denotes an integer variable, “nat ⇒ u8” represents the mapping of an integer to 8-byte data, and “(nat × instruction)” represents the “(position, instruction)” tuple. The model of “instruction” will be introduced below.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3047411, IEEE Access

IEEE Access

Qian *et al.*: Verification of Operating Systems for Internet of Things in Smart Cities from the Assembly Perspective using Isabelle/HOL

D. OPERAND

The X86 architecture consists of 11 operand formats (seven addressing modes), and the possibility of all operands can be divided into three types: immediate data, registers, and memory references. To cover all nine memory reference types, eiz can be used when no registers are required, and 0 is used when no offset is needed. Considering the 32-bit operand as an example, the model can be represented as follows.

```
datatype operand32 =
   Imm32 int
 | Reg32 r32
 | Offset32 int r32 r32 int
```

The keyword "datetype" is used to define the "operand32" type, which represents the 32-bit operand. This type is addressed by the immediate data ("Imm32"), registers ("Reg32"), or memory references ("Offset32"). With respect to the memory reference type, "int r32 r32 int" is a set of variables to represent the value of the memory address "(integer 1 + register 1 + register 2 × integer 2)." We can use eiz when no registers are required, and 0 when no offset is needed.

E. INSTRUCTION

Depending on the data length of the operands, each type of instruction basically contains the 8-bit version (ending with "B"), the 16-bit version (ending with "W"), and the 32-bit version (to "L" end). Several instructions are involved and we have only listed a few here, as demonstrated below.

```
datatype x86_instruction =
   (* nop *)
   NOP
   (* add *)
 | ADDB operand8 operand8
 | ADDW operand16 operand16
 | ADDL operand32 operand32
...
```

We divide instructions into blocks according to the type of assembly instructions, such as "add" instruction and "call" instruction. Most types of instructions contain an 8-bit version (ending with "B"), a 16-bit version (ending with "W"), and a 32-bit version (ending with "L").

F. CODE

The assembly code segment represents the collection of instructions, shown as follows.

```
type_synonym x86_code = "(nat × (nat ×
x86_instruction)) list"
```

Here, "code" is of type "list," which is similar to the array type in most programming languages.

G. SYSTEM STATE

Finally, the system state is composed of three parts, i.e., registers, memory for storing data, and memory for storing instructions. The code of the model is given below.

```
record x86_state = R :: x86_register
                   M :: x86_memory
                   I :: x86_instrmem
```

IV. SIMULATION OF EXECUTION

A. READING AND WRITING REGISTERS

First, we implement the 32-bit version of the auxiliary function to read and write registers as follows.

```
definition gr32::"x86_state ⇒ r32 ⇒ u32"
where
"gr32 s r ≡ (case r of
  EAX ⇒ eax (R s)
| EDX ⇒ edx (R s)
| ECX ⇒ ecx (R s)
| EBX ⇒ ebx (R s)
| EBP ⇒ ebp (R s)
| ESI ⇒ esi (R s)
| EDI ⇒ edi (R s)
| ESP ⇒ esp (R s)
| EFLAGS ⇒ eflags (R s)
| EIP ⇒ eip (R s)
| EIZ ⇒ 0
)"
```

```
definition sr32::"x86_state ⇒ r32 ⇒
u32 ⇒ x86_state"
where
"sr32 s r v ≡ (case r of
  EAX ⇒ s (| R := (R s) (| eax := v |) |)
| EDX ⇒ s (| R := (R s) (| edx := v |) |)
| ECX ⇒ s (| R := (R s) (| ecx := v |) |)
| EBX ⇒ s (| R := (R s) (| ebx := v |) |)
| EBP ⇒ s (| R := (R s) (| ebp := v |) |)
| ESI ⇒ s (| R := (R s) (| esi := v |) |)
| EDI ⇒ s (| R := (R s) (| edi := v |) |)
| ESP ⇒ s (| R := (R s) (| esp := v |) |)
| EFLAGS ⇒ s (| R := (R s)
              (| eflags := v |) |)
| EIP ⇒ s (| R := (R s) (| eip := v |) |)
)"
```

We use the keyword "definition" to define functions that have no recursion, such as "gr32" and "sr32." For the "gr32" function, "state ⇒ r32 ⇒ u32" implies that "gr32" takes the data from the "state" and "r32" types as input, and outputs the data of "u32" type. Here, "gr32 s r" indicates that "s" and "r" are the function parameters, and "case r of" implies discussing the situation of "r." "EAX ⇒ eax (R s)" means that if "r" is "EAX," return the value of eax in the system status. For the "sr32" function, "(| R := (R s) (|eax := v|) |)" stands for setting "eax" in the system state to the variable "v."

