We are drowning in bugs. Security vulnerabilities, data corruption, crashes, freezes, and plain malfunctions have become a fact of life, with consequences ranging from minor annoyance to privacy violations, data theft, billions of dollars in losses, and even death. And yet, we are continuously increasing our dependence on critical computer systems, in all aspects of our lives: health (electronic patient records, medical devices), money (banking, digital currency), commerce (online shopping, supply chain management), manufacturing (industry 4.0), logistics (automated transit, shipping), public life (online voting, news feeds), relationships (communications, online dating), entertainment, etc.

The computing infrastructure that powers all these systems may have come together in just 60 years, but the conventional wisdom has been that it is unfixable: there is too much of it, and it is too complicated. Our collective approach is mostly to live with the bugs, fixing the most egregious ones when they threaten the whole apparatus (in 2014, after the discovery of the HeartBleed bug, enormous efforts were expended to re-secure the Internet by patching and eventually replacing OpenSSL; until that point, a critical piece of the modern web had been maintained exclusively by two individuals — both named Steve).

It does not have to be this way: over the last 10 years, program-verification technology has made great leaps, so great in fact that the dream of end-to-end verification of complex software systems (formally proving that machine code conforms to mathematical specifications) is now within reach. My research combines programming languages, systems engineering, and verification to build trustworthy software and hardware systems and permanently root out large classes of malfunctions and vulnerabilities.

Recent verification advances have made a few things clear:

a. We have the technology to verify small pieces of complex low-level code, such as high-performance implementations of critical data structures and algorithms written in C or assembly, using verification technology such as Bedrock, Dafny, Low+, or VST.

b. We also have the technology to verify large pieces of non-performance-critical software, written in high-level languages offering powerful abstractions, using interactive theorem provers like Coq and HOL.

c. There is strong industrial demand for verification technology: over the last few years, cryptographic primitives verified in Low+ and Coq were deployed to billions of users in Google Chrome and Firefox.

The field of verification has achieved significant successes, but none of the existing approaches provide a complete solution. Low-level verification technology is promising, but it does not yet scale to large systems without expending significant effort. High-level verification does scale, but two problems make high-level verification of large performance-critical systems impractical today: performance, and trust. First, despite advances, current compiler technology yields disappointing results on high-level languages used for verification: we do not have a robust way to generate high-performance, low-level code from high-level verified programs. Second, even when a performance penalty is acceptable, compiling verified high-level programs typically requires using an unverified process called program extraction, a form of compilation that translates verified programs into inputs comprehensible by existing compilers for languages such as OCaml or Haskell. For lack of alternatives, most verified systems assume the correctness of program extraction and subsequent compilation instead of proving it, and thereby enlarge their trusted computing base (the set of unverified artifacts — hardware, software, axioms — whose correctness a system’s proper operation depends on). Unverified program extraction severely weakens correctness claims because it breaks the chain of trust from specifications to binary code.

As a result, we are stuck verifying either small high-performance programs at relatively high cost, or large high-level programs at relatively low cost but with poor performance and limited trustworthiness.

As part of my PhD, I led a successful effort at MIT to prototype a solution to these two problems. In 2019, we demonstrated the first end-to-end verified pipeline from high-level specifications to low-level code [1]. We did so by combining high-level verification with advanced compiler technology to obtain excellent overall performance with a very small trusted base.
Instead of the usual approach of manually verifying high-level code, extracting it, and using an unverified compiler to obtain binaries, we used metaprogramming to refine formal specifications into functional code, compiled this code into efficient imperative programs using custom-built proof-producing domain-specific compilers (Box 1), and linked the results with high-performance data structures written and verified in Bedrock. The end result of this process has come to be known as binary code extraction: instead of extracting high-level code from the theorem prover, we verify the entire compilation pipeline and only extract the final binaries.

The key insight is to abandon traditional program extraction and monolithic off-the-shelf compilers, and instead build customizable compilers that can safely be extended with arbitrary domain-specific knowledge. This allows system designers to produce high-performance binaries from high-level programs without increasing verification effort or reducing trust: in a sense, my research seeks to solve issues of trust and performance by making verified compiler construction so easy as to be a regular part of software development.

The following section details parts of this pipeline, as well as my ongoing research and my future projects. I start by summarizing my research on the Fiat refinement system, which shaped my research tastes and vision early in my PhD. Then, I outline three ongoing projects, each supplying a distinct axis of my future research program, along with preliminary results: designing the next generation of extensible compilers, extending the chain of trust all the way to circuits by designing hardware-verification technology, and building up tooling for verification technology to enhance its trustworthiness, usability, and comprehensibility.

My long-term goal is to rebuild the foundations of computing on robust grounds. My group will get there by building increasingly sophisticated end-to-end verified systems, starting with artifacts with immediate applications such as networking software and embedded devices proven correct from first principles.

