PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

Research

Article submitted to journal

Subject Areas:
software, theory of computing

Keywords:
seL4, proof engineering, code/proof co-generation, cogent

Author for correspondence:
Gerwin Klein
e-mail: gerwin.klein@data61.csiro.au

Provably Trustworthy Systems

Gerwin Klein[1,2], June Andronick[1,2], Gabriele Keller[1,2], Daniel Matichuk[1,2], Toby Murray[1,3], and Liam O'Connor[2]

[1]Data61 CSIRO, Sydney, Australia
[2]UNSW, Sydney, Australia
[3]University of Melbourne, Australia

We present recent work on building and scaling trustworthy systems with formal, machine-checkable proof from the ground up, including the operating system kernel, at the level of binary machine code. We first give a brief overview of the seL4 microkernel verification and how it can be used to build verified systems. We then show two complementary techniques for scaling these methods to larger systems: proof engineering, to estimate verification effort, and code/proof co-generation, for scalable development of provably trustworthy applications.

1. Introduction

Recent years have seen a number of impressive breakthroughs in the formal code-level verification of real-life systems. They demonstrate the feasibility of provably trustworthy systems. In this paper, we show how we can address the remaining challenge of making formal proofs more repeatable and scalable.

These breakthroughs range from the verified C compiler CompCert [1,2], to the verified microkernel seL4 [3–5], the concurrency verification in CertiKOS [6], the verified conference system CoCon [7], the verified ML compiler CakeML [8], the verified interactive theorem provers Milawa [9] and Candle [10], the verified crash resistant file system FSCQ [11], the verified crypto implementation of OpenSSL HMAC [12], the verified distributed system Ironfleet [13], and more — not to mention significant mathematical theorems such as the Four Colour Theorem [14,15], the mechanised proof of the Kepler Conjecture [16,17], and the Odd Order Theorem [18]. These are not toy systems. For instance, the CompCert compiler is a commercial product, and the seL4 microkernel is used amongst others, in space [19], autonomous aviation [20], and as an internet of things platform.

THE ROYAL SOCIETY
PUBLISHING

This is a remarkable time. Even five years ago, many of these results were believed infeasible. Yet they exist. With formal verification of real systems finally in reach, a number of problems that plague the development of real-world dependable software have a chance for solution.
They include faster, more agile, and at the same time more meaningful software certification for high-assurance applications in areas such as automotive, rail, aviation, aerospace, autonomous vehicles, medical, and critical infrastructure. Currently such applications are hampered in what they can achieve by the low complexity that can be certified with high confidence. Complexity is the enemy of dependability, but fully analysed complexity can implement smarter safety and security functionality. More scalable certification and verification will improve safety. Foremost amongst these areas is cybersecurity. The current approach of the arms race between exploits and mitigations is guaranteed to fail: the increasing complexity of real systems strongly favours the attacker. Formal proof gives a potential way out — while it does not address human behaviour such as susceptibility to password phishing, it can fully eliminate almost all known classes of *software* attacks and reduce remaining attack vectors to avenues that are too expensive for anyone but well-funded state actors. It has the potential of shutting down the malware (and anti-virus) market because it would remove economic incentive. It has the potential to increase political stability, because it would dramatically reduce the risk and effects of cyber attack, which, if unchecked, is clearly developing into an increasing asymmetric threat. Seen from this angle, this is one of the rare opportunities where Computer Science could realistically contribute to world peace. It is almost a moral imperative to push this work forward into practical use.

The technique that all of the results above use is interactive, machine-assisted proof. The main systems employed in these results were Isabelle/HOL [21], HOL4 [22], Coq [23], and Dafny [24]. We posit that this is no accident. Automated methods are not yet powerful enough to discover deep and large proofs by themselves. Even fully automated provers require significant user input, such as invariants, or tailored settings, or tweaked theorem definitions that are amenable to automated proofs. However, automated verification has become strong enough to truly assist human proof engineers in conducting such large-scale verifications at a significant level of detail.

The involvement of human proof engineers points to the currently major limitation of such verifications. Despite all the successes enumerated above, verification is still expensive and hard to scale to larger systems. We need to get to the stage, where such results are not remarkable achievements, but instead routine, repeatable, and affordable. In previous work [4], we analysed the effort spent on the seL4 microkernel verification in relation to traditional high-assurance certification and to traditional good quality engineering. We concluded that the formal verification was already cheaper than full high-assurance software certification, but that it is about a factor of five more expensive than traditional good quality software engineering in embedded systems [4, Section 7.2.3]. A factor of five is not an unachievable goal for effort reduction.

In the remainder of this paper we give an overview of the seL4 microkernel verification to provide an impression of the level of depth, breadth, and realism that can be achieved with interactive formal verification. We briefly touch on how a verified operating system kernel on its own can already be used to provide system-level guarantees for specific classes of systems. We then discuss two directions of ongoing research that aim to make formal proof more repeatable and more scalable: Section 3 shows how *proof engineering* techniques, similar to software engineering techniques, can help making formal verification more predictable and repeatable, by analysing data from the seL4 verification. We summarise our findings from previous work on correlations between specification size, code size, proof size, and proof effort. While these are not final results yet for an estimation model, they indicate that one might exist, and that formal proof projects can be managed effectively. Section 4 presents an overview of our approach to code/proof co-generation to make proof more scalable. The basic idea is that wherever code is generated, a proof of correctness should also be generated, not only to justify the correctness of the generated code, but chiefly to make reasoning about the generated code possible at a higher, more productive level of abstraction. We demonstrate the approach using a functional language for the implementation of low-level systems code such as file systems.

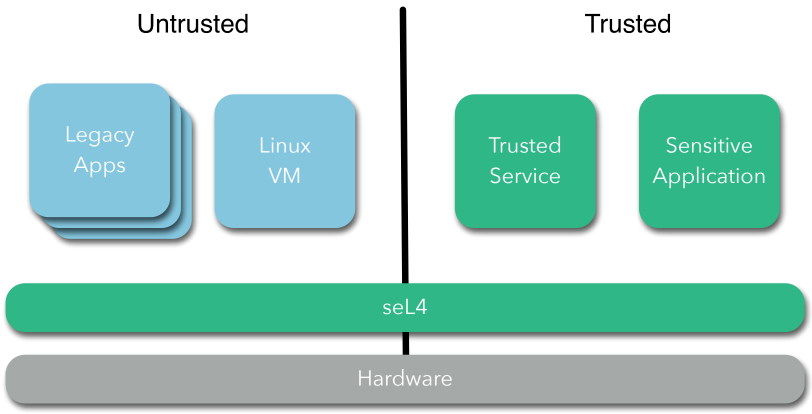


Figure 1. seL4 provides isolation.

2. seL4 Retrospective

The seL4 kernel [5] is a 3rd-generation microkernel [25] of the L4 [26] microkernel family. It is a general purpose microkernel, suitable for a wide range of applications. This is intentional. Instead of restricting the application to a basic separation kernel, a pure hypervisor, a security kernel, a server platform, or an aviation real-time kernel, we chose to maximise return of investment for the formal verification by designing a kernel that is general enough to be configured for all of these scenarios, without having to undergo re-verification.

For this reason, seL4 provides basic OS *mechanisms* such as interrupt delivery, threads, message passing, address space separation, and access control, instead of *policy*, which shapes what a specific system and application is allowed to do, how much memory is being used, how resources are separated from each other or not, and whether the system is static or dynamic. The mechanisms are formally verified once and for all, and policy can be configured. Higher-level theorems such as separation theorems, integrity, and confidentiality will have well-formedness assumptions about the policy in the system, but they apply to wide classes of policies and therefore to wide classes of systems. Other theorems, such as functional correctness apply to all policies, and therefore to all user-level systems that are run on the kernel.

Recent additions to the feature set of seL4 include real-time scheduling and direct support for hardware-virtualisation extensions such as Intel VT-x and VT-d and ARM hypervisor extensions and ARM SystemMMUs. The former simplify building hard-realtime systems on seL4 and enable logical temporal isolation in mixed-criticality systems, the latter simplify the use of seL4 as a type-1 hypervisor running unmodified guest operating systems.

The kernel is available on a number of ARM platforms (ARMv6, ARMv7, ARMv7a, and ARMv8), as well as Intel x86 and x64, but also on RISC-V. The verified platform has advanced from ARMv6 to ARMv7, with ARMv7a and Intel x64 close to completion. The kernel is available as open-source software, including its proofs and specifications [5].