Then, we implement the 16-bit version based on the 32-bit version, shown as follows.

```
definition gr16::"x86_state ⇒ r16 ⇒ u16"
where
"gr16 s r ≡ (case r of
  AX ⇒ ucast (eax (R s))
| DX ⇒ ucast (edx (R s))
| CX ⇒ ucast (ecx (R s))
| BX ⇒ ucast (ebx (R s))
| BP ⇒ ucast (ebp (R s))
| SI ⇒ ucast (esi (R s))
| DI ⇒ ucast (edi (R s))
| SP ⇒ ucast (esp (R s))
| IP ⇒ ucast (eip (R s))
)"

definition sr16::"x86_state ⇒ r16 ⇒ u16 ⇒
x86_state"
where
"sr16 s r v ≡ (case r of
  AX ⇒ sr32 s EAX
   ((gr32 s EAX AND 0xffff0000)
  OR ucast v)
| DX ⇒ sr32 s EDX
   ((gr32 s EDX AND 0xffff0000)
  OR ucast v)
| CX ⇒ sr32 s ECX
   ((gr32 s ECX AND 0xffff0000)
  OR ucast v)
| BX ⇒ sr32 s EBX
   ((gr32 s EBX AND 0xffff0000)
  OR ucast v)
| BP ⇒ sr32 s EBP
     ((gr32 s EBP AND 0xffff0000)
  OR ucast v)
| SI ⇒ sr32 s ESI
   ((gr32 s ESI AND 0xffff0000)
  OR ucast v)
| DI ⇒ sr32 s EDI
   ((gr32 s EDI AND 0xffff0000)
  OR ucast v)
| SP ⇒ sr32 s ESP
   ((gr32 s ESP AND 0xffff0000)
  OR ucast v)
| IP ⇒ sr32 s EIP
   ((gr32 s EIP AND 0xffff0000)
  OR ucast v)
)"
```

Similarly, we implement the 8-bit version based on the 16-bit version; this implementation has not been included in the paper.

B. READING AND WRITING MEMORIES

The functions of reading and writing memories are implemented in an opposite manner than those of the registers. We first implement the 8-bit version. The 16-bit and 32-bit versions are based on the 8-bit version, which is shown as follows.

```
definition gm16::"x86_state ⇒ nat ⇒ u16"
where
"gm16 s n ≡ ucast (gm8 s n)
+ (ucast (gm8 s (n + 1)) << 8)"

definition gm32::"x86_state ⇒ nat ⇒ u32"
where
"gm32 s n ≡ ucast (gm16 s n)
+ (ucast (gm16 s (n + 2)) << 16)"

definition sm8::"x86_state ⇒ nat ⇒ u8 ⇒
x86_state"
where
"sm8 s n v ≡ s (| M := (M s) (n := v) |)"

definition sm16::"x86_state ⇒ nat ⇒
u16 ⇒ x86_state"
where
"sm16 s n v ≡ sm8 (sm8 s n (ucast v))
(n + 1) (ucast (v >> 8))"

definition sm32::"x86_state ⇒ nat ⇒
u32 ⇒ x86_state"
where
"sm32 s n v ≡ sm16 (sm16 s n (ucast v))
(n + 2) (ucast (v >> 16))"
```

Here, "(M s) n" denotes the value stored in the memory address "n" in the system state. We use the function "ucast" provided by the "word" library for the converse type. This function can expand 8-byte data to 16-byte data, or truncate 16-byte data to 8-byte data.

