

# Research Statement

## Christopher League

I am interested in programming languages, systems, and environments – especially in the area of language-based security. My work is motivated by the conviction that the proper use of formal techniques will lead to more secure and reliable systems in the future. I enjoy working in this area because of the opportunity to do both theoretical and practical research.

### Background

For the past three years, I have collaborated with colleagues at Yale and Princeton on a DARPA-funded project called “Scaling proof-carrying code to production compilers and security policies.” First proposed by Necula and Lee [8], proof-carrying code (PCC) requires the producer of some piece of code to provide a formal *proof* that it adheres to some agreed-upon *safety policy*. Checking a proof is much easier than producing one, and the burden of proof is shifted to the producer. To ensure safety, the code consumer needs to trust relatively little code: just the proof checker, its axioms, and the runtime system. Importantly, the *compiler* is absent from the trusted computing base.

*Foundational* proof-carrying code (FPCC), proposed by Appel and Felty, goes a few steps further by using the smallest set of axioms and simplest verifier possible [1]. Typing rules for the source language – axioms in the original PCC – are proved from first principles as lemmas in FPCC. An essential component of this system is a *certifying compiler*. Starting from programs in a type-safe language, the certifying compiler translates both the code and the types, automatically yielding a proof of type safety.

Type safety is stronger than memory safety (the policy enforced by the virtual memory hardware). Type safety also means that a module respects its interfaces and cannot, for example, manufacture a pointer to the private fields of another module. Even more sophisticated properties can be encoded in a type system. Dan Wallach, for example, reformulated the Java *stack inspection* security mechanism as *security-passing style* and proved its soundness [11].

My dissertation research focused on building a certifying compiler *infrastructure*, in which one strongly-typed intermediate language can be used to certify programs in different source languages. Much is known about using intermediate languages based on typed lambda calculi to compile functional languages. I extended the capability of one such IL to support Java, an object-oriented language.

### An efficient object encoding

Encodings of objects in various typed lambda calculi were previously known; Bruce et al. compared several of them [2]. They were developed not for certifying compilers, but to explore the theoretical foundations of object-oriented languages. Unfortunately, they all had startling inefficiencies: extra levels of indirection, or superfluous function calls, for example. A certifying compiler would need an object encoding that does not interfere with the standard object layout or efficient implementation of virtual method calls.

My colleagues and I synthesized a new object encoding using well-understood components, each of which is useful on its own. Unlike its predecessors, our encoding ensures the safety of object operations without hurting performance. Our object layout is completely standard and there are no superfluous runtime calls or casts. Most importantly, the encoding integrates well with techniques for efficient compilation of other object-oriented features, such as access control, interfaces, mutual recursion, and dynamic cast. I wrote two different papers with formal translations of Java subsets [5, 7];

an expanded account, with a complete proof of type-preservation, was recently accepted by the journal TOPLAS.

## A better Java bytecode format

Java bytecode was designed to be interpreted by a stack-based virtual machine. The .class files contain just enough type information that the VM can verify type safety before running the code. Unfortunately, the verification procedure is overwhelmingly complex due to such aspects as subroutines (which are polymorphic and need not obey a stack discipline) and object initialization (which requires conservative alias analysis).

I used ideas from compilers for functional programming languages to design  $\lambda$ JVM, an alternate representation of Java bytecode [6].  $\lambda$ JVM makes data flow explicit, and is well-suited for translation into lower-level representations such as those used in optimizing compilers. Moreover, class verification reduces to simple type checking. All the difficult analyses are done during translation to  $\lambda$ JVM, and the results are preserved in compact type annotations. Subroutine return addresses are represented as first-class continuations, and uninitialized object types are tracked explicitly. My type checker for  $\lambda$ JVM in less than 260 lines of SML code. A Java system based on  $\lambda$ JVM would have a significantly smaller trusted computing base.

## A certifying compiler for Java (and ML)

I designed  $\lambda$ JVM as the first stage of a certifying compiler for Java. Its explicit types and data flow match those of FLINT, the strongly-typed intermediate language used in the SML/NJ compiler [10, 9]. It otherwise features the same primitive types (e.g., int, float, object) and operations (e.g., invokevirtual, putfield) as the Java virtual machine. Using our previous work on object encodings, the next stage of my compiler expands these Java-specific types and operations into the lower-level (but still well-typed) operations of the intermediate language.

The compiler is a prototype, but it runs realistic benchmarks with reasonable performance. One interesting feature is that it supports both Java and ML using the same typed intermediate language, optimizers, code generator, and runtime system. The IL provides a low-level abstract machine model, and flexible type constructors, general enough to prove the safety of various uses of the machine primitives. This design and implementation of this compiler is described in a new paper [4].

The original developers of PCC built Special J, also a certifying compiler for Java, using a different approach [3]. The typing rules for the Java language (at the machine code level) are *axioms* in their proof checker, and therefore part of the trusted computing base. In comparing their approach to mine, I discovered that their typing rules are *unsound*. I developed a simple malicious program and corresponding proof (using an unsound axiom) that is accepted by their proof checker. The code casts an arbitrary integer to a pointer. The existence of this bug, undetected for two years, raises suspicions about the entire approach taken by Special J. Foundational PCC does not use such high-level axioms. Rather, soundness of the source language is proven from first principles in higher-order logic.

## Future directions

There are now *two* major platforms (Java and Microsoft .NET) that each feature a common intermediate format and some notion of verification of security properties. I believe we are seeing the rudiments of an important conceptual shift in the computing industry: machine- and language-independence are desirable, and code safety is critical.

My colleagues and I are starting to collaborate with a group at Intel Labs on their IA64 just-in-time compiler for the Microsoft .NET platform. The Intel group already found that limited type information preserved in the compiler is useful for optimization (disambiguating memory references, for example) and for accurate garbage collection. They understand that a more rigorous typing discipline would bring security benefits, and are curious about the impact type-preservation might have on the performance, reliability, and maintainability of their system.

I am quite excited about this collaboration. First, .NET is a more ambitious platform than Java, less partial to a single source language. Also, JIT compilation imposes some new requirements, most notably the speed of the compiler. Finally, working with researchers and developers in industry will help bring pragmatic concerns to the foreground. The project will certainly yield an exciting new batch of practical and theoretical problems to solve. I truly believe that technology based on type theory, formal logic, and proof will eventually lead to more flexible, principled, and secure mobile code systems.

## References

- [1] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. Symp. on Principles of Programming Languages*, pages 243–253, Boston, January 2000. ACM.
- [2] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2):108–133, 1999.
- [3] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proc. Conf. on Programming Language Design and Implementation*, Vancouver, June 2000. ACM.
- [4] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. **Under review:** submitted to ACM Programming Language Design and Implementation (PLDI 2002).
- [5] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *Proc. Int'l Conf. Functional Programming*, pages 183–196, Paris, September 1999. ACM.
- [6] 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.
- [7] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight Java. In *Proc. Int'l Workshop on Foundations of Object-Oriented Languages*, London, January 2001. Expanded version to appear in *ACM Trans. on Programming Languages and Systems*.
- [8] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996. ACM.
- [9] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. Conf. on Programming Language Design and Implementation*, pages 116–129, La Jolla, June 1995. ACM.
- [10] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. Int'l Conf. Functional Programming*, pages 313–323, Baltimore, September 1998. ACM.
- [11] D. S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, 1999.