Towards Formal Verification of Memory Properties using Separation logic

Nicolas Marti † Reynald Affeldt ‡ Akinori Yonezawa †‡

†Department of Computer Science, University of Tokyo

‡Research Center for Information Security (RCIS), National Institute of Advanced Industrial Science and Technology (AIST)

With the recent dissemination of embedded systems, it has become important to verifiy low-level software such as specialized operating systems. However, such verifications are notoriously made difficult by complex memory management operations such as pointer arithmetic. As a first step towards the implementation of a reusable verification tool, we show how one can formally verify an important property of memory management for the Topsy operating system, an existing operating system for active-network cards. Our approach consists in verifying individually each function involved in memory management using a formal encoding of separation logic in the Coq proof assistant. At the time of this writing, we are still in the process of verifying memory management for Topsy, but we already found issues in the implementation.

1 Introduction

With the recent dissemination of embedded systems, it has become important to verify low-level software such as specialized operating systems. However, such verifications are notoriously made difficult by complex memory management operations such as pointer arithmetic. The fact is that software developers lack tools for such verifications, as opposed to, say, successful applications of model-checking in the hardware industry.

Our goal is to provide a verification tool for low-level software. We have already laid down the foundations of such a tool by implementing in the Coq proof assistant the *separation logic* [1] (a Hoare logic-like language for verification of C-like programming languages). Despite the tediousness of proof assistant-based verification, we believe that the Coq proof assistant provides an effective way to verify low-level software, as exemplified, among other experiments, by the recent verification of the Java Card System at the Common Criteria EAL7 level (Trusted Logic, press release of November 18, 2003).

In this paper, we use separation logic to specify an important property of memory management for the Topsy operating system [2], an existing operating system for active-network cards. The property in question is *memory isolation*, which holds when user-level tasks cannot access kernel-level memory. Arguably, memory isolation is important because it sustained many security properties of operating systems. As usual in verification, we found it difficult to formalize this property, and this is certainly the main contribution of this paper to show a possible formalization approach, starting from the original source-code to encoding into a proof assistant.

The rest of this paper is organized as follows. In Sect. 2, we recall the basics of x86 memory models and define memory isolation for these processors. In Sect. 3, we give an overview of the source code of Topsy and explain our approach to verify memory isolation. In Sect. 4, we formally specify the functions involved in the memory or thread management, focusing on the management of privilege levels. In Sect. 5, we illustrate how we can mechanically verify above specifications using the Coq proof assistant. In Sect. 6, we conclude, and review related and future work.

2 Memory Isolation for x86 Processors

Intuitively, memory isolation is the property that user-level threads cannot access kernel-level memory. Although the distinction between user-level and kernel-level threads is common in operating systems, the way they access memory always relies on the underlying hardware mechanisms provided by the processor.

Programs for x86 processors access memory in terms of *segments*, i.e., fixed-size arrays of memory. This segment-based access to memory is enforced by the use of *segment registers* (so-called cs, ss, ds, es, fs, gs registers) that are used to access physical memory through the use of *logical addresses*. Logical addresses consist of a segment and an offset, and are translated to physical addresses by the hardware based on a conversion table loaded in physical memory and called the *global descriptor table* (GDT) and pointed to by a special-purpose register called gdtr[1].

x86 processors also distinguish between *privilege levels*, i.e., a flag that indicates whether or not the processor may accomplish some operations. The *current privilege level* of the code is stored in the cs register and is 0 for kernel-level threads and 3 for user-level threads. Also, each segment descriptor stored in the GDT contains a flag called *descriptor level privilege* that indicates the privilege of the segment.

The memory isolation property for x86 processors can be defined precisely as follows: An operating system based on a x86 segmentation memory model ensures the *memory isolation* property if all the kernel related memory areas are only covered by kernel privilege segments, and if all user tasks always run in user level privilege.

3 Memory Isolation for Topsy

In this section, we first give an overview of the implementation of Topsy (version 2) and then we explain our approach to verify memory isolation.

