Towards Formal Verification of Dynamic Memory Allocator Properties Using BIP Framework

Xiutai Lu
School of information and software engineering, University of Electronic Science and Technology of China, Chengdu,, China,
xtlu@std.uestc.edu.cn

Yang Gao
School of information and software engineering, University of Electronic Science and Technology of China, Chengdu,, China,
gyang@std.uestc.edu.cn

Wensheng Guo*
School of information and software engineering, University of Electronic Science and Technology of China, Chengdu,, China,
gws@uestc.edu.cn

Fengbo Zhang
School of information and software engineering, University of Electronic Science and Technology of China, Chengdu,, China,
1030337689@qq.com

Xia Yang
School of information and software engineering, University of Electronic Science and Technology of China, Chengdu,, China,
xyang@uestc.edu.cn

Jun Wan
School of information and software engineering, University of Electronic Science and Technology of China, Chengdu,, China,
1457221049@qq.com

ABSTRACT

Dynamic storage allocation (DSA) algorithms play an important role in the Real-Time Operating systems (RTOSs) community. It allows the RTOS to use limited memory efficiently. To ensure the DSA properties of a dynamic memory allocator, it is important to verify the implementation of its DSA algorithms. However, most previous works ignore memory interactive behaviors and just verify individually each function involved in DSA. Our main contribution in this paper is to verify the consistency of the memory interactive properties and its implementation. For this purpose, we use the BIP (Behavior, Interaction, Priority) Framework to deal with abstract behaviors, properties, and cross references to implementation code. We chose the TLSF as a testbed for formal verification of dynamic memory allocator properties and have produced a verification of TLSF. Both the behavior operations and property requirements of the TLSF have been specified in the BIP framework and the entire verification process is automated.

CCS CONCEPTS

• **Security and privacy**; • **Formal methods and theory of security**; • **Logic and verification**;

KEYWORDS

Dynamic memory allocator, BIP framework, Automated verification

ACM Reference Format:
Xiutai Lu, Yang Gao, Wensheng Guo*, Fengbo Zhang, Xia Yang, and Jun Wan. 2021. Towards Formal Verification of Dynamic Memory Allocator Properties Using BIP Framework. In *The 5th International Conference on Computer Science and Application Engineering (CSAE 2021), October 19–21, 2021, Sanya, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3487075.3487122

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CSAE 2021, October 19–21, 2021, Sanya, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8985-3/21/10…$15.00
https://doi.org/10.1145/3487075.3487122

1 INTRODUCTION

In practical RTOS, the security of dynamic memory allocation algorithm largely determines the stability of system operation. The goal of the dynamic memory allocation algorithm is to dynamically provide the system program with the amount of memory required at runtime, which has complex DSA properties. The complexity mainly comes from the following aspects: 1) The diversity of memory allocation requirements. 2) Performance considerations of the memory allocation process. 3) Fragmentation management of memory blocks in memory pool. To ensure that the memory allocation can always run efficiently and stably, the memory management module needs to provide a good average response time and a low fragmentation rate. These behaviors and attributes of the dynamic memory allocator are an important starting point for this article. We need to adopt extremely strict formal methods to specify and verify the implementation of these behaviors and properties.

We chose TLSF as a testbed for formal verification of dynamic memory allocator properties. The TLSF algorithm provides explicit allocation and release of memory blocks with a temporal cost$\Theta(1)$. It has the features of automatic memory consolidation, flexibility, low memory fragmentation and has been applied in many systems, such as Amiga OS [1], Xtratum Hypervisors [2], Orocos [3] and so on. But at the same time, the complexity of the algorithm is greatly increased. Therefore, the formal verification work will be more complicated, bring more verification problems, mainly reflected in the following aspects: 1) Formally specify complex data structures; 2) Assertion definition and reasoning of behavior operations; 3) Formal specification of complicated DSA properties.

To deal with the above-mentioned problems, we plan to use the BIP Framework to model TLSF's abstract behaviors [4], properties, and cross references to implementation code, which is a more effectiveness method to verify the dynamic memory allocator used in RTOS. More precisely, our paper makes the following contributions:

- We present a standardized modeling process for quickly defining abstract behavior of a dynamic memory allocator.
- We formally define several critical DSA properties based on the Linear Temporal Logic and verify these properties automatically in the BIP framework.
- We chose the TLSF as a testbed for formal verification of dynamic memory allocator properties and have produced a verification of TLSF.

