A Formally Verified Buddy Memory Allocation Model

Ke Jiang [1] , David Sanán [1] , Yongwang Zhao [2] , Shuanglong Kan [1] , Yang Liu [1]
[1] *School of Computer Science and Engineering*
Nanyang Technological University, Singapore
{*johnjiang, sanan, slkan, yangliu*}*@ntu.edu.sg*
[2] *School of Computer Science and Engineering*
Beijing Advanced Innovation Center for Big Data and Brain Computing
Beihang University, Beijing, China
zhaoyw@buaa.edu.cn

***Abstract*—Buddy memory allocation algorithms are widely adopted by various memory management systems for managing memory layouts. Rigorous mathematical proofs provide strong assurance to improve confidence on the reliability of a memory management system. In this paper, we offer a formally verified buddy memory allocation model, which preserves functional correctness and security properties. Firstly, we construct a specification consisting of operations to allocate and dispose memory blocks according to a buddy memory allocation algorithm. Then we verify that the specification preserves key invariants over the memory to guarantee functional correctness of the algorithm. Finally, we verify that the specification also preserves the integrity of the memory. Therefore, they do not affect other memory blocks previously allocated. Our work is done in interactive theorem prover Isabelle/HOL.**

***Index Terms*—Memory Specification, Formal Verification, Functional Correctness, Security.**

1. Introduction

The correctness of applications and libraries of a system using dynamic memory allocation relies on properties the memory structures have to preserve. Errors at any stage of the development process of the memory management components, may break those properties, leading to critical issues in the rest of the system. Along the last decades, formal methods have been successfully applied to the verification of critical systems such as [1], [2]. To improve confidence on the reliability of a memory management, verification of functional correctness and security properties is applied from the top specification layers down to the implementation and the machine code.

Formal verification has been applied to different level of abstractions on memory management systems. For instance, the work in [3] formalizes an abstraction of the memory management as a sequence of write and read operations, and proves the sequential consistency over this abstraction. Also focusing on a high level specification, the work in [4] specifies the memory layouts and the operations that manage the memory including alloc, free, load and store at two levels of abstraction. An abstract specification defines a general memory model for most of imperative languages, and proves properties such as block-valid before and after a store operation. Then a more concrete specification focuses on a C-like language (a large subset of C) with pointer arithmetic. It implements the operations defined in the abstract specification, and proves that these operations satisfy the abstract specification. The properties that have been stated in the abstract specification are also proved in the concrete specification. In a similar way, [5] provides an axiomatic specification of the allocation and free operations for a sequential memory model characterizing well behaved programs. It then implements a sequential memory model with an LLVM-like language [6]. Finally, the axioms on well-behaved programs are used to reason about the correctness of the implementation, and to prove it satisfies a well-behaved program.

At the implementation level, the work from [7] specifies a heap manager modeling low-level implementation details and verifies its functional correctness. It develops a library for separation logic [8] to handle pointers in the C source code and to ensure the separation of memory blocks. The work from [9] specifies and verifies the memory allocation module of Contiki [10], a popular open source operating system for IoT (Internet of Things). This work specifies the memory layouts using Frama-C [11], together with the allocation and deallocation operations. Its verification detects errors like out-of-bounds accesses and potentially harmful situations by using automatic provers like Frama-C/Wp [12]. Also, the work from [13] formally verifies the correctness of a heap allocator at the C source code level, consisting of the memory layouts, and the allocation, free and initialization operations. They use the tools CParser [14] and AutoCorres [15] to translate the C source code into an implementation model in the Isabelle/HOL theorem prover [16].

Although an extensive number of works have formally specified and verified abstract models and implementations for dynamic memory allocation systems, up to our knowledge no work has been done to specify and verify buddy memory allocation. Originally described by [17], a buddy

memory allocation organizes the memory in logical layers with a power of two number of nodes, in which children of the same logical node are *buddies* one of each other. Free memory blocks are split during allocation to adequate the requested amount of memory to their size, and hence reduce the external fragmentation. During deallocation of a memory block, a set of free buddy memory blocks are merged into a free block, that belongs to an upper logical layer. This memory allocation system is widely used in many operating systems. For example, BSD [18] uses this memory system for blocks smaller than a page, i.e., 4 kilobytes. The Zephyr OS [19], a micro-kernel from the Linux Foundation focusing on IoT devices, also uses this memory system.

As a first step to achieve correctness of a buddy memory management, we develop and verify a specification for a buddy memory algorithm. The specification provides applications with services for allocation and disposal of memory blocks, which completely captures the behavior of the algorithm. Then we specify and prove properties over the memory layouts to achieve functional correctness of the operations and invariants related to the memory logical layouts. Finally, we show that the memory services preserve the integrity of other previously allocated memory blocks. The specification and proofs have been carried out in the Isabelle/HOL proof assistant and it is composed of about 5000 lines of specifications and proofs. The interested reader can access the models and proofs at [1].

The paper is organized as follows. The second section provides a background on the buddy memory allocation algorithms and the Isabelle/HOL verification environment. Section three is tackles with the buddy memory allocation model. The proofs to properties for functional correctness and the verification of integrity for security are arranged in section four. The last section we give the conclusions and future work.

2. Background

In this section, we give the necessary backgrounds for the buddy memory allocation algorithms and the Isabelle/HOL theorem prover.

2.1. Buddy Allocation Algorithms

The buddy allocation algorithms aim to minimize external fragmentation by partitioning the memory into blocks fitting as best as possible memory requests from applications. Starting from an initial block with capacity equal to the maximum available memory Ω, blocks are split into a power-of-two number of blocks Γ to ease address computation. Each of the split blocks are buddies of each other and will be coalesced into a single block whenever it is possible when disposing an allocated block.

This mechanism logically organizes the memory into levels, where each level l hosts Γ^l blocks of equal size Ω/Γ^l. Assuming the maximum memory Ω is a constant, it is easy to show that the level containing blocks with the necessary capacity to satisfy a memory request of size s is $\Delta_s \equiv \lceil log_\Gamma(\Omega/s) \rceil$ where $\lceil n \rceil$ is the upper natural number of n.

For efficiency when accessing the free available blocks of a given allocation request, implementations use a multi-level linked-list to manage the free blocks on a given level. Under an allocation request of size s, if the linked list for the level Δ_s is not empty, a block is directly picked from the head of list. If it is empty the mechanism will try to find a higher level with free blocks and the splitting process starts to generate a perfect fit for the requested size.

