

Verification of Unloadable C Modules—Status Report

Bart Jacobs Jan Smans Frank Piessens

Report CW567, October 2009



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verification of Unloadable C Modules—Status Report

Bart Jacobs Jan Smans Frank Piessens

Report CW 567, October 2009

Department of Computer Science, K.U.Leuven

Abstract

C programs 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 propose a separation-logic-based approach for the verification of such programs and modules. We propose proof rules for loading and unloading modules, and for dealing with pointers to functions in unloadable modules, that ensure soundness while imposing minimal verification overhead. We offer a formalization and we report on verifying a small kernel-like program using a prototype implementation of the approach in our verifier, VeriFast. To the best of our knowledge, ours is the first approach for sound modular verification of unloadable modules.

Verification of Unloadable C Modules

Status Report

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. C programs 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 propose a separation-logic-based approach for the verification of such programs and modules. We propose proof rules for loading and unloading modules, and for dealing with pointers to functions in unloadable modules, that ensure soundness while imposing minimal verification overhead. We offer a formalization and we report on verifying a small kernel-like program using a prototype implementation of the approach in our verifier, VeriFast. To the best of our knowledge, ours is the first approach for sound modular verification of unloadable modules.

1 Introduction

In statically typed safe programming languages (including Java, C#, the ML family, and Haskell), code is immutable and permanent.¹ 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 like LISP, Scheme, JavaScript, Ruby, or Python, and in unsafe languages like C and C++. In C, if at one point during execution a function pointer points to a function that satisfies a given contract, this does not mean it always will. The module (DLL, shared object, ...) containing the function 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 (VCC [4], Frama-C [1], HAVOC [5], Smallfoot [2], our own verifier VeriFast [7], Jahob [9]) assume that the program is unchanging and is not part of the mutable state. As a result, these

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

¹ Some of these languages may garbage collect code that is unreachable, but this is unobservable.

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 programs involving code unloading. The approach is simple: execution of unloadable code at a given address requires a permission to read at the address and a proof that the code at this address has the expected behavior. Specifically, execution of the body of a function in an unloadable module requires an abstract permission that states that the module is currently loaded.

The conventional approach for verifying dynamically bound calls in separation logic is through predicate families [8], i.e. abstract predicates indexed by the target function. VeriFast, for example, has used predicate families indexed by function pointer to verify function pointer calls. However, this is not applicable to unloadable code since the function pointer no longer immutably identifies a specific function. Therefore, in our approach, we drop the use of predicate families and instead we use parameterized function types combined with higher-order predicates.

We implemented the approach in our prototype verifier, VeriFast, and we verified a small server 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. The implementation and the example are online at <http://www.cs.kuleuven.be/~bartj/verifast/>.

The rest of the paper is structured as follows. First, in Section 2, we show how programs involving dynamic code loading, but not unloading, may be verified in VeriFast. Then, in Section 3, we present the extensions required to enable verification of code unloading. In Section 4 we further extend the example to get a fully abstract treatment. We provide a formal treatment in Section 5. We discuss the implementation in Section 6 and related work in Section 8. We conclude in Section 9.

2 Dynamically loaded code

The C program in Figure 1 illustrates dynamic code loading. It is safe, and VeriFast can confirm this thanks to the annotations shown on a gray background.

Execution proceeds as follows. The main program, `a.c`, dynamically loads the shared object `b.so`, generated from `b.c`, and looks up the `getIncr` function in that object. If the function is not found, the program aborts. Otherwise, it calls the function to retrieve another function, which it then calls as well.

The soundness of the verification approach used in this example, is based on the assumption that the mapping from function names to function types (with their associated contract), as specified in `b.spec` in the example, is globally unique. This seems realistic, especially if sufficiently long and distinctive names are chosen, and if the origin of the code is trusted, e.g. through code signing. An alternative approach would be to submit the module to a verification tool at load time.

```

// dlfcn.h
void *dlopen(char *name);
    requires string(name);
    ensures string(name) * lib(result); // b.spec
                                        getIncr : b.h#getIncrType

// b.h
typedef int incrType(int x);
    requires true;
    ensures result - 1 = x;

typedef incrType *getIncrType();
    requires true;
    ensures is_incrType(result);

// a.c
#include "dlfcn.h"
#include "b.h"
#include "b_proxy.h"

void main()
    requires true;
    ensures true;
{
    void *ℓ = dlopen("b.so");
    getIncrType *f = dlsym_getIncr(ℓ);
    if (f == 0) abort();
    incrType *g = f();
    int y = g(41);
    assert(y == 42);
}

// b_proxy.h (generated from b.spec)
#include "b.h"

getIncrType *dlsym_getIncr(void *ℓ);
    requires lib(ℓ);
    ensures lib(ℓ) ∧ (result == 0 ? true :
        is_getIncrType(result));

// b_proxy.c (generated from b.spec)
getIncrType *dlsym_getIncr(void *ℓ)
{ return dlsym(ℓ, "getIncr"); }

// b.c
#include "b.h"

int myIncr(int x) : incrType
{ return x + 1; }

incrType *getIncr()
: getIncrType
{ return myIncr; }

```

Fig. 1. An example of dynamic loading, but not unloading, in C. VeriFast annotations are shown on a gray background

3 Code unloading

Suppose we added the following contract to `dlfcn.h` in Figure 1:

```
void dlclose(void * $\ell$ );  
  requires lib( $\ell$ );  
  ensures true;
```

Clearly, this would be unsound, since this contract would allow the main program to unload the library and then perform calls through function pointers that point into the library.

