A Project Summary

High-Assurance Common Language Runtime

Andrew Appel (PI) David Walker Zhong Shao (PI) Valery Trifonov
Princeton University Yale University

Component software platforms, such as the Java Virtual Machine (JVM) and Microsoft's .NET Common Language Runtime (CLR), provide a secure virtual machine in which components developed independently and in different source languages can smoothly interoperate. Foundational Proof-Carrying Code (FPCC) and Certified Binaries (CB) are promising new technologies for building high-confidence software systems. We propose a new effort that extends and adapts ideas from FPCC and CB for use in a real VM such as JVM or CLR. We will apply the theoretical ideas from our current research, and the lessons learned from our prototypes, to a full-scale design. We will work with industrial laboratories (e.g., at Intel), advising them on design of low-level intermediate representations and type systems to use in their JIT compilers. We will also design new metadata protocols for more descriptive specification of interfaces, and use them to support interoperation between sophisticated components. Our proposed work is divided into three areas:

1. **High-level specifications for low-level software.** The bytecode verifier for CLR or JVM enforces only conventional object-oriented type safety. These types are too inflexible for real applications so developers often bypass the type system, producing code that steps outside the "managed" part of the VM; such components cannot be statically verified. We will show how to use higher-order logic to design much more general type systems.
2. **High-assurance virtual machine.** In principle the bytecode verifier enforces type safety and provides an ironclad guarantee that components respect their partners' API, but in practice this guarantee is only as strong as the implementation of the native libraries, the verifier, and the just-in-time (JIT) compiler. Experience has shown that bugs in the native interface, verifier, or JIT can lead to new system vulnerabilities and can be used by attackers to bypass protection measures. We plan to extend and apply our work on proof-carrying code to build higher-assurance, validated implementations of the JVM or CLR infrastructure.
3. **Certified libraries with application-specific properties.** We will extend our CLR or JVM implementation to support not only simple type safety but also more advanced properties such as resource bounds on memory and network bandwidth, consistent metadata protocols, generalized access control, and proper coordination of concurrent components. These application-specific properties are crucial for principled interoperation between large-scale systems. We'll build certified libraries and develop new technologies for specifying, composing, and verifying advanced properties.

We are well qualified for this work. Andrew Appel has more than 15 years experience in compiler design and implementation; he is a coauthor of the *Standard ML of New Jersey* (SML/NJ) research compiler, author of a leading textbook on compiler implementation, and is a Fellow of ACM. David Walker is the lead designer of "Typed Assembly Language," which is now the accepted state-of-the-art technology for certifying compilers. Zhong Shao has ten years experience in design of type-safe optimizing compilers, is a coauthor of SML/NJ, and designed its FLINT common typed intermediate language, with many influential publications on this technology. Valery Trifonov is an expert on object-oriented type systems and has done extensive work on designing new technologies for safe language interoperation.

B Project Description

B.1 Introduction

The goal of our research is to design platforms for secure operation of component-based systems where not all the components are fully trusted; and to have this technology be directly relevant to improving the security of commercial virtual machines such as Sun's Java or Microsoft's .NET.

Sun's Java architecture introduced a safe virtual machine in which an ensemble of software components developed independently of one another and possibly by different vendors could smoothly interoperate. The goal of Microsoft's Common Language Runtime (CLR) is to generalize this approach and allow components in many source languages to interoperate safely. CLR supports flexible interoperation by using a unified type system and by allowing components to share the same address space (which enables efficient communication and resource sharing). In both systems, each component is implemented as an object which specifies its services using a typed interface. A *bytecode verifier* ensures that the components implement the stated interfaces and satisfy predetermined safety properties. Since static checking enforces interface specifications, mutually untrusting components can interact knowing that their collaborators are well-behaved.

Type-based static checking for source languages such as Java and C# – and for their byte-code languages (JVM and the "managed CLR," respectively) – can provide a *soundness guarantee*, that if the program passes the check then certain (well specified) bad things cannot possibly happen at run time. This guarantee is at the heart of our research program: on the one hand, we will expand the guarantee so that even more bad things are explicitly prevented, and on the other hand we will use several techniques to validate the guarantee with very high assurance. In contrast, other static analysis techniques [51] are unsound: they do find many weaknesses but provide no guarantees. While unsound techniques can still be very useful in auditing legacy C and C++ code, they cannot provide the same high level assurance as sound static checking.

Sound static checking can detect and prevent many kinds of security weaknesses:

- **Viruses:** Many computer viruses exploit "buffer overrun" weaknesses in application programs; static checking completely prevents such attacks.
- **Inadvertent software bugs:** Some applications may be purchased from vendors with inadequate software-engineering practices. We may wish to run these on privileged machines (perhaps for less-than-critical calculations). But software bugs can cause a program not only to produce incorrect results, but also to read and/or corrupt data belonging to other programs. Static checking can guarantee confidentiality and integrity of other components' data. However, static checking will not (in general) be able to guarantee the correctness of the results computed by these applications.
- **Malicious insider at a software company:** A programmer might place intentionally malicious "Trojan horse" code in a software system to be run on a privileged machine. If the program is explicitly given sensitive data, it is difficult to protect the confidentiality and integrity of this data using any technique. But suppose the program is not explicitly given sensitive data – we would still like to prevent it from accessing sensitive data located elsewhere on the privileged host. In principle, static checking can enforce this "compartmentalization," but in practice if you let an attacker write part of the programs that will run on your system, he has an extremely powerful tool. Resistance against malicious program code is a weak point of existing static-checking systems [13]. *Our research directly addresses this weakness, providing strong compartmentalization even against malicious code.*
- **Systems built from components:** A component-based system is assembled from parts provided by many vendors. This increases the chance for buffer overruns, software bugs, and Trojan horses, and makes static checking even more useful.

Microsoft's .NET and Sun's JVM use static checking to enforce interfaces – this means they are good platforms for building more secure systems. But interface enforcement in CLR (and JVM) has two substantial limitations, which we propose to address in our research:

- Traditional verifiers enforce only conventional object-oriented type safety, so higher-level specifications cannot be enforced. Because conventional type systems are often too inflexible for real applications, developers often bypass the type system, producing code that steps outside the "managed" part of the VM; such components cannot be statically verified. Also, standard type systems cannot be extended to specify and check new sorts of program properties without re-engineering the verification software: properties such as resource bounds on memory and network bandwidth, proper coordination of concurrent components, generalized access control, and formal software protocols cannot be specified or checked. Our work on certified binaries [46] shows how to incorporate arbitrary assertions and proofs (in higher-order logic) into compiler intermediate languages, and how to use them to specify, compose, and verify advanced system properties.
- In principle the bytecode verifier enforces type safety and provides an ironclad guarantee that components respect their partners' API, but in practice this guarantee is only as strong as the implementation of the native-code libraries (for low-level services), the verifier, and the just-in-time (JIT) compiler. Experience has shown that bugs in the native interface, verifier, or JIT can lead to new system vulnerabilities and can be used by attackers to bypass protection measures. Our research in proof-carrying code [3] shows how to build high-assurance, validated implementations of this infrastructure.

