Software systems are complex. They coordinate components written at different times, by different people, perhaps even in different programming languages, to accomplish shared goals. Operating systems, for example, contain device drivers written by third parties, which must be trusted (or proved) not to crash the kernel. OSs are also typically implemented in multiple languages (e.g., context switching code in assembly interacting with schedulers written in C), which can lead to complicated system-level invariants and subtle bugs. Other widely used software systems, such as the implementations of languages like Java and OCaml, are just as heterogeneous: The compiler is written in a high-level language but generates code that is linked to garbage collectors and other runtime components written in C or C++.

At the same time, software systems have grown increasingly important to daily life. It is now common for cars, which can contain upwards of 40 onboard computers, to “drive by wire.” Other safety-critical systems, such as those used in avionics or in medical devices like pacemakers, rely fundamentally on software as well. Bugs in these systems can have disastrous consequences.

The goal of my research is to increase confidence in practical, widely used software systems such as compilers, runtime systems, and operating systems, especially in high assurance and safety-critical contexts, through the application of formal methods from programming languages theory—program logics, type systems, and interactive and automated theorem proving. In particular, my research seeks to tame the complexity inherent in these large systems by applying tools and techniques, such as verified separate compilation, high-level domain-specific languages, and verified decision procedures, to make composing and reasoning about highly heterogeneous systems more practical.

Verified Separate Compilation for C [SBCAar, BSDA14]

The verified (or verifying) compiler is one of the “grand challenges” [HM05] of computer science. By “verified,” I mean the compiler is proved—typically in a machine-checked logic such as Isabelle/HOL [Isa], Coq [Coq], or PVS [PVS]—to produce assembly programs that behave exactly the same as (or, in the case of nondeterministic languages, exhibit no more behaviors than) the source modules from which they were compiled. While there had been much early work, beginning with McCarthy and Painter in 1967 [MP67], on verified compilation, it was not until 2006 that Leroy [Ler06] demonstrated the first practical, fully verified and optimizing compiler for a real-world language, C. Since Leroy, there has been work extending the techniques he pioneered to other languages, e.g., to multithreaded Java [Loc12] or to weak-memory concurrency [SVN13]. However, all of these projects shared a common limitation: They assumed the program being compiled was defined monolithically, as a single translation unit in a single language. This “closed-world” [PA14] assumption made the compiler correctness proofs significantly easier, but did not scale to real-world systems.

In my dissertation, I addressed the closed-world problem by modifying Leroy’s verified but whole-program compiler for C, called CompCert, to support verified separate compilation and linking. The resulting system, called Compositional CompCert (Figure 1, [SBCAar]), provides strong guarantees about the correctness of separately compiled multimodule C programs, even in the context of linked assembly code (which may be written by hand or generated by a second compiler).
In addition to Compositional CompCert itself, my work on verified separate compilation, along with that of colleagues at Princeton, resulted in several new formal methods. The first is a novel operational semantics of language-independent linking that precisely models the interactions of code modules written in different languages, such as C and x86 assembly. Language-independent linking makes it possible to state the correctness of a separate compiler as a straightforward contextual equivalence, which increases confidence in, and the usefulness of, the overall separate compilation theorem. The second contribution is a new form of simulation proof, called structured simulations, that combines the assumption-guarantee reasoning that is typical in proofs of concurrent program transformations [LFF12] with fine-grained, or structured, invariants on memory. Structured invariants make it possible to distinguish compiler-managed memory regions—such as the stack locations reserved for spilled registers or return address—from programmer-visible stack locations such as those of addressed local variables. Distinguishing these regions is essential for proving correctness of compiler phases such as register allocation and spilling that may modify compiler-managed memory.

Future Directions. The Compositional CompCert project has so far focused on C and assembly. However, the theories of language-independent linking, structured simulations, and verified separate compilation that I co-developed in the course of my dissertation research are more general than just Compositional CompCert, C, and assembly. One immediate direction for future research is to apply the work on Compositional CompCert to the scenario in which source programs are written in a combination of C, assembly, and a high-level functional language such as the Coq proof assistant’s Gallina specification language (the language in which Compositional CompCert is itself programmed). The primary motivation here is to decrease the time and effort required to build large high-assurance software systems: Small, high-performance components can be written in C (or assembly), verified—with effort proportionate to the complexity of C—in a tool like the Verifiable C program logic [ADH+14], and then compiled with
Listing 1: Ziria WiFi 802.11a/g receiver pipeline

```haskell
let comp Decode(h : struct HeaderInfo) =
  DemapLimit(0) >>>
  (if (h.modulation == M_BPSK) then
    DemapBPSK() >>> DeinterleaveBPSK()
  else if (h.modulation == M_QPSK) then
    DemapQPSK() >>> DeinterleaveQPSK()
  else ...) // QAM16, QAM64 cases
  >>> Viterbi(h.coding, h.len*8 + 8)
  >>> scrambler()
in let comp detectSTS() = removeDC() >>> cca()
in let comp receiveBits() =
  seq { h ← DecodePLCP(); Decode(h) >>> check_crc(h.len) }
in let comp receiver() =
  seq { det ← detectSTS();
    params ← LTS(det.shift);
    DataSymbol(det.shift) >>>
    FFT() >>>
    ChannelEqualization(params) >>>
    PilotTrack() >>>
    GetData() >>>
    receiveBits() }
in read >>> repeat { receiver() } >>> write
```