C. GETTING AND SETTING INSTRUCTIONS

We must set instructions before the execution. We also need to get the current and next instructions during the execution. The related functions are shown as follows.

```
definition geti::"x86_state ⇒
x86_instruction"
where
"geti s ≡
snd ((I s) (u32_nat (gr32 s EIP)))"

definition nexti::"x86_state ⇒ nat"
where
"nexti s ≡ (u32_nat (gr32 s EIP)) +
fst ((I s) (u32_nat (gr32 s EIP)))"

primrec seti::"x86_state ⇒
x86_code ⇒ x86_state"
where
"seti s [] = s" |
"seti s (i # is) = seti (s (| I := (I s)
  ((fst i) := (snd i)) |)) is"
```

D. READING AND WRITING OPERANDS

To read the source operand, the three cases of immediate data, registers, and memory references must be considered. There are no instructions that take the immediate data as the destination operand; therefore, to write the destination operand, only two cases are required to be considered. For example, the 32-bit version has been elaborated below.

```
definition go32::"x86_state ⇒ operand32 ⇒
u32"
where
"go32 s op ≡ (case op of
  (Imm32 i) ⇒ int_u32 i
| (Reg32 r) ⇒ gr32 s r
| (Offset32 i1 r1 r2 i2) ⇒
gm32 s (u32_nat (int_u32 i1 + gr32 s r1
+ gr32 s r2 * int_u32 i2))
)"
```

```
definition so32::"x86_state ⇒ operand32 ⇒
u32 ⇒ x86_state"
where
"so32 s op v ≡ (case op of
  Reg32 r ⇒ sr32 s r v
| Offset32 i1 r1 r2 i2 ⇒
sm32 s (u32_nat (int_u32 i1 + gr32 s r1
+ gr32 s r2 * int_u32 i2)) v
)"
```

E. UPDATING INSTRUCTION REGISTER

It is easy to update the instruction register through the functions defined below.

```
definition ueip::"x86_state ⇒ x86_state"
where
"ueip s ≡ sr32 s EIP (nat_u32 (nexti s))"
```

F. UPDATING PROGRAM STATUS WORD

On observing the instructions related to updating the program status word, it is evident that most of the cases can be summarized in three cases. For some instructions such as "add," the OF, CF, SF, and ZF registers must be updated; for "sub," the CF register must be reversed; and for "or," the OF and CF registers must be cleared.

In the first case, we still take the 32-bit version as an example as follows:

```
definition uof32::"x86_state ⇒ u32
⇒ u32 ⇒ x86_state"
where
"uof32 s s1 s2 ≡ sf s OF
(u32_sint (s1 + s2) ≠
u32_sint s1 + u32_sint s2)"
```

```
definition usf32::"x86_state ⇒ u32
⇒ x86_state"
where
"usf32 s d ≡ sf s SF (u32_sint d < 0)"
```

```
definition uzf32::"x86_state ⇒ u32
⇒ x86_state"
where
"uzf32 s d ≡ sf s ZF (d = 0)"
```

```
definition ucf32::"x86_state ⇒ u32
⇒ u32 ⇒ x86_state"
where
"ucf32 s s1 s2 ≡
sf s CF (u32_uint (s2 + s1) ≠
u32_uint s2 + u32_uint s1)"
```

```
definition uf32::"x86_state ⇒ u32
⇒ u32 ⇒ u32 ⇒ x86_state"
where
"uf32 s s1 s2 d ≡ ucf32 (uzf32
(usf32 (uof32 s s1 s2) d) d) s1 s2"
```

The result of the corresponding register can be simulated with the built-in operation of Isabelle/HOL, such as updating the OF register here. If the result of adding two numbers as 32-bit signed numbers is different from that of adding the two numbers as integers, it implies that an overflow has occurred.

In the second case, we only need to modify the update of the CF register.

```
definition ucf32'::"x86_state ⇒ u32
⇒ u32 ⇒ x86_state"
where
"ucf32' s s1 s2 ≡ sf s CF
(u32_uint (s2 + s1) =
u32_uint s2 + u32_uint s1)"
```

```
definition uf32'::"x86_state ⇒ u32
⇒ u32 ⇒ u32 ⇒ x86_state"
where
"uf32' s s1 s2 d ≡ ucf32'
(uzf32 (usf32 (uof32
s (-s1) s2) d) d) (-s1) s2"
```

In the third case, we only need to modify the update of the OF and CF registers.

```
definition cof::"x86_state ⇒ x86_state"
where
"cof s ≡ sf s OF False"
```

```
definition ccf::"x86_state ⇒ x86_state"
where
"ccf s ≡ sf s CF False"
```

```
definition uf32''::"x86_state ⇒ u32
```

```
⇒ u32 ⇒ u32 ⇒ x86_state”
where
“uf32” s s1 s2 d ≡ ccf
(uzf32 (usf32 (cof s) d) d)”
```

