Verification of Unloadable Modules

Bart Jacobs, Jan Smans*, and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract. Programs in unsafe languages, like C and C++, may dynamically load and unload modules. For example, some operating system kernels support dynamic loading and unloading of device drivers. This causes specific difficulties in the verification of such programs and modules; in particular, it must be verified that no functions or global variables from the module are used after the module is unloaded.

We present the approach we used to add support for loading and unloading modules to our separation-logic-based program verifier VeriFast. Our approach to the specification and verification of function pointer calls, based on parameterizing function types by predicates, is sound in the presence of unloading, but at the same time does not complicate the verification of programs that perform no unloading, and does not require callers to distinguish between function pointers that point into unloadable modules and ones that do not.

We offer a machine-checked formalization and soundness proof and we report on verifying a small kernel-like program using VeriFast. To the best of our knowledge, ours is the first approach for sound modular verification of C programs that load and unload modules.

1 Introduction

In statically typed safe programming languages, code is immutable and permanent. That is, both statically bound and dynamically bound routine calls always succeed and are bound to code that satisfies the static type of the call. Also, if an object reference or function value satisfies a given contract at one point in time, it continues to do so forever.

This is not the case in dynamically typed languages and in unsafe languages like C and C++. In C, if at one point during execution at a given address there is a function that satisfies a given contract, this does not mean this will remain the case indefinitely. The module containing the function may be part of a dynamically linked library (DLL, also known as a shared object) that may be unloaded, or the function's code may reside on the stack or in a malloc'ed piece of memory.

Existing verification approaches for C programs (Caduceus/Frama-C [7], HAVOC [5], VCC [4], Smallfoot [1], our own verifier VeriFast [8]) assume that the program is unchanging and is not part of the mutable state. As a result,

* Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

these approaches cannot be used for sound verification of programs that involve the unloading of code.

In this paper, we propose a separation-logic-based approach for extending a verification approach for C programs to enable verification of the memory-safety, data-race-freedom, and compliance with preconditions, postconditions, and other assertions of programs involving code unloading. Specifically, our contributions with respect to existing verification tools for C are the following features, which none of the existing tools have:

- **Soundness in the presence of unloading** Verification of an unloadable module checks that when execution of code in the module is attempted, a permission indicating that the code is present is owned by the current thread.
- **Predicate-parameterized function types** In the absence of unloading, abstract predicate families [11] indexed by function pointer could be used to enable abstraction for function pointer contracts. However, in the presence of unloading, a function pointer no longer immutably refers to a specific function. Predicate-parameterized function types solve this problem, and furthermore allow callers to be agnostic as to whether a function pointer points into an unloadable module or not.
- **Modular support for global variables** A module, even an unloadable one, may declare global variables. It is checked that these are not used after the module is unloaded.

We implemented the approach in our prototype verifier, VeriFast, and we verified a small server written in C that allows clients to load modules, unload modules, and use services provided by the modules, mimicking operating system kernels that may dynamically load and unload device drivers. Also, we developed a formalization and a machine-checked soundness proof of the approach.

The remainder of the paper is structured as follows. In Section 2, we illustate the problem by means of an example. In Section 3, we formalize the relevant subset of C. In Section 4, we present our approach. In Section 5, we discuss the implementation. Finally, in Section 6, we conclude and discuss related work.

2 Problem Statement

We illustrate the verification challenges addressed by our approach using the example C program shown in Figure 1. The example program adopts some aspects of an operating system kernel that loads and unloads device drivers as kernel modules. It consists of a simple “kernel module”, `RamDisk.c`, that implements a file abstraction backed by memory, and a simple “kernel”, that loads the kernel module, tests its functionality, and unloads it. Like most kernel modules, the example module uses kernel resources that should be cleaned up properly when the client is done using the module. The example uses a simple byte vector resource (i.e. a growable array of bytes) as backing for its file abstraction. While the example is kept simple on purpose, it contains the essential ingredients of dynamic loading and unloading of modules that offer services to clients and that potentially use resources not directly visible to the client in the implementation of these services.

The kernel dynamically loads the `RamDisk` module using an API declared in `Modules.h`. This API assumes each dynamically loaded module exports a function called *module_init* and a function called *module_exit*, and it returns pointers to these functions through its by-reference parameters *init* and *exit*. The client program performs implicit casts of these void pointers to pointers to function types *module_init* and *module_exit*, respectively, declared in `KernelModule.h`.

After loading `RamDisk`, the client program tests its functionality using an assert statement, in function *testOps*. Specifically, it writes the zero-terminated string `"Hello"` to the file and then checks that reading the file yields the same string. If the condition of the assert statement evaluates to false, the program aborts. Finally, the program unloads the module and terminates.

The challenge we take up in this paper is to come up with an approach, suitable for implementation in a semi-automatic program verifier like VeriFast, for verifying modularly that programs that load and unload modules, such as the example program, execute safely. Executing safely is a strong property: it means that the program does not access (i.e., read, write, or execute) unallocated memory and that all assert statements succeed.

Support for unloadable modules introduces additional safety risks, including for instance:

- Programs should not use a function pointer pointing into an unloadable module after this module has been unloaded. For example, if the two final calls in function *main* are swapped, the program is unsafe.
- Programs should not access a global variable declared by an unloadable module after this module has been unloaded. For example, if an access of *o* is added at the end of function *main*, the program is unsafe.