This paper is organized as follows. Section 2 further discusses the background and related work on verification methods for the memory management module. Section 3 explains the proposed approach and how it is applied to the TLSF algorithm. Section 4 describes the implementation of the verification method and shows the verification results. Section 5 discusses conclusions and future work.

2 RELATED WORK

In this section, we will briefly outline related work. Yu et al. [5] of Yale University used CAP to build an authentication library for dynamic storage allocation, and further used Coq for verification. They defined the MIPS machine model, the formal specification of the memory allocation algorithm, and the nature of the final proof in Coq. But the authentication library only contains the simplest malloc and free operations, and assumes that the memory will not be exhausted. The verified dynamic allocation algorithm is too simple and not representative.

Sarra et al. [6] propose a verification method for smart contracts in a supply chain management system. The method is divided into two parts: (1) use the BIP framework to model the abstract behavior of the smart contract, (2) convert the BIP model to the NuSMV model and verify the security attributes. In the verification process, linear sequential logic is used to describe the security properties of the smart contract, and finally the NuSMV model checker is used to verify whether the security properties are satisfied. This verification method uses BIP language to describe the behavior of the system, and uses security attributes to describe the requirements of the system, which verifies the consistency of the implementation and the requirements.

The Japan Institute of Information Technology verified the memory management of the Topsy operating system [7]. The memory management module of the Topsy kernel uses heap management to provide basic dynamic memory allocation functions. This method also uses Coq as a formal tool to define the implementation of the memory allocation algorithm, uses separation logic to describe the assertion and specification of the algorithm, and finally proves the correctness of the code interactively in Coq. But the Topsy kernel is a kernel used for teaching, and its memory allocation algorithm is too simple. In addition, its verification is only for verification at the code level.

Qiao et al. [8] conducted a formal verification of the spacecraft memory management system based on the Event-B method. The verification process is based on the Rodin modeling tool, using the Event-B mathematical abstraction method to formalize the system memory management model. For the generated model, verify whether it can meet the properties extracted from the requirements of the memory management module, so as to illustrate the correctness of the operating system memory model. In the modeling process, the memory management requirements, design and implementation are hierarchically modeled from top to bottom. The method for determining the safety requirements of the memory management module in this project is worthy of reference, but the verification process is complicated and not universal.

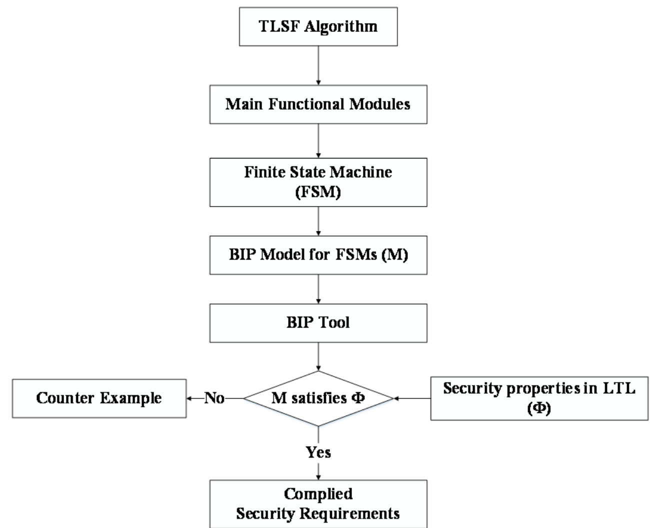

Figure 1: General Approach.

Frederic et al. [9] proposed a formal verification method for the memory allocation module of the IoT operating system. This method uses the deductive verification tool, Frama-C, to specify and verify the code. Frama-C provides various static and dynamic analyzers as separate plug-ins, and comes with its own behavioral specification language, ACSL. The paper uses ACSL to describe the function of the memory allocation module. After the automatic verification of Frama-C, an out-of-bounds access error was found. However, the memory allocation module verified by this project uses a static memory allocation algorithm, the amount of code is small, and the security attributes of the verification are few.