G. SIMULATING ARITHMETIC LOGIC OPERATIONS

We use the built-in operations to simulate binary operations in Isabelle/HOL. Here, we consider the 32-bit version as an example, as follows.

```
definition a8::“x86_state ⇒ operand8 ⇒
operand8 ⇒ operator ⇒ u8”
where
“a8 s o1 o2 op ≡ (case op of
  ADD ⇒ go8 s o2 + go8 s o1
| SUB ⇒ go8 s o2 − go8 s o1
| OR’ ⇒ go8 s o2 OR go8 s o2
...
)”
```

H. SIMULATING BEHAVIORS OF INSTRUCTIONS

According to the content of the second volume of Intel’s official documentation, we can model the behavior of instructions. Here, we use “push” as an example. The process of modeling the remaining instructions is similar and is not shown here.

First, we check the semantics of the instruction, shown as follows.

```
IF StackAddrSize = 64
THEN
IF OperandSize = 64
THEN
RSP ← RSP − 8;
Memory[SS:RSP] ← SRC;
(* push quadword *)
ELSE IF OperandSize = 32
THEN
RSP ← RSP − 4;
Memory[SS:RSP] ← SRC;
(* push dword *)
ELSE (* OperandSize = 16 *)
RSP ← RSP − 2;
Memory[SS:RSP] ← SRC;
(* push word *)
FI;
...
```

Second, we look for the 32-bit versions and find the specific behavior of these versions from the above behaviors, as follows.

```
ESP ← ESP − 4;
Memory[SS:ESP] ← SRC;
(* push dword *)
```

Then, we check the effect of the instruction on program status word. The effect was found to be “None,” that is, there is no need to update the program status word.

Finally, the model of instruction “pushl” is built according to the behavior shown below.

```
definition pushl::“x86_state ⇒
operand32 ⇒ x86_state”
where
“pushl s o1 ≡ ueip (sm32
(sr32 s ESP (gr32 s ESP − 4))
(u32_nat (gr32 (sr32 s ESP
(gr32 s ESP − 4)) ESP))
(go32 (sr32 s ESP
(gr32 s ESP − 4)) o1))”
```

I. SIMULATING SINGLE-STEP EXECUTION

Now, we combine the behaviors of all instructions. A function is designed to simulate single-step execution as follows.

```
primrec step::“x86_state ⇒
x86_instruction ⇒ x86_state”
where
“step s (NOP) = nop’ s” |
“step s (ADDB o1 o2) = addb s o1 o2” |
“step s (ADDW o1 o2) = addw s o1 o2” |
“step s (ADDL o1 o2) = addl s o1 o2” |
...
```

The function “step” will simulate the execution of an instruction with the help of the function defined above.

J. SIMULATING MULTI-STEP EXECUTION

Based on the aforementioned function, a function is designed to simulate multi-step execution as follows.

```
primrec exec::“x86_state ⇒ nat ⇒
x86_state”
where
“exec s 0 = s” |
“exec s (Suc n) = exec
(step s (geti s)) n”
```

The function “exec” will simulate the execution of the instruction recursively until n becomes 0.

V. INTRODUCTION TO MOS

A. FEATURES OF MOS

The OS MOS is a micro OS for IoT in smart cities, which has been illustrated in Fig. 1. It boots with GRUB and uses a two-level page table. It can handle common interrupts and has basic memory management and process management. It can interact with users through the console and can run user programs.

B. INTRODUCTION TO ALLOCATE MEMORY

Here, we aim to verify a part of the code of MOS for memory allocation. This OS uses the free linked list and first-fit

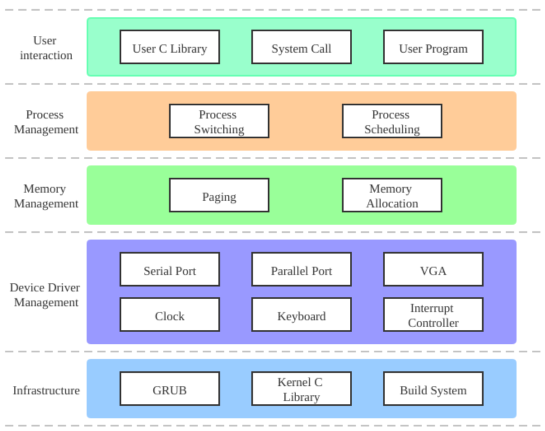


FIGURE 1. System framework of MOS.