The verification approach should be modular and sound. Modularity implies that:

- Unloadable modules and client programs should be verifiable separately, with minimal assumptions about each other and with proper information hiding. For example, the verification of a client program should not depend on whether a module it loads and unloads declares global variables.
- The verification of code that uses a function pointer should not depend on whether the function pointer points into an unloadable module or into static code. For example, verification of function *testOps* should be agnostic as to whether function pointer $o \rightarrow write$ points to an unloadable module or not.

Soundness in such a modular setting means: If all modules and client programs involved in a run of the system have been verified, then the run is safe (as defined above).

```
// Modules.h
struct module;

struct module *load_module
   (char *name,
    void **init, void **exit);
void unload_module
   (struct module *m);
```

```
// KernelModule.h
typedef int read(
   int offset, char *buffer, int count);
typedef void write(
   int offset, char *buffer, int count);

struct file_ops {
   read *read;
   write *write;
};

typedef struct file_ops *module_init();
typedef void module_exit();
```

```
// Kernel.c
#include "Modules.h"
#include "KernelModule.h"

void testOps(struct file_ops *o) {
   o->write(0, "Hello", 6); char b[10];
   int n = o->read(0, b, 10);
   assert(n == 6 &&
      memcmp(b, "Hello", 6) == 0);
}

void main() {
   module_init *init;
   module_exit *exit;
   struct module *m  = load_module(
      "RamDisk", &init, &exit);
   struct file_ops *o = init();
   testOps(o); exit();
   unload_module(m);
}
```

```
// ByteVector.h
struct vector;

struct vector *create_vector();
int vector_read(struct vector *v,
   int offset, char *buffer, int count);
void vector_write(struct vector *v,
   int offset, char *buffer, int count);
void vector_dispose
   (struct vector *v);
```

```
// RamDisk.c
#include "KernelModule.h"
#include "ByteVector.h"

struct vector *vector = 0;

int myRead(int o, char *b, int c) {
   return vector_read(vector, o, b, c);
}

int myWrite(int o, char *b, int c) {
   vector_write(vector, o, b, c);
}

struct file_ops o = {0, 0};

struct file_ops *module_init() {
   o.read = myRead;
   o.write = myWrite;
   vector = create_vector();
   return &o;
}

void module_exit() {
   vector_dispose(vector);
}
```

Fig. 1. Example C program that loads and unloads a module

3 Formal Programming Language

We will present our approach in the context of a simple formal programming language that retains only the relevant aspects of C. In this section, we introduce the syntax and the semantics of the programming language. In the next section, we present our specification formalism and proof system.

The formal programming language is an extension of the standard separation logic language [14] with function pointer call and module load and unload commands, and with function values L. The latter are used to represent pieces of code in the heap; they are a higher-level analog of assembly language instructions. The language's syntax is as follows:

$$
\begin{array}{l}
\quad n \in \mathbb{Z}, x \in \text{Vars}, \tau \in \text{FunTypeNames} \\
e ::= n \mid x \mid e + e \mid e - e \qquad\qquad b ::= e = e \mid e < e \\
c ::= x := \mathbf{cons}(\overline{e}) \mid x := [e] \mid [e] := e \mid \mathbf{dispose}(e) \mid x := e \mid (c; c) \\
\quad \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid x := \mathbf{call}\ e(\overline{e}) \mid x := \mathbf{load}\ e\ \mathbf{as}\ \tau \mid \mathbf{unload}(e) \\
L ::= \mathbf{lambda}\ (\overline{x})\ c
\end{array}
$$

We adopt the standard run-time state of separation logic, consisting of a store s, a total function that maps program variable names to integers, and a heap h, a partial function that maps positive integer addresses to integer values. The domain of the heap coincides with the allocated addresses. In order to be able to store function values in the heap, we assume an injective encoding $\lfloor\cdot\rfloor$ of function values L into integers; this corresponds to the encoding of assembly instructions as byte sequences on real architectures.

To model the loading and unloading of modules, we assume the existence of a module repository *Modules*, which is a finite map from module names to module definitions. A module name is simply an integer. A module definition consists of the module's *contract*, which is a function type name $\tau \in$ FunTypeNames, and a module image. The module image is simply a tuple of one or more integers. The first element of the tuple is the encoded function value for the module's entry point; the other elements may be encoded function values or data (corresponding to global variables in C).

We describe the semantics of the new commands. For a formal big-step semantics of the language, see the extended version of this paper [9].

Function pointer call command $x :=$ **call** $e(\overline{e})$ executes the function value at address e in the heap. Specifically, if at address e there is a function value with parameters $\overline{x}$ and body c, it executes c under a store that binds the parameters to the arguments specified in the call and the variable ip (for *instruction pointer*) to the target address, and it assigns the result of the call, conventionally stored in local variable result by the function, to variable x. The call aborts if address e is not allocated, if the value at address e is not an encoded function value, or if there are more or fewer parameters than arguments. It also aborts if c aborts.

Command $x :=$ **load** e **as** τ, where e is an expression and τ is a function type name, loads the module named e and stores its address in variable x. Specifically, loading a module whose image is $v_1, \ldots, v_n$ means allocating $n + 1$ consecutive addresses, storing the image size n at the first address (used by **unload**), and the image itself at the subsequent addresses. The address x returned by the **load** command is the address where the image size is stored; it follows that the first element of the image is at address $x + 1$.