To enable unloading, we extend the approach with four ingredients. Firstly, we add *module names* as first-class values inside annotations. For example, the module name of `b.c` is `b` and this name may be used as an expression in annotations. Secondly, we add a built-in predicate *module*, that takes a module name and states that this module is currently loaded. Thirdly, a module may declare itself to be **unloadable**. The precondition of each function of an unloadable module m must imply $module(m)$, and this permission is unavailable for the duration of the function’s execution. Also, each module that exports symbols for dynamic linking must be unloadable.² Finally, we allow function types to be parameterized by arbitrary values, including module names.

In Figure 2, we show the example of Figure 1, modified so that the main program now unloads the module after performing the function calls. The contracts have been updated to include the *module* permission.

4 Abstraction

The example of Figure 2 is lacking in abstraction. For example, suppose we want to write a module that logs calls to the *incr* function. This is not possible given the precondition of *incrType* in `b.h`, which provides access only to the module image.

To enable this implementation, we parameterize *incrType* not by a module name, but by an arbitrary predicate P . Further, in order to allow the logging infrastructure to be torn down after the client is done using the module, the *getIncr* function returns not just a pointer to the *incr* function, but a struct containing both a pointer to the *incr* function and a pointer to a *dispose* function.

The implementation, with annotations, is shown in Figure 3. The implementation now uses global variables. To support these, we extend the verification approach with three new ingredients. Firstly, we add another built-in predicate *code*, which takes a module name and represents the code of the named module. Secondly, the *module* predicate now represents not just the module’s code but its global variables as well. That is, it is the separate conjunction of the code and the globals. Thirdly, we introduce a variant of the *module* predicate,

² This may be relaxed by specifying in the `spec` file whether the module is unloadable or not, and adapting the proxy contract accordingly

```

// dlfcn.h
void *dlopen(char *name);
  requires string(name);
  ensures string(name) *
     $\exists m \bullet \text{lib}(\text{result}, m) * \text{module}(m)$ ;

void dlclose(void * $\ell$ );
  requires  $\exists m \bullet \text{lib}(\ell, m) * \text{module}(m)$ ;
  ensures true;

// b.h
typedef int incrType(m)(int x);
  requires module(m);
  ensures module(m)  $\wedge$  result - 1 = x;

typedef incrType *getIncrType(m)();
  requires module(m);
  ensures module(m)  $\wedge$ 
    is_incrType(result, m);

// a.c
#include "dlfcn.h"
#include "b.h"
#include "b_proxy.h"

void main()
  requires true;
  ensures true;
{
  void * $\ell$  = dlopen("b.so");
  getIncrType *f = dlsym_getIncr( $\ell$ );
  if (f = 0) abort();
  incrType *g = f();
  int y = g(41);
  dlclose( $\ell$ );
  assert(y = 42);
}

// b.spec
getIncr : b.h#getIncrType

// b_proxy.h (generated from b.spec)
#include "b.h"

getIncrType *dlsym_getIncr(void * $\ell$ );
  forall m;
  requires lib( $\ell, m$ );
  ensures lib( $\ell, m$ )  $\wedge$  (result = 0  $\vee$ 
    is_getIncrType(result, m));

// b.c
#include "b.h"
unloadable;

int myIncr(int x) : incrType(b)
{ return x + 1; }

incrType *getIncr()
  : getIncrType(b)
{ return myIncr; }

```

Fig. 2. The example of Figure 1, with unloading added, but lacking in abstraction

named $module_0$, which represents the same state, but additionally states that the globals hold their initial value. The contract of $dlopen$ is updated to provide $module_0$.

5 Formal System

In this section, we formalize the approach and state the soundness theorem. A detailed soundness proof is future work.

The rest of this section is structured as follows. In Section 5.1, we introduce the formal programming language and execution model, and we illustrate it using an example program and module. In Section 5.2, we define the syntax and the semantics of the specification language, and we show the specification for the example module. Finally, in Section 5.3, we introduce the proof system, we state its soundness theorem, and we show a proof outline for the example module.

5.1 Language Syntax and Semantics

The formal programming language is an extension of the standard separation logic language 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. Its syntax is as follows:

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

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. In order to be able to store function values in the heap, we assume a one-to-one encoding $[\cdot]$ of function values into integers.