Figure 1 shows one of the main application scenarios of seL4: providing isolation between trusted and untrusted components of a system, between critical and uncritical functionality. The idea is to reduce the trusted computing base (TCB) to the truly critical components of a system, while at the same time allowing rich functionality by running legacy guest operating systems and other large components next to the critical functionality. The seL4 kernel safe-guards any access to sensitive hardware resources and provides strong separation between components with configurable and access-controlled communication channels between them. These channels can be implemented by synchronous message passing, shared memory, or semaphore-like notifications. Authority to resources can be delegated to subsystems for dynamic use, can be

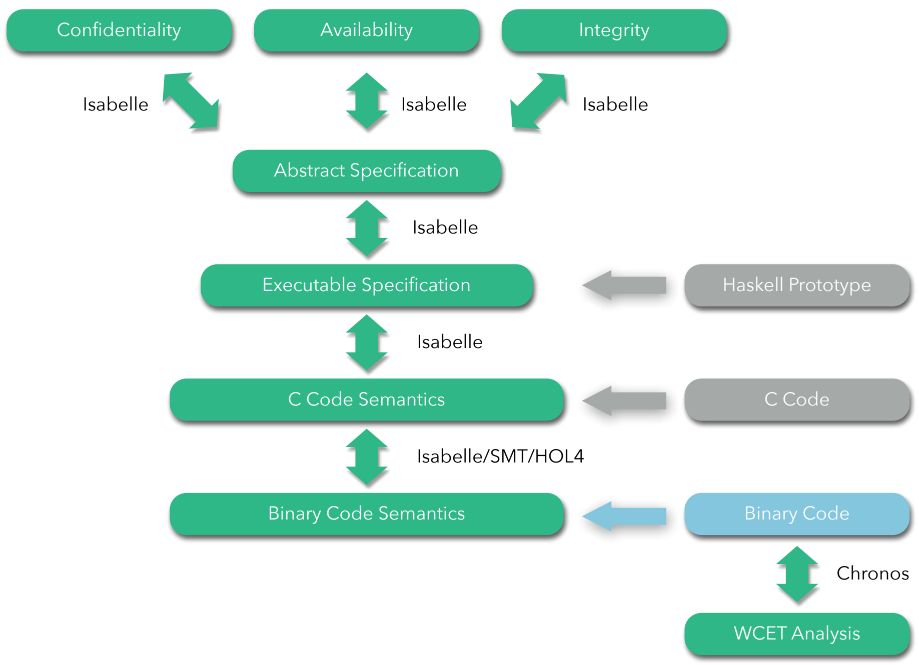


Figure 2. Simplified seL4 verification artefact overview.

managed from a supervisory component, or can be statically fixed at system initialisation time. As we will see later, the flexibility of seL4's general-purpose mechanisms allows us to employ it in securing a rich set of systems and applications without further formal verification work.

(a) The seL4 Verification

The main feature of seL4 is the degree to which we have formally analysed its behaviour. Historically, the initial formal verification [3] was a *functional correctness* proof between the C code of the kernel and its abstract formal specification in Isabelle/HOL. Functional correctness means that every behaviour of the C code is correct according to the abstract formal specification, i.e., the C code is a formal *refinement* of the abstract specification. In the following years, this analysis has both broadened and deepened. Figure 2 gives an overview. Earlier work [27] describes and analyses the verification process in detail.

The figure shows the main artefacts of the verification chain of seL4. To the right, we have from the bottom, the worst-case execution time analysis [28–31], which is performed on the binary of seL4 in a modified and enriched version of the Chronos tool [32]. The binary code of seL4 is the only code artefact that is part of the verification trust chain. Even though the C code and a high-level Haskell implementation of the kernel are imported into the theorem prover, neither of these nor their translation into the prover are required to be correct. The formal property is about the semantics of the binary [33], and the only trusted import step is the one from the real binary into the representation of the binary in the theorem prover. This eliminates the need to trust the compiler as well as the question whether the semantics of the C subset that we use in the seL4 verification is consistent with the C standard (seL4 uses C99). While it is still desirable for these to align, it is not necessary for correctness: if the compiler has a defect, and that defect is modelled accordingly in the semantics, the automated validation pass between C and binary semantics will

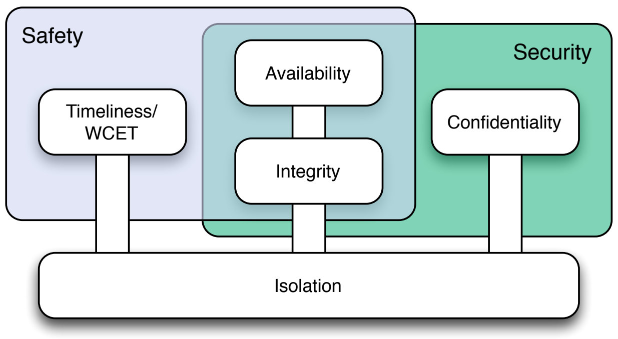


Figure 3. The relationship of OS properties to safety and security.

pass, and the rest of the proof will correctly take the non-standard behaviour into account and show that it still satisfies the specification.[1]

The binary validation process to C is automatic and relies on a set of tools, including Isabelle/HOL, HOL4, Z3 [34], and Sonolar [35]. From the C code up, we prove formal refinement between C code semantics and design-level executable specification, and between executable and abstract specification. At this stage we have functional correctness to a very deep level of detail. The intent of the abstract specification is to cover all user-visible behaviour, including error behaviour. This proof alone already yields many desirable properties such as termination, absence of undefined behaviour, absence of buffer overflows, and absence of code injection. However, functional correctness alone does not yet tell us that the specified behaviour is also the intended behaviour, that it is able to enforce higher-level security properties, and that there is no way to circumvent these properties by clever, unexpected use of the kernel API.

These are properties we prove in addition on the abstract specification. In Figure 2, this is indicated by the boxes Confidentiality (in this case intransitive non-interference [36]), Integrity [37], and Availability [37]. Integrity is the property that state cannot change without write-authorisation, non-interference is the property that says a separated component cannot tell the difference between different behaviours of other components, i.e. cannot *read* or even *infer* (via observations made on storage channels) spatial (memory) information without authorisation, and availability means that resources cannot be taken away without authorisation. Spatial availability is implied by integrity and authority confinement, the latter of which is also a (verified) precondition of our integrity property. Temporal availability is implied by worst-case execution time analysis on the kernel level and by either separation in the sense of non-interference or by real-time schedule enforcement for the system level.

While some properties, like worst case execution time, can only be reasoned about over the binary, we note that the security properties above were much easier and faster to prove on the abstract specification than by proving them directly on the code level. Functional correctness is the theorem that allows us to transfer abstract properties to binary-level properties[2] and thereby allows us to significantly reduce the amount of work necessary for future proofs. This is a general observation that also applies to other classes of code, in particular to generated code with generated proofs. The real power of a generated functional correctness proof is the effort reduction in reasoning at a higher level.

(b) Understanding the Verified Properties

To gain an understanding what these properties are useful for, Figure 3 shows the relationship between them. Availability and integrity are base-level properties that are important for both safety and security. Without integrity (i.e. with the possibility of arbitrary data changes), it is impossible to predict the behaviour of a system, and without availability of resources, the system will not do anything. Confidentiality is most important in the security domain, which attempts to prevent information leaks, and where it is sometimes even safe to stop execution to prevent such a leak. Timeliness (e.g. worst-case execution time) is a property that is most important in the safety space. If the flight-control system of a helicopter provides its output too late, safe operation cannot be guaranteed. All four properties together give us the full isolation that seL4 can provide.

Any proof about real-world systems must make fundamental assumptions. The seL4 proofs are no different. The model of execution makes a number of implicit assumptions, which we detail below, and the security proofs make additional explicit formal assumptions about the access control setup of the kernel. The latter are simple restrictions that can be checked explicitly in the theorem prover for each concrete system. The former are more difficult to pin down. The main implicit assumption of the functional correctness proof, and thereby of all seL4 proofs, is that the model of binary execution accurately reflects hardware behaviour. The model does not include details about execution time (as mentioned, WCET analysis is performed in a separate analysis), it does not contain the TLB and caches, it does not model the initialisation of the kernel and the boot process of the machine, and it contains only a very high-level abstraction of the context and mode switch, as well as of the devices the kernel accesses, in particular the interrupt controller and timer. Other research has shown that this list can be reduced further, e.g. by reasoning about devices more explicitly [38,39] or by modelling context switching in more detail [40]. The absence of time in the model is important for information flow: the proof does not show absence of timing side channels. Another implicit assumption is the soundness of the verification tool chain.