If there is no module named e in the repository, or if the module's contract is not τ, no module is loaded and the **load** command stores the value zero in x. We do not abort here; this, together with the module contract check, allows us to verify scenarios where the identity of the module being loaded is not known statically, such as when the module name is taken from user input.

Command **unload**(e) deallocates the loaded module at address e. It aborts if address e is not allocated, or if the value n at address e is not positive, or if any of the n subsequent addresses are not allocated; otherwise, it deallocates all of these $n + 1$ addresses.

3.1 Example program

We illustrate the language by translating the example C program of Figure 1 into it; the result is shown in Figure 2.

```
Modules = {
  (100, (module_init,
     (⌊MI⌋, ⌊ME⌋, ⌊MR⌋, ⌊MW⌋,
      $V_0$, $OR_0$, $OW_0$)))
}
main =
  m := load M as module_init;
  if m = 0 then skip else (
    init := m + 1; o := call init();
    write := [o + 1]; _ := call write(42);
    read := [o]; x := call read();
    assert(x = 42);
    exit := m + 2; _ := call exit();
    unload(m)
  )

where
  MI = lambda ()
        [ip + 5] := ip + 2; [ip + 6] := ip + 3;
        v := cons(0); [ip + 4] := v;
        result := ip + 5
  ME = lambda () v := [ip + 3]; dispose(v)
  MR = lambda () v := [ip + 2]; result := [v]
  MW = lambda (x) v := [ip + 1]; [v] := x
  $V_0$ = $OR_0$ = $OW_0$ = 0

  skip = x := x
  assert(b) = if b then skip else [0] := 0
```

Fig. 2. Example program in the formal language

The module repository of the example contains a single module, corresponding to module `RamDisk` of the C example. Its name is 100, its contract is module_init, and its image consists of seven values, the first four of which are the encodings of function values corresponding to functions *module_init*, *module_exit*, *myRead*, and *myWrite*, respectively, and the last three correspond to the global variables *vector*, *o.read*, and *o.write* of the C example.

There are four minor differences between the C example and the formal example. The first is that instead of specifying the specific module `RamDisk` as the module to be loaded, the formal example uses the value of variable M, whose initial value is arbitrary; imagine it was initialized by the user. If the value of this variable equals 100, the module named 100 in the repository will be loaded; otherwise, the **load** command will return zero and the rest of the program will be skipped. We will verify that the program is safe for arbitrary initial values of the variables; and during verification of the main program, we will make no assumptions about the module repository other than that each module has been verified.

The second difference is that in the formal example, the *module_init* and *module_exit* functions are at fixed offsets 0 and 1 in the module image, so their addresses can be obtained by adding 1, resp. 2 to the address returned by the **load** command.

The third difference is that in the formal example, for simplicity the contents of a "file" consist of a single integer. Therefore, instead of a byte vector, the module uses a simple memory cell to back its file abstraction. The write function takes the new file contents as its argument, and the read function returns the file contents as its return value.

The fourth difference is that the *testOps* function has been inlined into the main program.

Notice that the functions in the example module use their instruction pointer ip (i.e., the address of the function in memory) to compute the address of the module's other functions and global variables. For example, in the *module_init* function, the address of the global variable *vector* equals $\mathsf{ip} + 4$ since *module_init* is the first element of the module image and *vector* is the fifth element. This is a common technique for achieving position-independent code.

An example run of the example program is shown in Figure 3. The symbols *MI*, *ME*, *MR*, and *MW* refer to the function values defined in Figure 2. Notice that the init call initializes the module's global variables and allocates the "vector" (at address 50). The write call updates the vector, and the exit call de-allocates it.

4 Specification and Verification Approach

In this section, we present an approach for specifying and modularly verifying modules and programs that satisfies the soundness and modularity goals identified in Section 2. The approach is separation logic [14] with abstract predicates [11], extended with

