Research Statement

Kiran Gopinathan, April 2025

The need for trust in modern software has never been greater. Programs are being written and evolving at an unprecedented rates; modern systems are plagued by bugs with increasing frequency and substantially more severe consequences. Even as recent as a year ago, buggy patches to the CrowdStrike monitoring service lead to crashes and widespread outages at global scales, ultimately resulting in an estimated *$10 billion* worth of economic damages and untold human costs. With the increasing popularity of LLMs and adoption of ML-generated code, soon it may be the case that not even developers know what their programs do, painting a bleak picture of the future: the onslaught of bugs is only set to continue and worsen in severity.

Formal verification has long been developed as the solution to this problem, providing the strongest guarantees of software correctness that humanity is able to produce, and developments over the years have succeeded in bringing it to the cusp of mainstream adoption. In the past decade alone, the field has seen a substantial paradigm shift, as techniques and frameworks have matured to the point where it is now possible to verify practical real-world software: compilers, file systems, distributed programs, cryptography libraries and even whole operating systems. Most recently, the HACL verification effort has seen formally verified cryptography code deployed to popular web browsers (Firefox, Chrome, *etc.*), forming a crucial part of securing the daily browsing of billions of users. Sadly, in this shift from theory to practice, researchers are slowly uncovering and coming to face a more pernicious problem: that of how to maintain these verified software systems.

Code evolves; proofs do not. The fundamental problem facing existing verification is that it treats the systems to be verified as static artefacts: large scale verification efforts fix particular versions of the systems they target and rarely consider updates; any such changes often incur extensive manual effort to repair and major restructurings of respective proofs. In practice, many landmark verified systems are simply unmaintained, for example: FSCQ, a crash-safe verified file-system (339 citations), last updated 3 years ago; Verdi, a formalisation of the Raft distributed protocol (449 citations), last updated 2 years ago; JSCert, a verified ECMAScript reference interpreter (176 citations), last updated 10 years ago. The sad state of affairs for many real verified systems in practice is that their artefacts are often simply left to stagnate and fall into disrepair.

The past several decades of research have shown that formal verification can indeed scale to real code, but if verified software is to ever see wider adoption, it is vital that we solve this pervasive problem of proof maintenance. **In my PhD work, I pioneered forward-looking research into this very problem, developing the first state of the art automated techniques for the maintenance of verified software systems.**

My PhD thesis [3] proposed a general framework for handling evolution of verified systems, substantiating it over several case studies of maintaining real-world software. In my work, I built on the observation that verified software maintenance typically falls under one of a few categories. That is, maintenance often arises due to either (1) handling changes in specifications or data structures in response to shifting business requirements or (2) handling changes in implementations for performance. To handle the first case, I co-authored a tool, `certisuslik` [9], that could automatically synthesise certified C programs from specifications, removing the need for users to maintain implementations at all. Follow up work found that the `certisuslik` programs, despite being synthesised, had comparable performance to binaries from production compilers (GHC). For the second case, I developed a tool, `sisyphus` [7], for repairing proofs of verified OCaml programs. The tool was evaluated on a number of real OCaml programs and their changes from version control, including several functions from JaneStreet's core library, and found to drastically reduce manual maintenance efforts from hours to minutes. Following my PhD, I broadened the scope of my research to consider more general (3) non-functional issues related to verification tools themselves, such as verification times or handling version changes. As part of this, I introduced a novel proof maintenance optimisation, dubbed *proof-localisation*, intended to reduce verification times. I implemented this in a tool, `axolocl` [2], which was able to reduce re-verification times of long running queries by up to 1000x in several real-world verified code bases, including in AWS' cryptography libraries.