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 definition consists of a module contract and a module image. We assume each module has a single entry point; therefore, the module contract is simply the function contract for the single entry point. We assume a set $FunTypes$ of function type names, and a mapping from function type names to contracts. The run-time semantics is concerned only with the names and not the meanings of the function types. It uses them to perform a run-time type check. The module image is simply a tuple of one or more integers. The first element of the tuple is the encoded code for the module's entry point; the other

```

// b.h
typedef int incrType (P) (int x);
    requires P;
    ensures P ∧ result - 1 = x;

typedef void disposeType (o, P, m) ();
    requires obj(o, -, -) * P;
    ensures module(m);

struct obj {
    incrType *incr;
    disposeType *dispose;
};

predicate obj(o, d, p) =
    o → incr ↦ p * o → dispose ↦ d;

typedef struct obj *
getIncrType (m) ();
    requires module0(m);
    ensures
        ∃P, d, p • obj(result, P, m, d, p) * P
        ∧ is_incrType(p, P)
        ∧ is_disposeType(d, result, P, m);

// a.c
#include "dlfcn.h"
#include "b.h"
#include "b_proxy.h"

void main()
    requires true; ensures true;
{
    void *ℓ = dlopen("b.so");
    getIncrType *f =
        dlsym_getIncr(ℓ);
    if (f = 0) abort();
    struct obj *o = f();
    int y = o → incr(41); assert(y = 42);
    o → dispose(); dlclose(ℓ)
}

// dlfcn.h
void *dlopen(char *name);
    requires string(name);
    ensures string(name) *
        ∃m • lib(result, m) * module0(m);

void dlclose(void *ℓ);
    requires ∃m • lib(ℓ, m) * module(m);
    ensures true;

// b.spec
getIncr : b.h#getIncrType

// b_proxy.h (generated from b.spec)
#include "b.h"

getIncrType *dlsym_getIncr(void *ℓ);
    forall m;
    requires lib(ℓ, m);
    ensures lib(ℓ, m) ∧ (result = 0 ∨
        is_getIncrType(result, m));

// b.c
#include "b.h"
#include "logging.h"
unloadable;

struct logchannel *chan;

predicate Q = code(b) *
    ∃c • chan ↦ c * logchannel(c);

int myIncr(int x) : incrType(Q)
{ logCall(chan); return x + 1; }

struct obj o = {myIncr, myDispose};

void myDispose() : disposeType(o, Q, b)
{ disposeLogChannel(chan); }

struct obj *getIncr()
    : getIncrType(b)
{ chan = allocLogChannel(); return o; }

```

Fig. 3. The example of Figure 2, with better abstraction

elements may be code or data. We assume a one-to-one encoding $[\cdot]$ of module names to integers.

The dynamic semantics is given by the step rules in Figure 4. A run-time configuration consists of a state (i.e. a store and a heap) and a continuation. This is either a done continuation, denoting the successful termination of the program, a return continuation, denoting a caller stack frame, or a command continuation:

$$\kappa ::= \mathbf{done} \mid \mathbf{ret}(x, s, \kappa) \mid c; \kappa$$

A return continuation specifies the variable x that will receive the return value, and the saved store.

Most rules are standard. A function pointer call $x := \mathbf{call} e(\bar{e})$ looks for an encoded function value $\mathbf{lambda}(\bar{x}) c$ at the address given by e . It gets stuck if the address is not allocated, if the value at the address does not encode a function value, or if the length of the argument list does not match the length of the parameter list. Otherwise, it pushes a return continuation and executes the body of the function value under a store that binds the parameters to the corresponding arguments and the variable \mathbf{ip} (for *instruction pointer*) to the address given by e .

When the body of a function value completes, the caller's store is restored, and the return value of the function value, written by convention into variable \mathbf{result} , is assigned to the target variable of the call.

A command $x := \mathbf{load} e \mathbf{as} \tau$, where τ is a function type name, fails if there is no module whose name is encoded by the value of e , or if there is one but its contract is not τ . In that case, the command returns zero. Otherwise, it allocates $m + 1$ consecutive memory locations, where m is the size of the module image. It writes the size itself into the first location, and the module image into the subsequent locations. It returns the address of the first location. As a result, the module's entry point is at the returned address plus one.

Command $\mathbf{unload}(e)$ gets stuck if the memory location at the address given by e is not allocated, holds a negative value, or holds a value m such that some of the m next memory locations are not allocated. Otherwise, it deallocates these $m + 1$ memory locations.

Figure 5 shows an example main program and module repository holding a single module, called `myIncrLib`. The main program starts by loading the module whose name is given by the initial value of variable `libName`, which is assumed to be provided by the user; i.e., it is arbitrary. It then calls the loaded module's entry point, which it assumes returns a pointer to a struct that has two fields, the first of which points to a function that returns its argument incremented by one, and the second one serves to clean up any resources used by the module. It first calls the incrementor function, and asserts that it indeed incremented its argument. (It performs a null pointer dereference if it did not.) It then calls the dispose function, and finally unloads the module.

