Safe and Flexible Dynamic Linking of Native Code*

Karl Crary
Carnegie Mellon University

Michael Hicks
University of Pennsylvania

Stephanie Weirich
Cornell University

Abstract

We present the design and implementation of a 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, 7] and applet-based web browsers [16]. Extensible systems that lack safety typically suffer from a lack of robustness, the frequent crashes of Windows applications due to DLL incompatibility being a conspicuous example. Those 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 [22] (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.

*Submitted for publication

An alternative approach to verifiable bytecode is verifiable native code, first proposed by Necula and Lee [27] in the context of 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 [26, 18, 2].

While verifiable native code systems are fairly mature, all lack a well-designed methodology for dynamic linking, the mechanism used to achieve extensibility. Within PCC, for example, dynamic linking has only been performed in an ad-hoc manner, entirely within the TCB [27]. Most general-purpose languages support dynamic linking [3, 6, 8, 20, 28, 29], 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 operations, *e.g.*, loading and linking.
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 a dynamic linking framework for verifiable native code. We have developed this framework in the context of Typed Assembly Language [26] (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.[1]

To demonstrate the flexibility and efficiency of our framework, we have used it to program a type-safe implementation of DLopen [6], a UNIX library that provides dynamic linking services to C programs. Our version of

[1]The interested reader is referred to the companion technical report [14] for the full formal framework and soundness proof.

The first argument to the function do_op indicates the operation to perform, and the second argument contains a tuple of the two arguments. A client of this module might be something like:

```
fun add_int (x,y) = Arith.do_op (0,(x,y))
fun sub_int (x,y) = Arith.do_op (1,(x,y))
```

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

```
val Arith_do_op : int * (int * int) ref = ...
fun add (x,y) = !Arith_do_op (0,(x,y))
fun sub (x,y) = !Arith_do_op (1,(x,y))
```

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 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 generative type. Consider an implementation of Arith that uses datatypes:

```
structure Arith =
struct
  datatype op = Add | Sub | Mult | Div
  fun do_op (Add,(x,y)) = x + y
    | do_op (Sub,(x,y)) = x - y
    | do_op (Mult,(x,y)) = x * y
    | do_op (Div,(x,y)) = x / y
end
```

The code for the module is the same, but now we've used a generative datatype to implement the operation. If we attempt to close the client code as before, we get:

```
val Arith_do_op : Arith.op * (int * int) ref = ...
fun add (x,y) = !Arith_do_op (Arith.Add,(x,y))
fun sub (x,y) = !Arith_do_op (Arith.Sub,(x,y))
```

But we still have the external references to the datatype Arith.op itself, as well as to its constructors Add and Sub. The question becomes, how can we create holes for generative types and type constructors in the same way that we create holes for values? Essentially we need a way for the loaded code to state its assumptions concerning generative types, and a way for the running program to validate those assumptions. Unfortunately, it is not feasible to do so outside the trusted computing base. Because load typechecks the given module, types play a major role in its operation. Therefore it makes sense that to manipulate shared types effectively we must incorporate them into our framework. As we describe in the next subsection, our implementation does not greatly increase the TCB as these checks are based on TAL's framework for static link verification [10].

To allow loaded code to refer to externally defined types, we do the following. A module may declare a type interface (X_I, X_E), which is a pair of maps from type name to implementation. The intuition is that X_I mentions the generative types provided by other modules, and X_E mentions generative types defined by this one. By not including the implementation of the type inside the map X (just mentioning its name), we can use this mechanism to implement abstraction as well. As an example, the type interface of the client code above would be something like:

$$(\{op = \texttt{Add} \mid \texttt{Sub} \mid \texttt{Mult} \mid \texttt{Div}\}, \{\})$$

and the interface for Arith would be the reverse:

$$(\{\}, \{op = \texttt{Add} \mid \texttt{Sub} \mid \texttt{Mult} \mid \texttt{Div}\})$$

The runtime system 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 generative type imports of the new module are consistent with the program interface, and that the exports of the new module do not redefine, or define differently, any types in the program 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 allows a program to manipulate objects of an undefined type abstractly.

We have developed a formal calculus for our framework and have proven it sound. While this formalization is interesting, the 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 [14] for the full theoretical treatment.