[1] More exactly, the GDT is partially mapped to special-purpose registers (a.k.a. segment selectors) to speed-up the translation process.

3.1 Topsy Implementation Overview

The implementation of Topsy is standard enough to be readable by a programmer with little experience. In the following, we comment on the contents of files related to memory and thread management (around 2,000 lines, one half being assembly code). The whole source code of Topsy (around 18,000 lines, available online [3]) is depicted in Fig. 1.

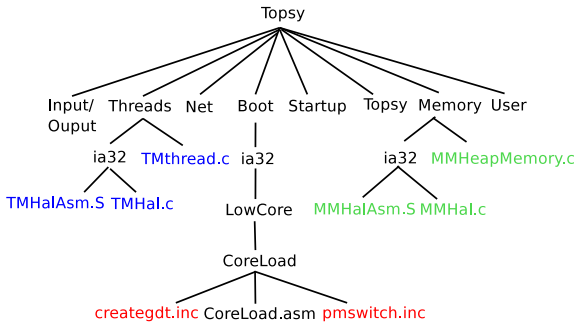

Figure 1: Topsy Source Code

3.1.1 Memory Model

Topsy implements two multi-threaded tasks for the kernel and the user. Accordingly, Topsy splits the memory into kernel-privilege and user-privilege segments by dividing the whole memory into four parts (each part being further divided into two segments). There are three kernel-privilege parts, that respectively contain (1) the GDT and other kernel data structures, (2) the kernel code, data and stack, and (3) a map to input/output ports; there is one user-privilege part that contains the user code, data and stack.

The memory model described above is set by the boot loader. The latter builds the GDT before switching the processor into segmented mode (files `creatgdt.inc` and `pm_switch.inc` in directory `Boot/ia32/LowCore/CoreLoad`). The user-privilege segment is set during kernel initialization (files `MMHal.c` and `MMHalAsm.s` in directory `Memory/ia32`).

3.1.2 Thread Manager

Thread management amounts to thread creation/destruction and context switching. In Topsy, these operations are implemented by the thread manager module (directory `Threads`).

Upon thread creation, the kernel allocates a *thread descriptor*, a data structure containing in particular the privilege of the thread. This data structure is built in the function `threadBuild` that initializes among others the privilege of the thread context (files `TMThread.c` and `ia32/TMHal.c`). Context switching is implemented by two functions to restore and save the thread context. The allocation of the processor to a thread is done by restoring the image of the processor state (function `restoreContext` in file `ia32/TMHalAsm.S`). When the thread has to release the processor, the kernel saves the thread context into the thread descriptor (function `_INTHandler` in file `ia32/TMHal.c`).

3.1.3 Memory Manager

The kernel initialization and the thread manager both rely on an internal memory allocation/deallocation mechanism called the *heap manager* (file `Memory/MMHeapMemory.c`).

The heap manager exploits a portion of memory called the *heap* that is reserved by the kernel. The function `hmInit` initializes the heap. The function `hmAlloc` is implemented by creating blocks of memory in the heap. The function `hmFree` implements deallocation by replacing “allocated” blocks in the heap with “free” blocks.

3.2 Memory Isolation for Topsy

In this section, we explain and discuss how we verify memory isolation for Topsy.

Our approach is to specify for each function involved in the memory or thread management of Topsy its intended behavior regarding privilege handling. The extraction from source code of involved functions and the specification of their intended behaviors is based on a careful reading of the implementation we saw above. For the sake of explanation, let us represent graphically the control flow in the kernel (Fig. 2). Following this control flow, we informally specify the intended behavior of each phase relevant to memory isolation:

- **Boot loader and kernel initialization** After kernel initialization, the GDT implements the memory model of Topsy and the processor is in segmented mode.
- **Heap manager** Under the hypothesis of a correct initialization of the heap, newly allocated blocks do not override previously allocated blocks, and only free blocks are marked as such.
- **Thread manager** Thread descriptors for user threads are initialized with user privilege, and context switching preserves this privilege.

In the next section, we refine these informal specifications to formal ones.