Research projects and future research plans

Fiat: Correct-by-construction refinement of high-level specifications into functional programs

Many domains lend themselves to crisp, simple specifications expressible in “small languages”: database queries using relational languages, packet parsing using packet diagrams from RFC protocol specifications, firewall policies using filtering languages like iptables or BPF, etc. Early in my PhD, I co-authored the Fiat system, the first correct-by-construction refinement framework embedded in the logic of the Coq theorem prover [2, 3]. In Fiat, power-users define domains such as databases or packet-parsing to support high-level specification of domain-specific programs, then implement refinement logic to automate the derivation of functional code implementing these specifications. End users write programs by specifying functionality and invoking the refinement logic to generate code. Because the logic produces proofs, users can trust the resulting code.

As a follow-up to the original Fiat paper, our team used refinement to derive verified binary encoders and decoders for network packets (ARP, TCP, UDP, IP, etc.), using non-determinism to capture encoding choices and automatically generating decoders proven to handle all allowable encodings [4]. Our project, Narcissus, provided the first step of an end-to-end pipeline, but it still generated high-level functional code: we had to extract generated programs to OCaml to run them, limiting performance and enlarging our trusted base.

Box 1 — Compilation inefficiencies

Consider the task of converting an ASCII string str to uppercase, for example to use it as a key in a table of DNS records. What might a fast implementation look like? If the original str is never reused, a good choice would be to represent str as a struct containing a length and a pointer to a contiguous array of characters:

```cpp
for (int pos = 0; pos < str.len; pos++) {
    str.chars[pos] = toupper(str.chars[pos]);
}
```

In a purely functional implementation of this task, like in Haskell or Coq, we would instead represent strings as linked lists of characters and write "String.map Char.toUpper str", which applies the function Char.toUpper to each character in str.

Could a general-purpose Coq or Haskell compiler translate this to the efficient code above? No: it could not assume that str can be mutated, nor would users be able to tell it. It would have to assume the worst and produce slow code. The implementation would pointer-chase through a linked list (creating data dependencies and cache pressure); it would build a fresh string (costing allocations, cache misses, and an extra traversal for garbage collection); and it would either stack-overflow on long strings or traverse the string twice (doubling allocation and pointer-chasing costs), or accumulate continuations (even more allocation).

The C version performs a single pass, occupies constant stack space, does not allocate, is cache-friendly, can be unrolled, and is trivially vectorizable (toupper on ASCII chars is just a comparison and a bitmask).

My research makes it possible to build custom compilers that apply optimizations like this one, translating String.map Char.toUpper to an efficient in-place loop. Such transformations are crucial to get optimal performance, but so narrow in scope that they would not be a good fit for a general-purpose compiler.

Instead of the usual approach of manually verifying high-level code, extracting it, and using an unverified compiler to obtain binaries, we used metaprogramming to refine formal specifications into functional code, compiled this code into efficient imperative programs using custom-built proof-producing domain-specific compilers (Box 1), and linked the results with high-performance data structures written and verified in Bedrock. The end result of this process has come to be known as binary code extraction: instead of extracting high-level code from the theorem prover, we verify the entire compilation pipeline and only extract the final binaries.

The key insight is to abandon traditional program extraction and monolithic off-the-shelf compilers, and instead build customizable compilers that can safely be extended with arbitrary domain-specific knowledge. This allows system designers to produce high-performance binaries from high-level programs without increasing verification effort or reducing trust: in a sense, my research seeks to solve issues of trust and performance by making verified compiler construction so easy as to be a regular part of software development.

The following section details parts of this pipeline, as well as my ongoing research and my future projects. I start by summarizing my research on the Fiat refinement system, which shaped my research tastes and vision early in my PhD. Then, I outline three ongoing projects, each supplying a distinct axis of my future research program, along with preliminary results: designing the next generation of extensible compilers, extending the chain of trust all the way to circuits by designing hardware-verification technology, and building up tooling for verification technology to enhance its trustworthiness, usability, and comprehensibility.

My long-term goal is to rebuild the foundations of computing on robust grounds. My group will get there by building increasingly sophisticated end-to-end verified systems, starting with artifacts with immediate applications such as networking software and embedded devices proven correct from first principles.

Research projects and future research plans

Fiat: Correct-by-construction refinement of high-level specifications into functional programs

Many domains lend themselves to crisp, simple specifications expressible in “small languages”: database queries using relational languages, packet parsing using packet diagrams from RFC protocol specifications, firewall policies using filtering languages like iptables or BPF, etc. Early in my PhD, I co-authored the Fiat system, the first correct-by-construction refinement framework embedded in the logic of the Coq theorem prover [2, 3]. In Fiat, power-users define domains such as databases or packet-parsing to support high-level specification of domain-specific programs, then implement refinement logic to automate the derivation of functional code implementing these specifications. End users write programs by specifying functionality and invoking the refinement logic to generate code. Because the logic produces proofs, users can trust the resulting code.

