Safe and Flexible Dynamic Linking of Native Code

Michael Hicks[1], Stephanie Weirich[2], and Karl Crary[3]

[1] University of Pennsylvania, Philadelphia PA 19104, USA,
mwh@dsl.cis.upenn.edu,
WWW home page: http://www.cis.upenn.edu/~mwh
[2] Cornell University, Ithaca NY 14853, USA,
sweirich@cs.cornell.edu,
WWW home page: http://www.cs.cornell.edu/sweirich
[3] Carnegie Mellon University, Pittsburgh PA 15213, USA
crary@cs.cmu.edu,
WWW home page: http://www.cs.cmu.edu/~crary

Abstract. We present the design and implementation of the first complete framework for flexible and safe dynamic linking of native code. Our approach extends Typed Assembly Language with a primitive for loading and typechecking code, which is flexible enough to support a variety of linking strategies, but simple enough that it does not significantly expand the trusted computing base. Using this primitive, along with the ability to compute with types, we show that we can *program* many existing dynamic linking approaches. As a concrete demonstration, we have used our framework to implement dynamic linking for a type-safe dialect of C, closely modeled after the standard linking facility for Unix C programs. Aside from the unavoidable cost of verification, our implementation performs comparably with the standard, untyped approach.

1 Introduction

A principle requirement in many modern software systems is dynamic extensibility—the ability to augment a running system with new code without shutting the system down. Equally important, especially when extensions may be untrusted, is the condition that extension code be *safe*: an extension should not be able to compromise the integrity of the running system. Two examples of systems allowing untrusted extensions are extensible operating systems [4], [11] and applet-based web browsers [22]. Extensible systems that lack safety typically suffer from a lack of robustness; for example, if the interface of a newer version of a dynamically linked library (DLL) changes from what is expected by the loading program, its functions will be called incorrectly, very possibly leading to a crash. These sorts of crashes are accidental, so in the arena of untrusted extensions the problem is greatly magnified, since malicious extensions may intentionally violate safety.

The advent of Java [3] and its virtual machine [29] (the JVM) has popularized the use of language-based technology to ensure the safety of dynamic extensions. The JVM bytecode format for extension code is such that the system may *verify* that extensions satisfy certain safety constraints before it runs them. To boost performance, most recent JVM implementations use just-in-time (JIT) compilers. However, because JIT compilers are large pieces of software (typically tens of thousands of lines of code), they unduly expand the *trusted computing base* (TCB), the system software that is required to work properly if safety is to be assured. To minimize the likelihood of a security hole, a primary goal of all such systems is to have a small TCB.

An alternative approach to verifiable bytecode is verifiable native code, first proposed by Necula and Lee [35] with Proof-Carrying Code (PCC). In PCC, code may be heavily optimized, and yet still verified for safety, yielding good performance. Furthermore, the TCB is substantially smaller than in the JVM: only the verifier and the security policy are trusted, not the compiler. A variety of similar architectures have been proposed [2], [25], [33].

While verifiable native code systems are fairly mature, all lack a well-designed methodology for dynamic linking, the mechanism used to achieve extensibility. In the PCC Touchstone system, for example, dynamic linking has only been performed in an ad-hoc manner, entirely within the TCB [35], and the current Java to PCC compiler, Special J, does not support dynamic linking [6]. Most general-purpose languages support dynamic linking [3], [9], [13], [27], [36], [37], so if we are to compile such languages to PCC, then it must provide some support for implementing dynamic linking. We believe this support should meet three important criteria:

1. **Security**. It should only minimally expand the TCB, improving confidence in the system's security. Furthermore, soundness should be proved within a formal model.
2. **Flexibility**. We should be able to compile typical source language linking entities, *e.g.*, Java classes, ML modules, or C object files; and their loading and linking operations.
3. **Efficiency**. This compilation should result in efficient code, in terms of both space and time.

In this paper, we present the design and implementation of the first complete framework for dynamic linking of verifiable native code. We have developed this framework in the context of Typed Assembly Language [33] (TAL), a system of typing annotations for machine code, similar to PCC, that may be used to verify a wide class of safety properties. Our framework consists of several small additions to TAL that enable us to *program* dynamic linking facilities in a type-safe manner, rather than including them as a monolithic addition to the TCB. Our additions are simple enough that a formal proof of soundness is straightforward. The interested reader is referred to the companion technical report [20] for the full formal framework and soundness proof.

To demonstrate the flexibility and efficiency of our framework, we have used it to program a type-safe implementation of DLopen [9], a UNIX library that provides dynamic linking services to C programs. Our version of DLopen has performance comparable to the standard ELF implementation [40], and has the added benefit of safety. Furthermore, we can program many other dynamic linking approaches within our framework, including Java classloaders [23], Windows DLLs and COM [7], Objective Caml's Dynlink [27], [37], Flatt and Felleisen's Units [13], and SPIN's domains [38], among others.

The remainder of this paper is organized as follows. In the next section we motivate and present our framework, which we call TAL/Load. In Section 3 we describe a type-safe version of DLopen programmed using TAL/Load. In Section 4 we compare the performance of our type-safe version to the standard version of DLopen. We discuss how we can program other linking approaches using TAL/Load in Section 5, and discuss other related work. We conclude in Section 6.

2 Our Approach

We begin our discussion by considering a straightforward but flawed means of adding dynamic linking in TAL, to motivate our actual approach, described later. Consider defining a primitive, load_0, that dynamically instantiates, verifies, and links TAL modules into the running program. Informally, load_0 might have the type:

$$\mathsf{load}_0 : \quad \forall\alpha : \mathsf{sig}.\, \texttt{bytearray} \to \alpha\ \texttt{option}$$

To dynamically load a module, the application first obtains the binary representation of the module as a `bytearray`, and provides it to load_0 preceded by the module's expected signature type α. Then load_0 parses the bytearray, checks it for well-formedness, and links any unresolved references in the file to their definitions in the running program. Next, it compares the module's signature with the expected one; if the signatures match, it returns the module to the caller. If any part of this process fails, load_0 returns `NONE` to signal an error. As an example, suppose the file "`extension`" contains code believed to implement a module containing a single function f of type `int` → `int`. In informal notation, that file is dynamically linked as follows:

```
case load0 [sig f : int -> int end]
     (read_file "extension") of
   NONE => ... handle error ...
 | SOME m => m.f(12)
```

There are many problems with this approach. First, it requires first-class modules; in the context of a rich type system, first-class modules require a complicated formalization (*e.g.*, Lillibridge [28]) with restrictions on expressiveness; as a result, in most ML variants (and TAL as well) modules are second-class [17], [26], [30]. Second, it requires a type-passing semantics as the type passed to load_0 must be checked against the actual type of the module at run-time. This kind of semantics provides implicit type information to polymorphic functions, contrary to the efforts of TAL to make all computation explicit. Third, all linking operations, including tracking and managing the exported definitions of the running program, and rewriting the unresolved references in the loaded file, occur within load_0, and thus within the TCB. Finally, we are constrained to using the particular linking approach defined within the TCB, diminishing flexibility. As we show in Sections 3 and 5, linking is the aspect of extensibility that differs most among source languages. For example, Java links unresolved references incrementally, just before they are accessed, while in C all linking generally occurs at load-time. Furthermore, extensible systems typically require more fine-grained control over linking. For example, in SPIN [4], only trusted extensions may link against certain secure interfaces, and in MMM [37], the runtime interface used during dynamic linking is a safe subset of the one used during static linking, a practice called module thinning.

Rather than place all dynamic linking functionality within the TCB, as we have outlined above with load_0, we prefer to place smaller components therein, forming a dynamic linking framework. Furthermore, these components are themselves largely composed of pre-existing TAL functionality. Therefore, this framework does not implement source-level dynamic linking approaches directly, but may be used to *program* them.

