

Verification of Unloadable Modules

Bart Jacobs, Jan Smans, and Frank Piessens

DistriNet Research Group, Department of Computer Science,
Katholieke Universiteit Leuven, Belgium

FM 2011 – Friday, June 24, 2011, 15:00

Contribution

We propose an approach
for
extending
a modular verification tool for C programs
to enable
the verification of
programs that load and unload modules

Our Motivation

Extending

VeriFast

to verify

unloadable device drivers

The Problem in a Nutshell

- How to modularly verify load and unload operations?
- How to modularly deal with function pointers?
- How to modularly deal with global variables?

Our Approach in a Nutshell

- Separation logic with abstract predicates
- **Module names** as first-class values
- **Module predicates** for abstractly denoting loaded module code and globals
- **Adapted function proof rule** that checks presence of code
- **Parameterized function types** for abstractly describing function pointers

Contents

- 1 An Example Program
- 2 Kernel Module Specification

Contents

- 1 An Example Program
- 2 Kernel Module Specification

Example Program

- Modules.h
- KernelModule.h
- Kernel.c
- ByteVectors.h
- RamDisk.c

Module Loading/Unloading API

```
// Modules.h
```

```
struct module;
```

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

Kernel Module Interface

```
// 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 *kernel_module_init();  
typedef void kernel_module_exit();
```

Kernel

```
/* 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() {
    kernel_module_init *init; kernel_module_exit *exit;
    struct module *m = load_module("RamDisk", &init, &exit);
    struct file_ops *o = init(); testOps(o); exit();
    unload_module(m);
}
```

Byte Vectors

```
// 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);
```

Example Unloadable Module

```
/* 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 ops = {myRead, myWrite};

struct file_ops *init() {
    vector = create_vector();
    return &ops;
}

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

Translation to Formal Language

main =

```
m, init, exit := load M as kernel_module_init;  
ops := call init();  
write := [ops + 1]; _ := call write(42);  
read := [ops]; x := call read();  
assert(x = 42);  
_ := call exit();  
unload(m)
```

Translation to Formal Language

```
module 100 : kernel_module_init {  
  vector = 0  
  myRead() = v := [vector]; result := [v]  
  myWrite(x) = v := [vector]; [v] := x  
  ops = myRead, myWrite  
  init() = v := cons(0); [vector] := v; result := ops  
  exit() = v := [vector]; dispose(v)  
}
```

Read and Write Function Specs (1st Attempt)

predicate ramdisk(x) =
 $\exists v.\text{vector} \mapsto v * v \mapsto x$

myRead()
req ramdisk(X)
ens ramdisk(X) \wedge result = X
= v := [vector]; result := [v]

myWrite(x)
req ramdisk(-)
ens ramdisk(x)
= v := [vector]; [v] := x

Function Proof Rule

$$\frac{M \vdash f(\bar{x}) = c \quad M \vdash \{P\} c \{Q\}}{M \vdash \{P\} f(\bar{x}) \{Q\}}$$

Function Proof Rule

$$\frac{M \text{ not unloadable} \quad M \vdash f(\bar{x}) = c \quad M \vdash \{P\} c \{Q\}}{M \vdash \{P\} f(\bar{x}) \{Q\}}$$

$$\frac{M \text{ unloadable} \quad M \vdash f(\bar{x}) = c \quad M \vdash \{P\} c \{Q\}}{M \vdash \{\text{mycode}() * P\} f(\bar{x}) \{\text{mycode}() * Q\}}$$

Read and Write Function Specs (2nd Attempt)

predicate ramdisk(x) =
 $\exists v. \text{vector} \mapsto v * v \mapsto x$

myRead()
req mycode() * ramdisk(X)
ens mycode() * ramdisk(X) \wedge result = X
= v := [vector]; result := [v]

myWrite(x)
req mycode() * ramdisk(-)
ens mycode() * ramdisk(x)
= v := [vector]; [v] := x

Function Proof Rule

$$\frac{M \text{ not unloadable} \quad M \vdash f(\bar{x}) = c \quad M \vdash \{P\} c \{Q\}}{M \vdash \{P\} f(\bar{x}) \{Q\}}$$

$$\frac{M \text{ unloadable} \quad M \vdash f(\bar{x}) = c \quad M \vdash \{P\} c \{Q\}}{M \vdash \{\text{mycode}() * P\} f(\bar{x}) \{\text{mycode}() * Q\}}$$

$$\frac{M \vdash P \Rightarrow P' \quad M \vdash \{P'\} f(\bar{x}) \{Q'\} \quad M \vdash Q' \Rightarrow Q}{M \vdash \{P\} f(\bar{x}) \{Q\}}$$

Read and Write Function Specs (Corrected)

predicate ramdisk(x) =
myCode() * $\exists v. \text{vector} \mapsto v * v \mapsto x$

myRead()
req ramdisk(X)
ens ramdisk(X) \wedge result = X
= v := [vector]; result := [v]

myWrite(x)
req ramdisk(-)
ens ramdisk(x)
= v := [vector]; [v] := x

Read and Write Function Specs, Abstract