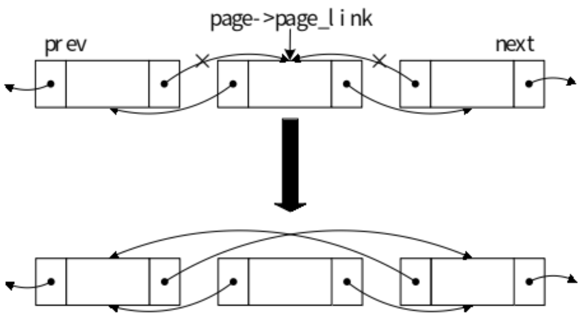


FIGURE 2. Delete page->page_link node.

algorithm to manage the free memory and allocate memory, respectively.

We consider the example of finding a free memory area of just size when allocating pages. Because the size of the free memory area is equal to that of the memory to be allocated, at this time, it is only necessary to delete the corresponding node in the free linked list and modify the size of its free memory.

For the delete operation of the linked list node, because the kernel linked list is a double-linked circular list, when deleting a node, we must modify the "next" pointer of the previous node and the "prev" pointer of the next node, as demonstrated in Fig. 2.

VI. CORRECTNESS VERIFICATION

A. RELATED ASSEMBLY CODE

The assembly code for allocating memory is given below. We have marked the system states before executing this code via comments.

```
80101170 <set_free_list_n_pages >:
# eax = num
80101170: 8b 44 24 04
mov 0x4(%esp),%eax
# free_list.n_pages = num
80101174: a3 c0 b1 12 80
mov %eax,0x8012b1c0
80101179: c3
ret

80101250 <alloc_pages >:
...
# eax = page->page_link->next
# ebx = &page->page_link
# edx = n
# ecx = free_list.n_pages
# esi = page->page_link->prev
80101299: 8b 43 04
mov 0x4(%ebx),%eax
8010129c: 83 ec 0c
sub    $0xc,%esp
# ecx = free_list.n_pages − n
8010129f: 29 d1
sub %edx,%ecx
# page->page_link->next->prev
#   = page->page_link->prev
801012a1: 89 30
mov %esi,(%eax)
# page->page_link->prev->next
= page->page_link->next
801012a3: 89 46 04
mov %eax,0x4(%esi)
# num = ecx
801012a6: 51
push %ecx
801012a7: e8 c4 fe ff ff
call 80101170 <set_free_list_n_pages>
801012ac: 83 c4 10
add $0x10,%esp
...
```

B. MODEL OF RELATED CODE

Because some codes have similar functions, the above codes are divided into blocks according to the functions, and the similar verification processes are omitted.

The code segment "set_n_pages_snippet1" is responsible for setting the number of free pages in the free list.

```
definition set_n_pages_snippet1 :: "code"
where
"set_n_pages_snippet1 ≡ [
  (0x80101170, 4,
    MOVL (Offset32 0x4 ESP EIZ 0)
         (Reg32 EAX)),
  (0x80101174, 5,
    MOVL (Reg32 EAX)
         (Offset32 0x8012b1c0 EIZ EIZ 0))
]"
```

The code snippet "set_n_pages_snippet2" is responsible for returning the function.

```
definition set_n_pages_snippet2 :: "code"
where
"set_n_pages_snippet2 ≡ [
  (0x80101179, 1, RET)
]"
```

The code snippet called "call_snippet1" is responsible for preparing the function call to allocate the space required by the function and calculate the parameters.

```
definition call_snippet1 :: "code"
where
"call_snippet1 ≡ [
  (0x8010129c, 3,
    SUBL (Imm32 0xc) (Reg32 ESP)),
  (0x8010129f, 2,
    SUBL (Reg32 EDX) (Reg32 ECX))
]"
```

The code snippet "list_del_snippet1" is responsible for deleting a linked list node.

```
definition list_del_snippet1 :: "code"
where
"list_del_snippet1 ≡ [
  (0x801012a1, 2,
    MOVL (Reg32 ESI)
         (Offset32 0 EAX EIZ 0))
]"
```

The code snippet "call_snippet2" is responsible for passing the parameters to the function.

```
definition call_snippet2 :: "code"
where
"call_snippet2 ≡ [
  (0x801012a6, 1, PUSHL (Reg32 ECX))
]"
```

The code snippet "call_snippet3" is responsible for actually calling the function.

```
definition call_snippet3 :: "code"
where
"call_snippet3 ≡ [
  (0x801012a7, 2,
    CALL (Imm32 0x80101170))
]"
```

C. VERIFICATION OF RELATED CODE

Now, we verify the correctness of the memory allocation module, that is, whether the assembly code block has satisfied the corresponding requirements.