Our framework defines a primitive load similar to load_0 above, but with the following simplifications:

1. Loaded modules are required to be *closed* with respect to terms. That is, they are not allowed to reference any values defined outside of the module itself. We can compile source-language modules that allow externally-defined references to be loadable by using a "poor man's functorization," which we describe below. Modules may refer to externally-defined (*i.e.*, imported) type definitions.
2. Rather than return a first-class module, load returns a tuple containing the module's exported term definitions (and thus the type variable α now is expected to be a tuple-type, rather than a signature). Any exported type definitions are added to the global *program type interface*, a list of types and their definitions used by the current program, used to resolve the imported type definitions of modules loaded later.
3. Rather than require a type-passing semantics for the type argument to load, we make use of term-level representations of types, in the style of Crary *et al.* [8].

These simplifications serve three purposes. First, by eliminating possible type components from the value returned by load, we avoid a complicated modular theory, at a small cost to the flexibility of the system. Second, the majority of the functionality of load—parsing binary representations and typechecking—is *already* a part of the TCB. By avoiding term-level linking (since loaded modules must be closed) we can avoid adding binary rewriting and symbol management to the TCB (we do have to manage type definitions, however, as we explain in the next subsection). Finally, by adding term-level type representations, we preserve TAL's type-erasure semantics. These representations also allow the implementation of a dynamic type, making it possible to program linking facilities outside of the TCB. We call our framework TAL/Load.

While TAL/Load only permits loading closed *TAL* modules, in practice we wish to dynamically load non-closed *source* modules by resolving their external references with definitions in the running program. One way to implement this linking strategy is by translating source-level external references into "holes" (i.e. uninitialized reference cells), in a manner similar to closure-converting a function. After the module is loaded via load, these cells are linked appropriately using a library added to the program. To track the running program's symbols, we can use term-level type representations, existential types [31] and a special checked_cast operator to implement type dynamics [1], amenable to programming a type-safe symbol table.

We defer a complete discussion of how to effectively use TAL/Load until Section 3, where we describe our implementation of a full-featured dynamic linking approach for C programs. For the remainder of this section, we focus on two things. First, we look more closely at the process of closing a module with respect to its externally defined types and terms. We explain the difficulty with closing a module with respect to named types, thus motivating our solution of using the program type interface. We then describe the implementation of TAL/Load in the TALx86 [32] implementation of TAL.

2.1 Comparing Types by Name

The complications with first-class structures arise because of their type components; if M and N are arbitrary expressions of module type having a type component t, it is difficult at compile-time to determine if $M.t$ is equal to (is the same type as) $N.t$. The problem arises because we do not know the identities of types $M.t$ and $N.t$, and therefore must use their names (including the paths) to compare them.

In the absence of these named types[1], closing a module with respect to its externally-defined terms is fairly simple. For example, consider the following SML module, perhaps forming part of an I/O library, that supports the opening and reading of text files.

```
structure TextIO =
struct
  type instream = int
  val openIn : string -> instream = ...
  val inputLine : instream -> string = ...
  ...
end
```

[1] Named types are also called branded types, and can be used to implement abstract types (as in first-class modules) and generative types (such as structs in C or datatypes in ML).

A client of this module might be something like:

```
fun doit =
  let val h = TextIO.openIn "myfile.txt" in
    TextIO.inputLine h
  end
```

If we want to close this client code to make it amenable for dynamic loading, we need to remove the references to the TextIO module. For example, we could do:

```
val TextIO_openIn :
  (string -> int) option ref = ref NONE
val TextIO_inputLine :
  (int -> string) option ref = ref NONE
fun doit () =
  let val h = getOpt (!TextIO_openIn)
                "myfile.txt" in
    getOpt (!TextIO_inputLine) h
  end
```

We have converted the externally referenced function into a locally defined reference to a function. When the file is dynamically loaded, the reference can get filled in. This strategy is essentially a “poor man’s” functorization. This process closes the file with respect to values. However, we run into difficulty when we have externally defined values of named type. Consider if TextIO wished to hold the type instream abstract. If we attempt to close the client code as before, we get:

```
val TextIO_openIn :
      (string -> TextIO.instream) option ref = ...
val TextIO_inputLine :
      (TextIO.instream -> string) option ref = ...
```

We still have the external references to the type TextIO.instream itself. We must have a way to load a module referring to externally defined, named types. Because types form an integral part of typechecking, a trusted operation, our solution is to support name-based type equality within the TCB. As we do not want to overly complicate the TCB, we base the support for named types on that of TAL’s framework for static link verification [15]. There, paths are disregarded altogether in comparing types; only one module may export a type with a given name. A related project, TMAL [10], approaches this problem differently, as we describe in Section 5.6.

Therefore, loaded code is not closed with respect to externally defined types, but instead declares a type interface (X_I, X_E), which is a pair of maps from type names to implementations. X_I mentions the named types provided by other modules, and X_E mentions named types defined by this one. By not including the implementation of the type inside a map X (just mentioning its name), we can use this mechanism to implement abstraction. As an example, the type interface of the client code above would be something like:

$$(\{\mathit{instream}\}, \{\})$$

and the interface for `TextIO` would be the reverse:

$$(\{\}, \{\mathit{instream}\})$$

Part of the implementation of load maintains a list of the imported and exported types of all the modules in the program, called the *program type interface.* When a new module is loaded, load checks that the named type imports of the new module are consistent with the program type interface, and that the exports of the new module do not redefine, or define differently, any types in the program type interface imports. We do not require that all of a module's type imports be defined by the program interface when it is loaded. This relaxation requires a uniform representation of named types; in our case, all named types are pointer-types. Not requiring defined imports facilitates loading a file that has mutually-recursive type definitions. In particular, the loaded file indicates the type it expects from another file to be loaded. When the other file is loaded, its export is confirmed to match the previously loaded import.

We have developed a formal calculus for our framework and have proven it sound. While this formalization is interesting, our real contribution lies in the way we can program type-safe dynamic linking within our framework. We refer the interested reader to the companion technical report [20] for the full theoretical treatment.

2.2 Implementation

We have implemented TAL/Load in the TALx86 [32] implementation of TAL. The key component of TAL/Load is the load primitive:

$$\mathsf{load} : \forall\alpha.\, (R(\alpha) \times \texttt{bytearray}) \to \alpha\ \texttt{option}$$

In addition to the `bytearray` containing the module data, load takes a term representation of its type argument, following the approach of Crary *et al.'s* λ_R [8]. Informally, λ_R defines term representations for types, called R-terms, and types to classify these terms, called R-types. For example, the term to represent the type `int` would be $R_{\texttt{int}}$, and the type of this term would be $R(\texttt{int})$. The type $R(\tau)$ is a singleton type; for each τ there is only one value that inhabits it—the representation of τ. Therefore the typechecker guarantees the correspondence between a type variable checked statically and the representation of that type used at runtime.

The actions of load are illustrated in Figure 1. In the figure, the square boxes indicate unconditional actions, and the diamond boxes indicate actions that may succeed or fail. Each square and diamond box has data inputs and outputs, indicated as wavy boxes; the arrows illustrate both data- and control-flow. Using components of the TALx86 system, load performs two functions:

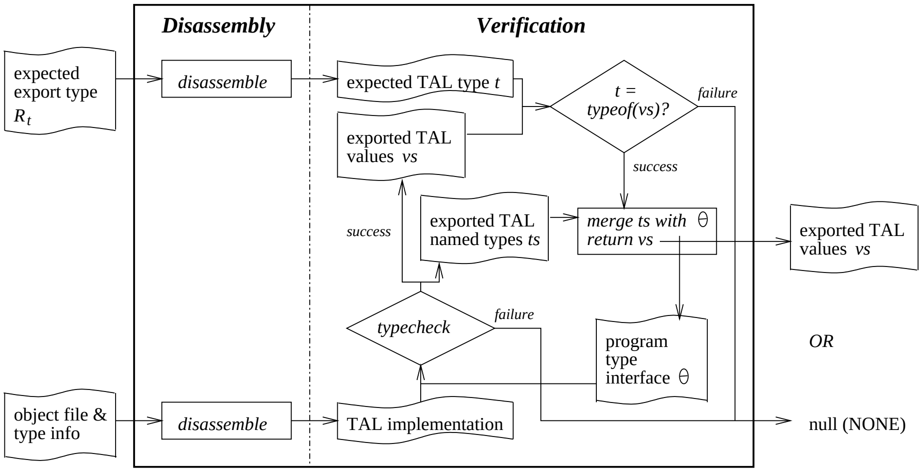

Fig. 1. The implementation of load

1. **Disassembly** The first argument R_t indicates the expected type t of the exports, and must be disassembled into the internal representation of TAL types. Type t should always be of tuple type, where each element type represents the type of one of the object file's exported values. The second argument to load is a byte array representing the object file and the typing annotations on it; while conceptually a single argument, in practice TALx86 separates the annotations from the object code, resulting in an *object file* and a *types file*. The contents of these two files, stored in buffers, are disassembled and combined to produce the appropriate internal representation: a *TAL implementation*.
2. **Verification** The TAL implementation is then typechecked in the context of the program's current type interface Θ, following the procedure described in the previous subsection. If typechecking succeeds, the result is a list of exported values and exported types. The values are gathered into a tuple, the type of which is compared to the expected type. If the types match, the tuple is returned (within an option type) to the caller, and the exported types are combined with Θ to form the new program type interface. On failure, *null* (*i.e.*, `NONE`) is returned.

The majority of the functionality described above results in no addition to the TAL trusted computing base. In particular, the TAL link verifier, typechecker, and disassembler are already an integral part of the the TCB; TAL/Load only makes these facilities available to programs through load. Three pieces of trusted functionality are needed, however, beyond that already provided by TAL: loading the object code into the address space of the running program, representing types as runtime values, and maintaining the program type interface Θ at runtime. We explain how these elements impact the TCB below.

Loading Following the verification process, before returning to the caller, some C code is invoked to load the object code into the address space. This loading code is based on that used by the Linux kernel to dynamically load modules. We describe the code for ELF object files, used in TALx86 Linux implementation; COFF files, used in the Windows implementation, are similar.

First, the file is parsed, performing well-formedness checks and extracting the ELF file's section headers, which describe the file's format. The file must be a relocatable object file, as is normally produced by a compiler for separate compilation, *e.g.* by `cc -c`. The sections of interest are the code and data sections, the relocations section, and the symbol tables. Second, the code and data are logically arranged in the order and alignment specified by the file and the ELF standard, and the total required size is computed. Third, any externally-defined symbols are resolved—more on this below. Finally, an appropriately-sized buffer is allocated and the code and data are copied to that buffer (TAL uses garbage collection, so the buffer is allocated using the GC allocator).[2] This code is then relocated to work relative to the allocated buffer's address. Finally, the address of the buffer is returned to the caller (which is the result of `load`).

It is troublesome that we resolve (*i.e.* link) external symbols during the loading process. Much of the motivation of our approach is to perform linking outside the TCB, in part to avoid the additional complexity. In fact, the overwhelming majority of symbols *are* linked by mechanisms outside the TCB, as we show in the next section. However, there are some *trusted* symbols that cannot easily be linked in this way. These symbols are part of the *macro* instructions of TALx86. Macro instructions do not map directly to a machine instruction, but instead to a machine instruction sequence; this sequence may include references to external symbols. For example, the macro for the TALx86 `malloc` instruction consists of six machine instructions, of which two are calls to external functions, one to `GC_malloc` (to actually allocate the memory), and the other to `out_of_memory` (in case the GC allocator returns *null*). The file cannot be closed with respect to these calls, because they are primitive.

As a result, when a file containing a `malloc` instruction is dynamically loaded, the external calls to must be resolved by the loader. We do this by rewriting the code directly, using the relocations provided in the object file. Patching symbols in this manner has the unfortunate consequence that loaded code cannot be shared between (OS-level) processes because the patched symbols, like `GC_malloc`, may be at different addresses in each process.

Given that we must link some symbols implicitly—that the module does not truly have to be 'closed'—it is reasonable to ask "why not link all symbols in this way?" The answer is that it would greatly reduce our flexibility and our security.

[2] Note that this allocation is necessary; we cannot reuse the buffer containing the object file data to avoid the copy. The reason is that `load` effectively changes the type of the buffer argument from `bytearray` to some type α. Placing the object file contents in a fresh buffer prevents surreptitiously modifying the given buffer via an alias still having `bytearray` type. We could avoid this copy by proving that no aliases exist, *e.g.* by using alias types [41].

As motivated in §2, by moving symbol management outside of the TCB, we can better control how symbols are stored (*i.e.* what datastructure), how they are apportioned among users of various privilege levels, how they are interfaced, *etc.*, without changing to trusted computing base; instead we can rely on the system to verify that this 'untrusted' code is safe.

While implicit linking seems to be necessary for TALx86 macro instructions, it may be that our approach could be improved. In particular, if the symbols referred to by macro sequences (*e.g.* `GC_malloc`) were always loaded at the same address, then we could share the code between processes. Given that most modern operating systems support separate, per-process address spaces, and that both ELF and COFF files allow the loaded address for a program component to be specified, this should be possible. It would furthermore allow the relocation process to take place outside of the TCB, preceding the call to load. The disassembler would then check for the particular, fixed address when checking the well-formedness of macro instruction sequences, rather than looking for an external symbol reference.

Passing Types at Runtime Term representations for types are used, among other things, to preserve TAL's type-erasure semantics. So that this addition to the TAL trusted computing base can be kept small, we do two things. First, we represent R-terms using the binary format for types already used by the TAL disassembler. Note that the binary representation of a named type is a string containing the name. Second, we do not provide any way within TAL to dynamically introduce or deconstruct R-terms, such as via appropriate syntax and `typecase` [8]. Doing so would require that we reflect the entire binary format of types into the type system of TAL. Instead, we only allow the introduction of R-terms in the static data segment by a built-in directive. Consequently, only closed types may be represented.

Aside from providing type information to load, R-types are also useful for implementing dynamic types. Dynamic types may be used to implement type-safe symbol management, as we describe in the next section. Therefore we allow limited examination of R-terms with a simple primitive called checked_cast:

$$\mathsf{checked\_cast} : \forall\alpha.\,\forall\beta.\,(R(\alpha) \times R(\beta) \times \beta) \rightarrow \alpha\ \mathtt{option}$$

Informally, checked_cast takes a value of type β and casts it to one of type α if the types α and β are equal. This operation is trivial to add as comparing types is part of the TAL typechecker. Therefore it does not add to the TCB. With a full implementation of λ_R including `typecase`, checked_cast does not need to be primitive [42].

Maintaining the Program Type Interface As explained in the previous subsection, the need to maintain the program's type interface at runtime derives directly from the presence of named types in TAL. We may use elements already within the TCB to implement the program type interface. Representations of type interfaces (X_I, X_E) already exist as a part of object files; they are used in verifying static link consistency. The initial Θ is initialized in a small bit of code generated by the TAL static linker after it has determined the program's type interface. Computing the new type interface at run time is done using this same trusted code for static link verification, so maintaining this information at run time does not significantly expand the TCB.

3 Programming Dynamic Linking

Having defined our dynamic linking framework TAL/Load, we now describe how to use TAL/Load to program dynamic linking services as typically defined in source languages like C and Java. As a concrete demonstration, we present a type-safe version of DLopen [9], a standard dynamic-linking methodology for C, that we have written using TAL/Load. Our version, called DLpop, provides the same functionality for Popcorn [32], a type-safe dialect of C. We chose to implement DLopen over several other dynamic linking approaches because it is the most general; we describe informal encodings of other approaches, including Java classloaders [23], in Section 5. We begin by describing DLpop and the ways in which it differs from DLopen, and then follow with a description of our implementation written in TAL/Load.