- special built-in abstract predicates **lib**, $\mathbf{module}_0$, and **module** to allow programs that load and unload modules to reason about loaded modules abstractly, and
- *parameterized function types* and *partial predicate applications* for reasoning about function pointers in a way that allows abstraction over whether a function pointer points into an unloadable module.

```
// $s = \{\mathsf{M}\colon 100, \ldots\}, h = \emptyset$
m := load M as module_init;
// $s = \{\mathsf{m}\colon 1, \ldots\}, h = \{1\colon 7, 2\colon \lfloor MI \rfloor, 3\colon \lfloor ME \rfloor, 4\colon \lfloor MR \rfloor, 5\colon \lfloor MW \rfloor, 6\colon 0, 7\colon 0, 8\colon 0\}$
if m = 0 then skip else (
   // $s = \{\mathsf{m}\colon 1, \ldots\}, h = \{1\colon 7, 2\colon \lfloor MI \rfloor, 3\colon \lfloor ME \rfloor, 4\colon \lfloor MR \rfloor, 5\colon \lfloor MW \rfloor, 6\colon 0, 7\colon 0, 8\colon 0\}$
   init := m + 1; o := call init();
   // $s = \{\mathsf{m}\colon 1, \mathsf{init}\colon 2, \mathsf{o}\colon 7, \ldots\}$
   // $h = \{1\colon 7, 2\colon \lfloor MI \rfloor, 3\colon \lfloor ME \rfloor, 4\colon \lfloor MR \rfloor, 5\colon \lfloor MW \rfloor, 6\colon 50, 7\colon 4, 8\colon 5, 50\colon 0\}$
   write := [o + 1]; _ := call write(42);
   // $s = \{\mathsf{m}\colon 1, \mathsf{init}\colon 2, \mathsf{o}\colon 7, \mathsf{write}\colon 5, \ldots\}$
   // $h = \{1\colon 7, 2\colon \lfloor MI \rfloor, 3\colon \lfloor ME \rfloor, 4\colon \lfloor MR \rfloor, 5\colon \lfloor MW \rfloor, 6\colon 50, 7\colon 4, 8\colon 5, 50\colon 42\}$
   read := [o]; x := call read();
   // $s = \{\mathsf{m}\colon 1, \mathsf{init}\colon 2, \mathsf{o}\colon 7, \mathsf{write}\colon 5, \mathsf{read}\colon 4, \mathsf{x}\colon 42, \ldots\}$
   // $h = \{1\colon 7, 2\colon \lfloor MI \rfloor, 3\colon \lfloor ME \rfloor, 4\colon \lfloor MR \rfloor, 5\colon \lfloor MW \rfloor, 6\colon 50, 7\colon 4, 8\colon 5, 50\colon 42\}$
   assert(x = 42);
   exit := m + 2; _ := call exit();
   // $s = \{\mathsf{m}\colon 1, \mathsf{exit}\colon 3, \ldots\}, h = \{1\colon 7, 2\colon \lfloor MI \rfloor, 3\colon \lfloor ME \rfloor, 4\colon \lfloor MR \rfloor, 5\colon \lfloor MW \rfloor, 6\colon 50, 7\colon 4, 8\colon 5\}$
   unload(m)
   // $s = \{\ldots\}, h = \emptyset$
)
```

Fig. 3. An example run of the example program. In this run, the value of M is 100, the module is allocated at address 1, and the "vector" is allocated at address 50.

We first introduce the specification language and we illustrate it with a specification for the example module. We then define the proof system and outline a proof of the example program.

4.1 Specification Language

As in separation logic and in Hoare logic, a correctness judgment is of the form $\{P\}\ c\ \{Q\}$, where c is a command and P and Q are *assertions*, i.e. conditions on the program state. It means: if command c is executed in an initial state that satisfies precondition P, then it executes safely and if it terminates, the final state satisfies Q.

Assertions may contain the usual logical operators $\wedge, \vee, \exists$, and equality between assertion expressions. The assertion expressions include the program expressions as well as *logical variable occurrences*. As in Hoare logic, logical variables are universally quantified across correctness judgments, and serve to connect the precondition and the postcondition.

In Hoare logic, an assertion is interpreted under a store and a logical variable interpretation (a total function from logical variable names to integers). In separation logic, an assertion is interpreted under a store, a logical variable interpretation, and a heap. Separation logic introduces three operators to describe the heap: **emp** states that the heap is empty; the points-to assertion $E \mapsto E'$

states that the heap consists of a single memory cell at address E containing value E'; and the separating conjunction $A * A'$ states that the heap can be split up into two disjoint parts, such that one part satisfies A and the other part satisfies A'.

Abstract predicates (or *predicates* for short) are named, parameterized assertions. They serve to describe a piece of state abstractly, without revealing the details. For example, the predicate $\mathsf{Q}(\ell, x)$ defined below describes the resources used by the example module after initialization, when loaded at address ℓ and when the file contents are x:

$$\begin{aligned}\textbf{predicate } \mathsf{Q}(\ell, x) &= \ell + 1 \mapsto \lfloor MI \rfloor * \ell + 2 \mapsto \lfloor ME \rfloor * \ell + 3 \mapsto \lfloor MR \rfloor \\ &\quad * \ell + 4 \mapsto \lfloor MW \rfloor * \exists v \bullet \ell + 5 \mapsto v * v \mapsto x\end{aligned}$$

It encompasses the module image (at $\ell + 1$ through $\ell + 7$), plus the vector (at v), minus the memory cells containing the function pointers (at $\ell + 6$ and $\ell + 7$).

To specify function pointers, we introduce *function type definitions* and *function type assertions*. A function type definition associates a function type name with a precondition and a postcondition. A function type assertion $E : \tau$ states that function pointer E may be safely called with the contract associated with function type τ. We allow function types to be parameterized by a list of integer-valued parameters. Therefore, the general form of function type judgments is $E : \tau(\overline{E})$, where $\overline{E}$ are the function type arguments.

For example, consider the function type definition

$$\textbf{funtype } \mathsf{addN}(\mathsf{n})(\mathsf{x}) \textbf{ req } \mathsf{P}() \textbf{ ens } \mathsf{P}() \land \mathsf{result} = \mathsf{x} + \mathsf{n}$$

where P is some predicate. It defines a function type addN with one function type parameter n. It applies to functions of one argument. Given this definition, the function type assertion $100 : \mathsf{addN}(5)$ implies that calling the function at address 100 with argument 10, in a state where P holds, returns value 15.

We need to be able to parameterize function types by predicates, in order to abstractly specify the state required by a function pointer. In order to avoid a type system, we assume an encoding of predicate names to integers, and we allow predicate assertions of the form $E(\overline{E})$, where E is the encoding of the predicate name and $\overline{E}$ are the predicate arguments. This way, we can use a predicate name as a function type argument.