3 FORMAL VERIFICATION OF TLSF ALGORITHM

The proposed approach devised for TLSF algorithm verification is depicted in Figure 1. The whole verification process is divided into four steps. In step 1, we analyze the TLSF algorithm, determine the main functions, and remove the unnecessary parts of the code. To reduce the difficulty of modeling and determine the security requirements, we simplify the algorithm flow and divide the TLSF algorithm into three functional modules: creating a memory pool, allocating memory, and releasing memory. In step 2, a set of conversion rules is used to convert each functional module of the TLSF algorithm into a finite state machine (FSM), which provides an appropriate level of abstraction for BIP modeling.

In step 3, we use the Behavior Interaction Priority (BIP) framework [10] to model the abstract behavior of the finite state machine

(FSM) to build the BIP model of the TLSF algorithm. The BIP framework is a general system-level formal modeling framework that supports the hierarchical structure and includes a set of tools that support modeling, model conversion, simulation, verification, and code generation. The BIP framework has strong design correction features, which are convenient for modeling the interaction between FSMs. It has been used many times to verify the consistency of system design and implementation, such as robotic systems and real-time systems [11-12].

In step 4, to check the consistency of the TLSF algorithm and security requirements, the expected security requirements of the TLSF algorithm are formalized into linear temporal logic (LTL) security properties and applied to the generated BIP model. The BIP tool will check whether each state transition path of the BIP model meets the security properties definition to achieve the purpose of verification. If any security attribute is violated, the path inspection will stop and a counterexample will be given to show the error state, otherwise, it will prove that the TLSF algorithm meets the requirements of all security attribute.

3.1 TLSF Algorithm Analysis

In this section, we will analyze the TLSF algorithm, identify the main functional modules, and related security requirements. TLSF is a dynamic memory allocation algorithm that is very suitable for the embedded field, its goal is to dynamically provide the required amount of memory to the application at runtime [13].

In order to satisfy the constraints of embedded real-time systems such as high real-time requirements, less physical memory, and not supporting MMU. TLSF combines two mechanisms: Segregated list and Bitmaps fit. TLSF maintains a doubly linked-list of multiple free blocks to store free blocks of different sizes. When receiving a memory request, TLSF first performs a two-level index according to Bitmaps fit, finds the corresponding free link list, and takes out the free block pointed to by the linked-list header for allocation.

Compared with other DSA algorithms, TLSF has two notable features: bounded and short response time, bounded and low fragmentation, to fulfill the most important real-time requirements. In general, TLSF operates on memory, while memory management has only two operations: allocation and release. Meanwhile, before performing memory operations, TLSF needs to initialize a large memory block as the memory pool. Therefore, the main functional modules of TLSF can be divided into the following three parts:

- **Creating Memory Pool (CMP):** The module is used to complete various initialization operations, including initializing the free linked-list, initializing the two-dimensional array storing the head of the free linked-list, initializing the index value of the two-level index, etc. After completing a series of initialization operations, insert the initial memory block into the corresponding free linked-list, and set the size of the memory block to complete the creation of the memory pool.
- **Allocating Memory (AM):** The module needs to allocate an appropriate free memory block according to the request. After receiving the memory request, first judge whether the size of the requested memory block meets the requirements. Then according to the two-level index, find the corresponding free block linked-list, and obtain the appropriate free block. If the free block is larger than the requested size, and the free block after cutting is larger than the specified minimum size, the cutting operation will be performed.
- **Releasing Memory(RM):** The module needs to determine whether the released memory block can be merged into a larger free memory block and inserted into the corresponding free linked-list. After receiving the release request, check whether the previous or the next physical block of the memory block is free. If it is free, perform the merge operation, mark the merged memory block as free and insert it into the corresponding free linked list to complete the release operation.

Combined with the above design criteria and functional modules, we put forward the following security requirements. TLSF must at least fulfill these security requirements to ensure the security and reliability of memory management operations.