When a block is deallocated, the algorithms check whether the block can be merged with its buddy blocks. Together with the list of free nodes of a level, the implementation keeps a bitmap with the allocation state of the nodes of a level for quickly coalescing of free buddy nodes. Otherwise the coalescing algorithm would need to traverse the free node list of the level the coalesce is happening at to check that all the buddies blocks are in the list. On levels with a large number of free nodes using a bitmap can significantly increase the performance of the algorithm.

In this work, we provide a specification for the buddy allocator implemented in Zephyr, which uses quad-trees so $\Gamma = 4$. Fig. 1 represents the memory system using the multilevel free linked-list and the multilevel free bitmap with $\Gamma = 2$ to ease the representation. It is composed of 3 level 0 memory blocks numerated from 0 to 2. Only the level 0 block has allocated memory requests, hence the level multilevel free link on level 0 only contains block zero nodes 1 and 2. Allocated nodes 3 and 4 at level 1, and node 7 at level 2 are logical nodes, that is they do not represent physical addresses. Leafs at nodes 5, 6, 8, 9, and 10 do represent physical nodes that have been, or can be, allocated.

2.2. Isabelle/HOL

We use the Isabelle/HOL interactive theorem prover [16] to conduct the specification and verification of the memory management. Isabelle/HOL is a higher order logic theorem prover, using a typed lambda calculus-like functional language for specifications.

Isabelle/HOL includes a specification for simple common types such as naturals (*nat*), integers (*int*) and booleans (*bool*). It also specifies some composed data types like tuples, records, lists and sets that are parametrized with other types. Isabelle provides the interface *datatype* for the creation of user defined types based on type constructors.

Isabelle provides functions on predefined types to access their members or to provide additional operations over them. In the following we describe those functions and types that we use along this work. A tuple is denoted as $(t_1 \times t_2)$, projection functions *fst* and *snd* respectively return elements t_1 and t_2. Lists are defined as a datatype with an empty construct denoted with *NIL* or $[]$, and a concatenation construct denoted with $\#$, where $x\#xs$ adds x to the front of xs. The ith component of a list as is written as $as!i$. Isabelle/HOL

1. www.ntu.securify.es/buddyallocator

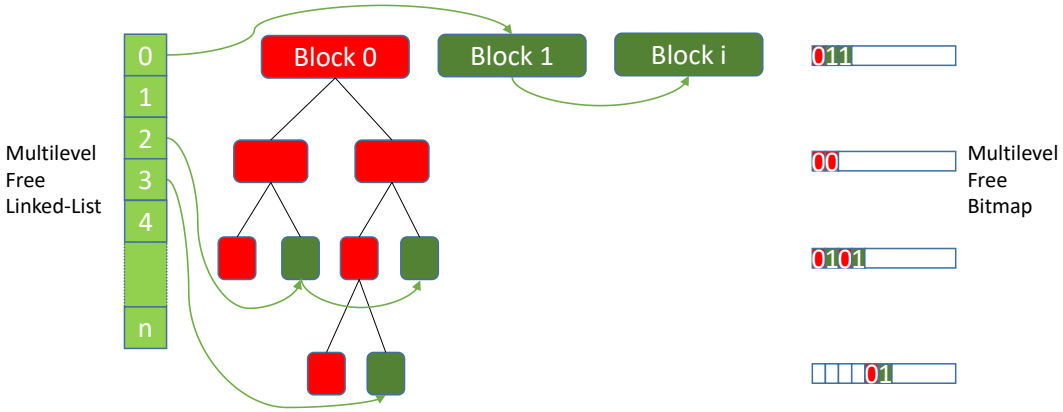

Figure 1. Structures in Buddy Allocation Algorithms

provides functions for definite and indefinite descriptions. Definitive description is represented by $THE\ x.\ P\ x$ and returns the element uniquely described by the predicate P, else it returns and undefined value. Indefinite description is represented by $SOME\ x.P\ x$, selecting a random element from the predicate P that must describe at least one element, else it returns an arbitrary value.

Isabelle/HOL allows users to create non-recursive specifications using the command *definition*, and to create recursive specifications using commands *primrec* and *recursive*.

3. Specification of Buddy Allocation Model

The specification of the buddy memory allocation consists of a model for the necessary data structures to represent the memory layouts, as well as the allocation and disposal operations. This specification follows the algorithm for the buddy memory management in Zephyr OS, which applies a quartering split over blocks.

3.1. State Representation

In the specification, the state models the memory as a set of quad-tree each of them representing a memory pool. At this level of the specification, we assume that applications work with our block entity, so requesting a memory block returns the block itself, which will be used later during the deallocation.

$$\begin{aligned}(set:\ 'a)\ tree\ =\ & Leaf\ (L:\ 'a)\ \mid \\ & Node\ (LL:\ 'a\ tree)\ (LR:\ 'a\ tree) \\ & (RL:\ 'a\ tree)\ (RR:\ 'a\ tree)\end{aligned}$$

We define the structure of a quad-tree inductively. A quad-tree is parametrized by a variable type $'a$ and it has two constructors: *Leaf* and *Node*. A *Leaf* is a terminal node storing values of the parametrized type $'a$, and a *Node* has four (sub)trees that are built recursively. Notations *LL*, *LR*, *RL* and *RR* return the corresponding subtrees of the *Node* tree. Notation **set** represents a function that gathers values of the parametrized type $'a$ from all *Leaf* nodes.

In this specification, we use the tuple (*block_state_type* $\times$ *ID*) to instantiate the polymorphic type *'a* in the quad-tree structure. Type *block_state_type* indicates the usage state of a block and it is constructed using an Isabelle/HOL *datatype*. It consists of two subtypes: *ALLOC* and *FREE*. The former is used to mark memory blocks that have been allocated to applications, whilst the latter is used to mark those unallocated blocks (hence they are free to be assigned to applications requesting memory). The type *ID* is a natural number representing the address identifier of a memory block. Finally, type *BlockTree* represents an instantiated quad-tree in which terminal nodes represent allocated or free memory blocks identified by a natural number. We will indistinctively use the terms of Leaf and memory block along the document. Non-terminal *Node* represents the splitting process of the algorithm.

$$\begin{aligned}block\_state\_type\ &=\ FREE\ \mid\ ALLOC \\ ID\ &=\ nat \\ BlockTree\ &=\ (block\_state\_type\ \times\ ID)\ tree\end{aligned}$$

The allocation and free services are defined as a number of operations over the *quad-tree* data structure representing the memory. These operations manipulate a *BlockTree*, accessing and modifying its structure and the data it stores. Function **level** takes two *BlockTree*, *btree* and *b*, and it returns a natural number *level* that represents the layer number where *b* is located in *btree* from the root node. The level of a node with regards to itself is 0, and if the *blocktree* b does not belongs to $btree$ the function also returns 0. Functions **allocsets** and **freesets** take a *BlockTree btree* and returns a *BlockTree* set *aset* of all the Leaf nodes from *btree* whose *block_state_type* are well *ALLOC* for the function **allocsets**, well *FREE* for the function **freesets**. Function **free_lvl** takes a *Blocktree btree* and a natural number l, and it returns a set with all the free Leaf nodes located at level l in *btree*. We use the notation *idset* to represent the collection

of all used *IDs*. To create a new Leaf node, we pick up as new *ID* any natural number not belonging to *idset*.