For example, we can abstract the function type addN defined above over the predicate P by adding a function type parameter p:

$$\textbf{funtype } \mathsf{addN}'(\mathsf{p}, \mathsf{n})(\mathsf{x}) \textbf{ req } \mathsf{p}() \textbf{ ens } \mathsf{p}() \land \mathsf{result} = \mathsf{x} + \mathsf{n}$$

Given this definition, we can restate the earlier assertion as $100 : \mathsf{addN}'(\mathsf{P}, 5)$.

In general, we wish to instantiate predicate-parameterized function types not just by fixed predicate names, but also by predicate names to which one or more arguments have already been applied. To enable this, we assume an encoding $\lfloor\cdot\rfloor$ of partial predicate applications, of the form $p(\overline{n})$, where $\overline{n}$ are the pre-applied predicate arguments, to integers. The meaning of a predicate assertion $E(\overline{E})$, where E is the encoding of a partial predicate application $p(\overline{n})$ and $\overline{E}$ evaluates to $\overline{m}$, is $p(\overline{nm})$.

For example, we can specify the read and write functions from the example program using the following function types:

$$\begin{array}{ll}
\textbf{funtype}\ \mathsf{read}(\mathsf{filePred})() & \textbf{funtype}\ \mathsf{write}(\mathsf{filePred})(\mathsf{x}) \\
\quad \textbf{req}\ \mathsf{filePred}(\mathsf{X}) & \quad \textbf{req}\ \mathsf{filePred}(\_) \\
\quad \textbf{ens}\ \mathsf{filePred}(\mathsf{X}) \land \mathsf{result} = \mathsf{X} & \quad \textbf{ens}\ \mathsf{filePred}(\mathsf{x})
\end{array}$$

The function types are parameterized by a predicate that describes the resources that implement the file abstraction. The predicate takes as an argument the contents of the file. The contract of read states that it returns the current contents; the contract of write does not care about the old contents (denoted by the underscore, shorthand for $\exists y \bullet \mathsf{filePred}(y)$) and sets its argument x as the new contents.

As we will prove later, the read and write functions of the example module satisfy this contract when instantiated with the predicate Q defined above, partially applied to the location ℓ where the module was loaded. Formally, we will prove $\ell + 3 : \mathsf{read}(\mathsf{Q}(\ell))$ and $\ell + 4 : \mathsf{write}(\mathsf{Q}(\ell))$. (Remember that if the module is loaded at ℓ, then the *myRead* function is at $\ell + 3$ and the *myWrite* function is at $\ell + 4$.)

Besides parameterized function types and partial predicate applications, we introduce three special built-in abstract predicates to reason abstractly about modules. $\mathbf{lib}(E, E')$ describes the memory cell holding the image size of a module named E' loaded at address E. $\mathbf{module}_0(E, E')$ describes the memory cells holding the module image of the module named E' loaded at address E, in their initial state. Finally, $\mathbf{module}(E, E')$ describes the memory cells that initially held the module image of the module named E' loaded at address E. The latter predicate states only the allocatedness of these cells; it does not describe their contents.

Formally, if $(M, (\tau, (v_1, \ldots, v_n))) \in \mathit{Modules}$, i.e., there is a module named M with contract τ and whose image consists of the n values $v_1, \ldots, v_n$, then we have

$$\begin{aligned}
\mathbf{lib}(\ell, M) &= \ell \mapsto n \\
\mathbf{module}_0(\ell, M) &= \ell + 1 \mapsto v_1 * \cdots * \ell + n \mapsto v_n \\
\mathbf{module}(\ell, M) &= \ell + 1 \mapsto \_ * \cdots * \ell + n \mapsto \_
\end{aligned}$$

Using these constructs, we can now specify the *module_init* and *module_exit* functions of the example program:

$$\begin{array}{ll}
\textbf{funtype}\ \mathsf{module\_init}(\mathsf{l}, \mathsf{m})() & \textbf{funtype}\ \mathsf{module\_exit}(\mathsf{o}, \mathsf{filePred}, \mathsf{l}, \mathsf{m})() \\
\quad \textbf{req}\ \mathbf{module}_0(\mathsf{l}, \mathsf{m}) & \quad \textbf{req}\ \mathsf{o} \mapsto \_ * \mathsf{o} + 1 \mapsto \_ * \mathsf{filePred}(\_) \\
\quad \textbf{ens}\ \exists \mathsf{filePred}, \mathsf{r}, \mathsf{w} \bullet & \quad \textbf{ens}\ \mathbf{module}(\mathsf{l}, \mathsf{m}) \\
\qquad \mathsf{result} \mapsto \mathsf{r} * \mathsf{result} + 1 \mapsto \mathsf{w} * \mathsf{filePred}(0) & \\
\qquad \land\ \mathsf{r} : \mathsf{read}(\mathsf{filePred}) \land \mathsf{w} : \mathsf{write}(\mathsf{filePred}) & \\
\qquad \land\ \mathsf{l} + 2 : \mathsf{module\_exit}(\mathsf{result}, \mathsf{filePred}, \mathsf{l}, \mathsf{m}) &
\end{array}$$

The function type $\mathsf{module\_init}$ serves as the module's contract; the auxiliary function types $\mathsf{module\_exit}$, read, and write are referred to in the definition of

$\mathsf{module\_init}$. As with all function types that serve as module contracts, $\mathsf{module\_init}$ is parameterized by the address l where the module is loaded and the module name m. The precondition requires the module's image in its initial state. The postcondition states that the return value points to two consecutive memory cells, the first holding a pointer to a read function and the second a pointer to a write function. It further provides the resources $\mathsf{filePred}(0)$ that the read and write functions require; the predicate argument 0 indicates the file contents. The $\mathsf{module\_exit}$ function takes back the memory cells holding the function pointers, as well as the resources denoted by $\mathsf{filePred}(\_)$, and yields back the module image, in an unspecified state, ready to be unloaded.