- The memory block calling the release interface must have been used
- The memory block calling the allocation interface must be free
- Ability to select the most suitable memory block during allocation
- If the allocated memory block is larger than the requested memory block, the allocated memory needs to be divided
- If the allocation request is fulfilled, the allocated address space is continuous
- Any two adjacent memory blocks cannot overlap
- When the state of the memory block changes, the data structure of the free block in the memory management will be updated immediately
- If two adjacent memory blocks are free, the merge operation is performed

3.2 Finite Automatic State Machine Modeling

In this section, we convert the three functional modules of the TLSF algorithm into finite state machines (FSM). A FSM is a mathematical model composed of a finite number of states and a set of transition rules. Its function is to describe the state sequence of the object in its life cycle, and how to respond to various external events. In computer science, FSM is widely used to model application behavior.

FSM operates by responding to a series of events. Each event is within the control range of the transition function belonging to the current state, where the range of the function is a subset of the state. The function returns the next state (perhaps the same state). At least one of these states is the final state, and when the final state is reached, the state machine stops.

In the process of constructing FSM, the function of the functional module is defined as a conversion rule. After the function is executed, the current behavior of the functional module is defined as a state. Taking the AM function module as an example, the entire module is divided into ten states, and each state represents a behavior of the module. Different states jump to other states through their own conversion rules, and state changes also mean that the behavior of the module has changed, as shown in the Figure 2. There

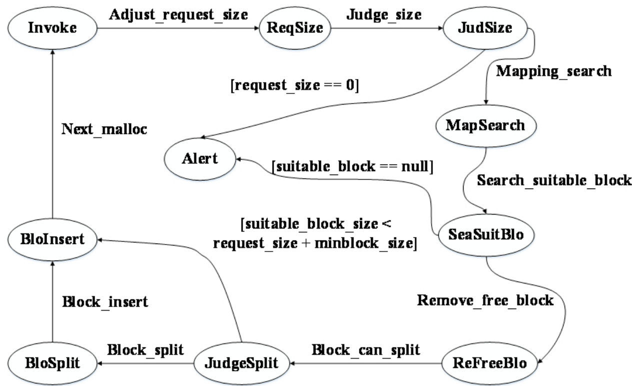

Figure 2: FSM of AM in TLSF Algorithm.

are two types of conversion rules, including general conversion and protection. Protection is a set of conditions placed in square brackets. The previous state must meet the conditions before it can be converted to the next state.

- **Invoke:** The invoke state is the initial state of a FSM, which in turn directs the input towards the appropriate state based on the required transition, i.e. the called function.
- **ReqSize:** Adjust an allocation size to be aligned to word size, and no smaller than internal minimum
- **JudSize:** Before allocating memory, the AM checks whether the adjusted memory size is valid.
- **MapSearch:** According to the adjusted memory size, find the index of the corresponding free linked-list.
- **SeaSuitBlo:** According to the index, find the suitable free memory block. If no free memory block is found, the memory allocation request cannot be satisfied and a warning is issued.
- **RemFreeBlo:** Remove a free block from the free list.
- **JudgeSplit:** When a suitable free memory block is found, the AM checks whether the free memory block can be divided.
- **BloSplit:** The free memory block is divided into two parts, the first part is used to satisfy the memory allocation request, and the second part is marked as Free.
- **BloInsert:** Insert the remaining free memory blocks into the corresponding free linked-list.
- **Alert:** Once it detects that the requested memory size is 0 or no suitable free memory block is found, AM will trigger an alert.

The CMP module has five states: invoke, initializing control structure, initializing free block linked list, setting initial memory block, and setting adjacent memory block. The RM module has five states: invoke, release memory block, merge memory block, mark memory block, and insert free linked-list. Since the modeling process is the same as the AM module, it will not be further explained in this paper.

3.3 Abstract State Behavior Modeling

BIP (Behavior, Interaction, Priority) framework is a combined modeling framework for complex systems, including BIP language and BIP tools. Among them, BIP language is a component-based language for modeling and programing complex systems. The BIP tools include the compiler and the engine used to compile and execute BIP programs. In this paper, we primarily adopt two concepts from BIP language: atomic components and connectors.

Atomic components are the basic elements that describe module behavior and interfaces. They form larger components through the definition of interaction and priority. Atomic components are described by Petri nets or FSM models with added variables and ports. Among them, variables are used to store internal data; ports are migrating tags, which are used to define interfaces for interaction with other modules. The synchronization and data exchange among components are described by connectors. A connector is a stateless entity that supports interaction through a set of interface ports of components. For every interaction, the connector provides the guard and the data transfer to exchange data across the ports involved in the interaction. BIP defines two types of interaction: rendezvous means that all ports must participate in the interaction; broadcast means that there are an initiating port and a series of receiving ports for the interaction.