In our current research we are taking the first key steps to solving both of these problems. At Princeton we have developed the theory and prototype of *foundational proof-carrying code* [5, 3], a technology for removing the entire JIT compiler (and most of the byte-code verifier) from the trusted computing base. This means that any security-critical (or interface-compromising) bugs in the JIT will be detected by our proof-checker. Our "foundational" checker itself is only about 3,000 lines of code – two orders of magnitude smaller than an optimizing JIT compiler, and one order of magnitude smaller than the trusted base of any competing system. Our theoretical work in semantic models of machines and type systems [30, 55, 6] establishes a basis on which we are engineering a prototype system.

At Yale we have developed *typed* common intermediate languages [46, 23, 44] that can accommodate not only the standard object-oriented model, but also generic (polymorphic) programming and Java-style reflection. The latter is crucial to dynamic extensibility and adaptive interoperation. Unlike JVM and CLR, our type system [46] is capable of expressing all valid propositions and proofs in higher-order predicate logic, so it can be used to capture and verify higher-level specifications of software components. The fact that we can internalize a very expressive logic into our type system means that formal reasoning, traditionally performed at the meta level, can now be expressed inside the intermediate language itself. For example, much of the past work on program verification using Hoare-like logics can now be made explicit in our intermediate languages.

We propose a new effort that extends and adapt these ideas for technology transfer to a real VM such as the Microsoft CLR (or the open-source "Mono Project" version of the CLR), or Sun's JVM (or IBM's open-source Jalapeno research platform for Java). We will apply the theoretical ideas from our current system, and the lessons learned from our prototypes, to a full-scale design. We will work with industrial laboratories (e.g., at Intel), advising them specifically on design of low-level intermediate representations and type systems to use in their JIT compilers. We will also design new metadata protocols for more descriptive specification of interfaces, and use them to support seamless interoperation between sophisticated software components.

B.2 Overview of Planned Research and Collaboration

The expected results from this project are: (1) a general framework for accurately specifying, composing and verifying higher-level properties for low-level software; (2) a prototype implementation of certifying compiler for the Microsoft Common Intermediate Language (CIL) and (together with Intel[1]) a realistic industrial-strength certifying JIT compiler for CIL; and (3) a set of general recipes for writing certified libraries and a type-safe dynamic linker. We divide our work into the following three areas:

1. **High-level specifications for low-level software.** Work in this area focuses on basic research. Our goal is to tackle the fundamental problem in constructing high-assurance software components, i.e., how to accurately specify, compose, and verify high-level specifications for low-level programs (including the "unmanaged" code in the .NET VM). We will work on **Logic-based type systems (LTS)** at Yale and Princeton to specify and verify low-level constructs such as stack allocation, mutable recursive data structures, exception handling, data layout with alignment and flexible tagging, and array-bounds checking elimination. We will extend the current **FPCC infrastructure** at Princeton to support new and more powerful type systems, develop semantic models for LTS, and build a highly efficient FPCC verifier. We will develop **LTS/FPCC interface** technologies for translating compiler intermediate languages annotated with LTS types into foundational proof-carrying code.
2. **High-assurance virtual machine.** Work in this area focuses on the technology transfer of results from Part 1 into realistic CLR implementations. We plan to extend our FLINT certifying compiler to support Microsoft Common Intermediate Language. We'll also work with people at Intel Labs, advising them on design of low-level intermediate representations and type systems to use in their JIT compilers. At Yale we'll develop a prototype certifying **FLINT/CIL** compiler that compiles CIL bytecodes into the logic-based typed assembly language (which uses LTS developed in Part 1). At Princeton we'll develop a prototype **CIL PCC** generator that translates FLINT/CIL assembly code into typed machine code. And we'll all work with Intel Labs to transfer our technologies developed under our prototype implementation into realistic industrial-strength **Certifying JIT compilers**.
3. **Certified libraries with application-specific properties.** Large software systems often have very specific rules and constraints. To achieve seamless interoperability, we must specify and verify these advanced system properties. At Princeton and Yale we'll develop technologies for specifying, composing, and verifying various kinds of **memory management APIs** (e.g., reference counting, garbage collection, region-based memory management). We'll also develop technologies for **type-safe dynamic linking**, build a set of libraries for enforcing consistent **metadata protocols and type-safe reflection**, and develop the basic methodologies for building libraries that enforce **advanced security protocols and accurate resource usage**.

We give a detailed description for each of these three areas in Sections B.3–B.5. We then summarize the broader impact (see Section B.6) and compare our work with other related research (see Section B.7). Finally we present the PIs' results from prior NSF supported grants (see Section B.8).

B.3 High-Level Specifications for Low-Level Software

The bytecode verifier for Microsoft CLR (and Java VM) enforces only conventional object-oriented type safety. CIL types are too inflexible for real applications, so developers often bypass the type system, pro-

[1] Intel Labs has an "Open Runtime Platform" project, one component of which is an open-source research JIT compiler for the Microsoft Common Intermediate Language interface. We are now beginning a collaboration with researchers in this project to make this JIT "type-preserving" so that it can generate proof-carrying code. Intel plans to publish all its results and make all the source code available under a BSD-style open-source license. All of our own work will be unrestricted by proprietary licenses.

ducing code that steps outside the "managed" part of the VM; such components cannot be statically verified.

The primary reason for such deficiency is of course that the type systems for these languages are too weak to certify the safety or other advanced properties on "unmanaged" code. Past researches on programming languages have focused on developing type systems for high-level languages (e.g., ML, Java); however, real software systems need building blocks written in "low-level" programming or machine languages. In addition, component libraries are typically transmitted in low-level bytecode or machine-code formats; making them secure also requires new technologies for specification and verification of low-level code.

We believe that the most fundamental question in building high-assurance CLR is on how to specify, compose, and verify high-level specifications for low-level programs. In our previous work [3, 46], we developed several new technologies to tackle this problem.

- **Foundational proof-carrying code.** To certify machine code, we use proof-carrying code (PCC) [37, 35]. PCC allows a code producer to provide a machine-language program to a host, along with a formal proof of its safety. The proof can be mechanically checked by the host; the producer need not be trusted because a valid proof is a good certificate of safety.