4.2 Proof system

Our proof system extends separation logic's assertion logic with rules for deriving function type judgments and for folding and unfolding predicate assertions and module assertions; and it extends separation logic's program logic with rules for verifying function pointer call and module load and unload commands. The new rules are shown in Figure 4.

A-FunType

$$\frac{\begin{array}{c}\textbf{funtype}\ \tau(\overline{y})(\overline{x})\ \textbf{req}\ P\ \textbf{ens}\ Q \\ \vdash P[\overline{v}/\overline{y}] \Rightarrow \ell \mapsto \lfloor \textbf{lambda}\ (\overline{x})\ c \rfloor * \textbf{true} \\ \{P[\overline{v}/\overline{y}] \land \mathsf{ip} = \ell\}\ c\ \{Q[\overline{v}/\overline{y}]\}\end{array}}{\vdash \ell : \tau(\overline{v})}$$

A-PredAsn

$$\frac{\textbf{predicate}\ p(\overline{y}) = A}{\vdash p(\overline{v})(\overline{w}) \Leftrightarrow A[\overline{vw}/\overline{y}]}$$

A-Module-Unfold

$$\frac{(M, (\tau, (v_1, \ldots, v_m))) \in \mathit{Modules}}{\vdash \begin{array}{c}\textbf{module}_0(y, M)\ \Rightarrow \\ y + 1 \mapsto v_1 * \ldots * y + m \mapsto v_m\end{array}}$$

A-Module-Fold

$$\frac{(M, (\tau, (v_1, \ldots, v_m))) \in \mathit{Modules}}{\vdash \begin{array}{c}y + 1 \mapsto \_ * \cdots * y + m \mapsto \_ \\ \Rightarrow\ \textbf{module}(y, M)\end{array}}$$

C-Call

$$\frac{\textbf{funtype}\ \tau(\overline{y})(\overline{x})\ \textbf{req}\ P\ \textbf{ens}\ Q}{\{e : \tau(\overline{y}) \land \overline{e} = \overline{z} \land P[\overline{z}/\overline{x}]\}\ x := \textbf{call}\ e(\overline{e})\ \{Q[\overline{z}/\overline{x}, x/\mathsf{result}]\}}$$

C-Load

$$\begin{array}{c}\{\textbf{emp} \land e = y\} \\ x := \textbf{load}\ e\ \textbf{as}\ \tau \\ \left\{\begin{array}{c} x = 0 \land \textbf{emp}\ \lor \\ x > 0 \land \textbf{lib}(x, y) * \textbf{module}_0(x, y) \land x + 1 : \tau(x, y)\end{array}\right\}\end{array}$$

C-Unload

$$\begin{array}{c}\{\textbf{lib}(e, y) * \textbf{module}(e, y)\} \\ \textbf{unload}(e) \\ \{\textbf{emp}\}\end{array}$$

Fig. 4. Proof rules

Per rule A-FunType, proving a function type assertion $\ell : \tau(\overline{v})$ requires proving a) that the function type's precondition implies that there is some function value with the correct number of parameters at location ℓ, and b) that this function value's body satisfies the function type's contract.

Using this rule and rule A-PredAsn, we can easily prove the assertions $\ell + 3 : \mathsf{read}(\mathsf{Q}(\ell))$ and $\ell + 4 : \mathsf{write}(\mathsf{Q}(\ell))$, which express the correctness of the read and write functions of the example module. Indeed, $\mathsf{Q}(\ell, \mathsf{X})$ implies $\ell + 3 \mapsto \lfloor MR \rfloor * \mathbf{true}$, and it is a straightforward separation logic exercise to verify the body of MR against the contract $\{\mathsf{Q}(\ell, \mathsf{X})\} \cdot \{\mathsf{Q}(\ell, \mathsf{X}) \wedge \mathsf{result} = \mathsf{X}\}$; similarly for the write function.

By additionally using rule A-Module-Fold, we can prove the assertion $\ell + 2 : \mathsf{module\_exit}(\ell + 6, \mathsf{Q}(\ell), \ell, 100)$, where 100 is the name of the example module. This states the correctness of the module exit function of the example module. Finally, using all of these results and rule A-Module-Unfold, we can prove the correctness of the example module: $\ell + 1 : \mathsf{module\_init}(\ell, 100)$.

This correctness condition, which our proof system imposes on all modules in the module repository, justifies Rule C-Load: it states that the module's entry point satisfies the module's contract, instantiated with the address where the module is loaded and the module's name.