3.1 DLpop: A type-safe DLopen

```
extern handle;
extern handle dlopen(string fname);
extern a dlsym<a>(handle h, string sym, <a>rep typ);
extern void dlclose(handle h);

extern exception WrongType(string);
extern exception FailsTypeCheck;
extern exception SymbolNotFound(string);
```

Fig. 2. DLpop library interface

Most Unix systems provide some compiler support and a library of utilities (interfaced in the C header file `dlfcn.h`) for dynamically linking object files. We call this methodology DLopen, after the principal function it provides. We have implemented a version of DLopen for our type-safe C-like language, Popcorn [32], which we call DLpop. The library interface is essentially identical to DLopen except that it is type-safe; it is depicted in Figure 2. We describe this interface in detail below, noting differences with DLopen; a thorough description of DLopen may be found in Unix documentation [9]. DLpop and DLopen both provide three core functions:

handle dlopen(string fname)

Given the name of an object file, dlopen dynamically loads the file and returns a handle to it for future operations. Imports in the file (*i.e.,* symbols declared extern therein) are resolved with the exports (*i.e.,* symbols not declared static) of the running program and any previously loaded object files. Before it returns, dlopen will call the function _init if that function is defined in the loaded file. In DLpop (but not DLopen), dlopen typechecks the object file, throwing the exception FailsTypeCheck on failure. In addition, the exception SymbolNotFound will be raised if the loaded file imports a symbol not present in the running program, or WrongType if a symbol in the running program does not match the type expected by the import in the loaded file. DLopen functions, in general report errors with an errno-like facility.

a dlsym<a>(handle h, string sym, <a>rep typ)

In DLpop, dlsym takes a handle for a loaded object file h, a string naming the symbol s, and the representation of the symbol's type typ, dlsym returns a pointer to the symbol's value. The syntax <a> refers to the type argument a (not its representation) to dlsym. In lambda-calculus notation, dlsym therefore has the type

$$\mathtt{dlsym} : \forall a.\ \mathtt{handle} \times \mathtt{string} \times R(a) \to a$$

In DLopen, dlsym does not receive a type argument, and the function returns an untyped pointer (*null* on failure), of C-type void *, which requires the programmer to perform an unchecked cast to the expected type. The fact that our version takes a type representation argument typ to indicate the expected type means that this type can be (and is) checked against the actual type at runtime. In practice, this type always has the form of a pointer type since the value returned is a reference to the requested symbol. As in TAL, we have extended Popcorn with representation types (<a>rep), implementing them with TAL *R*-types. The term representing type t in Popcorn is denoted repterm@<t>. Because we cannot create the representation of a type with free type variables in TAL, the type argument a to dlsym must also be a closed type. If the requested symbol is not present in the object file, the exception SymbolNotFound is thrown; if the passed type does not match the type of the symbol, the exception WrongType is thrown.

void dlclose(handle h)

In DLopen, dlclose *unloads* the file associated with the given handle. In particular, the file's symbols are no longer used in linking, and the memory for the file is freed; the programmer must make sure there are no dangling pointers to symbols in the file. In DLpop, dlclose only removes symbols from future linkages; if the user program does not reference the object file, then it can be garbage collected.

The current version of DLpop does not implement all of the features of DLopen, most notably: DLopen automatically loads object files upon which a dynami-

Dynamically linked code: `loadable.pop`

```
extern int foo(int);

int bar(int i) {
  return foo(i);
}
```

Static code: `main.pop`

```
int foo(int i) {
  return i+1;
}

void pop_main(){
  handle h = dlopen("loadable");
  int bar(int) = dlsym(h,"bar", repterm@<int(int)>);
  bar(3);
  dlclose(h);
}
```

Fig. 3. DLpop dynamic loading example

cally loaded file depends, allowing for recursive references; DLopen supports the ability to optionally resolve function references *on-demand*, rather than all at load-time, assuming the underlying mechanisms (*e.g.* an ELF procedure linkage table [40]) are present in the object file; and DLopen provides a sort of finalization by calling the user-defined function `_fini` when unloading object files. We foresee no technical difficulties in adding these features should the need arise. In a later version of DLpop, we implemented a variant of `dlopen` that allows the caller to specify a list of object files to load, and these files may have mutually-recursive (value) references. On-demand function symbol resolution is also feasible; a possible compilation strategy to support it is described below, and another approach is described in Section 5.1. Finally, finalization is implemented in most garbage collectors, in particular the Boehm-Demers-Weiser collector [5] used in the current TAL implementation.

Figure 3 depicts a simple use of DLpop. The user statically links the file `main.pop`, which, during execution, dynamically loads the object file `loadable.o` (the result of compiling `loadable.pop`), looks up the function `bar`, and then executes it; the type argument to `dlsym` is inferred by the Popcorn compiler. The dynamically linked file also makes an external reference to the function `foo`, which is resolved at load time from the exports of `main.pop`.

```
struct got_t {
  int (int) foo;
}
struct got_t GOT = { dummy };

static int dummy(int i) {
  raise (Failure);
}

static int bar(int i) {
  return GOT.foo(i);
}

void dyninit(a lookup<a>(string, <a>rep),
             void update<a>(string,a,<a>rep)) {

  int (int) foo = lookup("foo",repterm@<int (int)>);
  GOT.foo = foo;

  update("bar",bar,repterm@<int (int)>);
}
```

the type of the global offset table

the global offset table itself

to avoid null checks, all fields have dummy values

the function recompiled to reference the global offset table

initialization function called by dlopen

resolve file's imports

add the exported function to the symbol table

Fig. 4. Compilation of dynamically loadable code

3.2 Implementing DLpop in TAL/Load

Our implementation of DLpop is similar to implementations of DLopen that follow the ELF standard [40] for dynamic linking, which requires both library and compiler support. In ELF, dynamically loadable files are compiled so that all references to data are indirected through a *global offset table* (GOT) present in the object file. Each slot in the table is labeled with the name of the symbol to be resolved. When the file is loaded dynamically, the dynamic linker fills each slot with the address of the actual exported function or value in the running program; these exported symbols are collected in a *dynamic symbol table*, used by the dynamic linker. This table consists of a list of hashtables, one per object file, each constructed at compile-time and stored as a special section in the object file. As files are loaded and unloaded, the hashtables are linked and unlinked from the list, respectively.

We describe our DLpop implementation below, pointing out differences with the ELF approach. We first describe the changes we made to the Popcorn compiler, and then describe how we implemented the DLpop library.

Compilation As in the ELF approach, dynamically loadable files must be specially compiled, an operation that we perform in three stages. First, the compiler must define a GOT for the file, and translate references to externally defined functions and data to refer to slots in the GOT. In ELF, the GOT is

```
int foo(int i) {
  return i+1;
}
void pop_main() {

  handle h = dlopen("loadable");
  int bar(int) = dlsym(h,"bar",repterm@<int (int)>);
  bar(3);
  dlclose(h);
}

void dyninit(a lookup<a>(string, <a>rep),
             void update<a>(string,a,<a>rep)) {

  update("foo",foo,repterm@<int (int)>);
}
```

foo is still exported (not static) so statically linked files may refer to it

initialization function called at startup

add the exported function to the symbol table

Fig. 5. Compilation of statically linked code

a trusted part of the object file, while in DLpop the GOT is implemented in the verifiable language, TAL. As a consequence, the table is well-typed with the compiler initializing each slot to a dummy value of the correct type, where possible. For slots of abstract type, we cannot create this dummy value, so we initialize the slot to null and insert null checks for each table access in order to satisfy the typechecker.

Second, the compiler adds a special `dyninit` function that will be called at load-time to fill in the slots in the GOT with the proper symbols. This approach differs from ELF, in which the GOT is filled by a dynamic linker contained in the running program. From the loading program's point of view, the `dyninit` function abstracts the linking process. The `dyninit` function takes as arguments two other functions, `lookup` and `update`, that provide access to the dynamic symbol table. For each symbol address to be stored in the GOT, `dyninit` will look up that address by name and type using the `lookup` function, and fill in the appropriate GOT slot with the result. Similarly, `dyninit` will call `update` with the name, type, and address of each symbol that it wishes to export. Because the `dyninit` function consists only of TAL code, all linking operations are verifiably type-safe. This verification prevents, for example, `lookup` from requesting a symbol by name, then receiving a symbol of an unexpected type. In an untypechecked setting, as in DLopen, this operation could result in a crash.