```
funtype read(filePred)()  
  req filePred(X)  
  ens filePred(X)  $\wedge$  result = X
```

```
funtype write(filePred)(x)  
  req filePred(-)  
  ens filePred(x)
```

```
myRead()  
  req ramdisk(X)  
  ens ramdisk(X)  $\wedge$  result = X  
  = v := [vector]; result := [v]
```

```
myWrite(x)  
  req ramdisk(-)  
  ens ramdisk(x)  
  = v := [vector]; [v] := x
```

```
myRead : read(ramdisk)  
myWrite : write(ramdisk)
```

Init and Exit Function Specs, Concrete

module_initial(myAddress, 100) \equiv
 mycode() * ops \mapsto myRead * ops + 1 \mapsto myWrite * vector \mapsto 0

init()
req **module_initial**(myAddress, 100)
ens result \mapsto myRead * result + 1 \mapsto myWrite * ramdisk(0)
 \wedge myRead : read(ramdisk) \wedge myWrite : write(ramdisk)
 = v := **cons**(0); [vector] := v; result := ops

exit()
req ops \mapsto _ * ops + 1 \mapsto _ * ramdisk(_)
ens **module**(myAddress, 100)
 = v := [vector]; **dispose**(v)

module(myAddress, 100) \equiv
 mycode() * ops \mapsto _ * ops + 1 \mapsto _ * vector \mapsto _

Init and Exit Function Specs, Abstract

```
funtype kernel_module_init(l, M, exit)()  
  req module_initial(l, M)  
  ens  $\exists r, w, \text{filePred}. \text{result} \mapsto r * \text{result} + 1 \mapsto w * \text{filePred}(0)$   
     $\wedge r : \text{read}(\text{filePred}) \wedge w : \text{write}(\text{filePred})$   
     $\wedge \text{exit} : \text{kernel\_module\_exit}(l, M, \text{ops}, \text{filePred})$ 
```

```
funtype kernel_module_exit(l, M, ops, filePred)()  
  req  $\text{ops} \mapsto \_ * \text{ops} + 1 \mapsto \_ * \text{filePred}(\_)$   
  ens module(l, M)
```

Further Proof Rules

$$\{\mathbf{emp}\}$$

$$m, \mathit{init}, \mathit{exit} := \mathbf{load} \ M \ \mathbf{as} \ \tau$$

$$\{\mathbf{lib}(m, M) * \mathbf{module_initial}(m, M) \wedge \mathit{init} : \tau(m, M, \mathit{exit})\}$$

$$\{\mathbf{lib}(m, M) * \mathbf{module}(m, M)\} \ \mathbf{unload}(m) \ \{\mathbf{emp}\}$$

$$\frac{\mathbf{module} \ M : \tau \ \{ \dots \} \quad M \vdash \mathit{init} : \tau(\mathit{myAddress}, M, \mathit{exit})}{M \ \mathbf{valid}}$$

Client Proof

```

main =
  {emp}
  m, init, exit := load M as kernel_module_init;
  {lib(m, M) * module_initial(m, M) ∧ init : kernel_module_init(m, M, exit)}
  ops := call init();
  {
    ∃r, w, P. lib(m, M) * ops ↦ r * ops + 1 ↦ w * P(0)
    ∧ r : read(P) ∧ w : write(P) ∧ exit : kernel_module_exit(m, M, ops, P)
  }
  write := [ops + 1]; _ := call write(42);
  read := [ops]; x := call read();
  assert(x = 42);
  {
    ∃P. lib(m, M) * ops ↦ _ * ops + 1 ↦ _ * P(42)
    ∧ exit : kernel_module_exit(m, M, ops, P)
  }
  _ := call exit();
  {lib(m, M) * module(m, M)}
  unload(m)
  {emp}

```

Conclusion

- Implemented in VeriFast
- Verified a kernel-like program
- Soundness proof in TR, checked using Coq
- Related work:
 - Reasoning about self-modifying machine code in a proof assistant
 - Cai, Shao, Vaynberg. Certified self-modifying code. PLDI 2007.
 - Myreen. Verified just-in-time compiler on x86. POPL 2010.
 - Separation logic for higher-order store
 - Schwinghammer, Birkedal, Reus, Yang. Nested Hoare triples and frame rules for higher-order store. CSL 2009.