  The proofs in Necula's PCC systems [36, 11] are written in a logic extended with many language-specific typing rules. They can guarantee safety only if there is no bug in the verification-condition generator (VCgen), or in the typing rules, or in the proof checker. The VCgen is a fairly large program (23,000 lines of C code in the Cedilla Systems' Special J implementation of a JVM compiler [11]) so establishing its full correctness is a daunting task. The typing rules are also error-prone: League *et al* [24] recently discovered a serious bug in the Special J typing rules that would wreck the integrity of the entire PCC-based system.

  Foundational Proof-Carrying Code (FPCC) [5, 3] tackles these problems by constructing and verifying its proofs using strictly the foundations of mathematical logic (with no type-specific axioms). FPCC is more flexible and more secure because it is not tied to any particular type system and it has a smaller trusted base.

- **Certified binaries.** FPCC is expressive because it can certify any property expressible in higher-order predicate logic. However, logical specifications are not as modular and as user-friendly as type information.

  To generate FPCC we build certifying compilers that can automatically turn type information in the source programs into proofs for the FPCC machine code. The key idea here is to use typed intermediate languages (TIL) and typed assembly languages (TAL) to serve as intermediate steps during compilation. However, the type systems for existing TILs and TALs are too weak to typecheck complex low-level programming constructs.

  A *certified binary* (CB) [46] is a value together with a proof that the value satisfies a given specification. Existing compilers that generate certified code have focused on simple type safety only. Certified binaries provide a general framework for explicitly representing complex propositions and proofs in typed intermediate and assembly languages. The idea is to use the formulae-as-types principle [21] to represent proofs and assertions in higher-order predicate logic as explicit terms in a typed lambda calculus, and then to use propositions to express program invariants, while their proofs serve as static capabilities.

Our current proposed research is to extend and adapt ideas from FPCC and CB for technology transfer to a real VM such as the Microsoft CLR. We want to apply the theoretical ideas from our current system, and

the lessons learned from our prototypes, to a full-scale design. Our proposed work in this area consists of the following three components: logic-based type systems (LTS), prover/checker for FPCC, and interface between LTS and FPCC.

Logic-based Type Systems. A logic-based type system is a type system with a type language in which the propositions of an advanced logic, e.g. higher-order predicate logic, can be expressed. It represents a natural step in the development of more powerful type systems, trying to capture more program invariants in machine-verifiable form. A central advantage of LTS is that the constructs of ad hoc type systems can be encoded in it as logical propositions and proofs. We propose to develop logic-based type systems which enable the specification and verification of low-level constructs, including stack allocation, exception handling, mutable recursive data structures, data layout with alignment and flexible tagging, and array-bounds check elimination (to name just a few):

- **Stack allocation and exception handling:** Compiling the higher-level constructs of CIL and JVML (such as method invocation) into *jump* or *call* assembly instructions and explicit parameter passing exposes the details of the calling convention, which is typically stack-based. Thus it becomes necessary to verify the type-correctness of the explicit allocation and initialization of stack frames. Further complications arise due to the support for exceptions, since they require resetting the stack pointer to a value which must be shown statically to correspond to an existing stack frame. Pointers to stack-allocated data add another level of complexity since code often uses them in the same way as pointers to heap-allocated data, thus their types must be "compatible" with the latter use, yet convey extra validity information. In LTS this information, providing the access capability for the pointer, is expressed as a proof of the proposition defining validity of the pointer, and is represented as a type. The work of Morrisett *et al.* [33], addressing some of these issues in the context of a typed assembly language based on System *F*, is a starting point for our development.

- **Mutable recursive data structures:** Type-checking of memory allocation and initialization is non-trivial due to aliasing [34], but the problem is amplified in the case of mutable data of recursive type. Walker and Morrisett [53] propose a solution using *alias types*, expressing propositions about the shape of the store during execution. Encoding these propositions in LTS would not only allow proving the safety of its integration with the other language features, but also removing the limitations of the approach due to the insufficient expressiveness of the logic used in [53].

- **Data layout:** Control over the layout of records in the memory is crucial for language interoperability, and CLR provides mechanisms to obtain it. However this flexibility taxes the runtime system, specifically the memory management components, which must accommodate records with padding and alignment requirements. Compiler-generated metadata makes these requirements available to the runtime in the form of *tags*, but before the code using these tags is trusted, in a type-safe system it must be shown that the tags indeed correspond to the types and layout of the data they are claimed to describe. The idea for the solution in LTS is in the use of singleton types [57], relating the tag values to type-level descriptors, which in turn are used to define the type (including layout information) of the data.

- **Array-bounds check elimination:** The proofs-as-capabilities concept finds another application in removing runtime checks which verify that a value used as a subscript is within the index range of an array. This can be achieved by defining the indexing constructs so they expect a proof that the subscript is within bounds, and defining the integer comparison operations so that they generate such proofs [46]. These proofs can be reused in all indexing constructs referring to the same value, and

in the constructions of derived proofs; since proofs are represented as types, their manipulation adds no runtime overhead. The semantics of the language again rely on singleton types to ensure that the proofs at the type level adequately represent meta-proofs about the subscript values.

FPCC Infrastructure. The strength of Foundational PCC is that it separates proving from checking even more effectively than conventional PCC. The FPCC checker will be described in section B.4. The FPCC prover proves the safety of a program from information provided by the compiler's low-level type-checker. The usual problem with provers, as Gödel showed, is that it's hard to make them complete. Recently we have found a way to structure such provers: we design a syntax-directed low-level type system, and we prove each rule of that system as a derived lemma in our foundational logic. Then proving a particular program is just the Prolog-like application of syntax-directed rules: it's efficient and easy to prove complete.

All the interesting mathematical and technical work is in the proofs of those lemmas. To do this we construct a semantic model of the type system as sets of executions on the target machine. We first managed this for a simple type system with immutable records, first-class functions, recursive types, and quantified types [5]. We then extended our technique to handle contravariant recursive types [6] and mutable record fields [2]. Now we are ready to scale it up to an object-oriented type system such as Java; our recent work in expressing object types using simpler quantified primitives [24] shows the way.

LTS/FPCC Interface. One of the steps in implementing our program is to develop technologies for translating compiler intermediate languages annotated with LTS types into FPCC. This process involves generating program invariants which hold for all states reachable during program execution, and proofs which guarantee that the satisfaction of the invariants implies that the safety policy is observed. One approach to generating these invariants (from the available type information) and the soundness proofs is to interpret each type using its semantic model and to prove each typing rule as a lemma in the founational logic. Another approach is to interpret the types of program terms *syntactically* by representing them as inductive type definitions, and encode the meta-level proof of the soundness of the type system of the intermediate (and, eventually, assembly) language as a higher-order type which produces the proofs of progress and preservation of the invariants when given the syntactic representation of the program type [56].