Finally, because the exports of dynamically linked files are designated by `dyninit`, the object file should only export `dyninit` itself; therefore the compiler makes all global symbols `static`. Figure 4 shows the entire translation for the dynamic code in Figure 3.

Statically linked files are only changed by adding a `dyninit` to export symbols to dynamically linked files. At startup, the program calls the `dyninit` functions

```
struct got_t {
  int (int) foo;
}
struct got_t GOT = { dummy };

static int dummy(int i) {
  int (int) foo = dynlookup("foo",repterm@<int (int)>);   look up foo

  GOT.foo = foo;                                  replace dummy in the GOT

  return GOT.foo(i);                                             call it
}

static int bar(int i) {
  return GOT.foo(i);
}
                               saved lookup function as passed to dyninit
static a dynlookup<a>(string, <a>rep) = ...;

void dyninit(a lookup<a>(string, <a>rep),
             void update<a>(string,a,<a>rep)) {

  dynlookup = lookup;                            note the lookup function

  update("bar",bar,repterm@<int (int)>);
}
```

Fig. 6. Compilation of dynamically loadable code to resolve functions on-demand. Only the parts that differ from Figure 4 are commented.

of each of its statically linked files. Figure 5 shows the static code of Figure 3 compiled in this manner.

Rather than add the `dyninit` function to fill in the GOT's of loaded files and note their exported symbols, we could have easily followed the ELF approach of writing a monolithic dynamic linker, called at startup and from `dlopen`. However, we have found that abstracting the process of linking to calling a function in the loaded file has a number of benefits. First, it allows the means by which an object file resolves its imported symbols to change without affecting the DLpop library. For example, in order to save space, we could allow GOT entries to be null by changing them to `option` type, or we could eliminate the GOT altogether by using runtime code generation, as described in Section 5. If we knew that many symbols may not be used by the loading program (as is likely with a large shared library), we could resolve them on-demand by making the dummy functions perform the symbol resolution, rather than doing so in the `dyninit` function; this approach is shown in Figure 6.

Second, `dyninit` simplifies the implementation of policy decisions made by the loading code with regard to symbol management. For example, the loading code may wish to restrict access to some of its symbols based on security criteria [38]; in this case, it could customize the `lookup` function provided to `dyninit` to throw an exception if a restricted symbol is requested.

Finally, using `dyninit` allows the loaded file to customize operations performed at link-time. For example, by adding a flag to prevent calls to `update` from occurring on subsequent calls to `dyninit` (and thus only the `lookup` calls are performed), we can enable code *relinking*. This allows us to *dynamically update* the module in a running program: we load a new version of a module, link it as usual, and then relink the other modules in the program to use the new module by calling their `dyninit` functions. Any needed state translation can be performed by the new module's `_init` function. Though not described here, we have fully explored this idea with an alternative version of DLpop [19], [18], and used it to build a dynamically updateable webserver, *FlashEd* [12].

The DLpop Library The DLpop interface in Figure 2 is implemented as a Popcorn library. The central element of the library is a type-safe implementation of the dynamic symbol table for managing the symbols exported by the running program. We first describe this symbol table, and then describe how the DLpop functions are used in conjunction with it.

DLpop encodes the dynamic symbol table as in ELF, as a list of hashtables mapping symbol names to their addresses, one hashtable per linked object file. Each time a new object file is loaded, a new hashtable is added. The dynamic symbol table is constructed at start-up time by calling the `dyninit` functions for all of the statically linked object files.

Each entry of the hashtable contains the name, value, and type representation of a symbol in the running program, with the name as the key. So that entries have uniform type, we use existential types [31] to hide the actual type of the value:[3]

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

To update the table with a new symbol (the result of calling `update` from `dyninit`), we pack the value (say of type β) and type representation (of type $R(\beta)$) together in an existential package, hiding the value's type, and insert that package into the table under the symbol's key. When looking up a symbol expected to have type α, and given a term representation `r` of type $R(\alpha)$, we do the following. First, the symbol's name is used to index the symbol hashtable, returning a package having type $\exists\beta.\beta \times R(\beta)$. During unpacking, the tuple is destructed, binding a type variable β, and two term variables, `table_value` and `table_rep`, of type β and $R(\beta)$, respectively. We then call

checked_cast[α][β]`(r, table_rep, table_value)`

which compares `r` and `table_rep`, and coerces `table_value` from type β to type α if they match. This value is then returned to the caller. Otherwise, the exception `WrongType` is raised.

The DLpop library essentially consists of wrapper functions for `load` and the dynamic symbol table manipulation routines:

[3] The type `<`τ_1, τ_2`> hashtable` contains mappings from τ_1 to τ_2.

`dlopen`

Recall that `dlopen` takes as its argument the name of an object file to load. First it opens and reads this object file into a `bytearray`. Because of the compilation strategy we have chosen, all loadable files should export a single symbol, the `dyninit` function. Therefore, we call load with the `dyninit` function's type and the `bytearray`, and should receive back the `dyninit` function itself as a result. If load returns `NONE`, indicating an error, `dlopen` raises the exception `FailsTypeCheck`. Otherwise, a new hashtable is created, and a custom `update` function is crafted that adds symbols to it. The returned `dyninit` function is called with this custom `update` function, as well as with a `lookup` function that works on the entire dynamic symbol table. After `dyninit` completes, the new hashtable is added to the dynamic symbol table, and then returned to the caller with abstract type `handle`.

`dlsym`

This function receives a type argument (call it α) and three term arguments: a `handle`, `h`; a string representing the symbol name, `s`; and the representation of the type α, `r`. Because the `handle` object returned by `dlopen` is in actuality the hashtable for the object file, `dlsym` simply attempts to look up the given symbol in that hashtable, following the procedure outlined above, raising the exception `SymbolNotFound` if the symbol is not present, or `WrongType` if the types do not match.

`dlclose`

The `dlclose` operation simply removes the hashtable associated with the `handle` from the dynamic symbol table. Future attempts to look up symbols using this handle will be unsuccessful. Once the rest of the program no longer references the handle's object file, it will be safely garbage-collected.

As a closing remark, we emphasize the value of implementing DLpop. We have not intended DLpop to be a significant contribution in itself; rather, the contribution lies in the *way* in which DLpop is implemented. By using TAL/Load, much of DLpop was implemented within the verifiable language, and was therefore provably safe. Only load and λ_R constitute trusted elements in its implementation, and these elements are themselves small. If some flaw exists in DLpop, the result will be object files that fail to verify, not a security hole.

We should point out that the implementation described here (and measured in the next section) is the first of two DLpop implementations. Our most recent implementation, described fully in [18], differs in two key ways from the one described here. First, rather than perform the dynamic transformation for files within the compiler, we do it source-to-source, preceding compilation. Decoupling the transformation from the compiler results in a more modular and flexible implementation, but required the addition of some features to Popcorn. Second, the newer implementation is more full-featured. It supports loading modules with mutually-recursive references, and allows for dynamically updating a module, as described above. The principles behind the two implementations are essentially the same.

4 Measurements

Much of the motivation behind TAL and PCC is to provide safe execution of untrusted code without paying the price of byte-code interpretation (as in the JVM) or sandboxing (as in the Exokernel [11]). Therefore, while the chief goal of our work is to provide flexible and safe dynamic linking for verifiable native code, another goal is to do so efficiently.

In this section we examine the time and space costs imposed by `load` and DLpop. We compare these overheads with those of DLopen (using the ELF implementation) and show that our overheads are competitive. In particular, our run-time overhead is exactly the same, and our space overhead is comparable. The verification operation constitutes an additional load-time cost, but we believe that the cost is commensurate with the benefit of safety, and does not significantly reduce the applicability of dynamic linking in most programs. All measurements presented in this section were taken on a 400 MHz Pentium II with 128 MB of RAM, running Linux kernel version 2.2.5. DLopen/ELF measurements were generated using `gcc` version egcs-2.91.66.