It is easy to think that traditional testing does not make such assumptions. This would be wrong. Although tests are usually conducted on the binary code, the test harness is rarely the deployment platform, and the main assumption of tests is that the test vectors are representative of real inputs under real operating conditions. In particular for operating systems code, this is hard to achieve. For instance, to observe effects of TLB misbehaviour, it would not be enough to achieve the highest level of code coverage that aviation certification such as DO178-C demands for Level A code: coverage of all decisions on all code paths. The misbehaviour would be caused by wrong data written to page tables, not by taking a particular path in the control flow. Testing is necessary to validate model assumptions and confirm user expectations with respect to specifications, but it is not free of assumptions itself.

(c) Verified Systems

So far, we have looked at the formal properties of seL4 itself. To lift these properties to entire systems running on top of the kernel, we need to configure it such that it satisfies the policy preconditions of the security theorems and implements the desired isolation and communication setup of the system. We cover these in more detail in separate work on system initialisation [4,41, 42] and the verified CAmkES component platform [43,44] for seL4.

An example of a setup where seL4 is used to provide security properties for the entire system is the mission computer, or vehicle-specific module, of Boeing's Unmanned Little Bird (ULB), an autonomous optionally piloted helicopter. The helicopter, which performed successful test flights on seL4, is used in the DARPA HACMS (High Assurance Cyber Military Systems) program as a demonstration platform to show how Formal Methods can be used to secure autonomous vehicles against cyber attack. To provide such properties while also running legacy code and proprietary software, the architecture of the system is divided into critical parts such as flight planning,

[1]Of course, the more common scenario is that the binary validation fails and the defect is corrected or circumvented.

[2]Note that some care needs to be taken when using functional correctness to transfer non-interference properties, like confidentiality; see [36].

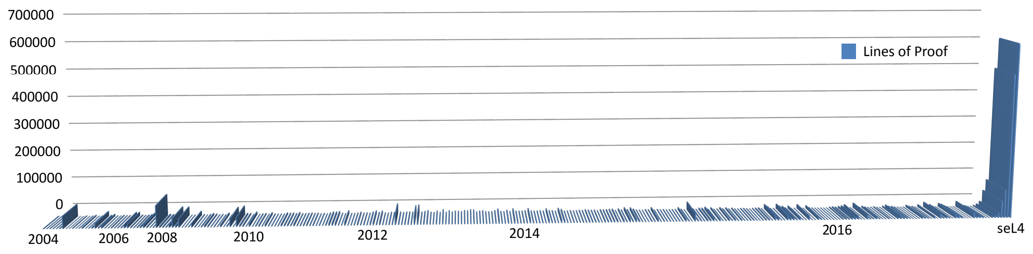


Figure 4. Article size in lines of proof in the AFP by submission date compared to Four-Colour theorem, Odd-Order theorem, Verisoft, seL4 (right-most four).

potential attack vectors such as outside communication, and untrusted software such as Linux virtual machines with proprietary code. The architecture is constructed such that formal analysis and high-assurance software construction can be concentrated on the potential attack vectors, and such that untrusted software does not have enough access control privilege to damage the system even if it behaves fully maliciously. The kernel enforces this access control setup, and the security properties above can be used to reason about the integrity of the trusted critical components. Together with the isolation properties we can make predictions about bounds on the behaviour of the system.

To validate the realism of the assumptions made in building the system, the HACMS program contains a Red Team that conducts penetration tests against the systems built in the program. Even with full root-level access to the Linux virtual machine, simulating a previous successful penetration through an external attack vector on legacy software, the Red Team was not able to break out of the isolation and adversely affect the behaviour of the system. The same Red Team performed successful attacks against baseline versions of the test vehicles in the program. This does not prove security, of course, but it gives indication that the scenario is realistic.

(d) Scale and Complexity

With such success, what is left to be done? The main challenge for formal verification remains scale and complexity. Figure 4 gives an impression of the scale of the seL4 proofs (including user-level components) compared to other proof developments in the Archive of Formal Proofs (AFP) [45], an archival library of Isabelle/HOL developments.

While the entire archive currently contains over 1.4 million lines of proof script, the average size of an entry in the archive is only about 4,700 lines. In comparison, the entirety of all seL4 proofs consist of over 600,000 lines of proof script, almost half the size of the entire archive and two orders of magnitude larger than an average entry. While the level of depth and breadth of the seL4 verification is extensive, seL4 is with about 10,000 lines of C code still not a very large code artefact. The graph in Figure 4 contains the Verisoft project [46,47], which also worked on operating system kernel verification and reached a similar number of lines of proof. This means the size of the seL4 proofs is not especially excessive. Even large mathematical theorems such as the Four Colour theorem and the Odd Order theorem (also shown in the figure) are smaller than these software verifications, although they reach a similar order of magnitude.

To make formal verification more routine, these numbers need to shrink and at the same time need to require less human effort. In previous work we have analysed basic proof engineering aspects that could be improved [48], and increased the automation that is available for code verification [49,50]. The following two sections investigate the issue from two further angles: predicting cost and effort, so proof projects can be managed effectively, and drastically reducing verification effort by generating proofs automatically — thereby enabling us to reason about higher levels of abstraction.

3. Proof Estimation

Before formal software verification can truly enter the mainstream we need better predictive models for its required cost, duration and effort. "How much and how long?" is a question that all software engineering project leaders must be able to answer, that all software companies must be able to provide to customers. If we want verified software to become the default in specific application areas, we need to provide project leaders with means to produce quotes on verification, to put a price on a proof, to commit to a delivery date in a contract and to estimate the resources needed for the project.

Despite the recent successes in formal verification, there has been little effort in gathering the empirical data required to build such models. Additionally, significant differences between verification frameworks make it difficult to directly compare the resulting artefacts. Finally, understanding the cost and effort required to produce a verified artefact is only the beginning. Like all software, verified code needs to be updated to satisfy changing requirements, and we need to understand how difficult it will be to adapt the corresponding proofs.

To begin answering these questions, we performed several in-depth investigations into the seL4 proofs. As mentioned, proof maintenance is an important issue, but to start with, we focus on the initial development of proofs only. The initial verification took approximately 20 person-years to complete over 4 years and produced over 200,000 lines of Isabelle proof script. Today the seL4 proof has grown to over 600,000 lines of proof and now includes proofs of confidentiality and authority confinement; it is continually updated to integrate new seL4 features and support new hardware platforms. Together, all the seL4 proofs constitute a rare and large data-set of verification artefacts, with detailed, version-controlled information about their evolution, maintenance, contributors, etc.

Through a retrospective analysis, we sought to identify *leading indicators* of verification effort and cost. A software verification project will iterate through several phases, ranging from expressing the requirements, formalising one or more levels of specification, developing the implementation, and proving that the implementation satisfies its specification. Each of these phases may lead to changes and require re-work in other ones: the proof may uncover a flaw in the requirements, or the implementation a flaw in the specification. At the start of the project, and often at each of these stages, the project leader needs to estimate the remaining cost required to complete the verification. We analysed the existing seL4 specification, code, and proofs to understand the impact of each artefact on the total effort of the project.

Before we describe these analyses in more detail, we should acknowledge that the overall problem is clearly not solvable in general. There is no reliable function that takes a set of requirements and predicts the overall duration and cost for a proof project. Many mathematical theorems look deceptively straightforward: e.g., Fermat's Last Theorem, the Kepler Conjecture, or the Four Colour Theorem. These are all simple to to state and yet took decades or even centuries to prove. Such problems can be encoded into programs and would have their required cost significantly under-estimated by any predictive mechanism. What we *can* strive towards, is a predictor that is applicable to certain classes of programs: with specifications that are not mathematically complex and for which historical data of similar verified programs exists. This data will become available as the successes of software verification continue. In this respect estimating proof effort is no different than estimating implementation effort: incomplete or uncertain predictions are better than none at all, and novel projects are at greater risk of having their costs underestimated.

With that in mind, we performed three investigations, relating specification size to code size, then proof size to effort, and finally specification size to proof size.

Specification vs. code: linear. The first investigation [51] revealed a linear correlation between the number of lines of Isabelle required to specify each seL4 API call, and the number of lines of C in its implementation. We additionally calculated the COSMIC function point (CFP) [52] count for each API call and compared it against its implementation size, but this turned out to correlate only poorly. Both analyses were done by "slicing" the seL4 implementation for each API call, only keeping code along the call path for that call.