For the first code snippet, we must verify that the free linked list has been correctly set and the address of the second code snippet is stored in the instruction register.

```
lemma set_n_pages_snippet1_correctness:
"set_n_pages_snippet1_state s"
apply (simp add:
        set_n_pages_snippet1_state_def)
apply (simp add:
        after_set_n_pages_snippet1_def
        before_set_n_pages_snippet1_def)
apply (simp add:
        get_n_pages_def get_num1_def
        set_n_pages_snippet2_address_def)
apply (simp add: set_n_pages_snippet1_def)
apply (simp add: exec mov set_get
        cast update_eip)
apply (simp add: add.commute)
apply (word_bitwise)
done
```

In Isabelle/HOL, various strategies can be applied to prove a theorem through the keyword "apply." After applying a strategy, the theorem prover simplifies the theorem into several sub-goals. Then, we can continue to apply different strategies to the sub-goals until no more sub-goals are generated. Next, we use the keyword "done" to indicate the conclusion of the proof.

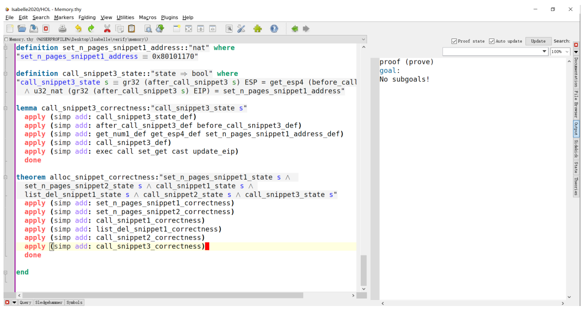
For the second code snippet, we must verify that the function returns correctly and the instruction register stores the return address saved in the stack.

```
lemma set_n_pages_snippet2_correctness:
"set_n_pages_snippet2_state s"
apply (simp add:
        set_n_pages_snippet2_state_def)
apply (simp add:
        after_set_n_pages_snippet2_def
        before_set_n_pages_snippet2_def)
apply (simp add: get_esp1_def
        ret_address_def)
apply (simp add:
        set_n_pages_snippet2_def)
apply (simp add: exec ret set_get
        cast update_eip)
done
```

For the third code snippet, we must verify that the space required by the function is allocated, the function parameters are calculated correctly, and the instruction register stores the address of the fourth code snippet. For the fourth code snippet, we must verify that the node with the linked list item is set correctly and the instruction register stores the address of the fifth code snippet. For the fifth code snippet, we must verify that the function parameters are correctly pushed onto the stack and the instruction register stores the sixth code snippet. The verification for all code snippets is similar; therefore, it has not been included in this paper.

For the sixth code snippet, the correctness of the function call process must be verified.

```
lemma call_snippet3_correctness:
"call_snippet3_state s"
apply (simp add:
        call_snippet3_state_def)
```


Isabelle2020/HOL - Memory.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Memory.thy (%USERPROFILE%\Desktop\Isabelle\verify\memory\)
Proof state
Auto update
Update
Search:
100%
Documentation
File Browser
Output
Sidekick
State
Theories
Query
Sledgehammer
Symbols

```
definition set_n_pages_snippet1_address::"nat" where
"set_n_pages_snippet1_address ≡ 0x80101170"

definition call_snippet3_state::"state ⇒ bool" where
"call_snippet3_state s ≡ gr32 (after_call_snippet3 s) ESP = get_esp4 (before_call
  ∧ u32_nat (gr32 (after_call_snippet3 s) EIP) = set_n_pages_snippet1_address"

lemma call_snippet3_correctness:"call_snippet3_state s"
  apply (simp add: call_snippet3_state_def)
  apply (simp add: after_call_snippet3_def before_call_snippet3_def)
  apply (simp add: get_num1_def get_esp4_def set_n_pages_snippet1_address_def)
  apply (simp add: call_snippet3_def)
  apply (simp add: exec call set_get cast update_eip)
  done

theorem alloc_snippet_correctness:"set_n_pages_snippet1_state s ∧
  set_n_pages_snippet2_state s ∧ call_snippet1_state s ∧
  list_del_snippet1_state s ∧ call_snippet2_state s ∧ call_snippet3_state s"
  apply (simp add: set_n_pages_snippet1_correctness)
  apply (simp add: set_n_pages_snippet2_correctness)
  apply (simp add: call_snippet1_correctness)
  apply (simp add: list_del_snippet1_correctness)
  apply (simp add: call_snippet2_correctness)
  apply (simp add: call_snippet3_correctness)
  done

end
```

```
proof (prove)
goal:
No subgoals!
```