The key insight across my research is in exploiting the latent information that is present but overlooked when working in a formally verified setting. While the challenge of maintenance and repair in software has traditionally been in inferring developer intent, the main observation that drives my work is that in formal developments, much of this information is available within verification artefacts: proof terms, SMT traces, synthesis trees. As formal verification has become more practical and verified systems have proliferated, mature and established formal artefacts are emerging for the first time, exposing the potential for an entire new field of research into how to maintain these systems. **My research vision combines techniques from formal verification, programming languages and program repair to tackle the problem of proof maintenance.**

Currently, verification carries a heavy opportunity cost. Not only does verification require developers to write specifications and proofs of their programs, but it then also effectively doubles the maintenance burden, requiring subsequent changes to update both code as well as proofs. **The long term vision of my research is to reduce the costs of formal verification and radically shift this paradigm**: verification should not be a burden — as my research shows, verified code has the potential to both be easier to write and maintain than unverified software — and my broader research goals are to bring this vision to reality.

Research Projects & Future Work

The following section presents a high level overview of my current research work and future research plans. I break the problem of proof maintenance into three main axes in which verified systems can evolve, and present key work I have done into tackling changes of each class: changes in implementations, changes in specifications, and finally, non-functional changes, such as slow verification times or migrating artefacts across tool versions.

Axis 1: Proof Maintenance of Implementations

One of the most common forms of evolution in real world software is changes in implementations, especially so in software exposing libraries and APIs for other downstream clients. In practice, library developers frequently refactor implementations of a library to improve performance or code quality while preserving the public API and specifications to ensure backwards compatibility. Given the ubiquity of these changes in unverified software, exploring this problem in verified systems is a natural place to start investigating automated proof maintenance.

The key problem of proof maintenance of implementations is the fundamental difficulty of verification: proving an arbitrary updated implementation satisfies a specification is in general an undecidable problem. For practical and scalable maintenance of verified implementations, tools must be developed that somehow leverage the unique information present in a repair setting: the latent information within the old program and proofs. Intuitively, developers find it easier to modify and refine a working program than to write one from scratch; the same *should* hold for verified systems: it should be easier to fix a broken proof of a verified program than to verify it from scratch, but sadly no prior work had succeeded in extracting and making use of this information.

This problem was the main focus of one of the projects of my PhD research. I developed a tool, `sisyphus`, to provide automated repair of verified OCaml programs over changes in their implementations [7]. Combining ideas from dependent type theory and program logics, I developed a novel technique based on the Curry-Howard Correspondence, dubbed *proof-driven testing*, that allowed efficiently repairing proofs when implementations changed. The core insight was that repair of verified OCaml programs could be framed as an invariant inference problem, *i.e.* inferring an invariant to verify the new implementation, and by analysing the proof terms provided by the proof assistant, it was possible to extract executable testing functions that could be used to quickly validate candidate invariants and thereby scale up the repair process to real-world programs. I evaluated `sisyphus` on a corpus of real-world OCaml code and its changes taken from the version control of popular OCaml libraries, including widely-used industrial code bases such as JaneStreet's `core` or the venerable `containers` library. Overall, `sisyphus` was able to reduce the maintenance burden of real-world code from hours to minutes. The contributions of this work were awarded a **SIGPLAN Distinguished Paper Award** at PLDI'23.

The first axis of my planned research will be to tackle proof maintenance of implementations, with the focus of developing novel approaches to reuse information in the repair process. The old adage goes "a complex system that works is invariably found to have evolved from a simple system that worked"[1]. Changing a program is easier than writing it from scratch. Repairing a broken verified system *should be* easier than verifying it from scratch, however this is currently beyond the state of the art verification techniques, and the goal of my research in this direction is to broach that gap. The next steps in this direction will be to extend my techniques of proof maintenance of implementations beyond the context of sequential OCaml programs to languages with more complex semantics, such as concurrent, distributed or probabilistic programs.

Axis 2: Proof Maintenance of Specifications

Another major form of evolution in real-world software is found through changes in specifications, typically brought about as the business requirements and design goals for a system naturally shift over its lifetime. Evolution of this form typically involves far more substantial changes across a program, including major restructurings and changes in algorithms, so reuse of old code, and therefore old proofs, is often not possible. In this case, the best we can hope for in maintenance is automatically synthesising code to satisfy new specifications.