The example module's image consists of six values, the first three of which are encoded function values, and the latter will serve as global variables. To illustrate that our approach performs strong abstraction and allows the module

$$\begin{array}{c}
\text{CONS} \\
\frac{0 < \ell \quad \text{dom}(h) \cap \{\ell, \dots, \ell + m - 1\} = \emptyset}{\langle s, h, x := \mathbf{cons}(e_1, \dots, e_m); \kappa \rangle \rightsquigarrow \langle s[x := \ell], h[\ell := \llbracket e_1 \rrbracket_s, \dots, \ell + m - 1 := \llbracket e_m \rrbracket_s], \kappa \rangle} \\
\\
\text{LOOKUP} \qquad \frac{(\llbracket e \rrbracket_s, v) \in h}{\langle s, h, x := [e]; \kappa \rangle \rightsquigarrow \langle s[x := v], h, \kappa \rangle} \qquad \text{MUTATE} \qquad \frac{\llbracket e \rrbracket_s \in \text{dom}(h)}{\langle s, h, [e] := e'; \kappa \rangle \rightsquigarrow \langle s, h[\llbracket e \rrbracket_s := \llbracket e' \rrbracket_s], \kappa \rangle} \\
\\
\text{DISPOSE} \qquad \frac{(\llbracket e \rrbracket_s, v) \in h}{\langle s, h, \mathbf{dispose}(e); \kappa \rangle \rightsquigarrow \langle s, h \setminus \{(\llbracket e \rrbracket_s, v)\}, \kappa \rangle} \qquad \text{ASSIGN} \qquad \frac{}{\langle s, h, x := e; \kappa \rangle \rightsquigarrow \langle s[x := \llbracket e \rrbracket_s], h, \kappa \rangle} \\
\\
\text{IFTRUE} \qquad \frac{\llbracket b \rrbracket_s = \mathbf{true}}{\langle s, h, \mathbf{if } b \mathbf{ then } c \mathbf{ else } c'; \kappa \rangle \rightsquigarrow \langle s, h, c; \kappa \rangle} \\
\\
\text{IFFALSE} \qquad \frac{\llbracket b \rrbracket_s = \mathbf{false}}{\langle s, h, \mathbf{if } b \mathbf{ then } c \mathbf{ else } c'; \kappa \rangle \rightsquigarrow \langle s, h, c'; \kappa \rangle} \qquad \text{SEQ} \qquad \frac{}{\langle s, h, (c; c'); \kappa \rangle \rightsquigarrow \langle s, h, c; (c'; \kappa) \rangle} \\
\\
\text{CALL} \qquad \frac{(\llbracket e \rrbracket_s, [\mathbf{lambda} (\bar{x}) c]) \in h \quad |\bar{x}| = |\bar{e}|}{\langle s, h, x := \mathbf{call } e(\bar{e}); \kappa \rangle \rightsquigarrow \langle (\lambda _ \bullet 0)[\text{ip} := \llbracket e \rrbracket_s][\bar{x} := \bar{e}], h, c; \mathbf{ret}(s, x, \kappa) \rangle} \\
\\
\text{RETURN} \qquad \frac{}{\langle s, h, \mathbf{ret}(s', x, \kappa) \rangle \rightsquigarrow \langle s'[x := s(\mathbf{result})], h, \kappa \rangle} \\
\\
\text{LOADSUCCESS} \qquad \frac{(\llbracket e \rrbracket_s, (\tau, (v_1, \dots, v_m))) \in \text{Modules} \quad 0 < \ell \quad \text{dom}(h) \cap \{\ell, \dots, \ell + m\} = \emptyset}{\langle s, h, x := \mathbf{load } e \mathbf{ as } \tau; \kappa \rangle \rightsquigarrow \langle s[x := \ell], h[\ell := m, \ell + 1 := v_1, \dots, \ell + m := v_m], \kappa \rangle} \\
\\
\text{LOADFAIL} \qquad \frac{\neg \exists \bar{v} \bullet (\llbracket e \rrbracket_s, (\tau, (\bar{v}))) \in \text{Modules}}{\langle s, h, x := \mathbf{load } e \mathbf{ as } \tau; \kappa \rangle \rightsquigarrow \langle s[x := 0], h, \kappa \rangle} \\
\\
\text{UNLOAD} \qquad \frac{\llbracket e \rrbracket_s = \ell \quad \{(\ell, m), (\ell + 1, v_1), \dots, (\ell + m, v_m)\} \subseteq h}{\langle s, h, \mathbf{unload}(e); \kappa \rangle \rightsquigarrow \langle s, h \setminus \{(\ell, m), (\ell + 1, v_1), \dots, (\ell + m, v_m)\}, \kappa \rangle}
\end{array}$$