An atomic component is used to model each functional module's behavior. After modeling the functional modules as FSM, we translate each FSM into BIP atomic components, which are expanded by variables and ports. The **Place** keyword is used to list the available states, while **Initial** is used to identify the initial state. To create the transitions between states, keywords **on**, **from**, and **to** are used. The **Provided** keyword is used to declare the restriction conditions of state transition. The condition preceded by provided represents the guard while statements after do represents the actions or the function's body statement. The **Port** represents the state transition function in the FSM, specifies the unique name of the transition, and is also the interface for interaction. The data exchange and synchronization among different atomic components are realized through the **Port**. Besides, the internal data and interactive data of the component are described using variables. Types of variables are either native or external. In addition, some basic functions of functional modules can be declared using extern, such as judgment functions and bit manipulation functions, making atomic components closer to the actual execution process of functional modules. The BIP notation of the AM module is detailed as follows. The BIP notation of the CMP module and the RM module are similar to the AM module and will not be displayed.

```
Code 1: The BIP notation of the AM module
Atomic component: atom type AM()
Local variable: data size_t malloc_size,
data size_t adjust_size, data size_t
remain_size, data blo remain_blo
Port (Functions inside the module): port
BlockPort_t insize(), port BlockPort_t
index(), port BlockPort_t suitblo(), port
BlockPort_t unsuitblo()
Interactive port: export port OnePort_t
loop(suit_blo), export port BlockPort_t
jug(), export port BlockPort_t fmalloc()
Ten states of the module: place Invoke,
ReqSize, JudSize, SeaIndex, SeaSuit,
```

```
RemFree, BloSplit, RemainBlo, InsRM
Define the state transition process
and variable values: initial to Invoke
do { malloc_size = malloc_size();
adjust_size = set_zero(); remain_size =
set_zero();fl=0; sl=0;id=0;} on insize
from Invoke to ReqSize do {adjust_size
= adjust_request_size(malloc_size);
} on fmalloc from ReqSize to
JudSize provided (id == 0) do
{j=judge_size(adjust_size);} on jug from
ReqSize to JudSize provided (id == 1) do {
id =0;j=judge_size(adjust_size);} on index
from JudSize to SeaIndex provided (j==1)
do {j = judge_suitblo(suit_blo); suit_blo =
search_suitable_block(suit_fl,suit_sl);} on
suitblo from SeaIndex to SeaSuit provided
(j==0) do {j=judge_block(suit_blo);}
```

Connectors connect ports from different functional modules to indicate the interaction mode among them. The rendezvous mode only needs to declare the ports participating in the interaction, while the broadcast mode needs to define the data transmission process. There are three interactions between functional modules:

- After CMP creates the memory pool, AM can start to allocate memory.
- After AM finishes allocating memory once, RM starts to release memory.
- After RM releases memory, AM continues to allocate memory.

The interaction between CMP and AM, RM and AM does not involve data transmission and adopts rendezvous mode, while the interaction between AM and RM needs to ensure that the same memory block is allocated and released. When data interaction is involved, the broadcast mode is adopted. We show the interaction between the creating memory pool module CMP and the allocating memory module AM and RM using connectors (circles) in Figure 3

A snippet of our BIP code to synchronize the connectors between CMP, AM, and RM is detailed as follows.

```
Code 2: BIP Connector code
compound type Compound()
Atomic component of CMP: cmp()
Atomic component of AM: am()
Atomic component of RM: rm()
Sync connector: Syn c1(interactive port of
cmp, interactive port of am)
Sync connector: Syn c2(interactive port of
am, interactive port of rm)
Sync connector: AMRM c3(interactive port of
rm, interactive port of am)
end
```