3.2. Allocation Model

The allocation takes as input a set of quad-trees representing the available memory pools $bset$, and a natural number s representing the requested size of the memory to allocate. If s is bigger than the maximum size Ω for any memory pool, then the allocation fails and the state is not modified. If s is smaller or equal than Ω, then it calls function **alloc** over $bset$ and the level rlv containing blocks of size bigger or equal to s, given by Δ_s as defined in Section 2.1. Function **alloc** carries out the necessary operations to find a block of size bigger or equal than s, and conducts the necessary modifications on $bset$ as we describe below.

We explain first functions that **alloc** uses to obtain the level on a system with free blocks with capacity to allocate a requested size: **exists_freelevel** and **freesets_maxlevel**. We first provide a description of these functions. Function **exists_freelevel** is a predicate taking as input a set of *BlockTree* $bset$ (the collection of all quad-trees in memory system) and a natural number rlv, and returns true if there exist a *BlockTree* $b \in bsets$ and a level l in b such that l has at least a free node. Function **freesets_maxlevel** has the same inputs, and it returns the maximum level less or equal than *rlv* having at least a free node. Formally:

Definition 1 (Existence of Free Leaf Nodes).

$$exists\_freelevel\ bset\ rlv \triangleq \exists l\ b.\ l \leq rlv \wedge b \in bset \wedge freesets\_level\ b\ l \neq \emptyset$$

Definition 2 (Maximum Level of Free Leaf Nodes).

$$\begin{aligned} freesets\_maxlevel\ bset\ rlv \triangleq\ & THE\ lmax. \\ & lmax \leq rlv \wedge \\ & \exists b \in bset.\ freesets\_level\ b\ lmax \neq \emptyset \wedge \\ & (\forall l \leq rlv.\ \exists b \in bset.\ freesets\_level\ b\ l \neq \emptyset \\ & \quad \longrightarrow l \leq lmax) \end{aligned}$$

During the allocation process, if there is not any free Leaf node available at the best fit level l for the requested size, but there exists a higher level $hl \geq l$ with free blocks, it is necessary to start the splitting process from hl down to l. Function **split** recursively divides $hl - l$ times a Leaf node *b* into a Node tree *btree*. It uses the function **divide** that takes a Leaf node *b* and returns a new non-terminal Node tree *n* with four terminal Leaf nodes. The division operation is always conducted on the leftmost subtree of n marking it as allocated, while the rest are marked as *FREE*. A simple example describing this process is shown in Fig. 2. We define the *split* operation as follows.

Definition 3 (Split a Leaf Node).

$$\begin{aligned} split\ b\ lv \triangleq\ & if\ lv = 0\ then\ b\ else \\ & Node\ (split\ (LL\ divide\ b)\ (lv-1))\ (LR\ divide\ b) \\ & \quad (RL\ divide\ b)\ (RR\ divide\ b) \end{aligned}$$

In addition, the allocation uses functions **set_type**, **replace** and **replace_leaf**. Function *set_type* takes a Leaf node *b* and a target *block_state_type s* as inputs, and returns a leaf *b'* resulting of changing the state of *b* to *s*. Function *replace* takes a *BlockTree btree*, two Leaf nodes *b* and *b'* as inputs, and returns a tree replacing the Leaf node *b* with *b'* in *btree*. Function *replace_leaf* takes a *BlockTree btree*, a Leaf node *b* and a Node tree *btr* as inputs, and returns the tree that replaces the Leaf node *b* with *btr* in *btree*.

To allocate a block of size s as described in Section 2.1, **alloc** firstly uses the function *exists_freelevel* to check whether there is a *BlockTree* in *bset* with free blocks in a level $l \leq$ *rlv*. If there is not, then the allocation process fails and returns original *bset*. Otherwise, the function *freesets_maxlevel* returns the maximum level $lmax$ with $lmax \leq rlv$ containing free blocks. From here there are two options.

(a) there is a *BlockTree* with free nodes at the requested level (hence *lmax* is equal to the requested level *rlv*). In this case the allocation operation selects a *BlockTree btree* from $btree$ such that there are free blocks at level *rlv*. After this, it randomly picks up a *FREE* Leaf node l in level *rlv* from *btree*, and obtains a new $btree'$ by modifying in $btree$ the type of l as *ALLOC* using the functions *set_type* and *replace_btree*. After that, the allocation returns an updated *bset* by replacing the previous *btree* with *btree'*. Note that this execution branch do not adds any additional nodes into the tree structure, and only modifies the type of a terminal node at level *rlv*.

(b) there is not a *BlockTree* with free nodes at the requested level (hence *lmax* is smaller to the requested level *rlv*). Since *lmax* is a lower level than *rlv*, it means that free nodes have bigger size than what is requested. In that case it is necessary to modify the tree splitting a terminal Leaf node l at level *lmax* down to level *rlv* to create a block fitting better the requested size. The allocation operation selects a *BlockTree btree* from $btree$ such that there are free blocks at level *lmax*. Then it randomly selects picks up a *FREE* Leaf node l in level *lmax* from *btree* and splits l into a Node tree *btr* using the function *split*. As explained above, *split "breaks"* the Leaf node *l* into a new *BlockTree* for which *btr* is its root, and where there is block b at depth $rlv - lmax$ from btr that is allocated and its buddies are free. Function **alloc** then updates l with *btr* in *btree* using *replace_leaf* obtaining *btree'*, to finally update *bset* by replacing the previous *btree* with *btree'*. Finally, the operation returns updated Block set and *True*.

Note that **alloc** modifies a free block into an allocated one, so the number of Leafs remain unmodified in the branch (a); or breaks a free block into a *Node* tree to eventually create a number of free blocks and an allocated block as shown in Fig. 2. In both cases the rest of Leafs are from the original set do not change and there is a new allocated block that was not present. To obtain the block **alloc** allocates, it is only necessary to subtract the allocate nodes in the original set of free nodes from the allocate nodes in the resulting set.

