

# Safe and Flexible Dynamic Linking of Native Code

Michael Hicks

University of Pennsylvania

Stephanie Weirich  
Cornell University

Karl Crary  
Carnegie-Mellon University

## Extensible Systems

- Load new code at run-time
- Examples:
  - ◆ operating systems
  - ◆ web browsers
  - ◆ servers

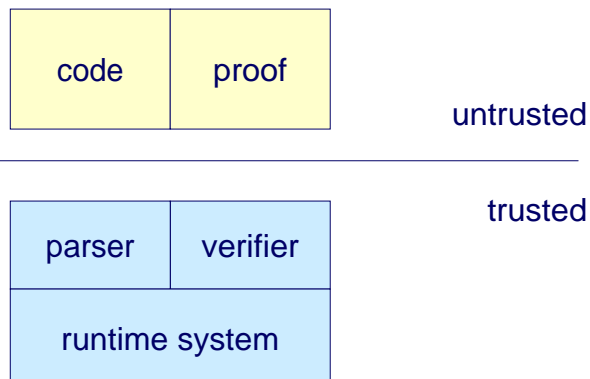
### Question:

- Is the loaded code safe?

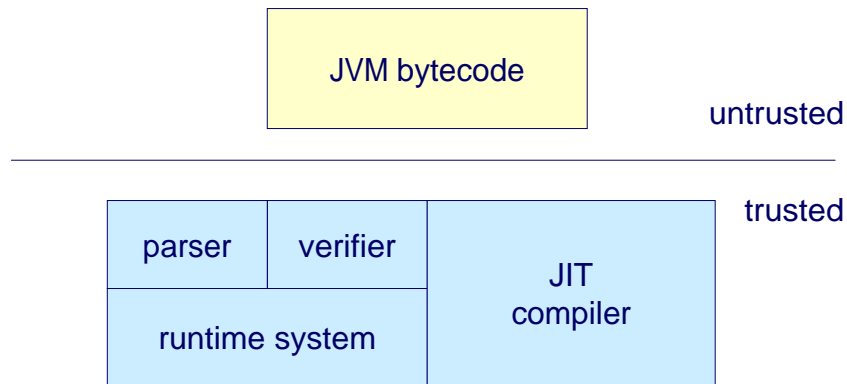
# Proof-Carrying Code

- Consists of
  - ◆ native code
  - ◆ safety proof
- Benefits of PCC
  - ◆ good performance
  - ◆ Security: small trusted computing base (TCB)
    - ◆ the larger the TCB, the more risk of a security hole

## PCC



## Compare to the JVM



## PCC runtime system

- System services
  - ◆ memory management (GC)
  - ◆ threads
  - ◆ dynamic linking
  - ◆ marshalling
  - ◆ ...
- This is not a well-developed area of PCC systems

# Our Work

## Dynamic linking for PCC

### Three goals:

- **Security**: minimize additions to the TCB
- **Flexibility**: support many source languages
- **Performance**: low overhead for loaded code

## Source-level Dynamic Linking

### Six operations:

- **Load** the new code into the running program
- **Verify** that the loaded code is type-correct
- **Link** unresolved symbols in the loaded code
- **Manage** symbols in the running program available for linking
- **Interface** new symbols in the loaded code from the running program
- **Unload** the new code when finished

## C/ Unix Dlopen

```
main () {  
    handle h = dlopen("foo");  
    int (*g)(int) = dlsym(h,"g");  
    int x = g(1);  
    dlclose(h);  
}
```

## Breakdown

- **dlopen:** loading  
(verification)  
linking  
symbol management
- **dlsym:** interfacing
- **dlclose:** unloading

# Example Execution

```
int f(int i) {  
    return i+1;  
}  
main () {  
    handle h = dlopen("foo");  
    int (*g)(int) = dlsym(h, "g");  
    int x = g(1);  
    dlclose(h);  
}
```

running program

"foo"

```
extern int f(int);  
int g(int i) {  
    return f(i)+2;  
}
```

filesystem

# Loading

```
int f(int i) {  
    return i+1;  
}  
main () {  
    handle h = dlopen("foo");  
    int (*g)(int) = dlsym(h, "g");  
    int x = g(1);  
    dlclose(h);  
}
```

running program

"foo"

```
extern int f(int);  
int g(int i) {  
    return f(i)+2;  
}
```

filesystem

```
int g(int i) {  
    return f(i)+2;  
}
```

# Verification

```
int f(int i) {
    return i+1;
}
main () {
    handle h = dlopen("foo");
    int (*g)(int) = dlsym(h,"g");
    int x = g(1);
    dlclose(h);
}
int g(int i) {
    return f(i)+2;
}
```

running program

"foo"

```
extern int f(int);
int g(int i) {
    return f(i)+2;
}
```

In a typed setting,  
**verification** would  
occur here as well.