Fig. 4. Step rules

flexibility in implementation, the module’s incremator function does not simply increment its argument, but additionally tracks the number of calls made to it in a dynamically allocated cell. The cell is allocated by the module’s entry point, and its address is stored in the third global variable. The first two global variables will serve as the struct that is returned to the client by the module’s entry point. The entry point writes a pointer to the incremator function and the dispose function into the struct, allocates the counter cell, and then returns the address of the struct. The incremator function, whose code resides in the second element of the module image, increments the counter cell and then returns its argument plus one. The dispose function, which resides in the third position, simply disposes the counter cell.

```

Modules = {
  (myIncrLib, (getIncrType, (
    [lambda () [ip + 3] := ip + 1;
      [ip + 4] := ip + 2; x := cons(0); [ip + 5] := x; result := ip + 3],
    [lambda (x) c := [ip + 4]; n := [c]; [c] := n + 1; result := x + 1],
    [lambda () x := [ip + 3]; dispose(x)],
    0,
    0,
    0
  )))
}
main =
  ℓ := load libName as getIncrType;
  if ℓ = 0 then x := x else (
    getIncr := ℓ + 1; o := call getIncr();
    incr := [o]; r := call incr(42);
    if r = 43 then x := x else [0] := 0; // assert(r = 43)
    dispose := [o + 1]; dummy := call dispose();
    unload ℓ
  )

```

Fig. 5. Example program in the formal language

5.2 Specification Language

The specification language is an extension of separation logic with special-purpose predicates **lib**, **module₀**, and **module**, for dealing with module loading and unloading, as well as with parameterized function types and assertion closures for dealing with dynamic code more generally.

Expressions in assertions may be constants, program variables, logical variables, operator applications, and assertion closure expressions. Evaluation of an

assertion closure expression yields an assertion closure C , which records the interpretation I of the free logical variables of the assertion and the store s that provides the value of the free program variables of the assertion.

The assertion $E : \tau(\overline{E})$ asserts that the address denoted by E satisfies the function type τ instantiated with arguments \overline{E} . The assertion closure application expression $E(\overline{E})$ applies arguments \overline{E} to the assertion closure denoted by E . $\mathbf{lib}(E, E')$ denotes the management information (i.e. the size field) for the loaded module named E' at address E . $\mathbf{module}_0(E, E')$ asserts that the module image of the module named E' is at address $E + 1$ and following, and it is in its initial state. $\mathbf{module}(E, E')$ asserts that there are m allocated memory locations at addresses $E + 1$ and following, where m is the size of the module named E' .

Assertions are interpreted in the context of a set of function type declarations and named predicate declarations. A function type declaration specifies a function type name, a list of function type parameters, a list of function parameters, a precondition, and a postcondition. A predicate declaration specifies a predicate name, a list of parameters, and a body.

$$\begin{aligned}
& n \in \mathbb{Z}, x \in \text{ProgVars}, y \in \text{LogVars}, op \in \text{Operators} \\
E & ::= n \mid x \mid y \mid op(\overline{E}) \mid \mathbf{lambda}(\overline{y}) A \\
C & ::= \mathbf{lambda}_{I,s}(\overline{y}) A \\
A & ::= \mathbf{emp} \mid E \mapsto E \mid E = E \mid A \wedge A \mid A \vee A \mid A * A \mid \exists y \bullet A \\
& \quad \mid E : \tau(\overline{E}) \mid p(\overline{E}) \mid E(\overline{E}) \\
& \quad \mid \mathbf{lib}(E, E) \mid \mathbf{module}_0(E, E) \mid \mathbf{module}(E, E) \\
ftdecl & ::= \mathbf{funtype} \tau(\overline{y})(\overline{x}) \mathbf{req} A \mathbf{ens} A \\
pdecl & ::= \mathbf{predicate} p(\overline{y}) = A
\end{aligned}$$

The semantics of assertion expressions, assertions, and function type judgments is given in Figure 6. The figure uses the following auxiliary notions derived from the semantics of the programming language.

$$\begin{aligned}
\mathbf{Stuck} & = \{\gamma \mid \neg \exists \gamma' \bullet \gamma \rightsquigarrow \gamma'\} \\
\mathbf{Bad} & = \{\gamma \mid \exists \gamma' \in \mathbf{Stuck} \bullet \gamma \rightsquigarrow^* \gamma'\} \\
\mathbf{Pre}(c) & = \{(s, h) \mid \langle s, h, c; \mathbf{done} \rangle \notin \mathbf{Bad}\} \\
\mathbf{Post}(c) & = \{((s, h), (s', h')) \mid \langle s, h, c; \mathbf{done} \rangle \rightsquigarrow^* \langle s', h', \mathbf{done} \rangle\}
\end{aligned}$$

We consider a configuration γ to be bad if it can lead to a stuck configuration. The precondition of a command is the set of initial states that will not lead to a stuck configuration. The postcondition relates the pre-states and the post-states.