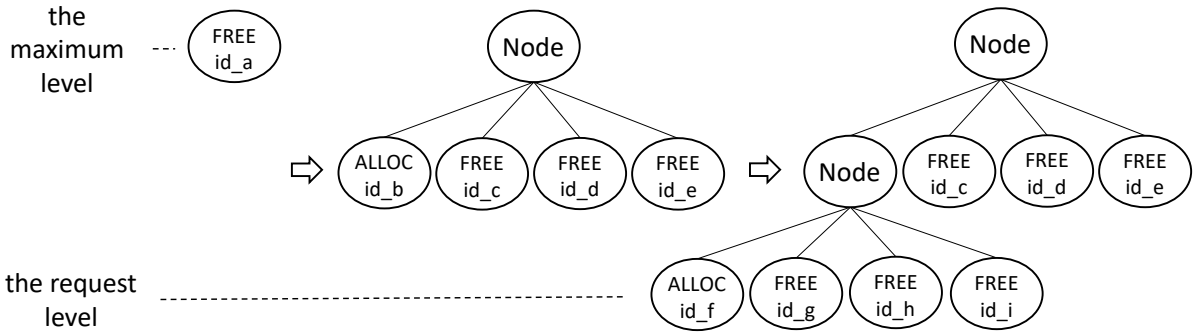

Figure 2. The progress of dividing a free leaf

Definition 4 (Allocation Operation).

$$
\begin{aligned}
&alloc\ bset\ rlv \triangleq \\
&if\ exists\_freelevel\ bset\ rlv\ then \\
&\quad lmax = freesets\_maxlevel\ bset\ rlv \\
&\quad if\ lmax = rlv\ then \\
&\quad\quad btree = SOME\ b.\ b \in bset \wedge \\
&\quad\quad\quad\quad freesets\_level\ b\ rlv \neq \emptyset \\
&\quad\quad l = SOME\ l.\ l \in freesets\_level\ btree\ rlv \\
&\quad\quad btree' = replace\ btree\ l\ (set\_type\ l\ ALLOC) \\
&\quad\quad return\ (bset - \{btree\} \cup \{btree'\},\ True) \\
&\quad else \\
&\quad\quad btree = SOME\ b.\ b \in bset \wedge \\
&\quad\quad\quad\quad freesets\_level\ b\ lmax \neq \emptyset \\
&\quad\quad l = SOME\ l.\ l \in freesets\_level\ btree\ lmax \\
&\quad\quad btr' = split\ l\ (rlv - lmax) \\
&\quad\quad btree' = replace\_leaf\ btree\ l\ btr' \\
&\quad\quad return\ (bset - \{btree\} \cup \{btree'\},\ True) \\
&else\ return\ (bset,\ False)
\end{aligned}
$$

3.3. Deallocation Model

The deallocation process takes a block b in a level l, and sets the state of b to *FREE*. During the deallocation process, if the state of all the buddies of b is already $free$ and l is not the root node, the buddy nodes are merged to avoid fragmentation. In the merging process, b' parent Node is transformed into a Leaf node at level $l-1$ that is set as free, causing all the buddies to be removed from the quad-tree. Hence they are not at level l anymore. The merging process is shown in Fig. 3.

Function **free** takes as input a set of *BlockTree* $bset$ and a block to dispose *b*. It firstly checks whether there is a $btree \in bset$ for which *b* belongs to and that its state is *ALLOC*. If the conditions are not met, the procedure fails returning the original *bset*. If they are, the procedure picks up the tree in *bset* as *btree* to which *b* belongs to, which must be unique. After this, *btree* is modified into *btree'* where the type of *b* is set to *FREE* using the functions *set_type* and *replace*. After that, the new tree is coalesced using the function *merge* and *bset* is updated replacing *btree* with the new memory pool. The definition of deallocation operation is as follows.

Definition 5 (Deallocation Operation).

$$
\begin{aligned}
&free\ bset\ b \triangleq \\
&if\ \exists btree \in bset.\ b \in set\ btree\ then \\
&\quad if\ fst\ b = FREE\ then \\
&\quad\quad return\ (bset,\ False) \\
&\quad else \\
&\quad\quad btree = THE\ t.\ t \in bset \wedge b \in set\ t \\
&\quad\quad btree' = replace\ btree\ b\ (set\_type\ b\ FREE) \\
&\quad\quad btree'' = merge\ btree' \\
&\quad\quad return\ (bset - \{btree\} \cup \{btree''\},\ True) \\
&else\ return\ (bset,\ False)
\end{aligned}
$$

At this point, we have finished the specification of the allocation and disposal operations for a buddy memory allocator. The next section tackles the verification of functional and security properties over the model for the allocation and deallocation operations.

4. Verification

In this section we will show that the formal specification introduced in the previous section preserves a set of properties guaranteeing its functional correctness in Section 4.1. As a security property, we also prove that the specification preserves integrity of the memory structures in Section 4.2 and Section 4.3. All the lemmas and theorems present in this section have been proven in the Isabelle/HOL theorem prover. We omit the proofs due to lack of space, but the interested reader can access the mechanize proofs in the project website.

4.1. Functional correctness

We first formulate and prove the desired properties for the allocation and deallocation services with regards to

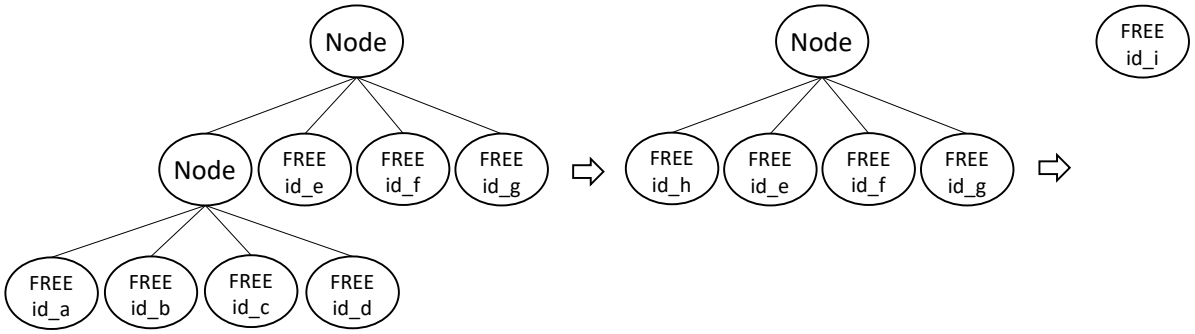

Figure 3. The progress of merging all free memory blocks

preservation of the memory layouts. Note that models for **alloc** and **free** return a tuple $(BlockTree \times Bool)$, where the first element is the new memory layouts, and the second one indicates the success or failure of the function. For simplicity on the presentation of the lemmas we consider that the models only return the new state of the memory layouts.