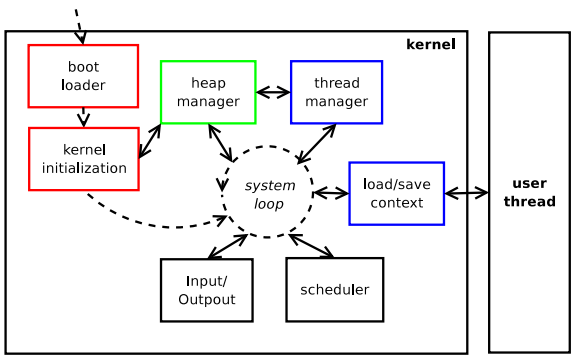

Figure 2: Topsy Control Flow

Arguably, our approach of specifying a selected part of the source code is questionable because it is always possible that some rogue function breaks memory isolation in an apparently unrelated part of the operating system. However, we think that our approach is still

relevant for several reasons. The first reason is that, as knowledgeable programmers, we certainly did specify the most important functions. The second reason is that code sharing limits the possibility for errors. For example, one can imagine that the message passing facilities for communication between threads may break memory isolation by mistakenly overriding thread privileges in the heap. However, since both the message passing facilities and the thread manager rely on the same heap manager, this is unlikely to happen as long as the allocation function allocates only fresh blocks, as we did specify above. Last but not least, the alternative approach of modeling, specifying, and verifying the whole source code is too complex for today's verification tools.

4 Formal Specification with Separation Logic

In this section, we formally specify memory isolation for Topsy by refining the informal requirements of the previous section.

4.1 The Specification Language

As stated in the introduction, we use separation logic for specification purposes. Separation logic is a language of Hoare triples $\{P\}\texttt{prg}\{Q\}$ where `prg` is an imperative program and P, Q are logical formulas. The intended semantics is that, in any state that satisfies formula P, the execution of program `prg` leads to a state satisfying formula Q. In order to build sound triples, there is a collection of syntax-directed inference rules.

The novelty of separation logic is to enable specification of memory operations allowed in imperative programs with advanced pointer usage. Let us briefly skim through some typical formulas of separation logic. The most basic formulas are the empty formula (noted Emp) that holds only for the empty heap (noted **emp**) and the mapsto formula (noted $l \mapsto e$, where l is an address and e a value) that holds for the singleton heap that associated the address l with the value e. The most important formula is the separating conjunction (noted $*$). Its purpose is to enable local reasoning by isolating parts of the heap between each other. More precisely, $P * Q$ holds for a heap h iff there is a partition h_1, h_2 of h such that P holds for h_1 and Q holds for h_2. For example, $l_1 \mapsto e_1 * l_2 \mapsto e_2$ never holds when $l_1 = l_2$.

4.2 Specification of the Heap Manager

The heap manager of Topsy implements allocation/deallocation primitives using a list-like data structure, hereafter called the *heap-list* to avoid confusion with the heap of separation logic[2]. Each element of a heap-list is a block that consists of a two-fields header and an array of memory. The header consists of the type of the block (free or allocated) and a pointer to the next block; the array of memory lies in between. The trailing header points to null and is not followed by any array of memory. Observe that the size of arrays of memory can be computed using the values of pointers.

Before specifying the heap manager, let us define a predicate to characterize the heap-list data structure. First, we define a simpler predicate to characterize arrays of memory rooted at some address l and of length sz:

$$\begin{aligned}&\mathsf{Array}\ l\ sz \stackrel{def}{=} (l{=}0\ \wedge\ \mathbf{emp}) \vee\\ &\quad (sz > 0\ \wedge (\exists e.(l \mapsto e)) * (\mathsf{Array}\ (l+1)\ (sz-1)))\end{aligned}$$