As a follow-up to the original Fiat paper, our team used refinement to derive verified binary encoders and decoders for network packets (ARP, TCP, UDP, IP, etc.), using non-determinism to capture encoding choices and automatically generating decoders proven to handle all allowable encodings [4]. Our project, Narcissus, provided the first step of an end-to-end pipeline, but it still generated high-level functional code: we had to extract generated programs to OCaml to run them, limiting performance and enlarging our trusted base.
Axis 1. Extensible extraction of efficient imperative code

In most cases, experienced programmers have a clear idea of what an efficient implementation of a given program might look like, even if their high-level language does not enable them to express it (in Narcissus, for example, we knew exactly where we were losing performance: lack of mutation and slow error handling). As a result, a common development strategy is to write and verify models and prototypes in high-level languages, and then rewrite critical parts by hand in a lower-level language, often without verification and at the cost of added bugs and lower maintainability.

The problem is that typical compilers are monolithic and inflexible: high-level languages offer powerful abstractions that compilers implement in full generality, yielding disappointing performance in exchange for completeness. When allowed, compiler extensions usually take the form of single-language transformations: users may be able, for example, to teach the compiler new algebraic laws such as $\text{map } f \ (\text{map } g \ l) = \text{map } (f \circ g) \ l$. More ambitious transformations, such as cross-language transformations that break the abstractions offered by the language (Box 1), are not even expressible: a program written in a garbage-collected pure functional language will not be magically transformed, regardless of the compilation hints that a user specifies, into code that mutates imperative data structures in place and uses pointer tricks to speed up its computations.

This is precisely what one of my main PhD projects is about. By phrasing compilation as an existentially quantified program-verification problem and deriving a witness through proof search, my research enables users to assemble custom compilers from collections of individual cross-language transformations [1]. Users provide context-dependent mappings of high-level types and operations to custom low-level data structures and algorithms, achieving excellent performance without having to rewrite their programs in a way that makes verification harder. As a result, the same verification-friendly inductive type of, say, lists of keys and values can be translated to a contiguous array storing its values inline, a binary search tree, a hash table, or any other low-level structure hand-crafted and verified for optimal performance; and a sequential lookup in that list of pairs will be alternatively implemented as a linear array scan, a tree traversal, a hash-table lookup, or any corresponding low-level operation. By specifying adequate mappings, users can leverage their optimization insights to build the perfect compiler for their program instead of manually rewriting it into a low-level language.

The first axis of my future research program will be to develop the next generation of extensible verified compilers and use them to verify critical system libraries, including a complete network stack. Connecting this stack to existing cryptographic implementations will provide the basic blocks to construct end-to-end verified secure communication systems. The next step will be bare-metal programs, targeting small embedded systems such as internet-of-things devices and medical devices.

Axis 2. Cycle-accurate semantics and verified compilation of hardware circuits

While the last decade has been rich in verification advances, they have also been rich in new bug discoveries; in particular, it has become increasingly clear that most hardware is riddled with bugs, and that even the most rigorous software proofs can be broken by untrustworthy hardware: hardware is implicitly part of the trusted base of any software project.

Consequently, as part of my PhD, I have designed and implemented the first high-concurrency verified compiler for a rule-based hardware language [5]. The language, Kôika, is a derivative of Bluespec: it provides a mix of modularity and control on timing to enable the rapid design of efficient hardware. Its semantics formally captures the notion of one-rule-at-a-time execution, a convenient abstraction that allows programmers to reason about the functionality of concurrent hardware modules as if they executed sequentially. Additionally, thanks to a precise model of ephemeral history registers (EHRs), Kôika allows users to reason about cycle counts, enabling proofs of hardware performance (Box 3). Thanks to verified circuit optimizations, my compiler generates circuits whose performance is comparable to that of the commercial Bluespec compiler, bsv.
In addition to a compiler, I implemented a comprehensive simulation and debugging suite for Köïka by leveraging software compilers and debuggers [6]: on typical designs, my simulator beats state-of-the-art circuit simulators by a factor of two to five, while enabling a wide-variety of workflows typically used with software (for example, collecting code-coverage metrics in simulation gives us extremely fine-grained information that would otherwise require adding hardware performance counters; similarly, placing conditional breakpoints within individual rules allows us to debug at sub-cycle precision).

The second axis of my future research program will be to build a hardware-verification toolkit robust enough to make it straightforward to verify realistic logic circuits and CPUs for embedded systems: embedded processors are ubiquitous in critical systems and are much simpler than desktop or server processors, making them realistic but high-impact targets for verification. Once complete, these will form an ideal platform to run end-to-end verified software on, yielding fully-verified devices with complete immunity to a large class of bugs and vulnerabilities.