As another project in my PhD, I developed tooling to tackle proof maintenance in this setting. I co-authored a tool `certisuslik` to automatically generate verified efficient C programs given only a user provided specification of the desired behaviour [9]. The tool was developed by extending an existing deductive synthesis tool, `suslik`, with a lightweight modification to produce certificates of any generated code. The key insight in this work was that the internal search trees that synthesis tools construct actually contain exactly the necessary latent information to verify any programs they generate, and designing a general algorithm to translate synthesis trees to verification proofs. My main contribution in this work was to extend the initial algorithm, which targeted programs in non-executable toy languages, to a real-world setting, extending the tool to produce executable certified C code, and bridging the gap from the synthesizer's simple internal logic to a production verification framework sufficient to certify practical programs. Follow up work by subsequent researchers[2] were able to use the tooling I developed to use `certisuslik` to generate efficient list and tree data structure manipulating

[1] John Gall (1975) Systemantics: How Systems Really Work and How They Fail p. 71

[2] Higher-Order Specifications for Deductive Synthesis of Programs with Pointers. David Young, Ziyi Yang, Ilya Sergey, Alex Potanin.

programs, which benchmarked competitively to battle-hardened production compilers such as the Glasgow Haskell Compiler (GHC), demonstrating the efficacy of this methodology for real-world proof maintenance.

The second axis of my future research is to tackle proof maintenance over changes in specifications, utilising latent information present within existing program synthesisers to certify their outputs. While program synthesis has been extensively studied, the question of generating certified code as usable for proof maintenance is relatively nascent, and my goal in this direction is to extend the insights from this work to other existing synthesis approaches. My next steps in this direction will be to investigate developing certifying versions of other synthesis strategies beyond deductive synthesis, such as CEGAR or LLM-based synthesis tools.

Axis 3: Non-Functional Proof Maintenance

Looking beyond traditional problems of software maintenance, more recently, proof engineers have started to run into new unseen problems unique to large verified developments. For example, as verified systems have grown in size, there has been increasing reports of developers running into issues with slowdowns or instability in verification times when using SMT solvers or proof automation. Similarly, changes in the verification tool itself often require extensive maintenance patches and updates spanning entire code-bases. Broadly, this class of problems can be thought of maintenance tasks that are *non-functional* in nature — the functionality remains the same, but patches are required for the system to remain usable — and sadly, given that systems large enough to encounter these issues have only recently emerged, research into this remains in a nascent state.

Following my PhD, I broadened the scope of my research to include these novel problems, and lead a project tackling one particular form non-functional maintenance: reducing slow re-verification times in program verifiers. As verified programs grow in size, the size of their verification queries and therefore verification times grows which can substantially hamper the maintainability of the system, as developers must wait increasingly long times to get feedback after each change. As part of this work, I developed a tool `axolocl` which implements a novel optimisation, dubbed *proof localisation*, for reusing information from the proof search across multiple verification runs, and thereby substantially reducing verification times when re-verifying a program [2]. The key insight in this work was utilise internal debugging output of the SMT solver to extract latent information about the proof search in an initial verification, such as UNSAT cores and axiom instantiations, and develop a light-weight encoding to store this information to reuse in subsequent verification attempts. In this process, I discovered a novel class of information, dubbed *lurking axioms*, which are required to reproduce SMT results, but are not logically relevant in the proof, and thus entirely absent from traditional SMT outputs. I implemented this technique as an extension to the Boogie Program Verifier, and evaluated the tool across a number of real-world production verified systems and found significant speedups for long-running files of up to 1000 times.

The third axis of my planned research vision is to look beyond the traditional forms of software evolution, and tackle the emerging non-functional maintenance tasks that uniquely arise in this setting. My next steps in this research direction are to characterise and investigate other forms of this class of this maintenance: automating evolving verified systems of changes in verifier versions, and tackling the problem of proof instability, where automated proofs become “flakey” and have high variance in their verification times.

Future Work

My current work captures broad swathes of the problem area of proof maintenance, though there are several open questions that still remain unaddressed. Below I list key problems that I intend to explore in future works.

Proof Stability in Automated Verifiers My prior work, `axolocl` [2], tackles the problem of reducing verification times of long-running proofs in automated verifiers but does not directly address proof instability. An increasingly common phenomenon observed in larger verified systems is of queries that have high variance in verification times, with small irrelevant changes to programs resulting in incommensurate changes in verification times. As follow up work, I plan to develop tools to tackle this problem, investigating whether the key ideas from proof-localisation can be extended and enhanced to also also improve the stability of verification queries.