Using the Array predicate, we can now define the $\mathsf{Heap\text{-}list}$ predicate:

$$\begin{aligned}&\mathsf{Heap\text{-}list}\ x \stackrel{def}{=} \exists st.(x \mapsto st, nil) \vee\\ &\quad \exists next.(next \neq nil)\ \wedge\ (x \mapsto free, next) *\\ &\qquad (\mathsf{Array}\ (x+2)\ (next - x - 2)) * (\mathsf{Heap\text{-}list}\ next) \vee\\ &\quad \exists next.(next \neq nil)\ \wedge\ (x \mapsto allocated, next) *\\ &\qquad (\mathsf{Array}\ (x+2)\ (next - x - 2)) * (\mathsf{Heap\text{-}list}\ next)\end{aligned}$$

The first clause captures "empty" lists (that contain only the trailing header). The second (resp. third) clause captures lists starting with a free (resp. allocated) block.

4.2.1 Initialization

In this section, we specify the function that initializes the heap-list (see Sect. 3.1.3).

Let us assume that the heap of the heap manager starts at address *hm_base* and has size *hm_size*. The initialization function `hmInit` turns this area into a heap-list consisting of only one big free block. Let $\mathsf{Heap\text{-}free}$ be the predicate that holds for heap-lists consisting only of free blocks. Using this predicate, the specification of `hmInit` becomes:

$$\begin{array}{c}\{\mathsf{Array}\ hm\_base\ hm\_size\}\\ \texttt{hmInit (}hm\_base\texttt{, }hm\_size\texttt{);}\\ \{\mathsf{Heap\text{-}free}\ hm\_base\}\end{array}$$

The formal definition of the $\mathsf{Heap\text{-}free}$ is similar to the definition of the predicate $\mathsf{Heap\text{-}list}$:

$$\begin{aligned}&\mathsf{Heap\text{-}free}\ x \stackrel{def}{=} \exists st.(x \mapsto st, nil) \vee\\ &\quad \exists next.(next \neq nil)\ \wedge\ (x \mapsto free, next) *\\ &\qquad (\mathsf{Array}(x+2)(next - x - 2)) * (\mathsf{Heap\text{-}free}\ next)\end{aligned}$$

4.2.2 Allocation/Deallocation

In this section, we specify the allocation/deallocation functions of the heap manager (see Sect. 3.1.3).

The allocation function `hmAlloc` is implemented by searching for a large-enough free block in the heap-list. If such a block can be found, it is split into an allocated block (whose address is returned) and a free block (available for further allocations).

The specification of the allocation function consists in checking that newly allocated blocks do not use already allocated addresses. Our idea is to define a predicate that characterizes heap-lists *and* isolate any previously allocated block: $(\mathsf{Heap\text{-}alloc}_{\{l,k,\ldots\}}\ hm\_base)$ is the same as $(\mathsf{Heap\text{-}list}\ hm\_base)$ without the arrays of memory starting at addresses $l,k,\ldots$ Using this predicate, we can specify that any allocated block (starting at address x and of size $size_x$) do not overlap with newly

[2]Readers familiar with separation logic will observe that we do not use the standard allocation/deallocation primitives.

allocated block (starting at address y —provided the function does not fail— and of size $size_y$):

$$\begin{array}{c}\{\ \mathsf{Heap\text{-}alloc}_{\{x-2\}}\, hm\_base * \mathsf{Array}\ x\ size_x \wedge y{=}0\ \}\\ \texttt{hmAlloc}(size_y,\ y);\\ \left\{\begin{array}{c}(\mathsf{Heap\text{-}alloc}_{\{x-2,y-2\}}\ hm\_base\ *\\ \mathsf{Array}\ x\ size_x\ *\ \mathsf{Array}\ y\ size_y\ \wedge y{\neq}0)\ \vee\\ (\mathsf{Heap\text{-}alloc}_{\{x-2\}}\ hm\_base * \mathsf{Array}\ x\ size_x \wedge y{=}0)\end{array}\right\}\end{array}$$