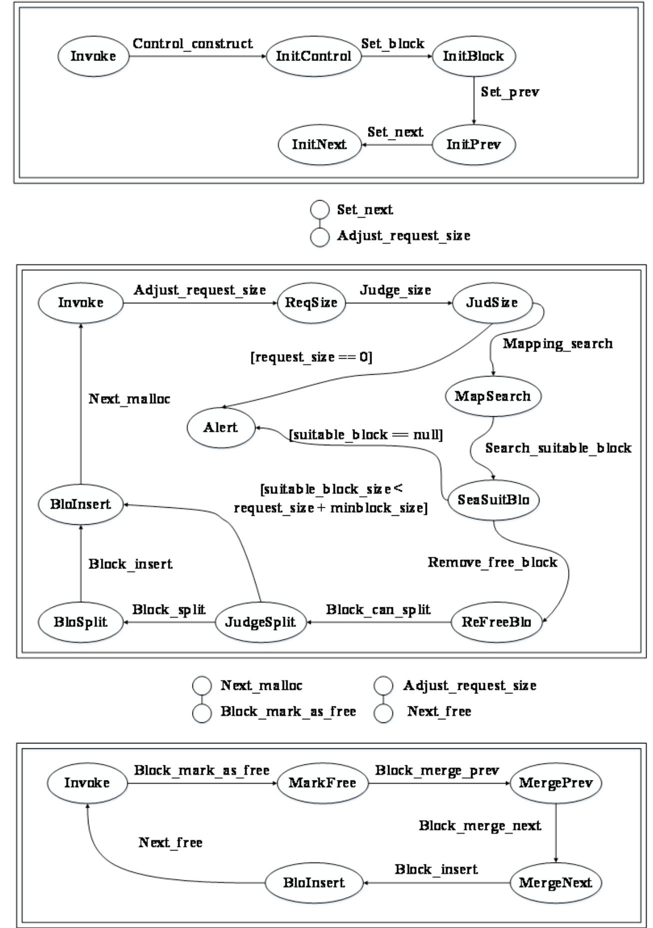

Figure 3: Interaction Modeling between AM, RM, CMP using Connectors.

3.4 Security Properties Modeling

After completing the abstract behavior modeling, we use linear temporal logic (LTL) to describe the security requirements of TLSF algorithm, construct the security attributes of BIP model, and then verify whether each state transition path meets the security attribute definition to achieve the purpose of verification. As a formal specification language, LTL can describe the properties of complex systems, and is widely used in program analysis and verification. In LTL, G {n} denotes the security attribute that any reachable state in n-step migration should maintain, and f {n} represents the active attribute that any reachable state in n-step migration will eventually satisfy. Using LTL, the security requirements of TLSF algorithm are modeled as follows:

Property 1: The memory block allocated by AM must be free:

- G{n}(AM.a==a_judmalloc⇒F{1}(AM.s==s_free))

Property 2: The memory block released by RM must be used:

- G{n}(RM.a==a_judfree⇒F{1}(AM.s==s_used))

Property 3: Am must select the most suitable memory block when it is allocating:

- G{n}(AM.a==a_malloc⇒F{1}(AM.suit==true))

Property 4: If the allocated memory block is larger than the requested memory block, AM must split the allocated memory block:

- G{n}(AM.a==a_sizelarge⇒F{1}(AM.s==s_split))

Property 5: If the allocation request is satisfied, the address space allocated by AM must be continuous:

- G{n}(AM.a==a_judcontinu⇒F{1}(AM.continu==true))

Property 6: Adjacent memory blocks must not overlap:

- G{n}(AM.a==a_judoverlap⇒F{1}(AM.overlap==true))

Property 7: After RM performs the release operation, if two free memory blocks are adjacent, they must be merged:

- G{n}(RM.prevfree==true||RM.nextfree==true⇒F{1}(RM.s==s_merge))

Property 8: After the memory block status changes, the free block data structure in the memory management must be updated immediately:

- G{n}(AM.a==a_statechange⇒F{1}(AM.freeblock==true))

4 IMPLEMENTATION AND VERIFICATION RESULTS

4.1 Operation Results of BIP Model

After using the BIP language to describe the abstract behavior of the TLSF algorithm and establish the BIP model, we use the BIP tool to process the BIP model and generate the executable system. The compiler reads the BIP model, converts it into an intermediate language that can be processed by the simulation engine, and links with the simulation engine. The simulation engine accepts the representation of the BIP model and simulates all possible interaction behaviors. Finally, the simulation engine will generate a complete behavior path and create an executable system. The system automatically simulates the state transition process according to the state and state transition function defined by the BIP language. The simulation results show the selected ports and state internal variables in each state transition process, as shown in Figure 4