4.1 Time Overhead

The execution time overhead imposed by dynamic linking, relative to Popcorn programs that use static linking only, occurs on three time scales: run-time, load-time, and start-time. At run-time, each reference to an externally defined symbol must be indirected through the GOT. At load-time, the running program must verify and copy the loaded code with `load`, and then link it by executing its `dyninit` function. At startup, statically linked code must construct the initial dynamic symbol table. DLopen/ELF has similar overheads, but lacks verification and its associated benefit of safety.

Run-time Overhead In most cases, the only run-time overhead of dynamic code is the need to access imported symbols through the GOT; this overhead is exactly the same as that imposed by the ELF approach. Each access requires one additional instruction, which we have measured in practice to cost one extra cycle. A null function call in our system costs about 7 cycles, so the dynamic overhead of an additional cycle is about 14%.

For imported values of abstract type, there is also the cost of the null check before accessing each GOT element. However, we have yet to see this overhead occur in practice. Most files do not export abstract values, but instead "constructor" functions that produce abstract values; an exception in our current code base is the Popcorn `Core` library, which defines `stdin`, `stdout`, and `stderr` to have abstract type `FILE`. These cases typically define the abstract type to allow a null value (a sort of abstract `option` type), meaning that a null-check would have occurred anyway.

Load-time Overhead The largest load-time cost in DLpop is verification. Verification in `load` consists of two conceptual steps, disassembly and verification, as pictured in Figure 1, and described in Section 2.2. Verification itself is performed in two phases: consistency checking (labeled *typecheck* in the figure) and interface checking (labeled $t = \mathit{typeof}(vs)$? in the figure). For the `loadable.pop` file, presented in Figure 3, the total time of these operations is 47 ms, where 2% is disassembly, 96% is consistency checking, and the remaining 2% is interface checking. Detailed measurements concerning the cost of TAL verification may be found in [16], which notes that in general, verification costs are linear in the size of the file being verified.

The remaining cost is to copy the verified code and to execute the file's `dyninit` function. For `loadable.pop`, the total cost of these two operations is negligible: about 0.73 ms. This time is roughly twice the time of 0.35 ms for DLopen/ELF. The main difference here is simply that the ELF loader is more optimized. Because of its small weight relative to verification, there is little reason to optimize linking in DLpop.

Verification is by far the most expensive load-time operation, but its cost could be reduced, in three ways. First, the verification code could be more optimized for speed. In particular, proof-carrying code's Touchstone compiler [34] has demonstrated small verification times, albeit with a different type system, and even TAL's implementors recognize that further gains could be made [16]. Furthermore, disassembly has not been optimized. Second, verification could be performed in parallel with normal service. After verification completes, only linking remains, which has negligible overhead. Finally, in the case of a trusted system, we could turn off the consistency-checking phase during verification, since it can be run for each loaded file on some other machine. Leaving on link-checking and interface-checking still ensures that the loaded code meshes with the running program at the module level, but trusts that the contents of the loaded module are well-formed. Since consistency-checking is the most time-consuming operation, we greatly reduce our total update times as a result. Breaking up the verification operation onto server and client machines has been explored for Java in [39].

Even with current overheads, verification occurs but once per extension, and so should not pose a problem for most applications. Applications that load code at larger time scales, and/or for which loaded code is long-lived, will amortize the cost of verification over the entire computation. Long running systems that load extensions or updates, such as operating systems and network servers, and productivity applications that use dynamically loaded libraries fall into this category. Even those applications for which loaded code is short-lived, *e.g.*, agent systems, could be accommodated, because while verification time may be large, execution time (thanks to native code) will be small, balancing out the total cost.

Start-time Overhead At start-time, before execution begins, each statically linked file's `dyninit` function is executed to create the initial dynamic symbol table for the program. In addition, the program type interface, generated by the linker, is properly instantiated for use by load. The costs of these operations depend on the number of symbols and type definitions exported by each file, and which libraries are used. A typical delay is on the order of tens of milliseconds, which is meaningless over the life programs that will perform dynamic linking.

In contrast, ELF imposes no start-time cost, because no type interface is used, and because the static linker generates the hashtables that make up the dynamic symbol table, storing them in the object file. This implementation trades space for time.

4.2 Space Overhead

Both DLpop and DLopen/ELF increase the size of object files relative to their compilation without dynamic linking support. Based on some simple measurements, they appear to be fairly comparable in practice. For the most part the per-symbol costs for DLpop are higher than that of DLopen/ELF, but there is a significantly smaller fixed cost. For the remainder of this section we break the down the space costs of DLpop, and compare them to those of DLopen/ELF.

For both imported and exported symbols, DLpop imposes three space costs: the string representation of the symbol name,[4] its type representation, and the instructions in the dyninit function that perform its linking. For imported symbols, there is the additional cost of the symbol's GOT slot and its default value. These costs are summarized in Table 1, and compared to the overheads DLopen/ELF. DLopen/ELF overheads were determined from [40] and from examining object files on our platform. The fixed cost was estimated by subtracting the per-symbol costs from the total calculated overhead shown in Figure 7.

The per-symbol cost of DLpop is about one and a half times as much as DLopen/ELF when not including type representations t. Type representations tend to be large, between 128 and 200 bytes for functions, increasing total overhead when they are considered. We mitigate this cost somewhat by sharing type representations among elements of the same type. One factor that adds to function type representation size is that the representation encodes not only the types of the function arguments and returned values, but also the calling convention. This fact suggests that sharing type components among representations would net a larger savings, since the calling convention is the same for all Popcorn functions. We could also reduce per-symbol overhead by eliminating dyninit and moving the linking code into the DLpop library. However, dyninit is a convenient, flexible way to perform linking, justifying the extra space cost.

DLopen/ELF has a much higher fixed space cost than DLpop. This comes from a number of sources, including load-time and unload-time code sequences, and datastructures that aid in linking. In ELF, each of the hashtables of the dynamic symbol table is constructed at *compile-time* and stored in the object file. Some of the hashtable overhead is per-symbol, but there is also a large fixed

[4] Popcorn strings have a length field and an extra pointer (for easier translation to/from C-style strings), adding 2 words to a C-style representation.

	DLpop						DLopen
	symbol	`dyninit`	type	GOT	default		(ELF)
	name	function	rep	slot	value	total	
import function	$8 + l$	24	t	8	8	$48 + l + t$	$58 + l$
data	$8 + l$	24	t	8	$8^\star$	$48 + l + t$	$32 + l$
export	$8 + l$	24	t	-	-	$32 + l + t$	$24 + l$
fixed						4	˜2500

Table 1. Object file overheads, in bytes, for both DLpop and ELF. DLpop overheads are broken down into component costs; l is the length of a symbol's name and t is the size of its TAL type representation.

cost for the empty buckets in order to improve the hash function accuracy. In DLpop, these tables are constructed at start-time, creating a start-up penalty but avoiding the extra space cost per object file.

Figure 7 compares DLpop to DLopen/ELF for some benchmark files. Each of the four clusters of bars in the graph represents a different source file, with varying numbers of imported and exported functions, notated x**i** y**e** at the bottom of the cluster, where x and y are the number of imports and exports, respectively. When there is one exported function, its code consists of calling all of the imported functions; when there are fifteen functions, each one calls a single imported function. All functions are `void (void)` functions.[5] Each bar in the cluster represents a different compilation approach. The leftmost is the standard DLpop approach, and the rightmost is DLopen/ELF. The center bar is DLpop without the sharing of type representations, to show worst case behavior (when sharing, only one type representation for `void (void)` is needed). Each bar shows the size of object files when compiled statically, compiled to export symbols to dynamic code, and compiled to be dynamically loadable (thus importing and exporting symbols). The export-only case is not shown for ELF, as this support is added at static link time, rather than compile-time.