The formal definition of the Heap-alloc predicate follows:

$$\begin{array}{l}\mathsf{Heap\text{-}alloc}_L\ l \stackrel{def}{=} \exists st.(l \mapsto st, nil) \vee\\ \exists next.(next \neq nil) \wedge (l \mapsto free, next) *\\ \quad \mathsf{Array}(l{+}2)(next{-}\,l{-}2) * \mathsf{Heap\text{-}alloc}_L\ next \vee\\ \exists next.(next \neq nil) \wedge (l \mapsto allocated, next) *\\ \quad (l \notin L \rightarrow \mathsf{Array}(l{+}2)(next{-}\,l{-}2)) * \mathsf{Heap\text{-}alloc}_L next \vee\\ \exists next.(next \neq nil) \wedge (l \mapsto allocated, next) *\\ \quad (l \in L \rightarrow \mathsf{Emp}) * \mathsf{Heap\text{-}alloc}_L\ next\end{array}$$

The specification of the deallocation function `hmFree` (omitted here by lack of space) makes use of the same predicates for the pre/post-conditions.

4.3 Specification of the Thread Manager

4.3.1 Initialization

In this section, we specify the function that initializes the thread descriptors (see Sect. 3.1.2). Recall that we want to ensure that thread descriptors initialized on behalf of the user space are properly set.

The initialization of thread descriptors is performed by the function `threadBuild` which takes two parameters: the privilege of the thread (parameter *space*, that can be either *KERNEL* or *USER*) and the location of the thread descriptor to be initialized (parameter x, of type *threadPtr*). The following specification states that the creation of user-space thread initializes the fields corresponding to the registers in the thread descriptor to user-privilege (& represents the binary-and operator):

$$\begin{array}{c}\{(space{=}USER) \wedge (\mathsf{Array}\ x\ (sizeof\ threadPtr))\}\\ \texttt{threadBuild (}space\texttt{, }x\texttt{);}\\ \{\underbrace{(\exists \_cs.\ x.contextPtr.tf\_cs \mapsto \_cs \wedge \_cs\&3 = 3) * \cdots *}_{\text{for the fields } cs,ds,ss,es,fs,gs}\mathsf{True}\}\end{array}$$

4.3.2 Context Switch

In this section, we specify the functions that save and restore the processor context during context switch (see Sect. 3.1.2).

The processor context consists of the value of its registers-set. There is a data structure to store these values in thread descriptors. The context-switch functions make the translation between these data structures and the processor context. In order to ensure memory isolation, we need to verify that (1) after restoring a user-privilege thread descriptor, the processor runs in user privilege, and (2) after saving a user-privilege processor context, the corresponding thread descriptor has user-privilege.

Before specifying context switch, we introduce notations to accommodate assembly code. The idea is to translate assembly code to the imperative language of separation logic. More precisely, we identify a set of variables to match registers (for example, the variable *CR0* represents the CR0 register), and we model memory-based operations using pointer-based operations (for example, load operations become dereferences). The only difficulty are stack-based operations. In contrast, branching operations are easily translated because the assembly code in question does not exhibit complicated control flow.

We now formally specify the restore function (the save function is similar). The restore function consists of assembly code starting after the label `restoreContext`; it receives the data structure containing the processor context through the esp register. The specification below ensures that restoring a user-privilege processor context data structure does lead to a user-privilege processor context:

$$\begin{array}{c}\left\{\begin{array}{c}\underbrace{(esp \mapsto \ldots, \_cs, \ldots) \wedge (\_cs\&3 = 3\ \wedge\ \ldots)}_{\text{for the fields } \_cs,\_ds,\_ss,\_es,\_fs,\_gs}\end{array}\right\}\\ \texttt{restoreContext:}\ldots\\ \{\quad \underbrace{cs\&3 = 3\ \wedge\ \ldots}_{\text{for the registers } cs,ds,ss,es,fs,gs}\quad\}\end{array}$$

4.4 Specification of the Boot Loader

In this section, we formally specify that the boot loader sets a GDT implementing the intended memory model (see Sect. 3.1.1).