The semantic approach seems to be more powerful (as it can be extended to prove more sophisticated safety policies), but building faithful semantic models for all logic-based types may be challenging. The syntactic approach may allow us to avoid the complexity of the semantic definitions of types in FPCC, and take advantage of the already developed meta proofs of soundness of the intermediate languages. We plan to explore both approaches and develop a general methodology for compiling logic-based typed intermediate languages into FPCC.

B.4 High-Assurance Virtual Machine

In principle the bytecode verifier enforces type safety and provides an ironclad guarantee that components respect their partners' API, but in practice this guarantee is only as strong as the implementation of the native libraries, the verifier, and the just-in-time (JIT) compiler. Experience has shown [13] that bugs in the native interface, verifier, or JIT can lead to new system vulnerabilities and can be used by attackers to bypass protection measures. We plan to extend and apply our work on proof-carrying code to build higher-assurance, validated implementations of the CLR infrastructure.

A high-assurance virtual machine (for .NET or Java) would have an untrusted *certifying JIT compiler* which translates bytecodes to native code with proof; the trusted *policy and axioms*; and a trusted *checker*.

The output of the compiler is some machine-language program p; the "theorem" is $safe(p)$, where $safe$ is a predicate defined in the policy. The trusted checker verifies that the proof produced by the certifying compiler actually proves the statement $safe(p)$.

Such a system provides useful security guarantees – in particular, that program components respect their APIs – as long as all the trusted components are actually trustworthy. Our trusted components are:

- **Axioms of logic:** Our axioms are the 10 rules of Church's higher-order logic, which is very well understood by mathematicians since the 1950s; and about 20 rules of arithmetic (such as the associative law for addition) which have been trustworthy for even longer.[2]

- **Instruction-set architecture:** We also have axioms describing the behavior of machine instructions on the target machine. In order to make these trustworthy, they should be concise, readable, and testable. We have already made progress in writing declarative specifications of machine instruction sets concisely and readably in higher-order logic [30]. The encoding and semantics of SPARC instructions takes us about 1500 lines to describe, for example. In the proposed research we will extend these methods to non-RISC machines, and we will develop other methods for increasing the trustworthiness of these specifications. For example, it should be possible to automatically check the encoding specification for consistency with the vendor's assembler [15], and we plan to experiment with checking semantics specifications for consistency with test executions on the actual machine.

- **Policy:** Simple (but useful and nontrivial) safety policies can be specified very concisely and clearly in higher-order logic. For example, "don't read from addresses outside your own heap," and "don't execute any illegal instructions" can both be specified clearly and concisely, in about 100 lines [3]. However, the specification of complex APIs will be more of a challenge; we have done some preliminary work on how to do this in a trustworthy way [4] and it will be a real focus of our proposed effort.

- **Checker:** Our logic, theorems, and proofs are all encoded in the LF logical notation. A significant advantage of LF is that proof-checking is a simple and well understood process. We have a prototype LF proof checker that's about 500 lines of cleanly written C code (with no libraries), adapted from a core developed by Aaron Stump at Stanford University. A disadvantage of this checker is that it requires a verbose proof representation.

  On the other hand, Necula [39] has shown how to encode proofs very efficiently (in both size and checking time), but a disadvantage of his checkers is that they rely on much more complex axioms and policies than in our work.

  In our proposed work we plan to build checkers that are as trustworthy as Stump's and as efficient as Necula's. The methodology will be to use higher-order logic to yield machine-checkable correctness proofs for first-order logic programs; these logic programs would fit into Necula's "oracle-based" scheme.

These components – axioms, ISA, policy, checker – add up to less than 3000 lines of code. With a trusted computing base this small, it is realistic to claim that code audits, automated model checking, and other methods can yield a really *trustworthy* trusted computing base that guarantees safety properties of the untrusted components. In contrast, the trusted bases of other approaches are an order of magnitude larger,[3] or two orders of magnitude larger.[4]

[2] The arithmetic of our logic is the mathematical integers; but in our policy we explicitly specify the properties of 32-bit machine arithmetic, as necessary for the specification of what real machines do.

[3] For example, the TCB of the SpecialJ system [11] (excluding API specs) is about 36,000 lines of code

[4] A typical optimizing compiler for Java is over 100,000 lines of code.

Certifying Compiler. The other crucial component of the high-assurance VM is the compiler (even though it's not in the TCB). The Yale FLINT compiler is a research infrastructure for certifying (type-preserving) compilation. FLINT/Java is an instantiation of this infrastructure, compiling Java byte codes to native machine code; we propose to develop a FLINT/CIL compiler from Microsoft's Common Intermediate Language to machine code.

The work on FLINT/CIL will be in three stages: *CoreCIL*, a subset of managed CIL including features such as value types, boxing and unboxing, stack allocation, and all object-oriented constructs; *SafeCIL* includes all of managed CIL; and *UmCIL* is an extension of SafeCIL with many unmanaged features. Managed CIL is the type-safe part of the CIL language, and can serve as a target language for source programs written in type-safe languages such as Java, C#, Visual Basic, and the "managed subset" of C++; unmanaged CIL serves as a target for C and C++, but does not enjoy all the safety and security properties of managed CIL. The main research issues involved in FLINT/CIL will be:

- **A front end parsing the .NET CIL binary format.** We will also experiment with other front ends (i.e. not restricted to Microsoft's CIL) for this compiler.

- **Type-preserving compilation of managed code.** This phase will leverage the results of our previous work on compiling subsets of Java and the JVM language [23, 26, 25, 24] for compiling the object-oriented fragment of CIL. It is based on representing object types transparently as recursive record types, making use of existential quantification and row polymorphism for efficiently modeling the subtyping between class and interface types, and for avoiding coercions and auxiliary access functions with runtime overhead, found in the standard typed translations of objects [10]. The framework covers also various implementations of secondary language features, including interfaces, access modifiers, dynamic casts etc. Using our results on intensional type analysis [50], the framework will be extended with support for reflection and static verification of metadata. Compiling the complete CIL will involve also extending the model with new features such as value types, method pointers (implementing delegates), and explicit data layout.

- **Designing logic-based type systems.** The intermediate (term) languages, used as communication vehicles between the optimization phases of the compiler, will be equipped with a common logic-based type system, which will allow high-level types and other invariants (e.g. layout, different calling conventions) to be expressed uniformly as propositions at the level of kinds (the types of types), with their proofs provided as type-level arguments. Furthermore the sharing of the LTS between different term languages, combined with the use of inductive types for defining the language-specific type constructors for each term language, will enable us to translate between the term languages while preserving the structure of the high-level invariants and their proofs [46]. This strategy makes it possible to maintain the higher level type information during the compilation, and eventually use it to generate proofs for the emitted foundational proof-carrying code.