Proof Maintenance beyond Sequential Code While my work has substantially developed the state of the art of proof maintenance, the field as a whole is still largely in its nascence, and all prior works in the direction, including my own, have focused purely on maintenance of sequential programs. As future work, I plan to extend also tackle proof maintenance when applied to programs in languages with more complex semantics, such as:

1. **Randomised Algorithms** - Early in my PhD, I developed a large-scale verified proof of a class of complex probabilistic data structures known as approximate membership query structures, a class which includes the widely-used *Bloom filter* data structure, and provided the first formally verified proof of the false positive rate [10]. While the development was carefully organised, the entangled nature of reasoning used for randomised programs meant that the overall development was still brittle, and has become unmaintained over time. Building on my experience in this domain, I plan to investigate how proof maintenance techniques can be adjusted to handle the kinds of proofs found in randomised algorithms.

2. **Distributed Systems** - I have previously worked on formal verification of distributed protocols, in particular, developing an initial partial formalisation of the safety properties of the Nakamoto consensus protocol in the Rocq theorem prover [11]. In a similar style to randomised algorithms, safety and liveness properties of these distributed algorithms tend to be deeply entangled and whole-system properties, which makes modular and robust verification challenging. As future work, I plan to investigate developing proof maintenance techniques for maintaining proofs of verified protocols, focusing both on proofs in interactive proof assistants such as Rocq or Lean as well as automated tools such as Ivy.

Proof Maintenance in Unverified Programming Languages As type systems have become more complex and type checking has grown closer to verification, there is a growing opportunity to extend techniques from proof maintenance from verified systems back to unverified code. In particular, can formal artefacts of type checking in languages with non-trivial static analyses, such as Rust and its borrow checker, be used to automate the process of maintaining programs in these languages as they evolve? I have previously investigated developing tooling to provide traditional editing operations such as refactoring in Rust where maintaining type-correctness over such transformations is non-trivial [8]. As future work in this direction, I plan to investigate how the insights from my proof maintenance work can be applied to maintaining Rust programs, and how the latent information stored by Rust's type and borrow checker can be exploited to automatically repair Rust programs.

Proof Repair for Meta-Programming Another area in which techniques from proof maintenance could be applied more broadly is in the maintenance of macro-based meta-programs. Meta-programming facilities in a language allow users to change and extend a programming language to embed custom domain specific languages and tune the language to their problem domain. I have previously worked on studying the use of macro-systems in practice across the Racket Programming Language ecosystem [4]; a common problem in found by meta-programming users is in maintaining these macro-based DSLs: in particular in how DSL designers can maintain their DSL-based programs as the DSL itself updates and evolves. As future work, I plan to investigate how techniques from proof maintenance can be applied in meta-programming based DSLs, and how latent information stored by the compiler can again be used to automate repairing such programs.

Beyond Maintenance to Verification Guided Optimisation Currently, when verifying a program, beyond ensuring its correctness, verification does not provide any further benefits to the end user, but does it have to be this way? As my research has shown, the information stored in proof artefacts can capture insights about the intents a developer had in mind when writing a piece of code and it seems within reason that this information could even be used to *optimise* programs themselves. One area where this information would be particularly useful is concurrent programming, where correctness of code often depends on larger whole-system properties. Previously, I have worked in this domain, developing a library to simplify the construction of concurrent data structures through a technique known as *batching*, where multiple concurrent requests grouped and processed together in a batch [5]. As future work, I plan to investigate how the information in formally verified code might be used to automatically optimise it into a more efficient concurrent implementation, potentially using batching.

Language Design for Proof Maintenance In the long term, looking beyond immediate approaches of responsive solutions to proof maintenance, *i.e.* fixing proofs when they break, it is important to also consider *preventative* approaches to proof maintenance: how verification languages themselves can be designed to make them robust and resistant to change. In prior work, I have worked on a number projects on language design, helping to design the Rhombus programming language [6] and build tools for synthesising programming languages to describe potential attack surfaces [1], and over the course of my research career have built an insight into the ways in which languages can be crafted to satisfy different design goals. As part of my larger research vision, I plan to investigate how existing verification languages can be extended to aid proof maintenance, for example, providing new language features for users to constrain proof search in automated solvers, or to ways to instruct the solver to explicitly reuse results from previous verification runs.