The specification of the GDT amounts to an arithmetic characterization of Topsy memory model. Initially, there is an image of the GDT in the boot loader data beginning at GDT00. First, the boot loader copies the image of the GDT to its final location. Then, it fills the gdtr register with the address of the GDT (called *gdtr_base*) and its size (equal to 5). Finally, it switches the processor into segmented mode by setting the first bit of the control register CR0. Let Valid-Segment-Descriptor be a predicate that characterizes valid segment descriptors. The specification of the boot loader becomes:

$$\begin{array}{c}\left\{\begin{array}{c}\mathsf{Array}\ 1\ memory\_size\ \wedge\ (\forall x.\ 0 \leq x < 5 \rightarrow\\ \mathsf{Valid\text{-}Segment\text{-}Descriptor}(GDT00{+}x{*}8))\end{array}\right\}\\ \texttt{Create\_GDT:}\ldots\\ \texttt{PM\_Switch:}\ldots\\ \left\{\begin{array}{c}CR0\&1 = 1\ \wedge\ (\forall x.\ 0 \leq x < 5 \rightarrow\\ \mathsf{Valid\text{-}Segment\text{-}Descriptor}(gdt\_base{+}x{*}8))\end{array}\right\}\end{array}$$

The definition of the Valid-Segment-Descriptor follows. Intuitively, (Valid-Segment-Descriptor x) holds when x is the starting address of a segment descriptor that is either in kernel mode or the corresponding segment is inside the user working area:

$$\begin{array}{l}\mathsf{Valid\text{-}Segment\text{-}Descriptor}\ x \stackrel{def}{=}\\ \exists y_0, \ldots, y_7.\ x \mapsto y_0, \ldots, y_7 * \mathsf{True} \wedge\\ ((y_5 \div 32)\&3{=}0 \vee (y_2{+}2^8{*}y_3{+}2^{16}{*}y_4{+}2^{24}{*}y_7) \geq 2^{14})\end{array}$$

5 Formal Verification with Coq

In this section, we illustrate how above specifications can be verified formally using the Coq proof assistant. For this purpose, we have implemented a version of separation logic in Coq that we use to translate as faithfully as possible the original source code of Topsy. At

the time of this writing, we have already translated and verified some parts of the specifications presented in the previous section and already found some issues. The corresponding Coq scripts (around 7,000 lines of scripts) are available online [4].

By way of example, let us show how we verify the initialization of the heap manager described in Sect. 4.2.1. First, we translate the source code of the `hmInit` function into Coq, see Fig. 3. Second, we formalize the pre/post-conditions. This requires the encoding in Coq of the predicates Array and Heap-free:

```
Fixpoint Array (x:loc) (sz:nat) : assert :=
 match sz with
   0 => Emp
 | S n => (fun s => fun h => ∃y,
     ((int_e (loc2val x)) |-> (int_e y)) s h) **
       (Array (S x) n)
 end.

Inductive Heap_free: expr->store.s->heap.h->Prop :=
  heap_nil: ∀e s h status next, eval next s = 0 ->
   (e|-->status::next::nil) s h -> Heap_free e s h
| heap_cons: ∀e s h next h1 h2 x y, y ≠ 0 ->
   disjoint h1 h2 ∧ equal h (union h1 h2) ->
   eval e s = loc2val x -> eval next s = loc2val y ->
   (e|-->Free::next::nil ** Array(x+2)(y-x-2) s h1 ->
   Heap_free next s h2 -> Heap_free e s h.
```

Using above predicates and the Coq translation of the `hmInit` function, we can input the corresponding Hoare triple in Coq and check whether it holds:

```
Lemma hmInit_verif: ∀adr size, adr>0 -> size>4  ->
 { Array adr size }
 (hmInit adr size)
 { Heap_free (nat_e adr) }.
```

Actually, it happens that the above lemma does not hold because the original `hmInit` builds the trailing header outside of the heap. Indeed, the addition in line 8 of source code in Fig. 3 generates an out of range address. This problem is not an error in the sense that it does not lead to any abnormal behavior, but it certainly undermines reusability of functions.