filesystem

# Linking

```
int f(int i) {
    return i+1;
}
main () {
    handle h = dlopen("foo");
    int (*g)(int) = dlsym(h,"g");
    int x = g(1);
    dlclose(h);
}
int g(int i) {
    return f(i)+2;
}
```

# Symbol Management

```
int f(int i) {  
    return i+1;  
}  
main () {  
    handle h = dlopen("foo");  
    int (*g)(int) = dlsym(h,"g");  
    int x = g(1);  
    dlclose(h);  
}  
int g(int i) {  
    return f(i)+2;  
}
```

# Interfacing

```
int f(int i) {  
    return i+1;  
}  
main () {  
    handle h = dlopen("foo");  
    int (*g)(int) = dlsym(h,"g");  
    int x = g(1);  
    dlclose(h);  
}  
int g(int i) {  
    return f(i)+2;  
}
```

# Interfacing

```
int f(int i) {
    return i+1;
}
main () {
    handle h = dlopen("foo");
    int (*g)(int) = dlsym(h,"g");
    int x = g(1);
    dlclose(h);
}
int g(int i) {
    return f(i)+2;
}
```

# Unloading

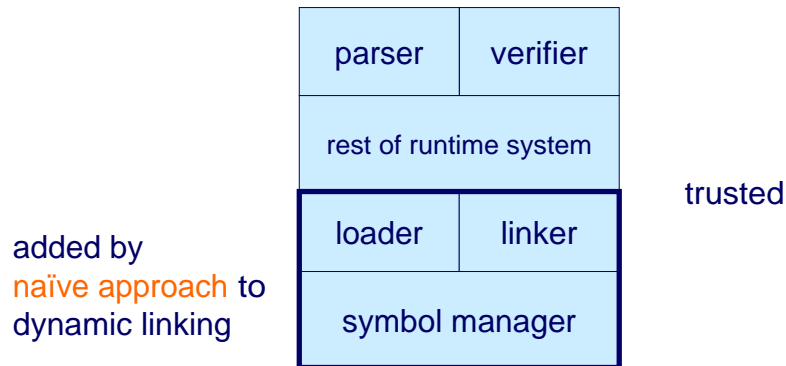
```
int f(int i) {
    return i+1;
}
main () {
    handle h = dlopen("foo");
    int (*g)(int) = dlsym(h,"g");
    int x = g(1);
    dlclose(h);
}
int g(int i) {
    return f(i)+2;
}
```

## Unloading

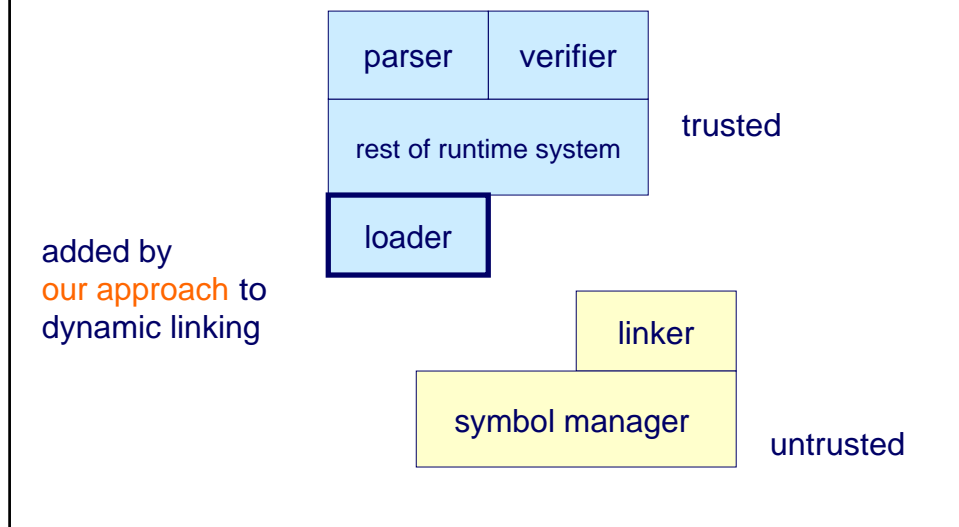
```
int f(int i) {  
    return i+1;  
}  
main () {  
    handle h = dlopen("foo");  
    int (*g)(int) = dlsym(h,"g");  
    int x = g(1);  
    dlclose(h);  
}
```