4.2 Security Properties Verification Results

We use the property detection engine to detect the executable system, to verify whether the requirements of each security attribute are met. The property detection engine monitors the behavior path of the BIP model based on the security attribute specification. The behavior analysis engine gives the verification results of all legal execution paths or illegal paths. After verification by the property detection engine, the BIP model of TLSF only fulfills the requirements of seven security attributes. Figure 5 shows the verification result for Property 1. To facilitate attribute validation, we define some state variables using constants is detailed as follows.

```
Code 3: Definition of state variables
Block state id definitions:
const data int s_init = 0
const data int s_free = 1
const data int s_used = 2
const data int s_split = 3
const data int s_merge = 4
```

```
var bool AM.overlap false
var bool AM.continu false
var bool AM.freeblo false
var int AM.ma 0
var int AM.ms 0
var int CMP.ca 0
var int CMP.cs 0
var int AM.ma 0
var int AM.ms 0
var int RM.fa 0
var int RM.fs 0
var bool AM.suit false
var bool AM.overlap false
var bool AM.continu false
var bool AM.freeblo false
var bool RM.prevfree false
var bool RM.nextfree false
[BIP ENGINE]: state #0: 3 internal ports:
[BIP ENGINE]:   [0] ROOT.am.insize
[BIP ENGINE]:   [1] ROOT.cmp.initblo
[BIP ENGINE]:   [2] ROOT.rm.init
[BIP ENGINE]: -> choose [1] ROOT.cmp.initblo
[BIP ENGINE]: state #1: 3 internal ports:
[BIP ENGINE]:   [0] ROOT.am.insize
[BIP ENGINE]:   [1] ROOT.cmp.setcur
[BIP ENGINE]:   [2] ROOT.rm.init
[BIP ENGINE]: -> choose [2] ROOT.rm.init
```

Figure 4: Simulation Results.

```
Simulation
SMC type : Hypothesis testing
SMC parameters : Alpha (0.1) - Beta (0.1) - Delta (0.1)
Simulation command : /home/sbip/Desktop/sbip-2.2.3/SMC_Solver/./Workspace/tlsf/Models/system -
Evaluated property : P>=0.9{ G{500} (((AM.ma==10) && (F{1} (AM.ms==1))) || (AM.ma!=10)) }

SMC verdict : true
Required number of traces : 11
Execution time : 00:00:08:909 ms
```

Figure 5: Verification result of Property 1.

```
Action id definitions:
const data int a_init = 0
const data int a_judmalloc = 10
const data int a_judfree = 11
const data int a_malloc = 12
const data int a_searchblo = 13
const data int a_sizelarge = 14
const data int a_adjacentfree = 15
const data int a_judoverlap = 16
const data int a_statechange = 17
const data int a_judcontinu = 18
```

As shown in Figure 6, **Property 3** is not satisfied. After path inspection, it is found that the problem occurs in state 89, as shown in Figure 7. No suitable memory block was found in this state, but the allocation operation was still performed. Combined with code inspection, it was found that in the process of extern TLSF source code, there was a problem with the function of finding a suitable memory block. This function misses the code for judging whether the current free block linked-list is empty, so by default, it returns the first free block linked-list head pointer found. As a result, AM did not select the most suitable memory block when allocating, and a security hole appeared. Therefore, when we were performing security attribute testing, there was a failure result. After modifying

```
Simulation
SMC type : Hypothesis testing
SMC parameters : Alpha (0.1) - Beta (0.1) - Delta (0.1)
Simulation command : /home/sbip/Desktop/sbip-2.2.3/SMC_Solver/./Workspace/tlsf/Models/system
Evaluated property : P>=0.9{ F{1000} ((AM.ma==12) && (AM.suit)) }

SMC verdict : false
Required number of traces : 1
Execution time : 00:00:01:365 ms
```

Figure 6: Verification Result of Property 3.

```
Parsed trace : trace_0
Monitor trace
Trace | Monitoring tab
State89:
@GlobalTime (ns) : 0.0
        bool AM.malloc = true
        bool AM.search_suit_block = false
        bool AM.judge_overlap = false
        bool AM.overlap = true
        bool AM.judge_continu = false
        bool AM.continu = true
        int AM.ma = 12
        bool AM.suit = false
```