Figure 5 shows a proof outline for the main program. This proof makes no assumptions about the module repository, other than that each module satisfies its contract.

```
$\{\mathbf{emp}\}$
$\mathsf{m} := \mathbf{load}\ \mathsf{M}\ \mathbf{as}\ \mathsf{module\_init};$
$\{\mathsf{m} = 0 \wedge \mathbf{emp} \vee \mathbf{lib}(\mathsf{m}, \mathsf{M}) * \mathbf{module}_0(\mathsf{m}, \mathsf{M}) \wedge \mathsf{m} + 1 : \mathsf{module\_init}(\mathsf{m}, \mathsf{M})\}$
$\mathbf{if}\ \mathsf{m} = 0\ \mathbf{then\ skip\ else}\ ($
   $\{\mathbf{lib}(\mathsf{m}, \mathsf{M}) * \mathbf{module}_0(\mathsf{m}, \mathsf{M}) \wedge \mathsf{m} + 1 : \mathsf{module\_init}(\mathsf{m}, \mathsf{M})\}$
   $\mathsf{init} := \mathsf{m} + 1; \mathsf{o} := \mathbf{call}\ \mathsf{init}();$
   $\left\{\begin{array}{l} \mathbf{lib}(\mathsf{m}, \mathsf{M}) * \exists \mathsf{p}, \mathsf{r}, \mathsf{w} \bullet \mathsf{o} \mapsto \mathsf{r} * \mathsf{o} + 1 \mapsto \mathsf{w} * \mathsf{p}(0)\ \wedge \\ \quad \mathsf{r} : \mathsf{read}(\mathsf{p}) \wedge \mathsf{w} : \mathsf{write}(\mathsf{p}) \wedge \mathsf{m} + 2 : \mathsf{module\_exit}(\mathsf{o}, \mathsf{p}, \mathsf{m}, \mathsf{M}) \end{array}\right\}$
   $\mathsf{write} := [\mathsf{o} + 1]; \_ := \mathbf{call}\ \mathsf{write}(42);$
   $\left\{\begin{array}{l} \mathbf{lib}(\mathsf{m}, \mathsf{M}) * \exists \mathsf{p}, \mathsf{r}, \mathsf{w} \bullet \mathsf{o} \mapsto \mathsf{r} * \mathsf{o} + 1 \mapsto \mathsf{w} * \mathsf{p}(42)\ \wedge \\ \quad \mathsf{r} : \mathsf{read}(\mathsf{p}) \wedge \mathsf{w} : \mathsf{write}(\mathsf{p}) \wedge \mathsf{m} + 2 : \mathsf{module\_exit}(\mathsf{o}, \mathsf{p}, \mathsf{m}, \mathsf{M}) \end{array}\right\}$
   $\mathsf{read} := [\mathsf{o}]; \mathsf{x} := \mathbf{call}\ \mathsf{read}();$
   $\left\{\begin{array}{l} \mathbf{lib}(\mathsf{m}, \mathsf{M}) * \exists \mathsf{p}, \mathsf{r}, \mathsf{w} \bullet \mathsf{o} \mapsto \mathsf{r} * \mathsf{o} + 1 \mapsto \mathsf{w} * \mathsf{p}(42)\ \wedge \\ \quad \mathsf{r} : \mathsf{read}(\mathsf{p}) \wedge \mathsf{w} : \mathsf{write}(\mathsf{p}) \wedge \mathsf{m} + 2 : \mathsf{module\_exit}(\mathsf{o}, \mathsf{p}, \mathsf{m}, \mathsf{M}) \wedge \mathsf{x} = 42 \end{array}\right\}$
   $\mathbf{assert}(\mathsf{x} = 42);$
   $\mathsf{exit} := \mathsf{m} + 2; \_ := \mathbf{call}\ \mathsf{exit}();$
   $\{\mathbf{lib}(\mathsf{m}, \mathsf{M}) * \mathbf{module}(\mathsf{m}, \mathsf{M})\}$
   $\mathbf{unload}(\mathsf{m})$
$)$
$\{\mathbf{emp}\}$
```

Fig. 5. Proof outline of the example program

Our proof system is sound: if each module in the module repository and the main program are provably correct, then the main program does not abort.

A full formal treatment of the specification and verification approach is in the extended version of this paper [9]. We developed a mechanically checked proof of its soundness in Coq (see http://www.cs.kuleuven.be/~bartj/unload/).

5 Verification Tool

We implemented the approach in our prototype verifier, VeriFast [8]. VeriFast takes a set of C and Java source files, annotated with preconditions, postconditions, loop invariants, mathematical datatype and function definitions, separation logic predicate definitions, inductive proofs in the form of lemma routines, and in-line explicit proof steps, and then symbolically executes each function in turn, where the symbolic state consists of the symbolic heap, the symbolic store, and the path condition. The symbolic heap is a separating conjunction of *heap chunks* of the form $p(\overline{a})$, where p is a term of first-order logic denoting the name of a separation logic predicate or a partially applied predicate, and $\overline{a}$ are terms denoting the predicate arguments. The symbolic store maps local variable names to terms, and the path condition is a set of formulae that are true on the current execution path. The SMT solver Z3 [6] is used to check boolean conjuncts in assertions, whereas spatial conjuncts, i.e. predicate assertions, are dealt with in the tool itself through simple pattern matching with the symbolic heap.

In order to fit more naturally with the C language, the approach as implemented in VeriFast differs from the formalization in this paper as follows.