FIGURE 3. Proof results.

```
apply (simp add:
          after_call_snippet3_def
          before_call_snippet3_def)
apply (simp add: get_num1_def
          get_esp4_def
          set_n_pages_snippet1_address_def)
apply (simp add: call_snippet3_def)
apply (simp add: exec call set_get
          cast update_eip)
done
```

The final result is shown in Fig. 3. The correctness of each code snippet indicates the correctness of the code of the allocating memory.

```
theorem alloc_snippet_correctness:
"set_n_pages_snippet1_state s ∧
set_n_pages_snippet2_state s ∧
call_snippet1_state s ∧
list_del_snippet1_state s ∧
call_snippet2_state s ∧
call_snippet3_state s"
apply (simp add:
          set_n_pages_snippet1_correctness)
apply (simp add:
          set_n_pages_snippet2_correctness)
apply (simp add:
          call_snippet1_correctness)
apply (simp add:
          list_del_snippet1_correctness)
apply (simp add:
          call_snippet2_correctness)
apply (simp add:
          call_snippet3_correctness)
done
```

VII. CONCLUSION

In this paper, we proposed a method to verify OSs from the assembly perspective using Isabelle/HOL. We established a system model of the X86 architecture that includes registers, memory, and common assembly instructions. This model can simulate the execution of assembly instructions. Subsequently, we considered a micro OS, named MOS, as an example to verify the correctness of the proposed method. However, the model has limitations; the verification of the whole OS may require a significant amount of resources. We expect the proposed method to help in the verification of OSs.

In the following work, for the completeness of the whole system, we will use different logics according to the characteristics of different modules, such as separation logic and temporal logic.

REFERENCES

[1] Z. Lu, G. Qu, and Z. Liu, "A survey on recent advances in vehicular network security, trust, and privacy," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 2, pp. 760–776, 2018.

[2] T. Rosenstatter and C. Englund, "Modelling the level of trust in a cooperative automated vehicle control system," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 4, pp. 1237–1247, 2017.

[3] B. Brecht and T. Hehn, "A security credential management system for v2x communications," in *Connected Vehicles*. Springer, 2019, pp. 83–115.

[4] G. O'Regan, "Overview of formal methods," in *Mathematics in Computing*. Springer, 2020, pp. 333–354.

[5] L. Schnieder, *Communications-Based Train Control (CBTC)*, Dec. 2019.

[6] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso, "Formal verification of a railway interlocking system using model checking," *Formal aspects of computing*, vol. 10, no. 4, pp. 361–380, 1998.

[7] A. E. Haxthausen and J. Peleska, "Formal development and verification of a distributed railway control system," *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 687–701, 2000.

[8] Z. Yuan, X. Chen, J. Liu, Y. Yu, H. Sun, T. Zhou, and Z. Jin, "Simplifying the formal verification of safety requirements in zone controllers through problem frames and constraint-based projection," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 11, pp. 3517–3528, 2018.

[9] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou, "A type system for certified binaries," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 1, pp. 1–45, 2005.

[10] R. Gu, Z. Shao, H. Chen, J. Kim, J. Koenig, X. Wu, V. Sjöberg, and D. Costanzo, "Building certified concurrent os kernels," *Communications of the ACM*, vol. 62, no. 10, pp. 89–99, 2019.

[11] M. Liu, L. Rieg, Z. Shao, R. Gu, D. Costanzo, J.-E. Kim, and M.-K. Yoon, "Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation," in *Proceedings of the ACM on Programming Languages*, vol. 4. ACM New York, NY, USA, 2019, pp. 1–31.