A function type judgment $\ell : \tau(\overline{v})$ holds, i.e., an address ℓ satisfies function type τ instantiated with arguments \overline{v} , if and only if the instantiated precondition of the function type implies that there is a function value at address ℓ that satisfies the function type's instantiated contract.

In general, the equations of Figure 6 do not qualify as definitions, since they may have multiple solutions, or no solution at all. To avoid this, we impose the following restrictions. We impose an order on function types and predicates, so that each of these may mention only earlier ones in its definition. Further, we do not allow assertion closure applications inside assertion closures. We believe

$$\begin{aligned}
\llbracket x \rrbracket_{I,s} &= s(x) \\
\llbracket y \rrbracket_{I,s} &= I(y) \\
\llbracket op(\overline{E}) \rrbracket_{I,s} &= \llbracket op \rrbracket(\llbracket \overline{E} \rrbracket_{I,s}) \\
\llbracket \mathbf{lambda} (\overline{y}) A \rrbracket_{I,s} &= \llbracket \mathbf{lambda} \rrbracket_{\text{freeLogVar}(A) \setminus \{\overline{y}\}, s | \text{freeProgVar}(A)} (\overline{y}) A
\end{aligned}$$

$$\begin{aligned}
I, s, h \models \mathbf{emp} &\Leftrightarrow h = \emptyset \\
I, s, h \models E \mapsto E' &\Leftrightarrow h = \{(\llbracket E \rrbracket_{I,s}, \llbracket E' \rrbracket_{I,s})\} \\
I, s, h \models E = E' &\Leftrightarrow \llbracket E \rrbracket_{I,s} = \llbracket E' \rrbracket_{I,s} \\
I, s, h \models A \wedge A' &\Leftrightarrow I, s, h \models A \wedge I, s, h \models A' \\
I, s, h \models A \vee A' &\Leftrightarrow I, s, h \models A \vee I, s, h \models A' \\
I, s, h \models A * A' &\Leftrightarrow \exists h_1, h_2 \bullet h = h_1 \uplus h_2 \wedge I, s, h_1 \models A \wedge I, s, h_2 \models A' \\
I, s, h \models \exists y \bullet A &\Leftrightarrow \exists n \in \mathbb{Z} \bullet I[y := n], s, h \models A \\
I, s, h \models E : \tau(\overline{E}) &\Leftrightarrow \llbracket E \rrbracket_{I,s} : \tau(\llbracket \overline{E} \rrbracket_{I,s}) \\
I, s, h \models p(\overline{E}) &\Leftrightarrow \exists \overline{y}, A \bullet (\mathbf{predicate} \ p(\overline{y}) = A) \\
&\quad \wedge (\lambda _ \bullet 0)[\overline{y} := \llbracket \overline{E} \rrbracket_{I,s}], (\lambda _ \bullet 0), h \models A \\
I, s, h \models E(E_1, \dots, E_m) &\Leftrightarrow \exists I', s', y_1, \dots, y_m, A \bullet \llbracket E \rrbracket_{I,s} = \llbracket \mathbf{lambda}_{I',s'} (y_1, \dots, y_m) A \rrbracket \\
&\quad \wedge I'[y_1 := \llbracket E_1 \rrbracket_{I,s} \cdots [y_m := \llbracket E_m \rrbracket_{I,s}], s', h \models A \\
I, s, h \models \mathbf{lib}(E, E') &\Leftrightarrow \exists (M, (\tau, (v_1, \dots, v_m))) \in \mathbf{Modules} \bullet \\
&\quad \llbracket E' \rrbracket_{I,s} = \llbracket M \rrbracket \wedge h = \{(\llbracket E \rrbracket_{I,s}, m)\} \\
I, s, h \models \mathbf{module}_0(E, E') &\Leftrightarrow \exists (M, (\tau, (v_1, \dots, v_m))) \in \mathbf{Modules}, \ell \in \mathbb{Z} \bullet \\
&\quad \llbracket E \rrbracket_{I,s} = \ell \wedge \llbracket E' \rrbracket_{I,s} = \llbracket M \rrbracket \wedge \\
&\quad h = \{(\ell + 1, v_1), \dots, (\ell + m, v_m)\} \\
I, s, h \models \mathbf{module}(E, E') &\Leftrightarrow \exists (M, (\tau, (v_1, \dots, v_m))) \in \mathbf{Modules}, \ell, v'_1, \dots, v'_m \in \mathbb{Z} \bullet \\
&\quad \llbracket E \rrbracket_{I,s} = \ell \wedge \llbracket E' \rrbracket_{I,s} = \llbracket M \rrbracket \wedge \\
&\quad h = \{(\ell + 1, v'_1), \dots, (\ell + m, v'_m)\}
\end{aligned}$$