The left-hand side of Figure 5 shows the relationship between CFP count for each API function and the size of its implementation, measured in normalised line counts. The COSMIC Function Point method is an internationally standardised way of sizing software based on its requirements. CFP counts are notionally performed on a Functional User Requirements (FUR) document. There is no FUR for seL4, and so instead we used the seL4 user manual, which has a similar target audience (users) and level of abstraction as a FUR. The analysis shows a poor correlation between CFP count and code size in the case of seL4. We also compared CFP count to specification size for the two specifications (abstract and executable), leading to similar results.

In contrast, the right-hand side of Figure 5 shows the scatterplot relationships between the size of the abstract specification, the executable specification and the implementation, for each API function. These do reveal a strong linear relationship between each pair.

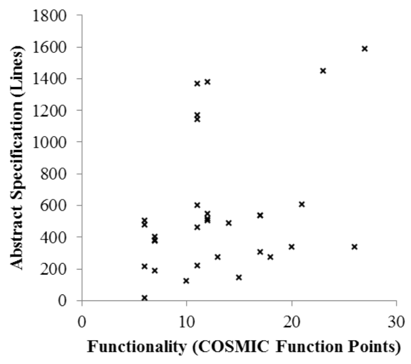
Formal specifications would typically be defined later than FURs, because they are much more precise. Nonetheless they are ideally defined relatively early in the project lifecycle, and so appear to be a promising target for code size estimation. Seen from another angle, these results also indicate that estimation on the verification effort could equally be based on the specification or the implementation. When the specification is indeed produced first (as it is the case for seL4, where the earliest artefact produced was the executable specification, then the abstract specification and finally the implementation), then it could be used as a base to estimate both code size and proof effort (see further investigations below). When the code happens to be produced first, then it could be used to estimate specification size, which in turns would lead to proof effort. Note the use of conditional: our analyses only reveal relationships on existing artefacts; deriving a predictive model is still an open question, as we will discuss below.

The linear relationship between code and specification size might appear worrisome: large code will have large specs. Note that this result is for specifications of functional behaviour. To gain confidence in large specifications of this nature, we should prove higher-level properties about them, such as integrity and confidentiality in seL4, which have much smaller, and more standardised descriptions.

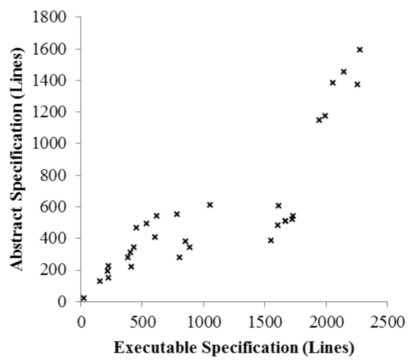
Proof vs. effort: linear. Our second investigation [53] showed another linear correlation, between the size of a proof, measured in lines of Isabelle, and the total effort required to write it (in person weeks). We divided the seL4 proof into 6 distinct proof artefacts, and compared the total effort and final size for each. We included 3 major specifications, also measured in lines of Isabelle, in the analysis. Additionally, historical data was taken from the repository where proof *change* size is correlated with self-reported effort. Both of these results showed strong linear correlations, shown in Figure 6, although with significantly different coefficients.

Specification vs. Proof: quadratic. Finally, our third investigation [54] showed a quadratic correlation between the size of a lemma statement and the total number of lines required to prove it. Here lemma statement size was measured as the total number of unique definitions in its statement. We investigated four sub-projects from seL4: the abstract invariants, the Haskell-to-abstract refinement, authority confinement, and confidentiality. Additionally, we included two proofs from Isabelle's Archive of Formal Proofs (AFP) [45] in the analysis: JinjaThreads [55] and SATSolverVerification [56]. We consider the set of lemmas that comprise each project, comparing the statement size of each lemma to its proof size. Within each project we observed a consistent quadratic relationship, but with a wide range of coefficients.

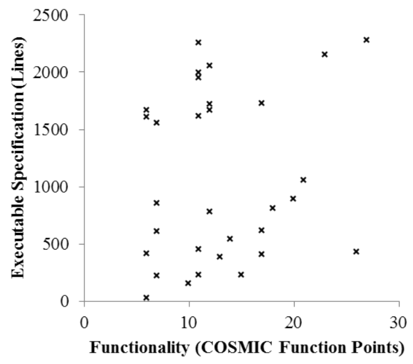
An important caveat to this result is that, in most cases, this correlation is much weaker unless lemma statement size is *idealised*. Often lemmas are stated in more detail than is formally required: concrete terms are given where parameters would suffice. This is done both for clarification and to facilitate automated reasoning. Such over-specified lemmas have larger statement sizes than their proofs would suggest, obscuring the observed relationship. To mitigate this, we discount



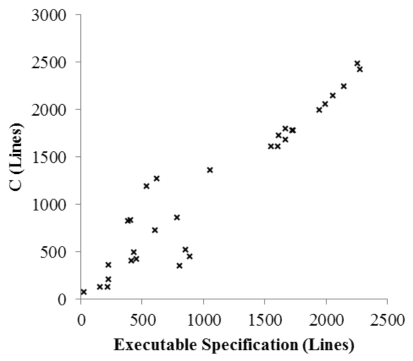
(a) Functionality vs. Abstract Spec.
(rho = 0.31, $p = 0.09$)



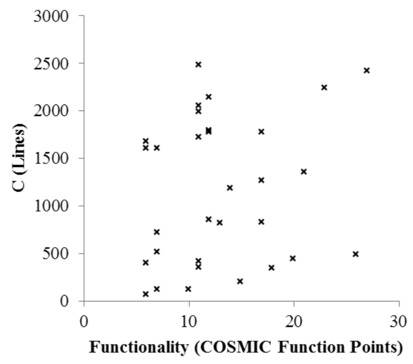
(b) Executable Spec. vs. Abstract Spec.
(rho = 0.89, $p = 0.00$)



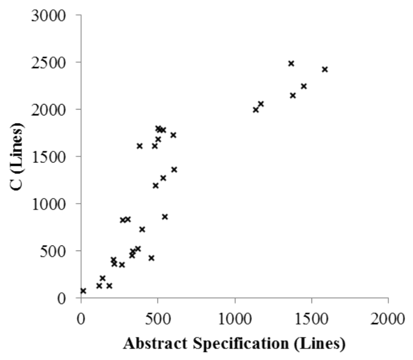
(c) Functionality vs. Executable Spec.
($R = 0.15$, $p = 0.42$)



(d) Executable Spec. vs. C
($R = 0.95$, $p = 0.00$)

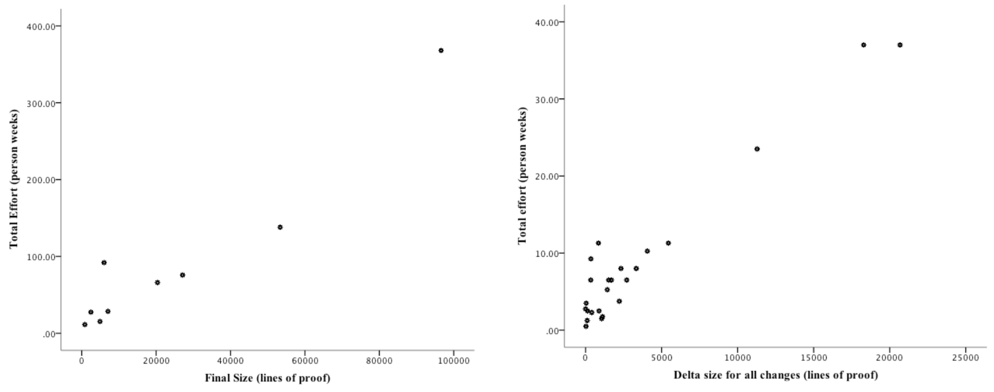


(e) Functionality vs. C
($R = 0.19$, $p = 0.32$)



(f) Abstract Spec. vs. C
(rho = 0.92, $p = 0.00$)

Figure 5. COSMIC Function Point versus Artefact Size (left) and Artefact Size Relationships (right). Each point is an seL4 API function.



(a) Scatter plot of project total effort in person weeks vs. final size in lines of proof (R^2=0.914, p<.001).

(b) Scatter plot of total effort (person weeks) vs. sum of repo-delta size (lines of proof) of all changes made in the 24 individual contributions to the four proofs and the specification (R^2=0.93, p<0.001).