[12] V. Sjöberg, Y. Sang, S.-c. Weng, and Z. Shao, "Deepsea: a language for certified system software," in *Proceedings of the ACM on Programming Languages*, vol. 3. ACM New York, NY, USA, 2019, pp. 1–27.

[13] J.-Y. Shin, J. Kim, W. Honoré, H. Vanzetto, S. Radhakrishnan, M. Balakrishnan, and Z. Shao, "Wormspace: A modular foundation for simple, verifiable distributed systems," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 299–311.

[14] H. Tuch, G. Klein, and G. Heiser, "Os verification: Now!" in *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, vol. 10. USA: USENIX Association, 2005, pp. 7–12.

[15] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser, "Towards a practical, verified kernel," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. USA: USENIX Association, 2007.

[16] G. Klein, P. Derrin, and K. Elphinstone, "Experience report: sel4: formally verifying a high-performance microkernel," in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, 2009, pp. 91–96.

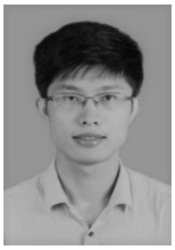
[17] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified cakeml compiler backend," *Journal of Functional Programming*, vol. 29, pp. 1–59, 2019.

[18] S. Owens, M. Norrish, R. Kumar, M. O. Myreen, and Y. K. Tan, "Verifying efficient function calls in cakeml," in *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP. ACM New York, NY, USA, 2017, pp. 1–27.

[19] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. Fox, "A verified certificate checker for finite-precision error bounds in coq and hol4," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–10.

[20] Z. Chen, M. Di Meglio, L. O'Connor, P. Susarla, C. Rizkallah, and G. Keller, "A data layout description language for cogent," in *Proceedings of PriSC 2019*, 2019, pp. 1–3.
[21] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell *et al.*, "Cogent: Verifying high-assurance file system implementations," vol. 44, no. 2, pp. 175–188, 2016.
[22] Z. Chen, L. O'Connor, G. Keller, G. Klein, and G. Heiser, "The cogent case for property-based testing," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, 2017, pp. 1–7.
[23] M. Fernandez, I. Kuz, and G. Klein, "Formalisation of a component platform," in *Proceedings of Operating Systems Design and Implementation (OSDI 2012)*, Hollywood, CA, USA, 2012.
[24] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.
[25] R. van Glabbeek, "Ensuring liveness properties of distributed systems: Open problems," *Journal of Logical and Algebraic Methods in Programming*, vol. 109, pp. 1–19, 2019.
[26] R. Sison and T. Murray, "Verifying that a compiler preserves concurrent value-dependent information-flow security," in *Proceedings of 10th International Conference on Interactive Theorem Proving (ITP 2019)*, Dagstuhl, Germany, 2019, pp. 27:1–27:19.
[27] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: the missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.
[28] G. Heiser, T. Murray, and G. Klein, "Towards provable timing-channel prevention," *ACM SIGOPS Operating Systems Review*, vol. 54, no. 1, pp. 1–7, 2020.
[29] H. T. Syeda and G. Klein, "Formal reasoning under cached address translation," *Journal of Automated Reasoning*, vol. 64, no. 6, pp. 1–35, 2020.
[30] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 1, pp. 1–29, 2016.

ZHENJIANG QIAN was born in Changshu, Suzhou, China, in 1982. He received the Ph.D. degree from Nanjing University, Nanjing, China, in 2013. He is currently the Deputy Dean of School of Computer Science and Engineering, Changshu Institute of Technology in China, where he is mainly responsible for the science research. His research interests include information security, cyper-physical systems, and theorem proving.



WEI LIU was born in Daping, Ganzhou, China, in 1986. He received the M.S. degree from Nanjing University, Nanjing, China, in 2012. He is currently the Director of the Security OS Research Office, NARI Group Corporation (State Grid Electric Power Research Institute), where he is mainly responsible for the underlying security research of the operating system, covering cloud security, mobile security, and industrial security to support the development of the entire department. His research interests include power system information security and operating system.

YIYANG YAO received the B.Eng. degree in software engineering and the master's degree in computer science from Northwestern Polytechnic University, Xi'an, China, in 2007 and 2010, respectively. Since 2010, he has been a member of the Information and Telecommunications Branch, State Grid Zhejiang Electric Power Co., Ltd. His research interests include network security and image processing.

• • •