The figure shows that DLpop is competitive with DLopen/ELF. The figure also illustrates the benefit of type representation sharing; the overhead for the 15**i** 15**e** when not sharing is almost twice that when sharing is enabled. As the number of symbols in the file increases, the ELF approach will begin to outperform DLpop, but not by a wide margin for typical files (exporting tens of symbols). In general, we do not feel that space overheads are a problem (nor did the designers of ELF dynamic linking, it seems). We could structure our object files so that the `dyninit` function, which is used once, and type representations, which are used infrequently, will not affect the cache, and may be easily paged out. Type representations are highly compressible (up to 90% using

[5] This is the Popcorn (C-like) notation for the type `unit` → `unit`.

[⋆] For one-word values, this is the cost of the value plus a pointer; structured values are larger.

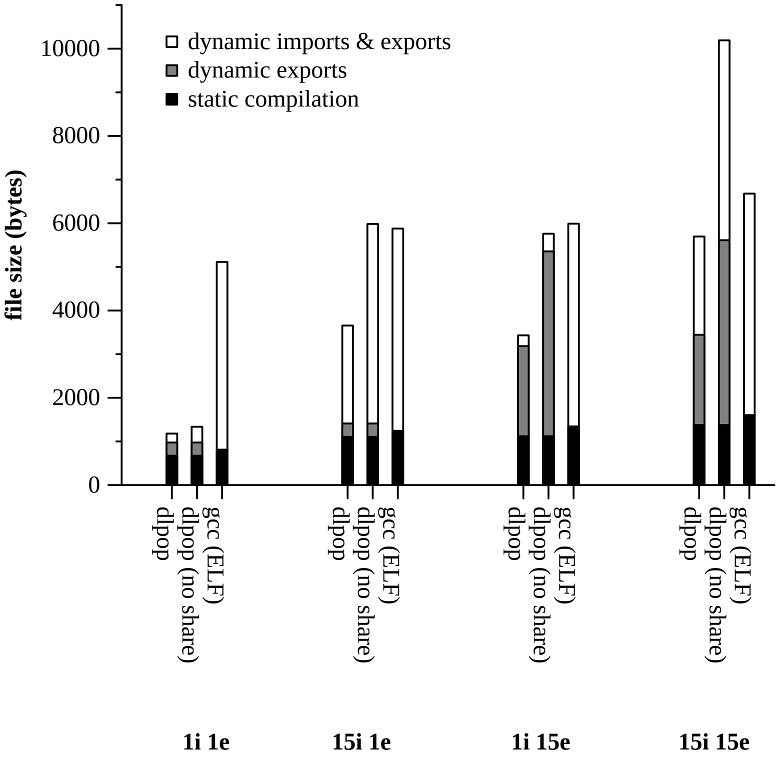

Fig. 7. Comparing the space overhead of DLpop, DLpop without type representation sharing, and DLopen/ELF for some microbenchmarks.

`gzip`), and therefore need not contribute to excessive network transmission time for extensions.

5 Programming other Linking Strategies (Related Work)

Using our framework TAL/Load, we can implement safe, flexible, and efficient dynamic linking for native code, which we have illustrated by programming a safe DLopen library for Popcorn. Many other dynamic linking approaches have been proposed, for both high and low level languages. In this section we do two things. First, we describe the dynamic linking interfaces of some high level languages, describe their typical implementations, and finally explain how to program them in TAL/Load, resulting in better security due to type safety and/or reduced TCB size. Second, we look at some low-level mechanisms used to implement dynamic linking, and explain how we can program them in our framework. Overall, we demonstrate that TAL/Load is flexible enough to encode typical dynamic linking interfaces and mechanisms, but with a higher level of safety and security.

5.1 Java

In Java, user-defined classloaders [23] may be invoked to retrieve and instantiate the bytes for a class, ultimately returning a `Class` object to the caller. A classloader may use any means to locate the bytes of a class, but then relies on the trusted functions `Classloader.defineClass` and `Classloader.resolveClass` to instantiate and verify the class, respectively. When invoked directly, a classloader is analogous to `dlopen`. Returned classes may be accessed directly, as with `dlsym`, if they can be cast to some entity that is known statically, such as an interface or superclass. In the standard JVM implementation, linking occurs incrementally as the program executes: when an unresolved class variable is accessed, the classloader is called to obtain and instantiate the referenced class. In the standard JVM implementation, all linking operations occur within the TCB: checks for unresolved class variables occur as part of JVM execution, and symbol management occurs within `resolveClass`.

We can implement classloaders in TAL/Load by following our approach for DLpop: we compile classes to have a GOT and an `dyninit` function to resolve and register symbols. A classloader may locate the class bytes exactly as in Java (*i.e.*, through any means programmable in TAL), and `defineClass` simply becomes a wrapper for a function similar to `dlopen`, which calls `load` and then invokes the `dyninit` function of the class with the dynamic symbol table.

To support incremental linking, we can alter the compilation of Java to TAL (hypothetically speaking) in two ways. We first compile the GOT, which holds references to externally defined classes, to allow null values (in contrast to DLpop where we had default values). Each time a class is referenced through the GOT, a null check is performed; if the reference is null then we call the classloader to load the class, filling in the result in the GOT. Otherwise, we simply follow the pointer that is present. As in the strategy depicted in Figure 6, the `dyninit` function no longer fills in the GOT at load-time; it simply registers its symbols in the dynamic symbol table. This approach moves both symbol management and the check for unresolved references into the verifiable language, reducing the size of the TCB.

5.2 Windows DLLs and COM

Windows allows applications to load Dynamically Linked Libraries (DLLs) into running applications, following an interface and implementation quite similar to DLopen and ELF, respectively, with some minor differences (see Levine [24, pps 217–222]). Like DLopen and ELF, DLLs are not type-safe and would therefore benefit in this regard from an implementation in TAL/Load.

DLLs are often used as vehicle to load and manipulate Common Object Model [7] (COM) objects. COM objects are treated abstractly by their clients, providing access through one or more interfaces, each consisting of one or more function pointers. All COM objects must implement the interface **IUnknown**, which provides the function `QueryInterface`, to be called at runtime to determine if the object implements a particular interface. `QueryInterface` is called with the globally unique identifier (GUID) that names the desired interface. GUIDs are not incorporated into the type-system (at least not for source languages like C and C++), and thus, as with `dlsym`, the user is forced to cast the object's returned interface to the type expected, with a mistake likely resulting in a crash.

Implementing COM in TAL/Load would be straightforward, with the added benefit of proven type-safety for interfaces. `QueryInterface` could be changed to take type parameter $R(t)$ in addition to the GUID of the expected interface, ensuring the proper type of the returned interface.

5.3 OCaml Modules

Objective Caml [27] (OCaml) provides dynamic linking for its bytecode-based runtime system with a special *Dynlink* module; these facilities have been used to implement an OCaml applet system, MMM [37]. Dynlink essentially implements `dlopen`, but not `dlsym` and `dlclose`, and would thus be easy to encode in TAL/Load. In contrast to the JVM, OCaml does not verify that its extensions are well-formed, and instead relies on a trusted compiler. OCaml dynamic linking is similar to that of other type-safe, functional languages, *e.g.* Haskell [36].

A TAL/Load implementation of the OCaml interface would improve on its current implementation [27] in two ways. First, all linking operations would occur outside of the TCB. Second, extension well-formedness would be verified rather than assumed.

5.4 Units

Units [13] are software construction components, quite similar to modules. A unit may be dynamically linked into a static program with the **invoke** primitive, which takes as arguments the unit itself (perhaps in some binary format) and a list of symbols needed to resolve its imports. Linking consists of resolving the imports and executing the unit's initialization function. **Invoke** is similar to `dlopen`, but the symbols to link are provided explicitly, rather than maintained in a global table.

Units could be implemented following DLpop, but without a dynamic symbol table. Rather than compiling the `dyninit` function to take two functions, `lookup` and `update`, it would take as arguments the list of symbols needed to fill the imports. The function would then fill in the GOT entries with these symbols, and then call the user-defined `_init` function for the unit. The implementation for **invoke** would call load, and then call the `dyninit` function with the arguments supplied to **invoke**.