- **Designing typed common intermediate languages** at lower levels (i.e., toward the back end of the compiler), and a typed portable assembly language. The critical step of this component is finding the exact set of primitives and their typing for each term language corresponding to its level of abstraction of the virtual machine features, e.g. explicit vs. implicit parameter passing conventions and memory management.

- **Certifying unmanaged code.** Certifying compilation of unmanaged code cannot be done solely through the type system, but such programs can often be proved safe through dataflow analysis. Recent progress in program analysis has unified flow analysis with type systems, however, and we plan

to aggressively investigate how to do certifying compilation using flow analysis, so that even C and C++ programs can achieve provable safety.

To bring our technology to the real world, we plan to work with researchers at Intel Labs to build a realistic industrial-strength **certifying JIT compiler**. We will help Intel Labs incorporate high-assurance static checking technology into their Open Research Platform. We will take their JIT compiler—which uses conventional untyped intermediate languages in the compilation process—and design type systems and type checkers, and we will assist them in building compiler components that use and generate these typed intermediate languages. We will design efficient representations for types and proofs, and pay particular attentions to efficiency and scalability. We will provide type-checking software useful in debugging their compiler. We will also design and build the high-assurance, trusted static checker for the machine code output of their JIT compiler.

B.5 Certified Libraries with Application-Specific Properties

We will extend our implementation of CLR to support not only simple type safety but also more advanced properties such as resource bounds on memory and network bandwidth, consistent metadata protocols, generalized access control, and proper coordination of concurrent components. These application-specific properties are crucial for principled interoperation between large-scale systems. We'll build certified libraries and develop new technologies for specifying, composing, and verifying advanced properties. The list of what we may want to tackle includes security and resource API, memory management API, safe dynamic linking, and reflection and metadata API.

Security and Resource API In previous work we have shown how to give a clean semantics to the security infrastructure of Java, and how to reflect that semantics into an implementation [8]. The basic idea is to encapsulate an abstraction of the security state (i.e., which permissions the process has explicitly enabled) and pass it as an argument to every function call. We showed that this has the advantage that changes in the permissions state are made explicit, so that conventional compiler optimizations need not be disabled, in contrast with prior techniques. We plan to use these ideas in a foundational, machine-checked verification of our VM's implementation of its security and access-control API.

Memory Management API The lack of control over memory resources, data structure layout, and memory reuse is one of the key reasons that programmers step outside the managed portion of a safe virtual machine and code in the unmanaged, insecure and unreliable portion of the virtual machine. Our certified binaries and foundational proof carrying code provide the fundamental framework for expressing the complex properties of data structures that are necessary for reasoning about memory management. However, the primitives they supply operate at a very low level of abstraction. Therefore, on top of the basic framework, we propose to build higher-level abstractions that a compiler can use to simplify the task of specifying memory management protocols and the task of generating safety proofs. Collectively, we call this system of abstractions the *Memory Management API*. Three central components of the Memory Management API involve:

- **Region abstractions:** Intuitively, a *region* is simply a group of objects that have common lifetimes. The region abstraction supports operations for allocating regions, allocating objects within a region and deallocating regions (which deallocates all objects within the region). This grouping structure and explicit deallocation provides an efficient alternative to standard tracing garbage collection.

Tofte and Talpin [49] were the first to demonstrate that region-based memory management for high-level languages could be captured in a static type system. More recently, Walker *et al.* [52] discovered how to control regions in a low-level compiler intermediate language and to represent many region-based optimizations efficiently using a calculus of static capabilities. We plan to leverage this earlier research and show how to capture these powerful invariants in our general-purpose framework of certified binaries.

Once we have developed our region abstractions, it will be possible to exploit results by Wang and Appel [55] who show how to code a conventional garbage collector in a type-safe language that includes region primitives.

- **Linear reasoning:** Reasoning with *linear logic* [17] makes it possible to track the number of references to an object. In the special case that there is only one reference to an object, the object may be safely deallocated without fear it may be accessed in the future. Linear reasoning also makes it possible to use reference counting memory management strategies, a very useful memory management strategy that is normally ignored in the implementation of type safe high-level languages like Java.

  Although linear logic dates back fifteen years, it has not been deployed in a practical virtual machine. We believe that when it is combined with the other aspects of our Memory Management API, it will be a powerful tool for specifying and reasoning about deallocation of data. In particular, recent theoretical results by Walker and Watkins [54] suggest that linearity interacts effectively with regions. However, much research in this direction remains. In particular, we must scale up the theory to a full language and incorporate it in the higher-order logic framework.

- **Memory algebra:** To specify data layout in a flexible and extensible fashion, we will need to develop a sophisticated *memory algebra*. Previous low-level type systems have had any number of restrictions on data layout that prevents a certifying compiler from performing memory management optimizations such as object inlining or from allocating and initializing arrays. To maximize the flexibility of our system we require a library of logical combinators that are capable of expressing a variety of invariants involving which sections of memory are readable or writable, initialized or not, aliased or unaliased, n bytes long or $n + 4$ bytes long, etc. The certified binaries framework and foundational proof carrying code were designed to be able to encode these sorts of invariants. However, there is a great deal of work to be done to design a higher-level Memory Management API that gives a clean and simple specification of all the useful invariants a compiler might require.

Safe Dynamic Linking On most computing platforms (the exceptions being some closed embedded systems) it is necessary to have mechanisms for dynamic extension and update of the executable code pool. High-assurance environments impose safety and security policies which must be shown to hold for new code before it is trusted for execution. Since determining automatically if these policies hold for arbitrary code is undecidable, and inferring even useful conservative approximations is unfeasible, code producers must help the environment to prove adherence to the policies by associating code with meta-information—type information in JVM class files, "metadata" in CLI, representations of proofs or hints in PCC. For program components this meta-information gives a partial description of their observable behavior; while in today's systems it is focused at the level of data layout and the types of a method's parameters, our proposal will make possible specifying higher-level properties such as resource consumption.

As indicated by the complexity of the system of rules for dynamic loading of JVM class files [29], in such an environment dynamic extension is far from trivial. Indeed, the subtle bugs in the initial JVM specification, which have devastating effects on the security of the system [42], are typical examples of

problems due to a large TCB. As a relatively complex piece of software involved in promoting data to executable code, the dynamic linker is a prime candidate for lifting out of the TCB.