4.1.1. Allocation. We first show that the allocation service does not change the state when none of the memory pools in a set b_set have any free block big enough to allocate a request of size rs.

Lemma 1 (Allocation Failure).

$$\neg\ exists\_freelevel\ b\_set\ \Delta_{rs} \longrightarrow (alloc\ b\_set\ \Delta_{rs}) = b\_set$$

As described in Section 2.1 to allocate a memory block of size rs, there must be a free block on a level lower or equal than Δ_{rs}. So under the assumption that there does not exist such a block, the allocation procedure leaves the state unchanged.

Then we show that if there is a memory pool with a block big enough to allocate the requested size rs, then **alloc** adds an allocated block at level Δ_{rs}. We first split the lemma in two cases. The first case to the case in which level Δ_{rs} has free blocks, and hence the free block is not split but modified, it is shown in lemma 2. The second case is when Δ_{rs} does not have free blocks, and hence a free block in a level $l < \Delta_{rs}$ is split till reach level Δ_{rs} and it is shown in lemma lemma 3.

Lemma 2 (Allocsets for Direct Allocation).

$$\begin{aligned}&exists\_freelevel\ b\_set\ \Delta_{rs} \wedge\\ &freesets\_maxlevel\ b\_set\ \Delta_{rs} = \Delta_{rs} \longrightarrow\\ &\exists l.\ l \notin allocsets\ b\_set \wedge l \in allocsets\ (alloc\ b\_set\ \Delta_{rs}) \wedge\\ &\quad allocsets\ (alloc\ b\_set\ \Delta_{rs}) = allocsets\ b\_set \cup \{l\} \wedge\\ &\quad level\ (alloc\ b\_set\ \Delta_{rs})\ l = \Delta_{rs}\end{aligned}$$

Lemma 3 (Allocsets for Indirect Allocation).

$$\begin{aligned}&exists\_freelevel\ b\_set\ \Delta_{rs} \wedge\\ &freesets\_maxlevel\ b\_set\ \Delta_{rs} \neq \Delta_{rs} \longrightarrow\\ &\exists l.\ l \notin allocsets\ b\_set \wedge l \in allocsets\ (alloc\ b\_set\ \Delta_{rs}) \wedge\\ &\quad allocsets\ (alloc\ b\_set\ \Delta_{rs}) = allocsets\ b\_set \cup \{l\} \wedge\\ &\quad level\ (alloc\ b\_set\ \Delta_{rs})\ l = \Delta_{rs}\end{aligned}$$

From lemmas 2 and 3 we show the final lemma 4 on the functional correctness of **alloc** with regards to allocated memory blocks.

Lemma 4 (Allocsets for Allocation).

$$\begin{aligned}&exists\_freelevel\ b\_set\ \Delta_{rs} \longrightarrow\\ &\exists l.\ l \notin allocsets\ b\_set \wedge l \in allocsets\ (alloc\ b\_set\ \Delta_{rs}) \wedge\\ &\quad allocsets\ (alloc\ b\_set\ \Delta_{rs}) = allocsets\ b\_set \cup \{l\} \wedge\\ &\quad level(alloc\ b\_set\ \Delta_{rs})\ l = \Delta_{rs}\end{aligned}$$

For the allocation algorithm, we also show in lemma 5 that any free block in the initial set of memory pools b_set, different than the block being allocated, remains free after allocation.

Lemma 5 (Freesets for Allocation).

$$\begin{aligned}&exists\_freelevel\ b\_set\ \Delta_{rs} \wedge\\ &freesets\_maxlevel\ b\_set\ \Delta_{rs} \neq \Delta_{rs} \longrightarrow\\ &\exists l.\ l \notin allocsets\ b\_set \wedge l \in allocsets\ (alloc\ b\_set\ \Delta_{rs}) \wedge\\ &\quad \forall l'.\ l' \neq l \wedge l' \in freesets\ b\_set \longrightarrow\\ &\qquad l' \in freesets\ (alloc\ b\_set\ \Delta_{rs})\end{aligned}$$

4.1.2. Deallocation. For deallocation, we first show in lemmas 6 and 7 that the deallocation service indeed does not change the memory layouts when the block to deallocate does not belong to any memory pool in the memory layouts, or when it exists but it has not been allocated.

Lemma 6 (Deallocation Failure 1).

$$\nexists btree \in b\_set.\ b \in set\ btree \longrightarrow (free\ b\_set\ b) = b\_set$$

Lemma 7 (Deallocation Failure 2).

$$\exists btree \in b\_set.\ b \in set\ btree \wedge fst\ b = FREE \longrightarrow (free\ b\_set\ b) = b\_set$$

Then we show in Lemma 8 that the deallocation process carries out a correct transition during a deallocation success process. It describes that the Leaf node, which is to be released, dose not belong to allocated set any more.

Lemma 8 (Allocsets for Deallocation).

$$\exists btree \in b\_set.\ b \in set\ btree \wedge fst\ b \neq FREE \longrightarrow$$
$$b \notin allocsets\ (free\ b\_set\ b)\ \wedge$$
$$allocsets\ b\_set = allocsets\ (free\ b\_set\ b) \cup \{b\}$$

With regards to free blocks, the deallocation process may remove free memory blocks during the coercing procedure. However, we have to ensure that any free memory block after deallocating, different than the block resulting of the deallocation process, has to be also free in the initial set of free blocks b_set. We show this in Lemma 9.

Lemma 9 (Freesets for Deallocation).

$$\exists btree \in b\_set.\ b \in set\ btree \wedge fst\ b \neq FREE \longrightarrow$$
$$(1)\exists b'\ lvl.\ b' \in free\_lvl\ (free\ b\_set\ b)\ lvl\ \wedge$$
$$b' \notin free\_lvl\ b\_set\ lvl\ \wedge$$
$$(2)\forall b''.\ b'' \neq b' \wedge b \in freesets\ (free\ b\_set\ b) \longrightarrow$$
$$b'' \in freesets\ b\_set$$

This property first says that there is a free block b' in the new set of memory pools that does not belong to the original set of memory pools (1). This is the free block resulting of deallocating the block. Then, we state that any other free block different than b' belonging to the new set it also belongs to the original set (2).