2.2 Implementation

We have implemented TAL/Load it in the TALx86 implementation of TAL. The key component of TAL/Load is the load primitive, whose actions are illustrated in Figure 1. Using components of the TALx86 system, load performs two functions:

1. **Disassembly** The first argument, which provides the expected type of the exports, must be disassembled into the internal representation of TAL types. This type should always be of tuple type, where each element type represents the type of one of the object file's exports. The second argument to load is a byte array representing the object file and the typing annotations on it, depicted in the figure as two distinct arguments. This information is parsed by the TAL disassembler to produce a TAL implementation.

2. **Verification** The TAL implementation is then verified using the current program's type interface, following the procedure described in the previous subsection. If verification succeeds, the result is a list of exported values and the new type interface (not shown). 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 new type interface replaces the original interface. On failure, 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. Two pieces of trusted functionality are needed, however, beyond that already provided by TAL: the ability to represent types as runtime values, and the maintenance of the program's "current" type interface at runtime.

Passing Types at Runtime As TAL uses a type-erasure semantics we need some way to pass a runtime representation of the type argument to load. We do this by extending

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

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

Figure 2: DLpop library interface

and compiler [25]. Our version, called DLpop, follows the interface in Figure 2, which we describe in detail below:

- `handle dlopen(string fname)`
  Given the name of a TAL object file, `dlopen` dynamically loads the file and returns a `handle` to it for future operations. Imports (*i.e.,* symbols declared `extern`) in the file 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.

- `a dlsym<a>(handle h, string symbol)`
  Given the handle for an object file and a string naming the symbol, `dlsym` returns a pointer to the symbol's value. In DLopen, this function returns an untyped pointer, of C-type `void *`, which requires the programmer to perform an unchecked cast to the expected type. By contrast, our version takes a type argument, denoted `<a>`, to indicate the expected (pointer) type; this type is checked against the actual type at runtime. 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.

  To implement type-passing style, `dlsym` uses a R-type internally, as described in Section 2.2. We hide this complexity from the user by making `dlsym` a special keyword. When encountering a call to `dlsym`, the compiler automatically constructs its type argument's representation, signaling an error when this cannot be done statically, and turns the `dlsym` call into a call to `dlsym_internal`, which has type:[3]

  ```
  a dlsym_internal<a>(handle h, string symbol,
                      R(a) typerep)
  ```

- `void dlclose(handle h)`
  In DLopen, `dlclose` *unloads* the file associated with the given handle. In particular, the file's symbols are removed from the DLopen symbol table, 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 the symbols from the symbol table; if the user program does not reference the object file, then it is unreachable and can be garbage collected.

[3] For purposes of exposition, we will include the R-type constructor in Popcorn syntax, even though it is an internal type. To mark this distinction, we will use TAL notation for type constructor application (`R(a)`) instead of that of Popcorn (`<a>R`).