Compositional CompCert. Components with lower performance requirements, such as “glue code,” can be implemented in Coq’s Gallina language, verified with much less effort than is typical for C programs, and then compiled using standard toolchains. The key technical challenges are: (1) building an assurance case for the additional runtime components required by the high-level language (such as the garbage collector), perhaps by verifying the runtime itself; and (2) increasing confidence in the compilation of the high-level Gallina code.

High-level Programming of Wireless PHYs [SGM+15, SGM+14]

The PHY, or “physical” layer, of a wireless protocol such as LTE or WiFi 802.11 is the hardware and/or software that translates between MAC-layer network packets and the signals that are transmitted and received over the radio. Although wireless PHYs are typically built in hardware as ASICs or FPGAs in order to meet the tight timing constraints of modern protocols, recent work on projects such as Sora [TLZ+11] has shown that it is possible to implement wireless protocols in software as well, while still meeting the latency and throughput requirements necessary for line-rate interoperation with hardware PHYs.

This work is encouraging, especially for researchers who experiment with prototype PHYs as part of their day-to-day research activities. However, the currently available tools for software radio (cf. [TLZ+11, Blo04]) do not yet make programming PHYs easy: in Sora, for example, implementing PHYs that meet the performance requirements of wireless standards like 802.11 requires extensive manual optimization and detailed knowledge of the target architecture, such as the size of CPU caches. The manual optimizations
that are typically performed in software tools like Sora include vectorization of DSP pipeline code, which can reduce modularity and can make pipelines brittle to modification, as well as the replacement of common components such as scramblers with lookup tables, which have to be optimized for CPU cache sizes. The resulting software system is performant but not easily modifiable without significant expertise.

Together with colleagues at Microsoft Research Cambridge, I co-designed and implemented a new programming language for building wireless PHYs, called Ziria, that removes many of the barriers to entry of the current tools—such as the need for extensive manual code optimization—while retaining the performance profile of the most efficient software radio platforms. The key innovations that make this possible include: a new, high-level abstraction of the control flow of DSP pipelines (Figure 2), which makes it possible to implement staged protocols such as frequency hopping without extensive use of global shared state; and, in the Ziria compiler, a novel automatic vectorization algorithm that produces close to optimally vectorized code even for full WiFi pipelines. Other optimizations performed by the compiler include automatic lookup-table insertion and annotation-guided pipeline parallelization onto SMPs. The end result is a system in which protocols such as WiFi can be implemented at a high level, as “executable specifications” (cf. Listing 1, which gives the toplevel of our Ziria WiFi receiver pipeline), and yet are compiled to code that approaches the performance of manually tuned versions (Figure 3 gives throughput of the Ziria wireless receiver in Listing 1 vs. manually optimized Sora code).

**Future directions.** Although the Ziria language was designed with high assurance applications in mind, the current Ziria compiler and runtime are not yet accompanied by machine-checked proofs of correctness. The most immediate direction for future work is to formalize the Ziria language and semantics in a proof assistant such as Coq, port the current compiler from Haskell to Coq, and then prove the compiler correct, e.g., with respect to the Compositional CompCert C semantics. Such a toolchain would provide an end-to-end link between high-level wireless code written in Ziria, such as our 802.11a/g WiFi transceiver, and the assembly code generated by the compiler.

**Verified Provers for Network Programming and Separation Logic [Ste13, SBA12]**

I have also done work to directly improve formal methods themselves—in particular, interactive theorem provers such as Coq—by building verified automatic theorem provers embedded inside proof assistants. Such provers, when built and verified in the Coq proof assistant, can be used for computational proof, in which properties are proved not in the usual step-by-step manner typical in Coq or Isabelle/HOL, but instead by just computing the result using a verified procedure (hence computational). Computational proof is still foundational, in the sense that it generates proofs that can be checked by a small, automated proof checker. However, the resulting proofs are typically much smaller than those generated by the usual means.

The first such system I built was a theorem prover, in resolution style, for a fragment of Separation Logic [Rey02] useful in shape analysis. This project, called VeriStar [SBA12], dovetailed with VeriSmall [App11], a verified forward symbolic executor for a C-like language, to produce a fully verified program analysis. More recently, I built a system in Coq [Ste13] for reasoning computationally about programs in NetCore [MFHW12], a new language for software-defined networking. The result was a proof-of-concept tool for doing fully automatic verification of network policy programs entirely within the Coq proof assistant.
References