Previous results by Hicks, Crary, and Weirich [20] have shown the feasibility of type-safe dynamic linking in the context of Morrisett *et al.*'s Typed Assembly Language (TAL) [32]. However there are new and significant problems in supporting dynamic linking for the proposed environment. For example, the linking models of JVM and CLI provide support for interface thinning and thus for evolving program components, which ensures compatibility between new versions of library code and clients of its old versions. The object-oriented bytecodes use name-based symbolic representation of the targets of all references to classes, methods, and fields, but their conversion to the more efficient pointer and offset representations requires a trusted JIT compiler. To achieve that at the lower level of our verifiable code, the linking model must allow the linker to "patch" the executable, *i.e.* to resolve references in a way which would permit an easy adaptation of the proof component. This requirement places the model beyond the expressiveness of conventional type systems.

A safe dynamic linker must have access to the meta-information for the component and the running system, which will be provided by the Reflection and Metadata library.

Reflection and Metadata API Of major importance for dynamically extensible applications and operating systems is the ability to inspect and interoperate with dynamically loaded components whose interfaces were unknown when compiling the application. Furthermore, the software responsible for extending the system (*e.g.* the dynamic linker) not only uses this meta-information but must also keep it up to date. These features are provided in frameworks like COM in an unsafe form—components can be queried about the interfaces they provide, but there is no control over whether the actual access complies with the supported interfaces.

In safe systems such as JVM and Microsoft's Common Language Infrastructure access to these components is granted through a Reflection Library, a part of the TCB which uses the meta-information to check dynamically that an access attempt is safe to execute. In theory this approach guarantees that the safety of the system cannot be violated, since in essence dynamic type-checks are performed on every reflective access. However, in addition to the performance penalties, there is the pervasive issue with having these services as part of the TCB: The safety guarantees are only as strong as our ability to prove the correctness of the implementation. Thus a highly reliable system must have the reflection library and access to the meta-information statically verified and thus taken outside the TCB.

The theoretical foundations of our approach to static type-checking of reflection are laid out in the framework of intensional type analysis, pioneered by Harper and Morrisett [18] and developed further by Crary *et al.* [12], which due to our work at Yale [50] can be applied to all types in a higher order system.

A significant hurdle to the static verification of reflection is that the type correctness of parts of the code depends on the values of the reflection data (representation of the metadata). This dependence makes type-checking undecidable in general, and even in systems with decidable restrictions poses serious problems for compilation [9]. Through the use of singleton types in the style of Xi and Pfenning [57] we have developed methods for avoiding this dependence in a system based on a higher-order predicate logic, and shown how its expressive power can be used to simplify the compilation [46].

While they are required features in a high assurance system, introspection and reflection can provide the power to break abstraction barriers imposed by static access control, such as abstract types or private fields, albeit without breaking the runtime safety. The necessity for finer control over deciding which parts of the system can be trusted with insider access levels is addressed by the security and resource API.

B.6 Summary of the Broader Impacts

Security Infrastructure. The nation's critical infrastructure including financial, communications, energy, transportation and defense capabilities rely upon highly connected interdependent information systems. These vast systems are made up of a multitude of interoperating and dynamically changing components. Currently these systems are at risk both from malicious terrorist attack and from simple software faults [43]. Our technology will help reduce the vulnerabilities in such large heterogeneous systems in three key ways:

1. Our core contribution, the high-assurance virtual machine, guarantees type safety with very high probability (much higher probability than any existing security infrastructure), which helps insulate components against one another, preventing buffer overrun and Trojan-horse attacks that allow one component to corrupt another.
2. Reflection and secure dynamic linking technology allow components to adapt to changing real-world circumstances such as security alerts or bugs. Consequently, systems managers will be able to respond to threats in real time.
3. Powerful higher-order logics provide a flexible underlying framework for reasoning about security and allow us to deploy and enforce new more expressive security policies than are currently possible.

Industrial Impact. We have chosen to focus our efforts on enhancement of mainstream commercial programming language technology such as Java and the .NET infrastructure. We have also developed a relationship with Intel Labs to help speed transfer of ideas and results from academia to industry. More specifically, Ali Adl-Tabatabai, Michal Cierniak, and Ken Lueh, at Intel, are currently developing an optimizing JIT that targets Intel's IA64 (Itanium) architecture. We are planning to work with them to increase the reliability of their compiler system by helping to design and prove the safety of their low-level compiler target language. This project will apply our research on low-level type systems and high-assurance infrastructure in an industrial setting. The resulting compiler will be one of the first to generate extremely reliable and secure code for next-generation commercial processors.

Education. The development, deployment and maintenance of high-assurance systems requires a broad range of intellectual skills ranging from highly applied compiler and system implementation abilities to a deep theoretical understanding of language principles and mathematical theorem proving techniques. We are actively developing educational curricula that will teach these skills to students as well as academic researchers and our industrial partners. At Princeton, Appel has written the well-known textbook *Modern Compiler Implementation in Java* and has developed courses to teach graduate and undergraduate students the fundamental theory underlying safe programming technology. Walker is currently developing a complementary course entitled "Foundations of Language-based Security" to teach the tools and techniques necessary for constructing a safe modern virtual machine and reasoning about security properties. Along with researchers from Microsoft research, ONR and a variety of academic institutions, Walker has been invited to teach at an upcoming ACM summer school on "Foundations of Internet Security." At Yale, Shao and his colleagues have developed courses on type-safe programming languages and compilers. Shao and Trifonov will be developing a new course on logic-based type system that would teach students to use the technologies developed under this proposal to write certified programs and libraries.

B.7 Comparison with Other Research

PIs' DARPA-funded research: A small part of this research is funded by DARPA through December 2003. Specifically, DARPA funds the beginning of the technology transfer effort to Intel Labs – the design of typed high-level and low-level intermediate languages for their JIT compiler, the implementation of type-checkers for those languages, and the teaching work in showing the Intel researchers (who are compiler experts but not type-theory experts) how to use these technologies. This funded research covers a significant portion of our proposed *certifying JIT compiler* (see the last paragraph of Section B.4). However, this effort—developing a typed optimizing JIT compiler—is really a several-year project that goes beyond the end of the DARPA funding; and many aspects of this new proposal, such as the *high-level specs for low-level software*, *certifying compiler for unmanaged CIL*, and *certified libraries* are outside the scope of our DARPA-funded research.

High-level specifications for low-level software: Necula and Lee's development of PCC [35, 36] is a significant theoretical achievement. As we have explained in the body of our proposal, we believe this concept can be applied in new domains (i.e., implementation of security/authentication policies for distributed computation and mobile code) and engineered for application to full-scale programming languages and language-independent code consumers. Morrisett *et al* [34] developed a typed assembly language, which allows a certifying compiler to produce native code with type annotations that can be checked to ensure safety. Typed assembly language (TAL) is less general than proof-carrying code, since proofs can be based both on types and on dataflow properties of the program. But Morrisett's TAL can be used as one of the proof techniques in a PCC system, and we intend to use this technique where appropriate.