$$\begin{aligned}
\ell : \tau(\overline{v}) &\Leftrightarrow \\
\forall I, s, h \bullet I(\overline{y}) = \overline{v} \wedge I, s, h \models P &\Rightarrow \\
\exists L, c \bullet (\ell, \llbracket L \rrbracket) \in h \wedge L \approx \mathbf{lambda} (\overline{x}) c & \\
\wedge (s, h) \in \mathbf{Pre}(c) & \\
\wedge \forall s', h' \bullet ((s, h), (s', h')) \in \mathbf{Post}(c) &\Rightarrow \\
I, s[\mathbf{result} := s'(\mathbf{result})], h' \models Q & \\
\text{where } \mathbf{funtype} \ \tau(\overline{y})(\overline{x}) \ \mathbf{req} \ P \ \mathbf{ens} \ Q &
\end{aligned}$$

Fig. 6. Semantics of assertions and function type judgments. \approx denotes equality of function values up to alpha conversion.

this ensures well-definedness of the semantics. Furthermore, they are sufficient for the programs we considered. Relaxing these restrictions is future work.

Figure 7 shows the definition of the contract of the example module of Figure 5. The entry point's contract `getIncrType` requires the module's image in its initial state, and returns the struct containing the pointer to the incrementor function and the dispose function, as well as the remainder of the module's state in the form of existentially quantified assertion closure value P . The incrementor function type `incrType` simply requires and ensures the module state P . The dispose function type `disposeType` takes back the struct with the function pointers and the module state, and returns the module image, ready for unloading.

```

funtype incrType( $P$ )( $x$ )
  req  $P$ 
  ens  $P \wedge \text{result} = x + 1$ 

funtype disposeType( $o, P, \ell, m$ )()
  req  $o \mapsto \_ * o + 1 \mapsto \_ * P$ 
  ens module( $\ell, m$ )

funtype getIncrType( $\ell, m$ )()
  req module0( $\ell, m$ )
  ens  $\exists P, p, d \bullet \text{result} \mapsto p * \text{result} + 1 \mapsto d * P$ 
       $\wedge p : \text{incrType}(P) \wedge d : \text{disposeType}(o, P, \ell, m)$ 

```

Fig. 7. Definition of function type `getIncrType`

5.3 Proof System

The proof system is an extension of the proof rules of separation logic with proof rules for deriving function type judgments, and for verifying function pointer call and module load and unload commands, and for folding and unfolding the module assertions. The new proof rules are shown in Figure 8.

Rule C-FUNTYPE allows one to derive a function type judgment guarded by a pure boolean expression ϕ . The latter serves to express constraints on the free logical variables that appear in the judgment. The other rules are straightforward.

We can now define validity of a module.

Definition 1. *A module is valid if its entry point satisfies the function type given by its contract, instantiated with the address where the module is loaded and the module name.*

$$(M, (\tau, \bar{v})) \in \text{Modules} \Rightarrow (\text{valid}(M) \Leftrightarrow \vdash \forall \ell \bullet \ell + 1 : \tau(\ell, M))$$

$$\begin{array}{c}
\text{C-FUNTYPE} \\
\frac{\text{funtype } \tau(\bar{y})(\bar{x}) \text{ req } P \text{ ens } Q \\
\text{freeLogVars}(\phi, E, \bar{E}) \cap \text{freeLogVars}(\bar{y}, P, Q) = \emptyset \quad \text{freeProgVars}(\phi, E, \bar{E}) = \emptyset \\
\phi \wedge P[\bar{E}/\bar{y}] \Rightarrow E \mapsto [\mathbf{lambda} (\bar{x}) c] * \mathbf{true} \quad \{\phi \wedge P[\bar{E}/\bar{y}] \wedge \text{ip} = E\} c \{Q\}}{\phi \Rightarrow E : \tau(\bar{E})} \\
\\
\text{C-CALL} \\
\frac{\text{funtype } \tau(\bar{y})(\bar{x}) \text{ req } P \text{ ens } Q \\
\{e : \tau(\bar{y}) \wedge \bar{e} = \bar{y}' \wedge P[\bar{y}'/\bar{x}]\} x := \mathbf{call} e(\bar{e}) \{Q[\bar{y}'/\bar{x}, x/\text{result}]\}}{\text{C-CALL}} \\
\\
\text{C-LOAD} \\
\frac{\{\mathbf{emp} \wedge e = y\} \\
x := \mathbf{load} e \text{ as } \tau \\
\{x = 0 \wedge \mathbf{emp} \vee 0 < x \wedge \mathbf{lib}(x, y) * \mathbf{module}_0(x, y) \wedge x + 1 : \tau(x, y)\}}{\text{C-LOAD}} \\
\\
\text{C-MODULE-UNFOLD} \\
\frac{(M, (\tau, (v_1, \dots, v_m))) \in \text{Modules}}{\mathbf{module}_0(y, M) \Rightarrow y + 1 \mapsto v_1 * \dots * y + m \mapsto v_m} \\
\\
\text{C-MODULE-FOLD} \\
\frac{(M, (\tau, (v_1, \dots, v_m))) \in \text{Modules}}{y + 1 \mapsto _ * \dots * y + m \mapsto _ \Rightarrow \mathbf{module}(y, M)} \\
\\
\text{C-UNLOAD} \\
\frac{\{\mathbf{lib}(e, y) * \mathbf{module}(e, y)\} \mathbf{unload}(e) \{\mathbf{emp}\}}{\text{C-UNLOAD}}
\end{array}$$