4.1.3. Correct Merging. The buddy memory allocation algorithm may reduce the fragmentation by invoking merge function during the process of deallocation to merge buddy blocks that are free, as shown in Fig. 3. Previous lemmas do not show that the merging procedure is correct, so we show here that the merging process is correctly carried out. This can be reduced to show that the memory layouts preserve an invariant saying that buddy memory blocks are not all of them free. To achieve this, we first define the notion of all free buddy memory block using the definition below, where function **child** gives us a set of all immediate child nodes of a *Tree* and **leaf** is a predicate checking whether a node is terminal or not:

Definition 6 (Four Free Leaves Belong to the Same Node).

$$is\_FFL\ btree \triangleq \forall chtree \in child\ btree.\ leaf\ chtree$$
$$\wedge\ fst\ chtree = FREE$$

Operating system initialize the whole memory into a series of free blocks in different sizes, and these memory blocks are in the form of root nodes which are not split at the very beginning. This initialization state satisfies not-existence of *FFL* trees. Then according to Lemma 10 and 11, any state after initialization satisfies non-existence of *FFL* trees no matter operating system performs allocation or deallocation operations. Therefore, we can make sure the whole memory system preserves this property. We have this theorem as follows. Furthermore, it proves that the buddy allocation specification reduces the fragmentation by merge operation. We then show in Lemmas 10 and 11 that after the allocation and deallocation procedures there are no buddy groups where all the members are free.

Lemma 10 (Non-existence of FFL during Allocation).

$$\forall b \in b\_set.\ \neg\ is\_FFL\ b$$
$$\longrightarrow \forall b \in (alloc\ b\_set\ \Delta_{rs}).\ \neg\ is\_FFL\ b$$

Lemma 11 (Non-existence of FFL during Deallocation).

$$\forall b \in b\_set.\ \neg\ is\_FFL\ b$$
$$\longrightarrow \forall b \in (free\ b\_set\ b).\ \neg\ is\_FFL\ b$$

4.1.4. Memory Isolation. Memory isolation ensures non-overlapping of the allocated address space. In Section 3.1 we show that the memory model uses a natural number as a block identifier *ID*. The *ID* helps us to uniquely identify each physical memory block in a memory pool.

Definition 7 introduces a judgment stating whether two Leaf nodes have the same *ID*. This definition checks that for any memory pool b in memory layouts b_set, and any memory block l in b, i.e. a terminal node of b, then there is not any other l', different than l, and other memory pool b' in b_set where the *ID* of l is equal the *ID* of l'.

Definition 7 (Different IDs).

$$is\_different\ b\_set \triangleq$$
$$\forall b \in b\_set.\ \forall l \in set\ b.\ \nexists l'\ b'.\ l' \in set\ b' \wedge b' \in b\_set\ \wedge$$
$$l' \neq l \wedge get\_id\ l' = get\_id\ l$$

Lemmas 12 and 13 shows that uniqueness of IDs is an invariant to the allocation and deallocation procedures.

Lemma 12 (Different IDs during Allocation).

$$is\_different\ b\_set \longrightarrow$$
$$is\_different\ (alloc\ b\_set\ \Delta_{rs})$$

Lemma 13 (Different IDs during Deallocation).

$$is\_different\ b\_set \longrightarrow$$
$$is\_different\ (free\ b\_set\ b)$$

Considering that a block ID allows us to uniquely identify a memory block, the lemma on uniqueness of blocks and Lemma 4 allows to infer non-overlapping of allocation of memory blocks. That is, that two different allocations do not assign the same memory block.

Note that although our level of abstraction does not consider memory addresses, it would be easy to refine this model to include memory addresses and identify the domain of addresses of a memory block as a function in terms of the location of the memory block in the tree.

4.2. A Security Model

Integrity is the assurance that information is trustworthy and accurate. To achieve this, data must not be changed in transit. In this subsection, we prove integrity of the data allocated by a buddy allocation specification. For this purpose, we follow the work from [20] to firstly design a security model, which consists of a nondeterministic state machine, and then to specify the integrity property and prove that it is an invariant on the memory allocator.

4.2.1. Memory State Machine. We define a memory state machine as tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{E}, \varphi, s_0 \rangle$. $\mathcal{S}$ represents the state space, $s_0 \in \mathcal{S}$ is the initial machine state, and $\mathcal{E}$ is the set of event labels. The state-transition function is characterized by φ, of type $\varphi : \mathcal{E} \to \mathbb{P}(\mathcal{S} \times \mathcal{S})$, where $\mathbb{P}(n)$ is the powerset of the set n.

Based on the state machine above, we introduce some auxiliary functions: function **execution(s, es)** takes a state *s* and a sequence of events *es*, then returns the set of final states; function **reachable(s)** (denoted as $\mathcal{R}(s)$) checks the reachability of a state *s* from the initialized state s_0 by the *execution* function: $\exists$*es. s* $\in$ *execution(s_0, es)*.

Next, we add the concept of domains, also referred as partitions, to represent the entities that execute the state-transition function. In addition, we introduce partition scheduling as a domain **scheduler**. Therefore, the domains ($\mathcal{D}$) in $\mathcal{M}$ are the configured partitions ($\mathcal{P}$) and the scheduler ($\mathbb{S}$), $\mathcal{D}$ = $\mathcal{P} \cup \{\mathbb{S}\}$.

4.2.2. Integrity Definition. We use the notion of integrity from [21], which provides a formalism for the specification of security policies. A domain *u* is non-interfering with domain *v* if no action performed by *u* can influence the subsequence outputs seen by *v*. We define the notion of integrity in our security model using the concepts of state equivalence and interfering.

Firstly, state equivalence is denoted as $\sigma_1 \sim \sigma_2$ is a relation in $(\mathbb{S} \times \mathbb{S})$. State equivalence of σ_1 and σ_2 on a domain d is denoted as $\sigma_1 \overset{d}{\sim} \sigma_2$.

Interference is a relation in $(\mathcal{D} \times \mathcal{D})$ and is denoted as $a \rightsquigarrow b$, representing that domain a interfere with domain b. We use $a \not\rightsquigarrow b$ to express that a domain a does not interfere domain b.

Finally, integrity of an event e is defined as:

Definition 8 (Integrity).

$$integrity(e) \triangleq \forall d\ s\ s'.$$

$$\mathcal{R}(s) \wedge dom(s,\ e) \not\rightsquigarrow d \wedge (s,\ s') \in \varphi(e) \longrightarrow (s \overset{d}{\sim} s')$$

$dom(\sigma, e)$ is a function from pairs $(\mathcal{S} \times \mathcal{E})$ to $\mathcal{D}$, specifying the domain executing an event e on a state σ. The notion of integrity states that for any domain d, and a reachable state s from the initial state s_0, if the domain executing the event e in s does not interfere with d, then for any state s' that e can transit to from s, s is equivalent to s' for the domain d. That is, if the domain executing e is not able to interfere with d, then the execution of e does not modify the information that domain d observes.