**How to add  
dynamic linking  
to PCC?**

## Naïve approach



## Our approach



# Advantages

## Flexibility

- Source languages often differ on **linking**
  - ◆ incremental vs. load-time linking
- and **symbol management**
  - ◆ module-thinning, interposition for security

## Security

- Fewer elements to trust

# Framework

- **load** primitive
  - ◆ performs loading and verification
- To implement linking and symbol management:
  - ◆ runtime term representations for types
  - ◆ **checked\_cast** primitive
  - ◆ existential types

# load primitive

**load:**  $\forall \alpha. R(\alpha) \times \text{bytearray} \rightarrow \alpha$

- Returns the module's exported values  $\alpha$ 
  - ◆  $\alpha$  is of tuple-type
- Loaded modules must be closed (no imports)
- Uses  $\lambda_R$  term representations for types:
  - ◆  $R_{\text{int}}$  is a term representing type `int`
  - ◆  $R(\text{int})$  is the type of  $R_{\text{int}}$

# Implementation

- TALx86 Typed Assembly Language
  - ◆ Compiler from type-safe C (Popcorn) to TAL
- Goal:**
- Show that we can compile typical, source-level dynamic linking
  - ◆ Present concrete demonstration for Popcorn
  - ◆ Can do other source languages

# DLpop

- Type-safe version of Dlopen

```
type handle
handle dlopen(string filename)
α dlsym<α>(handle, string sym, R(α))
void dlclose(handle)
```

## Implementation

Loadable files **compiled** to have:

- **indirection table** (like ELF Global Offset Table) for external references
- **dyninit** function to perform linking

**Library** implements DLPop interface:

- Initiates loading, unloading, linking
- Manages the type-safe dynamic symbol table

## Compilation Example

```
extern int f(int);  
int g(int i) {  
    return f(i)+2;  
}
```

## Creating the GOT

```
extern int f(int);
```

```
int g(int i) {  
    return f(i)+2;  
}
```

## Creating the GOT

```
struct tab_t {
    int f(int);
};
int dummy(int i) { raise Failure; }
struct tab_t TAB = { dummy };

int g(int i) {
    return f(i)+2;
}
```

## Indirecting

```
struct tab_t {
    int f(int);
};
int dummy(int i) { raise Failure; }
struct tab_t TAB = { dummy };

int g(int i) {
    return f(i)+2;
}
```

## Indirecting

```
struct tab_t {
    int f(int);
};
int dummy(int i) { raise Failure; }
struct tab_t TAB = { dummy };

int g(int i) {
    return TAB.f(i)+2;
}
```

## dyninit function

```
void dyninit( $\alpha$  l $\langle\alpha\rangle$ (string,R( $\alpha$ ))),
            void u $\langle\alpha\rangle$ (string, $\alpha$ ,R( $\alpha$ )))
{
    TAB.f = l("f",Rint $\rightarrow$ int);
    u("g",g,Rint $\rightarrow$ int);
}
```

## Loading with `dlopen`

`dlopen(file)`

- call `load(inittype, file)` which returns the file's `dyninit` function
- create a `symbol hashtable` for the module; add it to the dynamic symbol table (a list of hashtables)
- call `dyninit` with the appropriate lookup and update functions
- return the hashtable abstractly as a `handle`

## Symbol hashtable

`syntab`: `<string,  $\exists\alpha. (\alpha \times R(\alpha))$ >` hashtable

- looking up a symbol returns a value of type  $\exists\alpha. (\alpha \times R(\alpha))$
- use `checked_cast` to cast this value to the type  $R(\beta)$  indicated by the caller

## Performance Summary

Run-time overhead: one indirection  
DLpop = DOpen

Load-time overhead: verification  
Space overhead: type reps & **dyninit**  
DLpop > DOpen

## Performance Summary

Run-time overhead:  
**DLpop = DOpen**

In practice,

- Verification is amortized over total running time
- Additional space costs are paged out

## Other linking models

- Java
- Windows DLL's and COM
- Ocaml modules
- Units
- SPIN OS
- ...

## Incremental Linking

- Resolve symbols on-demand, not at load-time
- Instead, do the lookup in the **dummy** function

# Conclusions

## **Type-safe dynamic linking of native code**

- **Security**: minimal additions to the TCB
- **Flexibility**: may implement many source-level dynamic linking facilities
- **Performance**: equivalent to untyped approach

This document was created with Win2PDF available at <http://www.daneprairie.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.