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");
    bar(3);
    dlclose(h);
}
```

Figure 3: DLpop dynamic loading example

The current version of DLpop does not implement all of the features of DLopen. For example, it does not recursively load object files upon which a dynamically loaded file depends, and cannot load files with mutual dependencies. We also do not support user-defined finalization of unloaded object files. However, we foresee no technical difficulties in adding these features, and plan to do so in future work.

Figure 3 provides an example of this dynamic linking strategy in Popcorn. The user would statically link the file `main.pop`, which, during execution, dynamically loads the object file `loadable.o` (the result of compiling `loadable.pop`) to resolve the symbol "`bar`". The dynamically linked file also makes external references to the function "`foo`", which are resolved at load time from the exports of `main.pop`.

3.2 Implementing DLpop in TAL/Load

Our implementation of DLpop is similar to implementations of DLopen that follow the ELF standard [31] 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 datum in the running program. Exported symbols are collected in a dynamic symbol table to facilitate this task, first constructed at static link time and then added to as new files are loaded. The DLopen functions themselves are normally implemented in a library, `libdl`.

We describe our DLpop implementation below, which is similar in spirit to the ELF approach. However, DLpop is inherently more secure than DLopen: because it is written in TAL/Load, all operations are verifiably safe. A mistake in our implementation will result only in incorrect behavior,

DLpop encodes the dynamic symbol table as a string-keyed dictionary, mapping object file names to dictionaries of symbols; the statically linked program is treated as a single object file for our purposes here. Each time a new object file is loaded, a new symbol dictionary is added, indexed by its name.

Each entry of a symbol dictionary contains the name, value, and type representation of a symbol in the running program, with the name as the key. So that dictionary entries have uniform type, we use existential types [24] to hide the actual type of the value:[4]

$$\texttt{symbol\_dict} : \texttt{<string,}\ \exists\alpha.\ (\alpha \times R(\alpha))\texttt{>}\ \texttt{dict}$$

To update the table with a new symbol, we pack the value and type representation together in an existential package, hiding the value's type, and insert that package into the dictionary under the symbol's key. When looking up a symbol, the dictionary returns an entry containing a value of some abstract type and a representation of that type. We then use checked_cast to compare that type with the expected type and (if they match) coerce the value to the expected type.

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

- `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 `init` function. Therefore, we call load with the `init` function's type and the `bytearray`, and should receive back the `init` function itself as a result. If load returns `NONE`, indicating an error, `dlopen` raises the exception `FailsTypeCheck`. Otherwise, a new symbol dictionary is created, and a custom `update` function is crafted that adds symbols to this dictionary. The returned `init` function is called with this custom `update` function, as well as with a `lookup` function that works on the entire symbol table. After `init` completes, the new symbol dictionary is added to the global table, and then returned to the caller with abstract type `handle`.

- `dlsym_internal`
  Because the `handle` object returned by `dlopen` is in actuality the symbol dictionary for the object file, `dlsym_internal` simply attempts to look up the given symbol in that dictionary, raising an exception if the symbol is not found or has improper type.

- `dlclose`
  The `dlclose` operation simply removes the symbol dictionary 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.

[4] The type $\texttt{<}\tau_1, \tau_2\texttt{>}\ \texttt{dict}$ contains mappings from τ_1 to τ_2.

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 [7]). 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 [31]) and show that our overheads are competitive. In particular, our run-time overhead is exactly the same, and our space overhead is comparable for typical programs. The verification operation constitutes an additional load-time cost, but we believe that the cost is commensurate with the benefit of safety, and doesn't 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.

The impatient reader may wish to skip to Section 5 to read more about how TAL/Load can implement other dynamic linking strategies.

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: start-time, load-time, and run-time. At startup, statically-linked code must register its symbols in the DLpop symbol table. At load-time, the running program must verify and copy the loaded code with load, and then link it by executing its `init` function. At run-time, each reference to an externally defined symbol must be indirected through the GOT. DLopen/ELF has similar overheads, but lacks verification and its associated benefit of safety.

4.1.1 Run-time Overhead

At run time, the only additional overhead on dynamic code is the need to access imported symbols through the GOT; this is exactly the same overhead 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%.

4.1.2 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 *verify* in the figure) and interface checking (labeled *compare types* in the figure). For the `loadable.pop` file, presented in Figure 3, the total time

tion. All functions are `void (void)` functions.[6] 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, or better than, DLopen/ELF in most cases. The figure also illustrates the benefit of type representation sharing; the overhead for the 15i 15e 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. 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 `init` function, which is used once, and type representations, which are used infrequently, won't affect the cache, and may be easily paged out. Type representations are highly compressible (up to 90% using `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 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 [17] 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 `init` 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 `init` 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. The `init` function no longer fills in the GOT at load-time; it simply registers its symbols with the 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 OCaml Modules

Objective Caml [20] (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 [29]. 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 [28].

A TAL/Load implementation of the OCaml interface would improve on its current implementation [20] in two ways. First, all linking operations would occur outside of the TCB. Second, extension well-formedness would be verified rather than assumed.

5.3 Units

Units [8] 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 `init` 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 `init` function with the arguments supplied to **invoke**.

The current units implementation [8] is similar to the one we have described above, but is written in Scheme (rather

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

[12] 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.

[13] M. Hicks. Dynamic software updating. Technical report, Department of Computer and Information Science, University of Pennsylvania, October 1999. Thesis proposal. Available at `http://www.cis.upenn.edu/~mwh/proposal.ps`.

[14] M. Hicks and S. Weirich. A calculus for dynamic loading. Technical Report MS-CIS-00-07, University of Pennsylvania, 2000.

[15] 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.

[16] Hotjava browser. `http://java.sun.com/products/hotjava/index.html`.

[17] The basics of java class loaders, 1996. `http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html`.

[18] D. Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, Ithaca, NY 12853-7501, January 1998.

[19] 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.

[20] X. Leroy. *The Objective Caml System, Release 2.02.* Institut National de Recherche en Informatique et Automatique (INRIA), 1999. Available at `http://pauillac.inria.fr/ocaml`.

[21] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1997.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[24] 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.

[25] 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.

[26] 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.

[27] G. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[28] 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.

[29] 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.

[30] 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.

[31] Tool Interface Standards Committee. Executable and Linking Format (ELF) Specification. `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`, May 1995.

[32] S. Weirich. Type-safe cast. In submission, Mar. 2000.