Several ongoing projects (such as the ConCert project at CMU, the language-based security project at Cornell, the open source quality project at Berkeley, and the secure software systems project at Purdue) are also using advanced language technologies to build secure systems. Our project is unique in that we are attacking the security problem in the context of Common Language Runtime, thus we have to aim for a language-independent framework. Indeed, both FPCC and logic-based type systems are based on higher-order predicate logic which is not biased towards any programming language. In turn, we also get smaller trusted base because our prover and type-checker only need check against the higher-order logic which is much smaller than most language-specific type systems.

Recent efforts such as CCured [38], Cyclone [22], and Vault [14] aim to design new C dialects that are safe yet still capable of low-level programming. All three systems enforce simple type safety only and they do not support arbitrary logical assertions—so they can't use advanced integer constraints to eliminate redundant array bounds checking. All three systems only type-check the source language. Their compilers are not certifying so they have large trusted bases. Our project aims to push their technology into the compiler intermediate and assembly languages, in turn, we can use more expressive logic to specify and enforce more advanced safety properties.

High-assurance virtual machines: Typed intermediate languages have received much attention lately, especially in the ML community. Several ML compilers [27, 45, 48] maintain explicit type information inside their intermediate languages. However, none of them tried to extend the intermediate language to support multiple advanced languages. Neither have they seriously addressed the problems of how to scale up to handle large types, how to support efficient run-time type passing, and how to support efficient and principled language interoperations.

Franz [16] developed a machine-independent mobile-code system based on *slim binaries*, which are specially compressed abstract syntax trees from the Zurich Oberon compiler. These abstract syntax trees

would be difficult to target for another front end, but it might be very effective to apply the compression technology to our low-level FLINT language.

Lucco and his colleagues [1] developed a language- and machine-independent mobile-code system named *OmniWare*. OmniWare uses software fault isolation to achieve safety, efficiency, and language independence. The FLINT system, in contrast, achieves similar goals by using an expressive typed intermediate language. Using type system has the advantage of extensibility: types can to used to ensure not only execution safety but also more advanced security properties [19, 28].

C-- [40] is an untyped assembly language designed to provide portable support for features needed by advanced languages such as garbage collection, exception handling, and debugging, without building a particular garbage collector, exception semantics, or debugging model. FLINT shares the same goal but insist on using a strongly typed intermediate language.

Special J [11] is a certifying compiler for Java developed at the Cedilla Systems. Unlike FPCC and FLINT, Special J uses the source-level (object-oriented) type system to certify the low-level machine code, so it has a large trusted base containing many Java-specific typing rules. Indeed, League *et al* [24] recently discovered a serious bug in these typing rules that would break the integrity of their entire system.

Certified libraries with application-specific properties: Both CLR and JVML provide support for security, reflection, and memory management (via unmanaged code and native interface), but its extent is mostly in breadth. They have feature-rich libraries, but the interactions between them and the main language features are not studied well. Since they could be used to violate the safety of the system, they are subject to multiple runtime checks, which still does not guarantee that all potential loopholes are covered. The runtime checks also make these operations less efficient and makes them unsuitable for use in implementations of performance-critical runtime services.

A large number of dynamic-linking protocols have been designed, with various levels of safety. The Java and CLR virtual machines guarantee safety of programs constructed of bytecode components (classes and interfaces); furthermore, they provide for library evolution (binary compatibility) by recording symbolic references and type requirements only for the actually used members of external classes. However they cannot offer any safety guarantees for programs invoking native of "unmanaged" code. In addition, the bytecode verifier itself is a trusted piece of code of relatively large size, implementing a type inference algorithm for the JVML or CIL, which so far has not been shown sound (or, in the case of CIL, even formally defined), but the number of rules in its formal system would be in the hundreds. The TAL/Load framework [20] aims to reduce the size of the trusted code, but it only begin to explore the design space; questions such as how to handle higher-kind polymorphism (i.e. type variables which may be bound to type functions) and logic-based types (e.g. types with binding structure), and how to extend the framework with efficient support for binary compatibility, remain open.

B.8 Results from Prior NSF Support

NSF Award CCR-9974553 — Applying Compiler Techniques to Proof-Carrying Code (PI: Andrew W. Appel) Under this grant (Sept '99 to August '02, total amount $220,000) we have done basic research in proof-carrying code. This includes research in construction of efficient-sized proofs [7], specification of machine instruction-set architectures [30], and semantic models of type systems [6]. Our latest results are on "typed machine language" [47], a calculus that has a semantic model (so foundational proof-carrying code can be generated from it) but that is also at a high enough level so that it can be generated as the output of a compiler. We are nearing completion of a prototype proof-carrying code compiler for the core ML

language.

The graduate student supported has been Kedar Swadi, who is expected to finish his PhD in late 2002.

NSF Award CCR-9901011 — Typed Common Intermediate Format (PI: Zhong Shao) This grant (total amount: $320,000) was awarded to the PI starting from August 1999 for a duration of three years. The goal is to design and implement a language- and architecture-neutral typed intermediate format. Under this effort, we obtained several important results in the area of type-based language design and implementation.

Popular mobile code architectures (Java and .NET) include verifiers to check for memory safety and other security properties. Since their formats are relatively high level, supporting a wide range of source language features is awkward. Further compilation and optimization, necessary for efficiency, must be trusted. Our work on *type-preserving compilation of Java* [23, 26, 25, 24] (published in ICFP'99, IRE'01, and FOOL'01, and under review for PLDI'02) designed and implemented a fully type-preserving compiler for Java and SML. Our strongly-typed intermediate language, provides a low-level abstract machine model and a type system general enough to prove the safety of a variety of implementation techniques. Our research shows that precise type preservation is within reach for real-world Java systems.

Our work on *intensional type analysis* [50, 41] (published in ICFP'00 and TIC'00) solved a very difficult problem in the area of certifying compilation. Using our new techniques, type-analyzing operations can be applied to all runtime values in a language. This allows us to certify low-level runtime services such as garbage collection, pickling and marshaling, and persistent programming.

Both proof-carrying code and typed assembly languages aim to minimize the trusted computing base by directly certifying the actual machine code, but unfortunately, these systems cannot get rid of the dependency on a trusted garbage collector. Our result on *principled scavenging* [31] (published in PLDI'01) provided a set of general techniques for writing type-safe copying garbage collectors.