Figure 6. Proof size vs. effort.

definitions from lemma statements which never appear in their proof. In other words, the idealised size of a lemma statement is the size it would have had if it were written in its most general form. We argue that this idealisation is sound because it could be approximated by the application of domain-specific knowledge; a proof engineer seeking to apply a predictive model to her specification would first prune any obviously unrelated definitions. In Figure 7 we show the results of a quadratic regression of this idealised statement size against proof size for each of the 6 projects.

Using the same approach, a more in-depth study of the entire AFP by Blanchette et al. [57] showed, among other results, that this quadratic correlation only holds (with $R^2 > 0.7$) for $\sim$50 of the $\sim$200 articles examined. However, the best fitting articles were most significantly represented by those related to software verification. This gives additional evidence that the observed relationship holds for a larger class of software verification proofs, although not formal proofs in general.

Discussion. Put together, these results seem to (transitively) indicate a quadratic correlation between the size of the specification and the effort that will be needed to prove that an implementation will satisfy this specification. Such an effort indicator, in the growing world of verified software, is a first; but more work is needed to strengthen this claim, and provide reliable verification project management models and tools. Among the many open questions, the most significant is whether we can build a *predictive* model from these results. We list here a few of the challenges towards this goal.

Firstly, the artefacts measured in our three investigations do not strictly match. This is for both technical reasons and data availability. For instance, slicing the effort per API function (similar to slicing code size in our first investigation) would require much more fine-grained bookkeeping of which precise proofs each engineer was working on at any given time. The data is not available at this granularity for seL4, but it could be tracked on new projects for a more rigorous link between the analyses. Our work helps identifying the kind of data that should be recorded. Another instance is the lemma statement size from the third investigation: additional work is needed to show that it correlates with the specification size as measured in the first.

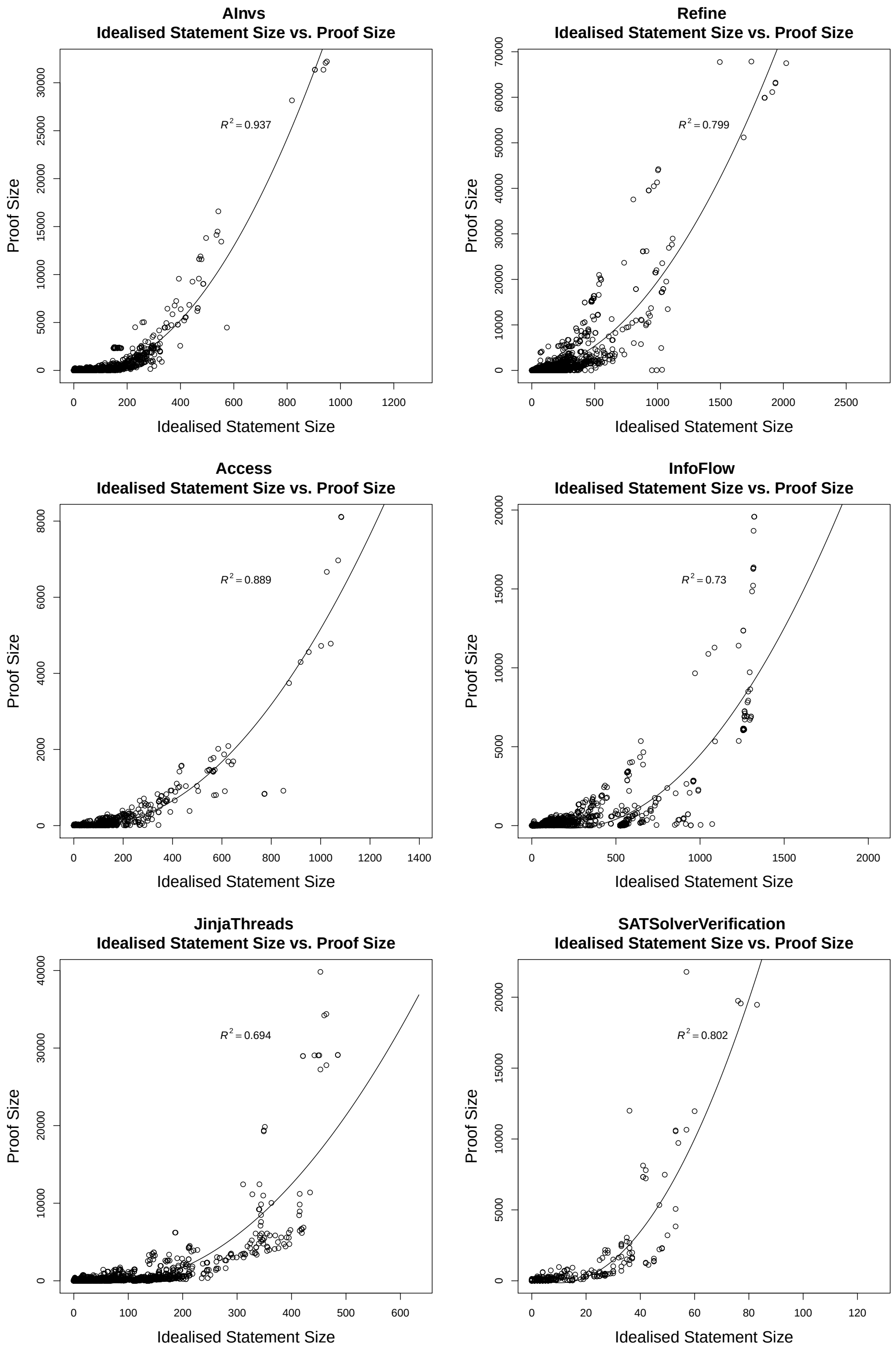


Figure 7. Relation between idealised statement size and proof size for the six projects analysed.

Secondly, although the third investigation demonstrates a correlation between lemma statement and proof size, their precise relationship varies between proofs; the coefficient computed for one proof cannot be immediately used as a predictive model for another. A potential solution would be to base an initial model on a previous project, but refine it iteratively as the proof is developed. Additionally, as more verified software is built, we might build a database of domain-specific models to draw upon.

Thirdly, a major difference between retrospective analyses and predictive model is the existence of the artefacts. Our analyses work on a snapshot of history: they take as input artefacts at a given point in time, typically at the point they are "done for the first time", and before they go into a maintenance mode where they get incrementally updated. And they compare the metrics of various artefacts at this one time. In particular, they assume the size of the "final" specification is fully known when trying to predict the size of its proof. In reality, the development of these artefacts is iterative and interleaved. Once again this indicates that an iterative model would be required to take this process into account.

Related to this, our analyses focused on the process of building verified software from the ground up, with little focus on predicting the effort required to modify existing proofs. Arguably understanding the maintenance cycle of proofs is even more important than understanding their initial development: seL4 took 4 years to develop and verify but has been maintained for over twice that time. Previously [3] we noted that, during the original verification of seL4, a change to less than 5 % of the code resulted in 17 % of the proof being reworked. However, other changes were relatively isolated and had a less dramatic impact. An in-depth analysis of a completed proof would likely be able to inform such a prediction. This might include, for example, how many lemmas reference a particular function, or how many proofs inspect a particular definition? A change could then be measured by the number of functions/definitions it touches, and then we could compute how pervasive that change is likely to be throughout the proof.

Finally, our investigations mainly focused on seL4, with some analysis of two developments from Isabelle's AFP. Although other work by Blanchette et al. supports the quadratic correlation between specification and proof size in the context of software verification, the other two relationships have little empirical evidence beyond seL4.

To further advance the field of understanding and predicting proof effort we propose that several investigations need to occur: an analysis of multiple large-scale verification projects, an experiment in building a predictive model in the early stages of a project, and a subsequent refinement of that model into an iterative one that incorporates proof updates and maintenance.

Although we have made good initial steps towards providing methods and tools for cost and effort estimation of verified software, we have also exposed a number of areas of future work required to strengthen their rigour, practicality and reliability. Our goal is for both new and ongoing verification projects to perform similar investigations, ultimately contributing towards building practical predictive models of developing verified software.

4. Code/Proof Co-Generation

Even as formal verification starts to become a more predictable activity, its cost still remains relatively high. Much of this cost arises due to the sheer volume of manual work required to-date to verify critical systems software like seL4. For example, the cumulative size of the seL4 proofs of functional correctness and security now exceeds 600,000 source lines of proof (see Section 2), which as we saw in Section 3 is directly correlated to the cumulative effort required to produce them (now in excess of 20 person years). Further, assuming that the level of effort required to verify a piece of software grows *quadratically* in response to the software's size, as suggested by the quadratic relationship between statement-size and proof-size mentioned in Section 3, there is clearly a limit to how far one can scale this style of manually intensive software verification.