4.2.3. Security Model. We define the security model as:

Definition 9 (Security Model).

The security model is a tuple

$$\mathcal{S}\_\mathcal{M} = \langle \mathcal{M}, \mathcal{D}, dom, \rightsquigarrow, \sim \rangle$$

assuming the following assumptions.

1) $\forall$d $\in \mathcal{D}$. $\mathbb{S} \rightsquigarrow$ d
2) $\forall$d $\in \mathcal{D}$. d $\rightsquigarrow \mathbb{S} \longrightarrow$ d = $\mathbb{S}$
3) $\forall$s t e. $s \overset{\mathbb{S}}{\sim} t \longrightarrow$ dom(s, e) = dom(t, e)
4) $\forall$s e. $\mathcal{R}(s) \longrightarrow \exists$s'. (s, s') $\in \varphi(e)$
5) $\forall$e. integrity(e)

The security model is tuple composed of a state machine $\mathcal{M}$, a set of execution domains $\mathcal{D}$, a function dom, and the relations for interference between domains and equivalence between states. The assumptions of the model represent that: the scheduler interfere with any execution domain (1); an execution domain d cannot interfere with the scheduler, unless d is the scheduler itself (2); two states s and t are equivalent on the scheduler only if the execution domain of s and t is the same on the execution of any event e (3); any event e be defined on any reachable state (4); all the events must preserve the integrity relation (5).

This security model constructs a sequential model for event-based specifications which ensures the preservation of the integrity property for any possible execution trace on $\mathcal{M}$. In the next section, we instantiate the security model with our buddy allocation specification.

4.3. Instantiation and Security Proofs

In this part, we construct an event specification by adding interfaces to the buddy allocation specification introduced in Section 3. We instantiate the security model specifying the function for execution domains, and the relations of domain interference and state equivalence. We also prove that the wrapped buddy allocation events satisfy the integrity property.

4.3.1. Instantiation. We first define the notion of state used in the memory state machine $\mathcal{M}$ as $\Sigma = \langle \mathcal{MP}, \xi, \mathcal{A} \rangle$, where $\mathcal{MP}$ is a set of memory pools, i.e., the memory state; ξ is the scheduled domain used to instantiate the function dom in the security model; and $\mathcal{A}$ is a function from execution domains to memory blocks identifiers, which holds the memory blocks that have been allocated to each execution domain. We use $\mathcal{A}$ to instantiate the equivalence function among domains.

The events $\mathcal{E}$ in the security model is a set composed of an allocation, deallocation, and scheduler event. The interfaces for the allocation and deallocation use the **alloc** and **free** functions defined in Section 3 to update the internal information of the memory and the blocks identifier used by the domain executing the event. The Scheduler is an event function that sets ξ non-deterministically with any execution domain. Note that we constrain the execution of allocation and free functions to be carried out by an execution domain

that is not the scheduler, and the execution of the scheduler by the scheduler domain. The transition relation is defined as:

Definition 10 (Allocate Memory Interface).

$$\begin{aligned}&\forall \Delta_{rs}.\ \varphi\ allocate\_memory\ \Sigma \triangleq \\ &\ if\ snd\ (alloc\ (\mathcal{MS}\ \Sigma)\ \Delta_{rs}) = True \wedge \xi\ \Sigma \neq \mathbb{S} \\ &\ then\ \langle (alloc\ (\mathcal{MS}\ \Sigma)\ \Delta_{rs}),\ \xi, \\ &\qquad \mathcal{A}[\xi := \mathcal{A}\ \xi \cup \{allocid\ (alloc\ (\mathcal{MS}\ \Sigma)\ \Delta_{rs})\}]\rangle \\ &\ else\ \Sigma\end{aligned}$$

Definition 11 (Deallocate Memory Interface).

$$\begin{aligned}&\forall b.\ \varphi\ free\_memory\ \Sigma \triangleq \\ &\ if\ snd\ (free\ (\mathcal{MS}\ \Sigma)\ b) = True \wedge \xi\ \Sigma \neq \mathbb{S} \\ &\ then\ \langle (free\ (\mathcal{MS}\ \Sigma)\ b),\ \xi\ \Sigma, \\ &\qquad \mathcal{A}[\xi := \mathcal{A}\ \xi - \{allocid\ (free\ (\mathcal{MS}\ \Sigma)\ b)\}]\rangle \\ &\ else\ \Sigma\end{aligned}$$

Definition 12 (Scheduler).

$$\begin{aligned}&\varphi\ scheduler\ \Sigma \triangleq \\ &\qquad if\ \xi\ \Sigma = \mathbb{S}\ then \\ &\qquad\quad \langle \mathcal{MS}\ \Sigma,\ SOME\ p.\ p \in \mathcal{D},\ \mathcal{A}\ \Sigma\rangle \\ &\qquad else\ \Sigma\end{aligned}$$

Where $\mathcal{MS}\ \Sigma$, $\xi\ \Sigma$, and $\mathcal{A}\ \Sigma$ are respectively the projections of $\mathcal{MS}$, ξ, $\mathcal{A}$ in Σ. The notation $f[i := v]$ sets the image of i to v in f, and for any $j \neq i$, $f\ j = f[i := v]\ j$. Function **allocid** returns the *ID* of the block allocated or freed by allocation and deallocation events.

Seeking integrity of the memory between domains, two different execution domains cannot interfere each with the other. On the other hand, since the scheduler $\mathbb{S}$ changes the current execution domain, it is necessary to set that it can interfere with any domain. We instantiate the interference relation as:

Definition 13 (Instantiation of $\rightsquigarrow$ by Domains).

$$d1 \rightsquigarrow d2 \triangleq (d1 = d2) \vee d1 = \mathbb{S}$$

The state equivalence for an execution domain different than the scheduler, hence a partition, must consider the set of block identifiers that have been allocated to it, given by $\mathcal{A}\ \Sigma\ d$ for the domain d in the state Σ. We then say that two states are equivalents in a partition d if the set of a allocated blocks for d is the same in both states. The scheduler must check that the projection of ξ is equal in the relation, that is the scheduler did not change. the Formally:

Definition 14 (Instantiation of $\sim$ by States and Domains).

$$\begin{aligned}s \overset{d}{\sim} t \triangleq\ & (d = \mathbb{S} \longrightarrow \xi\ s = \xi\ t) \wedge \\ & (d \neq \mathbb{S} \longrightarrow \mathcal{A}\ s\ d = \mathcal{A}\ t\ d)\end{aligned}$$

4.3.2. Security Proofs. To prove the instantiated model is a security model, we have to prove the assumptions of the security model $\mathcal{S}\_\mathcal{M}$. The first two assumptions are preserved by the interfering $\rightsquigarrow$ definition. The third assumption is preserved by the definition of equivalence for the *scheduler* domain. The fourth assumption on the reachability is preserved by the relation φ together with the following lemma:

To prove the last assumption *integrity(E)*, we prove the definition of integrity for each event.