A *certified binary* is a value together with a proof that the value satisfies a given specification. Our work on certified binaries [46] (to appear in POPL'02) is the first comprehensive study on how to incorporate higher-order predicate logic (with inductive terms and predicates) into typed intermediate languages. Our results are significant because they open up many new exciting possibilities in the area of type-based language design and compilation. The fact that we can internalize a very expressive logic into our type system means that formal reasoning traditionally done at the meta level can now be expressed inside the actual language itself. For example, much of the past work on program verification using Hoare-like logics may now be captured and made explicit in a typed intermediate language.

Our FLINT and SML/NJ compilers are used by many researchers worldwide to carry out advanced research on programming languages and compilers.

On the educational side, we have used the software developed under this project (FLINT/ML and SML/NJ) in the undergraduate and graduate curriculum at Yale. The use of these research results and software system artifacts helped students to learn the fundamental methodologies used in both programming language theory and practice.

On the human-resource side, this grant provided partial support to several graduate (Chris League, Stefan Monnier, Bratin Saha, and Nadeem Hamid) and undergraduate students (Yichen Xie and John Garvin) of mine. All graduate students are making excellent progress; three of them will defend their PhD dissertations in the next 12 months. Influenced by their experience (under this project), both undergraduate students decided to pursue PhD study in computer science (Xie at Stanford and Garvin at Rice).

C References Cited

References

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 127–136. ACM Press, 1996.

[2] A. J. Ahmed, A. W. Appel, and R. Virga. Semantics of general references by a hierarchy of Gödel numberings. Available at `www.cs.princeton.edu/~appel/papers`, July 2001.

[3] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[4] A. W. Appel and E. W. Felten. Models for security policies in proof-carrying code. Technical Report CS-TR-636-01, Princeton Univ., Dept. of Computer Science, March 2001.

[5] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symp. on Principles of Prog. Lang.*, pages 243–253. ACM Press, 2000.

[6] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, to appear.

[7] A. W. Appel, K. N. Swadi, and R. Virga. Efficient substitution in hoare logic expressions. In *4th International Workshop on Higher-Order Operational Techniques in Semantics (HOOTS 2000)*. Elsevier, Sept. 2000. Electronic Notes in Theoretical Computer Science 41(3).

[8] A. W. Appel, D. Wallach, and E. W. Felten. Safkasi: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, (to appear), 2002.

[9] G. Barthe, J. Hatcliff, and M. Sorensen. CPS translations and applications: the cube and beyond. *Higher Order and Symbolic Computation*, 12(2):125–170, September 1999.

[10] K. Bruce, L. Cardelli, and B. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, Dec. 1999.

[11] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, 2000. ACM Press.

[12] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. 1998 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 301–312. ACM Press, Sept. 1998.

[13] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. In D. E. Denning and P. J. Denning, editors, *Internet Beseiged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press, New York, 1997.

[14] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.*, pages 59–69, New York, 2001. ACM Press.

[15] M. F. Fernandez and N. Ramsey. Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering*, pages 326–336, May 1997.

[16] M. Franz. Beyond Java: An infrastructure for high-performance mobile code on the world-wide web. In *WebNet'97, World Conference of the WWW, Internet, and Intranet*, pages 33–38. Association for the Advancement of Computing in Education, Oct 1997.

[17] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[18] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd ACM Symp. on Principles of Prog. Lang.*, pages 130–141. ACM Press, 1995.

[19] N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Twenty-fifth Annual ACM Symp. on Principles of Prog. Languages*, pages 365–377, New York, Jan 1998. ACM Press.

[20] M. Hicks, S. Weirich, and K. Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2000.

[21] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[22] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. Available at `www.cs.cornell.edu/home/danieljg`, November 2001.

[23] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *Proc. 1999 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 183–196. ACM Press, September 1999.

[24] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Submitted to PLDI'02, November 2001.

[25] C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[26] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight Java. In *Proc. Int'l Workshop Found. Object-Oriented Languages*, London, January 2001.

[27] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[28] X. Leroy and F. Rouaix. Security properties of typed applets. In *Twenty-fifth Annual ACM Symp. on Principles of Prog. Languages*, pages 391–403, New York, Jan 1998. ACM Press.

[29] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[30] N. Michael and A. Appel. Machine instruction syntax and semantics in higher order logic. In *Proc. 17th International Conference on Automated Deduction*, pages 7–24. Springer Verlag, June 2000.

[31] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.*, pages 81–91, New York, 2001. ACM Press.

[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 *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[33] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proc. Second Int'l Workshop on Types in Compilation (TIC'98)*, volume 1473 of *LNCS*, pages 28–52, Kyoto, Japan, Mar. 1998.

[34] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, Jan. 1998.

[35] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119, New York, Jan 1997. ACM Press.

[36] G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Sept. 1998.

[37] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[38] G. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proc. 29th ACM Symp. on Principles of Prog. Lang.*, page (to appear). ACM Press, January 2002.

[39] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154. ACM Press, Jan. 2001.

[40] N. Ramsey and S. Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 285–298, New York, 2000. ACM Press.

[41] B. Saha, V. Trifonov, and Z. Shao. Fully reflexive intensional type analysis with type erasure semantics. In *Proc. 2000 ACM SIGPLAN Workshop on Types in Compilation*. Published as CMU School of Computer Science Technical Report, September 2000.

[42] V. Saraswat. Java is not type-safe, 1997. Web pages at: http://www.research.att.com/~vj/main.html.

[43] F. Schneider, editor. *Trust in Cyberspace*. National Academy Press, Washington, D.C., 1999.

[44] Z. Shao. Typed common intermediate format. In *Proc. 1997 USENIX Conference on Domain Specific Languages*, pages 89–102, October 1997.

[45] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. 1995 ACM Conf. on Prog. Lang. Design and Impl.*, pages 116–129, New York, 1995. ACM Press.

[46] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symp. on Principles of Prog. Lang.*, page (to appear). ACM Press, January 2002.

[47] K. N. Swadi and A. W. Appel. Typed machine language and its semantics, July 2001.

[48] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. 1996 ACM Conf. on Prog. Lang. Design and Impl.*, pages 181–192. ACM Press, 1996.

[49] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[50] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Proc. 2000 ACM SIGPLAN Int'l Conf. on Functional Prog.*, pages 82–93. ACM Press, September 2000.

[51] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, isoc.org, 2000. The Internet Society.

[52] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

[53] D. Walker and G. Morrisett. Alias types for recursive data structures. In *2000 ACM SIGPLAN Workshop on Types in Compilation*, Montréal, Canada, Sept. 2000.

[54] D. Walker and K. Watkins. On regions and linear types. In *ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, Sept. 2001.

[55] D. C. Wang and A. Appel. Type-preserving garbage collectors. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 166–178, London, UK, Jan. 2001.

[56] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[57] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.*, pages 214–227. ACM Press, 1999.