The current Units implementation [13] is similar to the one we have described above, but is written in Scheme (rather than TAL), a dynamically typed language. Therefore, while linking errors within `dyninit` may be handled gracefully in our system (since they will result in thrown exceptions), in Scheme they will result in run-time type errors, halting system service. Alternatively, run-time type checks would have to be provided for each access of the GOT.

5.5 SPIN

The extensible operating systems community has explored a number of approaches to dynamic linking. For example, the SPIN [4] kernel may load untrusted extensions written in the type-safe language Modula-3. In SPIN, dynamic linking operates on objects called domains [38], which are collections of code, data, and exported symbols. Domains are quite similar to Units, with the functionality of **invoke** spread among separate functions for creation, linking, and initialization, along with other useful operations, including unlinking and combining. All of these operations are provided by the trusted `Domain` module. Furthermore, all operations are subject to security checks based on runtime criteria. For example, when one domain is linked against the interface of another, the interface seen may depend on the caller's privilege.

We can implement domains using techniques described above, with the addition of filters to take security information into account. TAL/Load would improve on the security of the current SPIN implementation in the same ways as OCaml: less of the domain implementation must be trusted, and integrity of extensions can be verified, rather than relegated to a trusted compiler.

5.6 TMAL

The TAL module system implemented for TALx86, MTAL (Modular Typed Assembly Language [15]), provides a typed version of standard static linking facilities. Typed Module Assembly Language (TMAL) [10] is an alternative module system for TAL that provides a different model of linking, including dynamic linking. Our work in TAL/Load is an extension to TAL to allow dynamically linking MTAL modules. Therefore, TMAL and TAL/Load can be seen as two ways to solve similar problems. TMAL has not been implemented.

TMAL adds a simple notion of first-class modules to TAL; by using explicit coercions accompanied by runtime checks, the type system remains decidable. The operations provided for TMAL module values are much like those for SPIN domains, described above. Two modules can be linked together to form a third module, and the circumstances of linking can be customized. In particular, coercions are provided to remove exported names from a module, and to rename its types and/or values. In addition, modules can be linked with symbols from the program (rather than other modules).

TMAL also provides primitives for reflection. In particular, TMAL's `dlsym_v` is essentially the same as DLpop's `dlsym`. MTAL, and thus TAL/Load, makes the simplification that all named types are global, as we explained in §2.1. As a module is loaded, its type components are added into the global namespace. However, in TMAL, first-class modules can contain type components, which introduces a level of hierarchy. As a result, TMAL provides a `dlsym_t` operation for looking up a type component of a module, to be used prior to retrieving a value that has that type.

Finally, TMAL provides primitives for creating and loading dynamically-linked libraries, respectively; the latter operation is similar to `load`, and the former is something that we do at compile-time.

The major difference between TAL/Load and TMAL is that TAL/Load is intended for *programming* the sorts of operations that TMAL provides as primitive; the result is a smaller TCB. On the other hand, the goal of TMAL is to preserve and statically verify the constraints expressed by the source module language at the assembly language level. We could easily implement the majority of TMAL using TAL/Load, where the notion of `handle` as implemented in DLpop is analogous to a first-class module TMAL. Breaking the linking functionality out of DLpop's `dlopen` into the various TMAL linking primitives would be straightforward for values, but tricky for types, though still possible; *e.g.* our technical report [20] describes a way to implement load to hide global types from loaded modules, and we could use existential types to implement something like `dlsym_t`. However, in such an implementation, some properties that could be statically verified by TMAL, would have to be dynamically checked by load.

On the other hand, programming provides flexibility. In the case of values, we could even program additional module coercions, since they essentially control a module's symbol table. For example, we could add security information to the table to be used during linking, as is done in SPIN.

5.7 Low-level Dynamic Linking Mechanisms

A useful reference of low-level, dynamic linking mechanisms may be found in Franz [14]. One technique that he presents, which has been used to implement some versions of DLopen (as opposed to the ELF methodology [40]), is called load-time rewriting. Rather than pay the indirection penalty of using a GOT, the dynamic linker rewrites each of the call-sites for an external reference with the correct address.

This technique is a simple form of run-time code generation. Popcorn and the TAL implementation provide facilities for type-safe run-time code generation, called Cyclone [21], that we can use to implement load-time rewriting. Rather than compile functions to indirect external references through a GOT, we instead create template functions that abstract their external references. When `dyninit` is called, each template function is invoked with the appropriate symbols (found by calling `lookup`), returning a custom version of the original function, closed with respect to the provided symbols. This function is then registered with the dynamic symbol table using `update`. The advantage of this approach is that the process of rewriting can be proven completely safe.

There are two notable disadvantages. First, mutually recursive functions are problematic because their template functions must be called in a particular order. One possible solution is to use one level of indirection for recursive calls, backpatching the correct values. Another disadvantage is that template functions make copies of the functions they abstract, rather than filling in the holes in place; Cyclone's approach is more general, but not necessary in our context. However, the overall cost of doing this should be low (especially relative to verification). We plan to experiment with this approach in future work.

6 Conclusions

We have designed, implemented, and demonstrated TAL/Load, the first complete type-safe dynamic linking framework for native code. Our approach has many advantages:

- It supports linking of native code so dynamic extensions may be written in many source languages.
- It is composed largely of components already present in the TAL trusted computing base, therefore its addition does not overly complicate the code verification system.
- It is expressive enough to support a variety of dynamic linking strategies in an efficient manner.

Furthermore, there is nothing specific to TAL in this strategy—we believe that in principle it would also be applicable to Proof Carrying Code (with some changes to verification condition generation). We see this work as the first step in a larger study of type-safe extensible systems.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
2. A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, Jan. 2000.
3. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
4. B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain Resort, Colorado, 1995.
5. H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.
6. C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, June 2000.
7. Microsoft COM technologies. `http://www.microsoft.com/com/default.asp`.
8. K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, Sept. 1998. Extended version published as Cornell University technical report TR98-1721.
9. DLOPEN(3). Linux Programmer's Manual, December 1995.
10. D. Duggan. Sharing in Typed Module Assembly Language. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.
11. D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

12. Flashed webserver. `http://flashed.cis.upenn.edu`.
13. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of SIGPLAN International Conference on Programming Language Design and Implementation*, pages 236–248. ACM, June 1998.
14. M. Franz. Dynamic linking of software components. *IEEE Computer*, 30(3):74–81, March 1997.
15. N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, 1999.
16. D. Grossman and G. Morrisett. Scalable certification for Typed Assembly Language. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.
17. R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, Jan. 1990.
18. M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.
19. M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2001. To appear.
20. M. Hicks and S. Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, 2000.
21. L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Journal of Higher-Order and Symbolic Computation*, 12(4), 1999. An earlier version appeared in Partial Evaluation and Semantics-Based Program Manipulation, January 22-23, 1999.
22. Hotjava browser. `http://java.sun.com/products/hotjava/index.html`.
23. Basics of java class loaders, 1996. `http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html`.
24. John R. Levine. *Linkers and Loaders*. Morgan-Kaufman, 2000.
25. D. Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, Ithaca, NY 12853-7501, January 1998.
26. X. Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, Jan. 1994.
27. X. Leroy. *The Objective Caml System, Release 3.00*. Institut National de Recherche en Informatique et Automatique (INRIA), 2000. Available at `http://caml.inria.fr`.
28. M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1997.
29. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
30. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.
31. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
32. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

33. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.
34. G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, Jan. 1997.
35. G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Oct. 1996.
36. J. Peterson, P. Hudak, and G. S. Ling. Principled dynamic code improvement. Technical Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997.
37. F. Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28, pages 1365–1371. Elsevier, May 1996.
38. E. G. Sirer, M. E. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe dynamic linking in an extensible operating system. In *First Workshop on Compiler Support for System Software*, Tucson, February 1996.
39. E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, December 1999.
40. Tool Interface Standards Committee. Executable and Linking Format (ELF) specification. `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`, May 1995.
41. D. Walker and G. Morrisett. Alias types for recursive data structures. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.
42. S. Weirich. Type-safe cast. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming Languages*, pages 58–67, September 2000.