**Axis 3. Human-worthy verification**

While verification technology has made great progress, adoption remains limited: many of the tools most popular within the verification community are hard to use by non-experts.

I have expended significant effort throughout my PhD to develop tooling to improve the usability of the Coq theorem prover. My package to turn Emacs into a proper Coq IDE, Company-Coq [7], is used by over half of the entire Coq community, has received hundreds of stars on GitHub, and is routinely recommended in introductory Coq courses. I am also the author of the main F* IDE (fstar-mode), as well as Emacs modes for Dafny, Boogie, and Z3, and WASM builds of Z3 (with over a hundred Github stars).

Usability is not enough, however: non-experts will not trust verified systems unless experts can explain clearly what guarantees they provide. We formal method researchers love to talk about program proofs, but proofs are only as good as our capacity to explain what they are and what they mean.

Most recently, I have focused on making Coq proofs more comprehensible and easier to share and document: I recently published Alectryon, a literate programming system and proof visualization toolkit for Coq [8]. Alectryon is primarily a compiler: it creates interactive documents from Coq proof scripts by interleaving user input with the provers’ output, thus allowing readers to browse Coq developments without running Coq, optionally with advanced visualizations (Box 4). But it is also a literate programming system, which translates documents back and forth between proof-oriented and prose-oriented views, allowing authors to craft documents mixing code and proof seamlessly.

The third axis of my research program will be to reimagine interactive theorem prover tooling: we need better proof-editing and proof-exploration tools, better teaching materials, and a more systematic understanding of verified-software design and proof automation to turn verification and theorem-proving from a productive research area to a universally useful tool.

I consider this tool as an engineering jewel and a very important contribution to the Coq ecosystem: the description of its design and its implementation is really inspiring and, as far as I know, the tool significantly improves the state-of-the-art about literate programming tools [...]. I will not be surprised if this tool has a significant impact on the textbooks about mechanically formalized developments both in Mathematics and Computer Science.


**Research methods and philosophy**

I think of programming languages and systems engineering as enabling technologies: we envision tools and ideas to help other areas strive, while making fundamental discoveries about programs and languages.
It comes as no surprise, then, that **collaboration with academia and industry is at the heart of my research identity**: my favorite aspect of my PhD years was the diversity of ideas, fields, and collaborators that I was exposed to. In addition to giving demonstrations to startups, industry, and government labs like BedRock Systems, Galois, and Sandia, I have worked with researchers at Purdue [4], BAE, GrammaTech, U. Penn [9], Microsoft Research [10], INRIA [11], and MIT, and I have had many productive exchanges with researchers all over the world (I particularly enjoy the spontaneous discussions that are made possible by my authoring widely used tools).

**The freedom to pursue ambitious ideas, the collegiality of academia, and the intellectual stimulation of research are the key factors that drive me to apply for faculty positions.** My research program will open many avenues for fertile collaboration, spanning a wide range of domains: verification, programming languages, systems programming, security, hardware, networking, cryptography, databases, and even HCI.

**Conclusion**

The three axes of my research program — extensible proof-producing compilers, verified hardware cores, and reinvented verification tooling — will come together to enable end-to-end verified systems. My research is broadly applicable, and I will focus on two areas in the coming years:

**Secure networking**: I will start with trustworthy implementations of critical network services, like clock synchronization (NTP) or domain-name resolution (DNS). As the tools mature, my group will obtain robust and usable extensible-compilation technology ready for larger deployment: we will tackle increasingly complex programs, like a complete HTTP stack, verified VPN technology, parts of an OS kernel, and device drivers. In parallel, we will develop verified hardware to run these stacks on, and tooling to democratize this technology. With all these pieces in place, we could have secure and portable point-to-point communications, allowing anyone to connect a small fully-verified router to their hotel room's Ethernet and have instant access to an inviolable tunnel connecting them to another location across the world. With journalists, NGOs, and even academics under constant threat in many countries, this would provide invaluable safety.

**Trustworthy devices**: I will work with systems engineers, hardware designers, control theorists, and medical engineers to design embedded systems proven to operate correctly. Specifically, my first application domain will be verified **looping devices**. Patients suffering from severe forms of type-1 diabetes normally have to wake up every few hours at night to measure their blood glucose and program a pump to inject an adequate amount of insulin. Recently, programmers have found ways to **loop** these devices together, reprogramming pumps to inject insulin automatically based on glucose readings. Done well, this is an invaluable quality-of-life improvement. Done poorly, this is nearly a death sentence. One of my medium- to long-term targets will be to program and fabricate a complete prototype of a verified looping device.

These systems will constitute valuable research artifacts while offering direct real-world applications. **With the right research, we can rebuild the foundations of computing on robust grounds, and usher in a new era of computing that we can trust.**
References