A new approach is needed, therefore, if we are to scale software verification to entire critical systems. In this section, we report on recent work that suggests how to radically reduce the effort required to verify well-architected systems code. This approach is based on a simple realisation:

verifying a piece of code will always be more expensive than writing it, and at present is around half an order of magnitude more so. This suggests that a promising way to improve verification productivity is to invest more effort into *writing* code such that the verification burden is reduced later. For instance, use of an unfamiliar language in implementation can be outweighed by far by a significant reduction in formal verification cost later.

To make this effort-shift, we carefully designed a new programming language called Cogent for writing verified systems code [58]. Figure 8 depicts the Cogent approach. Cogent compiles to efficient C code, which easily integrates with existing systems APIs and abstract data types, while being designed as a purely functional language, making it very friendly to reason about formally. This makes proving properties of programs written in Cogent far easier than for traditional C code, because the level of abstraction is much higher. Additionally, Cogent's compiler *automatically* proves that the C code it produces is a correct compilation of the functional source program [59]. Therefore, Cogent facilitates obtaining verification guarantees on par with the seL4 approach by mere equational reasoning over pure functions, without unduly sacrificing performance — the remainder of the reasoning is entirely automatic. A key ingredient for achieving this double-act is Cogent's *linear* type system, which allows pure Cogent functions to be compiled to efficient C code without the need for a garbage collector [60] on one hand, while still providing the abstraction necessary to facilitate verification.

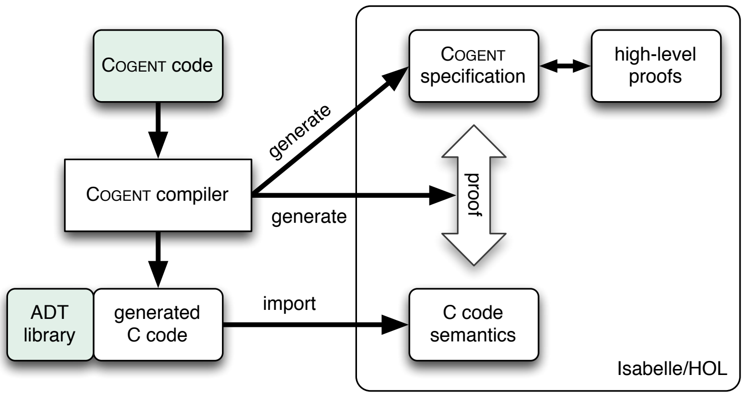


Figure 8. Code/Proof Co-Generation with Cogent.

To demonstrate Cogent's utility for producing efficient, verified systems software at much lower cost, we used it to implement two file systems, ext2fs and a custom flash file system called BilbyFs. For the latter, we then proved the functional correctness of two key file system operations. Our results indicate that Cogent allows verification effort to be meaningfully reduced (by around a third to a half) without unfairly trading away performance. We chose file systems as our case study since they are common but critical systems components of non-trivial complexity, whose functional correctness and performance is relied upon by the remainder of the operating system as well as all applications, but which in practice have been shown to be relatively error-prone [61]. Thus they present an ideal candidate for implementation and verification via Cogent.

Languages like Rust [62], Cyclone [63] and Habit [64] are examples of other languages based on the idea that more abstraction and more static checks through more expressive type systems in systems languages will result in safer, less buggy code. Cyclone does have a formal semantics, and one of the aims of the Rustbelt project is to produce such a semantics for Rust. However, none these exisiting more abstract systems languages provide a certified compilation or proof co-generation.

(a) Language Restrictions

Perhaps most important about Cogent's design is what it leaves out. We explicitly chose to exclude from Cogent's design certain features of mainstream systems languages, in order to favour verification productivity over expressiveness and to ease the job of the compiler to automatically generate correspondence proofs for the C code it emits. These restrictions include the absence of recursion or built-in iteration constructs, as well as the fundamental restriction imposed by Cogent's linear type system that all linearly typed variables be used exactly once in the program. Avoiding iteration eases the automatic C correspondence proofs, although we expect we can lift this restriction in the future to allow the addition of well-behaved iteration constructs to the language. As mentioned earlier, Cogent's linear type system and the restrictions it imposes are fundamental to allowing the programmer to work at a sufficiently high level of abstraction to ease proving properties of Cogent programs, while simultaneously allowing Cogent code to be compiled to efficient C.

Cogent provides a useful escape hatch from these restrictions, however, in the form of a C foreign function interface that allows Cogent programs to call functions implemented directly in C. Thus Cogent programs are written against a library of C abstract data types (ADTs), including appropriate iterators for collections and compound structures implemented as ADTs. Our experience writing the two file systems in Cogent has shown that this design is a sound trade-off: Cogent's existing C ADT library is reused between the two file systems, and is sufficiently general to be used by other file systems, such that we can amortise the cost of verifying these components separately over many different file systems.

The uniqueness restriction imposed by linear typing, requiring that all linearly typed variables be used exactly once in the program in order to ensure that no aliases can ever exist, is known to be too onerous for writing general-purpose code, and some data structures cannot be expressed at all. However, we found that it is manageable for the domain of well modularised file system implementations, given a suitable ADT library against which to program. Modularity is key here, as linear types purposefully restrict the kind of aliasing common in highly-coupled software. However, they also make many common data structures that naturally involve aliasing (e.g. cyclic graphs) impossible to encode in Cogent, and therefore they must be implemented directly in C. A benefit of linear types in the context of Cogent is that they can be used to thread state through the program in a very fine grained manner, which simplifies verification.

(b) A Taste of Cogent

To give a feel for what it is like to program in Cogent, Figure 9 depicts a snippet of Cogent from our ext2fs implementation, specifically the `ext2_inode_get()` function. This function looks up an inode on disk, given its inode number `inum`. Line 3 declares the type of this function. It takes 3 arguments: an `ExState`, which is a reference to an ADT embodying the outside environment of the file system; an `FsState`, a reference to a Cogent structure that holds the file system state, and `U32`, an unsigned 32-bit integer that in this case is the number of the inode to be looked up. It returns a result of type `RR` (defined on line 1), which is a pair. Its first component `c` is mandatory data that must always be returned, while the second component is a *tagged union* value which signals whether the function succeeded (`Success a`) or not (`Error b`), where `a` and `b` are result values for the respective cases. `ext2_inode_get()` (line 3) always returns both the `ExState` and `FsState`; when successful it also returns a `VfsInode` reference to the looked-up inode; otherwise it instead also returns a `U32` error code.

`ext2_inode_get()` first calls (line 6–7) the Cogent function `ext2_inode_get_buf()`, which, after more calculation, internally calls an ADT function to read the corresponding block from disk. We match (lines 8, 9, 22) on the result `res` to make a case distinction: for `Success` (line 9), the result contains a reference `buf_blk` to a buffer, plus the offset `offset` of the inode in that buffer. We use these to deserialise the inode from the buffer into a `VfsInode` structure.

```
1  type RR c a b = (c, <Success a | Error b>)
2
3  ext2_inode_get : (ExState, FsState, U32) -> RR (ExState, FsState) (VfsInode) (U32)
4
5  ext2_inode_get (ex, state, inum) =
6      let ((ex, state), res) =
7        ext2_inode_get_buf (ex, state, inum)
8      in res
9      | Success (buf_blk, offset) ->
10         -- read the inode, from the offset
11         let ((ex, state), res) =
12             deserialise_Inode (ex, state, buf_blk, offset, inum) !buf_blk
13         in res
14         | Success inode ->
15             let ex =
16               osbuffer_destroy (ex, buf_blk)
17             in ((ex, state), Success inode)
18         | Error () ->
19             let ex =
20               osbuffer_destroy (ex, buf_blk)
21             in ((ex, state), Error eIO)
22     | Error (err) -> ((ex, state), Error err)
23
24 osbuffer_destroy : (ExState, OsBuffer) -> ExState
```

Figure 9. Looking up an inode in ext2fs, in Cogent.