Modulo this correction, the proof is straightforward because the code amounts to structure assignments. In practice, we appeal to a weakest-precondition generator whose output can be solved using basic properties of separating connectives (namely, monotony and adjunction: the logical view of destructive update).

6 Conclusion

In this paper, we presented an approach to formal verification of memory properties of low-level software using separation logic. More precisely, we specified and partly verified the property of memory isolation for the Topsy operating system. Our approach was to extract from the source code the functions involved in thread and memory management (around 300 lines of code mixing C and assembly) in order to specify them individually. In addition, we illustrated mechanical verification in the Coq proof assistant, exhibiting an issue we found in the implementation.

Related Work The delta-core project [5] aims at verifying a micro-kernel written in a C-like language. Verification of properties of system calls have been specified and verified in the PowerEpsilon proof assistant after source code translation. The main difference with our work is that we focus on properties of memory management.

The VFiasco project [6] aims at verifying memory properties of a micro-kernel. The approach is to automatically translate a subset of the C++ language into the PVS proof assistant where a model of x86 processors has been implemented. The translation process seems to be the current challenge this project is facing.

The implementation of separation logic we did in the Coq proof assistant is similar to work by Weber in Isabelle [7]. The main difference is technical: we use an abstract data type implemented by means of modules for the heap whereas Weber uses partial functions.

Future Work Specifications written with lists are difficult to work with. In order to facilitate verification, we are currently implementing lemmas to prove properties of linked lists. In contrast, portions of code that only deal with assignments and branching instructions should be handled automatically. For that purpose, we plan to interface our Coq implementation of separation logic with existing work an automation of verification of separation logic [8].

References

[1] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, p. 55–74. Invited lecture.

[2] Lukas Ruf, Claudio Jeker, Boris Lutz, and Bernhard Plattner. Topsy v3: A NodeOS For Network Processors. In *2nd International Workshop on Active Network Technologies and Applications (ANTA 2003)*.

[3] Browsable Topsy Source Tree. `http://www.tik.ee.ethz.ch/~topsy/Source`.

[4] Formal Verification of Memory Isolation for Topsy. `http://web.yl.is.s.u-tokyo.ac.jp/~affeldt/seplog`. Coq scripts. Work in progress.

[5] Ming-Yuan Zhu, Lei Luo, and Guang-Zhe Xiong. A Provably Correct Operating System: delta-Core. *Operating Systems Review*, 35(1):17–33, January 2001.

[6] Michael Hohmuth and Hendrik Tews. The VFiasco Approach for a Verified Operating System. In *2nd ECOOP Workshop on Program Languages and Operating Systems (ECOOP-PLOS 2005)*.

[7] Tjark Weber. Towards Mechanized Program Verification with Separation Logic. In *13th Conference on Computer Science Logic (CSL 2004)*, volume 3210 of *LNCS*, p. 250–264. Springer.

[8] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. June 2005. Draft. `http://www.dcs.qmul.ac.uk/~berdine/drafts/execution.pdf`.

```
#define NULL (void*) 0
#define KERNELHEAPSIZE   (64*1024)

Error hmInit(Address addr)
{
    start = (HmEntry)addr;
    start->next = (HmEntry) ((unsigned long)addr +
        KERNELHEAPSIZE + sizeof(HmEntryDesc));
    start->status = HM_FREED;

    end = start->next;
    end->next = NULL;
    end->status = HM_ALLOCATED;

    hmLock = &hmLockDesc;
    lockInit( hmLock);

    return HM_INITOK;
}
```

```
Definition null := (int_e 0%Z).
Axiom size : nat.

Definition hmInit (adr:loc) :=
(
  hmStart <- (nat_e adr);
  hmStart -.> next *<- (nat_e adr) +e
    (nat_e size) +e (int_e 2);
  hmStart -.> status *<- Free;

  hmEnd <-* hmStart -.> next;
  hmEnd -.> next *<- null;
  hmEnd -.> status *<- Allocated

  (* locking operations elapsed *)
).
```

Figure 3: `hmInit` source code (original C code on the left, Coq translation on the right)