Lemma 14 (Integrity of alloc_memory).

$$\begin{aligned}\forall d\ s\ s'\ \Delta_{rs}.\ \mathcal{R}(s) \wedge (\xi\ s) \not\rightsquigarrow d \wedge \\ s' = alloc\_memory\ s\ \Delta_{rs} \longrightarrow s \overset{d}{\sim} s'\end{aligned}$$

Lemma 15 (Integrity of free_memory).

$$\begin{aligned}\forall d\ s\ s'\ b.\ \mathcal{R}(s) \wedge (\xi\ s) \not\rightsquigarrow d \wedge \\ s' = free\_memory\ s\ b \longrightarrow s \overset{d}{\sim} s'\end{aligned}$$

Lemma 16 (Integrity of scheduler).

$$\forall d\ s\ s'.\ (\xi\ s) \not\rightsquigarrow d \longrightarrow s \overset{d}{\sim} s'$$

Theorem 1. $\forall$E $\in$ {alloc_memory, free_memory, scheduler}. integrity(E)

In the end, we prove that the event specification based on the buddy allocation specification satisfies the integrity property and is really a security model. According to the instantiation of $\rightsquigarrow$, one partition domain (the *scheduler* domain can interfere any partition domain) cannot influence the memory address spaces of other partition domains. The integrity of memory address spaces ensures that the allocation and deallocation algorithms are trustworthy and accurate.

5. Conclusions and Future Work

In this paper, we have presented a specification for buddy allocation algorithms with two services for the allocation and disposal of memory blocks. Then we specify and prove a the necessary properties for functional correctness of the specification regarding the blocks allocated and disposed and the structure of the memory layout. After that, we introduce integrity property for the security assurance of this specification. To achieve this goal, we design a state-machine security model with the concepts of interfering and state equivalence. Through instantiating this security model by adding interface functions to the buddy allocation specification, we finally prove that the instantiation security model satisfies integrity property.

As future work, we plan to extend the current model with real address spaces, multilevel free linked-list and multilevel free bitmap to this buddy allocation specification to provide a design-level specification. Furthermore, we are plan to extend this work to an implementation-level specification using a modelling language able to capture the semantic

of system programming languages. In these processes, refinement technology is necessary to guarantee between two adjacent level specifications.

References

[1] J. M. Rushby and F. W. von Henke, "Formal verification of algorithms for critical systems," *IEEE Trans. Software Eng.*, vol. 19, no. 1, pp. 13–23, 1993. [Online]. Available: https://doi.org/10.1109/32.210304

[2] A. T. Luu, M. C. Zheng, and T. T. Quan, "Modeling and verification of safety critical systems: A case study on pacemaker," in *Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010*. IEEE Computer Society, 2010, pp. 23–32. [Online]. Available: https://doi.org/10.1109/SSIRI.2010.28

[3] L. Higham, J. Kawash, and N. Verwaal, "Defining and comparing memory consistency models," 1997.

[4] S. Blazy and X. Leroy, "Formal verification of a memory model for *C*-like imperative languages," in *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, ser. Lecture Notes in Computer Science, K. Lau and R. Banach, Eds., vol. 3785. Springer, 2005, pp. 280–299. [Online]. Available: https://doi.org/10.1007/11576280_20

[5] W. Mansky, D. Garbuzov, and S. Zdancewic, "An axiomatic specification for sequential memory models," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9207. Springer, 2015, pp. 413–428. [Online]. Available: https://doi.org/10.1007/978-3-319-21668-3_24

[6] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[7] N. Marti, R. Affeldt, and A. Yonezawa, "Formal verification of the heap manager of an operating system using separation logic," in *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Springer, 2006, pp. 400–419. [Online]. Available: https://doi.org/10.1007/11901433_22

[8] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. [Online]. Available: https://doi.org/10.1109/LICS.2002.1029817

[9] F. Mangano, S. Duquennoy, and N. Kosmatov, "Formal verification of a memory allocation module of contiki with frama-c: A case study," in *Risks and Security of Internet and Systems - 11th International Conference, CRiSIS 2016, Roscoff, France, September 5-7, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, F. Cuppens, N. Cuppens, J. Lanet, and A. Legay, Eds., vol. 10158. Springer, 2016, pp. 114–120. [Online]. Available: https://doi.org/10.1007/978-3-319-54876-0_9

[10] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - A lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE Conference on Local Computer Networks (LCN 2004), 16-18 November 2004, Tampa, FL, USA, Proceedings*. IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: https://doi.org/10.1109/LCN.2004.38

[11] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0326-7

[12] A. Blanchard, "Introduction to c program proof using frama-c and its wp plugin," 2017.

[13] A. Sahebolamri, S. D. Constable, and S. J. Chapin, "A formally verified heap allocator," in *Electrical Engineering and Computer Science - Technical Reports, 182*, 2018. [Online]. Available: https://surface.syr.edu/eecs_techreports/182

[14] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, M. Hofmann and M. Felleisen, Eds. ACM, 2007, pp. 97–108. [Online]. Available: https://doi.org/10.1145/1190216.1190234

[15] D. Greenaway, J. Andronick, and G. Klein, "Bridging the gap: Automatic verified abstraction of C," in *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, ser. Lecture Notes in Computer Science, L. Beringer and A. P. Felty, Eds., vol. 7406. Springer, 2012, pp. 99–115. [Online]. Available: https://doi.org/10.1007/978-3-642-32347-8_8

[16] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[17] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, pp. 623–624, 1965. [Online]. Available: https://doi.org/10.1145/365628.365655

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[19] The zephyr project. [Online]. Available: https://www.zephyrproject.org/

[20] Y. Zhao, D. Sanán, F. Zhang, and Y. Liu, "Refinement-based specification and security analysis of separation kernels," *IEEE Trans. Dependable Sec. Comput.*, vol. 16, no. 1, pp. 127–141, 2019. [Online]. Available: https://doi.org/10.1109/TDSC.2017.2672983

[21] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. [Online]. Available: https://doi.org/10.1109/SP.1982.10014