In this situation, simple linear types are unnecessarily restrictive. Passing the linear value `buf_blk` to `deserialise_Inode()` in line 12 would consume it, so `deserialise_Inode()` would have to return it as an additional result for it to be used again later on in the program. This is rather onerous, given that `deserialise_Inode()` never modifies the buffer. The "!" operator (in "`!buf_blk`", line 12) allows us to temporarily escape the use-once restriction of the linear type system: it indicates that within the right-hand side of = here, `buf_blk` is used read-only and can therefore be referenced multiple times without consuming it. The type system prevents `deserialise_Inode()` from modifying the buffer that is holding the inode being deserialised. Moreover, it ensures via a simple type-based alias analysis that no references to buffer or parts of it can be returned, as this would result in aliasing. Current development efforts in Cogent are focused on making this alias analysis more fine-grained.

For the `Error` case (line 22), the result includes an error code that is simply propagated to the caller along with the return values that were mandatory. The `Success`/`Error` case distinction is repeated following the attempt to deserialise the inode from the buffer. In either case, the buffer must be released, by calling the ADT function `osbuffer_destroy()`, whose type signature (line 24) suggests it consumes the buffer (since the buffer is absent in the return type).

Note that in this function, Cogent's linear type system would flag an error if the buffer `buf_blk` was never released. `ext2_inode_get()` will only type-check if all of the `Error` cases are handled. In this way, Cogent's type system helps to catch many common forms of errors in systems code.

Also observe that the mandatory `ExState` and `FsState` values are threaded through the function. The `ExState` value encompasses all external state, and so includes for instance Linux's buffer cache. Because Cogent is a purely functional language, it makes side effects explicit. This means this `ExState` is passed to `osbuffer_destroy()` explicitly, which then returns a new `ExState` reference that encompasses the external state of the world after the buffer has been released. This does not mean that large structures are copied at run-time. In practice, Cogent's linear type system allows the `ExState` update in `osbuffer_destroy()` to be performed in-place, as it is not possible to copy the state. The same is true for the other functions that each consume and return an `ExState` or `FsState`.

It is obvious from Figure 9 that Cogent's current surface syntax for error handling is unnecessarily verbose. The textual overhead for the regular error handling pattern can easily be reduced by additional syntactic sugar in a future language iteration, as alluded to above, without touching the verification infrastructure.

(c) Verification Effort

The Cogent language is by far not unique using programming language technologies and linear types in a systems context—for instance the Singularity OS [65] used both before. What is unique is Cogent's application of these techniques to increase formal verification productivity.

We refer the interested reader to existing publications [59,60] for a detailed explanation how the Cogent compiler generates the automatic proofs of correspondence between the C code it emits and the pure, functional Isabelle/HOL shallow embedding of the Cogent source program also generated by the compiler.

In this section, we concentrate on explaining how and why further reasoning over this shallow embedding is simpler than writing traditional functional correctness proofs of systems code, such as those for seL4. To do so, we draw on our experience proving functional correctness for two of the key file system operations of BilbyFs, our custom flash file system that we implemented in Cogent. Similar to seL4's functional correctness proofs, we prove functional correctness of these operations against a top-level *abstract* specification of their correct behaviour [66] phrased as a nondeterministic functional program in Isabelle/HOL.

Just as with seL4, the functional correctness proofs for BilbyFs relied on establishing global invariants of the abstract specification and its implementation. For BilbyFs, the invariants include e.g. the absence of link cycles, dangling links and the correctness of link counts, as well as the consistency of information that is duplicated in the file system for efficiency.

Importantly, unlike with seL4, none of the invariants had to include that in-memory objects do not overlap, or that object-pointers are correctly aligned and do point to valid objects. All of these details are handled automatically by Cogent's type system and are justified by the C code correctness proofs generated by the Cogent compiler. Thus Cogent considerably raises the level of abstraction for proving properties of systems programs.

Even better, when proving that the file system correctly maintains its invariants, we did so by reasoning over pure, functional embeddings of the Cogent code that the compiler has already automatically proved are correctly implemented by the generated C code. Because they are pure functions, these embeddings do not deal with mutable state (as e.g., the seL4 ones do). Thus there is no need to resort to cumbersome machinery like separation logic [67] to reason about them: each function can be reasoned about by simply unfolding its definition, and the presence of separation between objects x and y follows trivially from x and y being separate variables.

One way to attempt to quantify the verification effort-reduction afforded by Cogent is to compare the cost of performing the BilbyFs functional correctness proofs, per line of Cogent code verified, to that for the seL4 proofs, per line of C code verified. We performed such a comparison in [58], discovering that roughly 1.65 person months per 100 C source lines in seL4 are reduced to $\approx$0.69 person months per 100 Cogent source lines in BilbyFs.

We should note however that this comparison is between a partial proof of functional correctness (for BilbyFs) to a completed one (for seL4). It is possible that, were one to complete the BilbyFs proof, additional file system invariants would be discovered that would need to be proved, even for the two file system operations of BilbyFs that have already been verified. In this case, the effort per line of Cogent would necessarily increase. We believe that all such invariants have already been discovered, but the only way to be sure is to complete the functional correctness proof for the remaining operations of BilbyFs. Nevertheless, we believe it is clear that Cogent affords a meaningful reduction in verification effort by about one third to a half.

Application	C	Cogent
`git clone golang`	84 sec	88 sec
`make filebench`	48 sec	50 sec
`tar`	80 sec	96 sec
`du`	30 sec	31 sec

Figure 10. Running time for applications on top of BilbyFs

(d) Performance

In previous work [58], we conducted a detailed performance analysis of the two Cogent file system implementations, using various file system micro-benchmarks. We also looked at macro-benchmarks [60], and summarise the high-level trends here.

Overall, the Cogent file system implementations perform within 10-20% of their native counterparts. (For BilbyFs, which is a custom file system, we manually wrote a native C implementation which we compared the performance of its Cogent implementation against.) These performance overheads are due to the fact that the Cogent implementations tend to use data structures that strongly resemble those of the original C, rather than more idiomatic functional data structures, which are handled better by our compiler. For some operations, the C implementation would rely on unsafe features for performance, where the Cogent implementation uses slower, but easier-to-verify techniques, which lead to additional copying of data. In most cases, the overhead for these do not lead to a major performance penalty, but there are pathological cases, where copying happens inside a tight loop, which can lead to a slowdown of an order of magnitude for that specific operation. As we will discuss in Section (e), the upcoming version of Cogent addresses this shortcoming.

Occasionally, some overheads are introduced in C code generation: Generated code displays patterns which are uncommon in handwritten C and so might be missed by the C optimiser, even if trivial to optimise. One way to address this would be to generate code directly in the compiler's intermediate language that is consumed by its optimiser, or to explore different backends, as for example LLVM.

Figure 10 shows some application-level microbenchmarks that compare the performance of the Cogent implementation of BilbyFs against a hand-written C implementation of the same file system. The evaluation configuration is a Mirabox [68] with 1 GiB of NAND flash, a Marvell Armada 370 single-core 1.2GHz ARMv7 processor and 1 GiB of DDR3 memory, running Linux kernel 3.17 from the Debian 6a distribution. The first scenario clones the git repository of the language Go [69], creating 580 directories and 5122 files, for a total of 152MB. The next test compiles the Filebench-1.4.9.1 source code [70] with 44 C files for a total of 19K SLOC. The third application extracts a large tarball (an archlinux distribution), and the fourth measures the total size of a large directory structure. The `tar` command has a 20% slowdown, but all other scenarios show the formally verified Cogent version performing within 5% of the unverified native C implementation.

(e) Future work

Cogent is an ongoing project, and we intentionally omitted many features from the first version of the language, as we prioritised implementing the full compiler and verification pipeline over aggressive optimisation, expressivity and user friendliness. This enabled us to conduct the case studies, which allow us to make the decisions on the basis of actual data.

As we discussed in Section (d), we identified some usage patterns which lead to significant performance penalties. This occurs whenever we have to read a large data structure from the storage medium and traverse it in memory. In C, this can easily be done in-buffer, without deserialising and copying the data a second time. The downside is, however, that the resulting code is extremely hard to verify. Addressing this problem was our first priority, as Cogent's linear type system prevents us from using any kind of aliasing, making it impossible to avoid copying. Fortunately, region types, another flavour of substructural types, do exactly allow the type of controlled sharing we need, without losing assurances essential for verification. We are currently in the process of introducing region types to Cogent. A program which is well-typed in the linear type system is still well typed in the region type system, as the linear constraints are strictly stronger. The region information can be inferred and need not be provided by the user.