Conclusion

The past decades of research have brought formal verification to the cusp of mainstream adoption, but a final barrier to wider appeal still remains: the problem of how to maintain these verified systems as they change. The three axes of my research direction — automation for changes in implementations, changes in specifications and non-functional changes — will come together to tackle the problem of proof maintenance for the majority of programs. As my research shows, verified code has the potential to be easier to write and maintain than unverified software, and my research goals are to expose that potential to the world. The success of my broader research agenda will radically shift the balance between verified and unverified software, substantially reducing the cost of maintaining formal artefacts, and make verification the "obvious" option for software moving forward.

References

[1] Yuxi Ling, Gokul Rajiv, **Kiran Gopinathan**, and Ilya Sergey. 2025. Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA) *(SEC'25)*. USENIX Association, USA. URL: https://kirancodes.me/pdfs/doppler-usenix25.pdf

[2] **Kiran Gopinathan**, Dionysios Spiliopoulos, Vikram Goyal, Peter Müller, Markus Püschel, and Ilya Sergey. 2025. Accelerating Automated Program Verifiers by Automatic Proof Localization. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 21-26, 2025 (Lecture Notes in Computer Science)*. Springer.

[3] **Kiran Gopinathan**. 2024. *Scaling the Evolution of Verified Software*. PhD thesis. National University of Singapore, Department of Computer Science, Singapore, Singapore. URL: https://kirancodes.me/docs/thesis.pdf

[4] Yunjeong Lee, **Kiran Gopinathan**, Ziyi Yang, Matthew Flatt, and Ilya Sergey. 2024. DSLs in Racket: You Want It How, Now?. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering* (Pasadena, CA, USA) *(SLE '24)*. Association for Computing Machinery, New York, NY, USA, 84–103. doi:10.1145/3687997.3695645 URL: https://kirancodes.me/pdfs/racketnow-sle24.pdf

[5] Callista Le, **Kiran Gopinathan**, Koon Wen Lee, Seth Gilbert, and Ilya Sergey. 2024. Concurrent Data Structures Made Easy. *Proc. ACM Program. Lang.* 8, OOPSLA (2024), 1814–1842. doi:10.1145/3689775 URL: https://kirancodes.me/pdfs/obatcher-oopsla24.pdf

[6] Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, **Kiran Gopinathan**, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay A. McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. 2023. Rhombus: A New Spin on Macros without All the Parentheses. *Proc. ACM Program. Lang.* 7, OOPSLA (2023), 574–603. doi:10.1145/3622818 URL: https://kirancodes.me/pdfs/rhombus-oopsla23.pdf

[7] **Kiran Gopinathan**, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. doi:10.1145/3591221 URL: https://kirancodes.me/pdfs/sisyphus-pldi23.pdf

[8] Sewen Thy, Andreea Costea, **Kiran Gopinathan**, and Ilya Sergey. 2023. Adventure of a Lifetime: Extract Method Refactoring for Rust. *Proc. ACM Program. Lang.* 7, OOPSLA (2023), 658–685. doi:10.1145/3622821 URL: https://kirancodes.me/pdfs/rem-oopsla23.pdf

[9] Yasunari Watanabe, **Kiran Gopinathan**, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. doi:10.1145/3473589 URL: https://kirancodes.me/pdfs/CySuSLik-icfp21.pdf

[10] **Kiran Gopinathan** and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 279–303. doi:10.1007/978-3-030-53291-8_16 URL: https://kirancodes.me/pdfs/ceramist-draft.pdf

[11] **Kiran Gopinathan** and Ilya Sergey. 2019. Towards mechanising probabilistic properties of a blockchain. In *CoqPL'19: The Fifth International Workshop on Coq for PL*. URL: https://kirancodes.me/pdfs/probchain-coqpl19.pdf