Figure 7: Path Inspection Result of Property 3.

```
Simulation
SMC type : Hypothesis testing
SMC parameters : Alpha (0.1) - Beta (0.1) - Delta (0.1)
Simulation command : /home/lxt/LXT/bip/sbip2/./Workspace/tlsf/Models/system --limit 2000 --log-variables
Evaluated property : P>=0.9{ F{1000} ((AM.ma==12) && (AM.suit)) }

SMC verdict : true
Required number of traces : 11
Execution time : 00:00:00:629 ms
```

Figure 8: Verification Result after Correcting Errors.

the error code, we use the property detection engine to verify again. **Property 3** is satisfied, and the verification result is shown in Figure 8

5 CONCLUSION AND FUTURE WORK

This paper presents a formal verification method using the BIP framework and linear temporal logic (LTL), which applies model-checking to the dynamic allocation algorithm of the memory management module. We first converted the TLSF algorithm into FSMs and then used the BIP language to simulate their interactions. To verify the behavior correctness of the TLSF algorithm, the expected security requirements of the TLSF algorithm are formalized as LTL security attributes and applied to the generated BIP model. Finally, the property detection engine is used to give the verification result. The method proposed in [8] is based on the Event-B method for formal verification of the TLSF algorithm. For similar security requirements, an additional 105 manual proofs are required. In contrast, the verification method proposed in this paper achieves purely automated verification in terms of security requirements verification, with a higher degree of automation and better universality. In future work, we will further study the real-time problem of the algorithm. It is difficult to describe the strong real-time nature in the BIP language, and we plan to extend our approach to achieve this verification purpose.

REFERENCES

[1] Reimer J (2007). A history of the Amiga, part 2: The birth of Amiga[J]. last updated Aug, 12, 2.

[2] Masmano M, Ripoll I, Crespo A, Metge J (2009). Xtratum: a hypervisor for safety critical embedded systems[C]. 11th Real-Time Linux Workshop, Citeseer, 263-272.

[3] Bruyninckx H (2001). Open robot control software: the OROCOS project[C]. Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164). IEEE, 3: 2523-2528.

[4] Abdellatif T, Brousmiche K L (2018). Formal verification of smart contracts based on users and blockchain behaviors models[C]. 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), IEEE, 1-5.

[5] Yu D, Hamid N A, Shao Z (2004). Building certified libraries for PCC: Dynamic storage allocation[J]. Science of Computer Programming, 50(1-3), 101-127.

[6] Alqahtani S, He X, Gamble R, Papa M (2020). Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems[C]. Proceedings of the 53rd Hawaii International Conference on System Sciences, 5278-5287.

[7] Fankhauser G, Conrad C, Zitzler E, Plattner B, 2000 Topsy–a teachable operating system[J], Computer Engineering and Networks Laboratory, ETH Zürich, Switzerland.

[8] Qiao L, Yang MF, Tan YL, Pu GG, Yang H (2017). Formal verification of memory management system in spacecraft using Event-B. Ruan Jian Xue Bao/Journal of Software, 28(5), 1204-1220.

[9] Mangano F, Duquennoy S, Kosmatov N (2016). Formal verification of a memory allocation module of contiki with frama-c: A case study[C]. International Conference on Risks and Security of Internet and Systems. Springer, Cham, 114-120.

[10] Bliudze S, Cimatti A, Jaber M, Mover S, Roveri M, Saab W, Wang Q (2015). Formal verification of infinite-state BIP models[C]. International Symposium on Automated Technology for Verification and Analysis. Springer, Cham, 326-343.

[11] Basu A, Gallien M, Lesire C, Nguyen TH, Bensalem S, Ingrand F, Sifakis J (2008). Incremental Component-Based Construction and Verification of a Robotic System[C]. ECAI, 178, 631-635.

[12] Basu A, Bozga M, Sifakis J (2006). Modeling heterogeneous real-time components in BIP[C]. Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), Ieee, 3-12.

[13] Zhang Y, Zhao Y, Sanan D, Qiao L, Zhang J (2019). A Verified Specification of TLSF Memory Management Allocator Using State Monads[C]. International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. Springer, Cham, 122-138.