Firstly, module assertions unfold to a more abstract representation of the module image. In particular, all code of a module M is represented using a built-in predicate $\mathbf{code}(M)$. For example:

$$\mathbf{module}_0(\mathsf{RamDisk}) = \mathbf{code}(\mathsf{RamDisk}) * \mathit{vector} \mapsto 0 * \mathit{o.read} \mapsto 0 * \mathit{o.write} \mapsto 0$$

Note that a VeriFast module name such as RamDisk denotes both the address where the module was loaded and the module name proper, which identifies its contents.

Secondly, the derivation of a function type judgment $f : \tau(\overline{v})$ for a function f declared in an unloadable module M is split into two steps. In a first step, f is verified with respect to its declared contract, using the following proof rule.

$$\frac{P \Rightarrow [\pi]\mathbf{code}(M) * P' \qquad \{P'\}\ c\ \{Q'\} \qquad [\pi]\mathbf{code}(M) * Q' \Rightarrow Q}{M \vdash \mathbf{fun}\ f(\overline{x})\ \mathbf{req}\ P\ \mathbf{ens}\ Q\ \mathbf{do}\ c}$$

That is, the declared precondition P must imply some fractional (i.e., read-only) permission π for the module's code, and the function's body c is verified against the remainder of the precondition. The removed code fraction is added back to verify the declared postcondition Q.

In a second step, VeriFast simply verifies that the declared contract implies the function type contract:

$$\frac{\begin{array}{c}\mathbf{fun}\ f(\overline{x})\ \mathbf{req}\ P\ \mathbf{ens}\ Q\ \mathbf{do}\ c\\ \mathbf{funtype}\ \tau(\overline{y})(\overline{x})\ \mathbf{req}\ P'\ \mathbf{ens}\ Q' \qquad P'[\overline{v}/\overline{y}] \Rightarrow P \qquad Q \Rightarrow Q'[\overline{v}/\overline{y}]\end{array}}{f : \tau(\overline{v})}$$

Notice that this latter step does not need to take unloadability into account.

We used the approach of this paper to verify a small multithreaded server that allows clients to concurrently load DLLs, unload DLLs, and use services provided by the DLLs, mimicking operating system kernels that may dynamically load and unload device drivers. Specifically, a loaded DLL can register "devices" with the "kernel", by supplying function pointers that open, read, write, and close the device. Multiple concurrent clients can then access these devices. A reference counting mechanism prevents a module from being unloaded while devices provided by the module are open; furthermore, it is verified that a module unregisters all devices it registered before it unloads. Considering the complexity of this program, the annotation overhead is not excessive: 467 lines of annotations for 245 lines of code. Our tool verifies this program in 0.75 seconds. Website: http://www.cs.kuleuven.be/~bartj/verifast/.

6 Conclusion and Related Work

We presented the first approach for integrating verification of unloadable modules into a semi-automatic verification tool for C. It supports global variables, and its approach to function pointers enables mixing unloadable and non-unloadable code transparently. We reported on verifying a small but complex kernel-like program using our implementation. A formalization, a machine-checked soundness proof, the implementation, and the verified code are available on line.

Cai *et al.* [3] propose a separation-logic-based approach for verifying self-modifying assembly language programs. They formalize a generic target machine in Coq, and instantiate it with x86 and MIPS. They prove a number of assembly programs, including a boot loader, that loads a kernel image from disk and executes it. Their proof rule for well-formedness of a code block with respect to a given precondition is very similar to our A-FunType rule: the precondition must imply the presence of some code that executes safely. Their soundness proof uses a notion of safety for n steps, very similar to ours. A more recent work in a similar vein is Myreen's [10] on verifying a just-in-time compiler on x86, this time using Isabelle/HOL. Our main contribution with respect to these results is to adapt these ideas to C's system of modules, and to integrate it into a program verification tool for C.

Another, more theoretical, line of work on separation logic for stored code goes under the name of separation logic for *higher-order store* [13, 2, 15]. Instead of representing stored code as an integer-encoded syntactic lambda expression or machine instruction sequence, the authors adopt a higher-order heap: the heap maps addresses to integers and to *semantic commands*, which are themselves relations from heaps to heaps. This recursive domain equation is solved using techniques from category theory and domain theory, leading to a less accessible formalization. Another difference in focus is that the authors attempt not just to abstract over state required by stored code, e.g. using abstract predicates, but to *hide* such requirements, using *higher-order frame rules.* Whereas such hiding is preferable over mere abstraction, there are unsolved problems of modularity [12], and furthermore concurrency is not currently supported. Our contribution here is an instantiation in the context of C's modules, and a simpler formalization.

Acknowledgements

The authors would like to thank Raoul Strackx for helpful comments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven, and by the EU FP7 project SecureChange.

Bibliography

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.

[2] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. A simple model of separation logic for higher-order store. In *ICALP*, 2008.

[3] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *PLDI*, 2007.

[4] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proc. TPHOLs*, 2009.

[5] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In *POPL*, 2009.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[7] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, 2007.

[8] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, 2010.

[9] Bart Jacobs, Jan Smans, and Frank Piessens. Verification of unloadable modules (Extended version). Technical Report CW604, Dept. Computer Science, Katholieke Universiteit Leuven, March 2011.

[10] Magnus O. Myreen. Verified just-in-time compiler on x86. In *POPL*, 2010.

[11] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.

[12] François Pottier. Three comments on the anti-frame rule. Unpublished note, July 2009.

[13] Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In *Computer Science Logic*, 2006.

[14] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS 2002*, 2002.

[15] Jab Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, 2009.