Fig. 8. Proof rules

The proof of the example module is straightforward. Figure 9 gives an outline. The assertion closure parameter P is instantiated with predicate Q applied to the address where the module is loaded. This predicate encompasses the module image, plus the counter cell, minus the struct containing the function pointers.

We can now state the soundness theorem of our approach. It uses the notion of a valid command.

Definition 2 (Valid Command). *A command c is valid if it satisfies the following triple:*

$$\vdash \{\mathbf{emp}\} c \{\mathbf{true}\}$$

Theorem 1 (Soundness). *If each module in the module table is valid, and the main program is valid, then execution does not get stuck.*

A soundness proof is future work.

predicate $Q(\ell) =$
 $\ell + 1 \mapsto [\mathbf{lambda} () [\mathbf{ip} + 3] := \mathbf{ip} + 1;$
 $\quad [\mathbf{ip} + 4] := \mathbf{ip} + 2; x := \mathbf{cons}(0); [\mathbf{ip} + 5] := x; \mathbf{result} := \mathbf{ip} + 3] *$
 $\ell + 2 \mapsto [\mathbf{lambda} (x) c := [\mathbf{ip} + 4]; n := [c]; [c] := n + 1; \mathbf{result} := x + 1] *$
 $\ell + 3 \mapsto [\mathbf{lambda} () x := [\mathbf{ip} + 3]; \mathbf{dispose}(x)] *$
 $\exists c \bullet \ell + 6 \mapsto c * c \mapsto _$

$\forall \ell \bullet \ell + 2 : \mathbf{incrType}(\mathbf{lambda} () Q(\ell))$ by C-FUNTYPE (1)
 $\forall \ell \bullet \ell + 3 : \mathbf{disposeType}(\ell + 4, \mathbf{lambda} () Q(\ell), \ell, \mathbf{myIncrLib})$ by C-FUNTYPE, C-MODULE-FOLD (2)
 $\forall \ell \bullet \ell + 1 : \mathbf{getIncrType}(\ell, \mathbf{myIncrLib})$ by C-FUNTYPE, C-MODULE-UNFOLD, (1), (2)

Fig. 9. Proof outline of the validity of module `myIncrLib`

6 Verification Tool

We implemented the approach in our prototype verifier, VeriFast, and we verified a small server 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. The implementation and the example are online at <http://www.cs.kuleuven.be/~bartj/verifast/>.

7 Future work

The approach, as presented in this paper, is directly applicable to some important instances of dynamic code loading and unloading, notably loadable device drivers in the Linux kernel. However, in certain other cases, there are additional complexities that are not yet addressed by the approach.

Specifically, the user-space shared library mechanisms of most operating systems (notably Unix-like or Windows operating systems) are complicated by the fact that loading a module might yield a new reference to an existing instance of the module, rather than a fresh instance of the module. Therefore, in this context it is not sound for the proof rule for loading a library to grant full permission to the module's code and global variables.

Another item of future work is to add support for libraries or programs consisting of multiple modules. We envisage extending the specification language to allow a module B to *import* another module A . A module B 's **module** predicate would then denote not just B 's code and global variables, but would include A 's **module** predicate as well. A link-time check would ensure that there is no cycle in the module import graph.

A third item of future work is to investigate the interaction between the previous two: how to verify a program or library that specifies a load-time dependency on another library? Here, too, the library sharing issue arises: if two libraries B and C specify a dependency on some library A , then B and C will be linked at load time against the same instance of library A .

8 Related Work

The approach presented in this paper builds on and extends separation logic. Existing verification systems for separation logic include Smallfoot [2], jStar [6] and YNOT [3]. However, to the best of our knowledge, none of these systems can be used for verifying programs that use unloadable code.

9 Conclusion

Existing verification approaches for C programs (VCC [4], Frama-C [1], HAVOC [5], Smallfoot [2], our own verifier VeriFast [7], Jahob [9]) cannot be used for verifying programs that involve unloading of code as they assume that the code is unchanging and is not part of the mutable state. In this paper, we propose a novel separation-logic-based approach for extending a verification approach for C programs to enable verification of programs involving code unloading.

Bibliography

- [1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language.
- [2] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.
- [3] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.
- [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. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, number 5674 in LNCS, 2009. To appear.
- [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] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for java. In *OOPSLA*, 2008.
- [7] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.
- [8] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.
- [9] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.