Writing Cogent programs can be tedious, and while the introduction of region types will improve the situation for the user, there is still much we can do through straightforward means such as adding syntactic sugar or the introduction of specialised language constructs. Consider the `ext2_inode_get` function we discussed above. The code is quite typical for systems code in that there is a considerable amount of error checking and error handling involved. In Cogent, the treatment of values of union type (such as the result type in the example, which has two variants, `Success` and `Error`) has to be exhaustive. From a correctness perspective, this has the advantage that it statically ensures that all errors are handled. However, mostly, the error handling code just passes the error it received from the callee back to the caller after freeing all the memory allocated in the function so far. These default situations could be expressed much more concisely by adding specialised language constructs. Thanks to Cogent's type system, the compiler even knows which objects have to be deallocated before returning, so the default clean-up code could be automatically generated.

Independent of the language, we cannot always avoid serialisation and deserialisation of data. In fact, a significant portion of file systems code is typically concerned with this monotonous and error prone task. Data description languages address exactly this problem by generating the code automatically given specifications of the serialised and deserialised format. Following a similar approach as PADS [71], we are currently working on adding language-level support for data (de-)serialisation to Cogent.

The current version of Cogent already demonstrates the benefits of expressive type systems for automatic verification. This is not surprising. After all, type checking asserts properties of the code at compile time, and the more details are encoded in the type system, the more information we have. Cogent's type system at the moment only scratches the surface of what is possible in this respect, and it will be interesting to investigate this further by extending Cogent's type system to support refinement or even dependent types.

5. Conclusions

We have given a brief overview of more than a decade of work around the formal verification of the seL4 operating system microkernel. It remains to this date one of the largest and most detailed software verifications in the field.

We have argued that this kind of formally verified OS kernel can be directly applied, without further expensive formal verification, to enhance the security and safety of software systems. If we do invest further verification effort, we can reach an even larger class of systems, and have a realistic chance of significantly contributing to the solution of large-scale problems such as the current state of cybersecurity, which is developing into an existential threat to developed society.

To make this contribution, formal verification has to become more manageable and less expensive. We presented two areas of ongoing research that address these: proof effort estimation and code/proof co-generation.

There is much more to be done to reach cost parity with traditionally engineered high-quality, yet low-assurance software, but it can already be said that formally verified software is firmly within reach.

Acknowledgements. This article is a summary of work that has received many contributions. We would in particular like to acknowledge the contributions of Andrew Boyton, David Cock, Callum Bannister, Nelson Billing, Bernard Blackham, Timothy Bourke, Matthew Brassil, Adrian Danis, Matthias Daum, Jeremy Dawson, Philip Derrin, Dhammika Elkaduwe, Kevin Elphinstone, Kai Engelhardt, Matthew Fernandez, Peter Gammie, Xin Gao, David Greenaway, Gernot Heiser, Rafal Kolanski, Ihor Kuz, Corey Lewis, Jia Meng, Catherine Menon, Magnus Myreen, Michael Norrish, Sean Seefried, Thomas Sewell, Yao Shi, David Tsai, Harvey Tuch, Adam Walker, and Simon Winwood to seL4 and the surrounding formal verification. The proof effort estimation project additionally included Ross Jeffery and Mark Staples, and the following people contributed to the Cogent code/proof co-generation work: Joel Beeren, Peter Chubb, Alex Hixon, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Yutaka Nagashima, and Joseph Tuong.

Funding. The following note applies to Section 4 and partially Section 2. This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

Authors' Contributions. Gerwin Klein led the seL4 verification work, founded the proof estimation work and led the verification part of the code/proof co-generation project; June Andronick was involved in the proof engineering estimation work, in particular in collecting and measuring proof effort and data analysis; Gabriele Keller lead the code/proof co-generation project; Daniel Matichuk designed and implemented the spec and proof size metrics, and carried out their data collection in the proof estimation work; Toby Murray implemented and carried out metric evaluation work for proofs and specs and collected and measured proof effort for the proof estimation work, and co-led the verification part of the code/proof co-generation project; Liam O'Connor designed the Cogent programming language and type system in the code/proof co-generation project. All authors have contributed to the writing of this manuscript and all authors gave final approval for publication.

Data Accessibility. The seL4 code, specification and proofs, as well as the Cogent compiler and proofs are available on github [5,72].

Competing Interests. The authors have no competing interests.

References

1. Leroy X. 2009 Formal verification of a realistic compiler.
Communications of the ACM **52**, 107–115.
2. Leroy X. 2009 A formally verified compiler back-end.
Journal of Automated Reasoning **43**, 363–446.
3. Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. 2009 seL4: Formal verification of an OS kernel.
In *ACM Symposium on Operating Systems Principles*, pp. 207–220. Big Sky, MT, USA.
4. Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R, Heiser G. 2014 Comprehensive formal verification of an OS microkernel.
ACM Transactions on Computer Systems **32**, 2:1–2:70.
5. The seL4 microkernel code and proofs.
`https://github.com/seL4/`.
6. Gu R, Shao Z, Chen H, Wu XN, Kim J, Sjöberg V, Costanzo D. 2016 CertiKOS: An extensible architecture for building certified concurrent OS kernels.
In *USENIX Symposium on Operating Systems Design and Implementation*.
7. Kanav S, Lammich P, Popescu A. 2014 A conference management system with verified document confidentiality.
In *International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pp. 167–183.
8. Kumar R, Myreen M, Norrish M, Owens S. 2014 CakeML: A verified implementation of ML.
In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 179–191. San Diego.

9. Davis J, Myreen MO. 2015 The reflective Milawa theorem prover is sound (down to the machine code that runs it).
Journal of Automated Reasoning **55**, 117–183.
10. Kumar R, Arthan R, Myreen MO, Owens S. 2016 Self-formalisation of higher-order logic — semantics, soundness, and a verified implementation.
Journal of Automated Reasoning **56**, 221–259.
11. Chen H, Ziegler D, Chajed T, Chlipala A, Kaashoek MF, Zeldovich N. 2015 Using Crash Hoare logic for certifying the FSCQ file system.
In *ACM Symposium on Operating Systems Principles*, pp. 18–37. Monterey, CA.
12. Beringer L, Petcher A, Ye KQ, Appel AW. 2015 Verified correctness and security of OpenSSL HMAC.
In *Proceedings of the 24th USENIX Security Symposium*, pp. 207–221. Washington, DC, US.
URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer
13. Hawblitzel C, Howell J, Kapritsos M, Lorch JR, Parno B, Roberts ML, Setty STV, Zill B. 2015 IronFleet: proving practical distributed systems correct.
In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, pp. 1–17. Monterey, CA, USA.
14. Gonthier G. 2005.
A computer-checked proof of the four colour theorem.
http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf.
15. Gonthier G. 2008 Formal proof — the four-color theorem.
Notices of the American Mathematical Society **55**, 1382–1393.
16. Hales TC, Harrison J, McLaughlin S, Nipkow T, Obua S, Zumkeller R. 2010 A revision of the proof of the Kepler conjecture.
Discrete & Computational Geometry **44**, 1–34.
17. Hales TC, Adams M, Bauer G, Dang DT, Harrison J, Hoang TL, Kaliszyk C, Magron V, McLaughlin S, Nguyen TT, Nguyen TQ, Nipkow T, Obua S, Pleso J, Rute J, Solovyev A, Ta AHT, Tran TN, Trieu DT, Urban J, Vu KK, Zumkeller R. 2015 A formal proof of the Kepler conjecture.
Computing Research Repository **abs/1501.02155**.
18. Gonthier G, Asperti A, Avigad J, Bertot Y, Cohen C, Garillot F, Roux SL, Mahboubi A, O'Connor R, Biha SO, Pasca I, Rideau L, Solovyev A, Tassi E, Théry L. 2013 A machine-checked proof of the Odd Order Theorem.
In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pp. 163–179.
19. Atuah KD. 2017.
Securing outer space with seL4.
https://research.csiro.au/tsblog/securing-outer-space-sel4/.
20. Trustworthy Systems Team. 2017.
The SMACCM project.
https://ts.data61.csiro.au/projects/TS/SMACCM/.
21. Nipkow T, Paulson L, Wenzel M. 2002 *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*.
22. Slind K, Norrish M. 2008 A brief overview of HOL4.
In *International Conference on Theorem Proving in Higher Order Logics*, pp. 28–32. Montréal, Canada.
23. Bertot Y, Castéran P. 2004 *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*.
Texts in Theoretical Computer Science. An EATCS Series.
24. Leino KRM. 2010 Dafny: An automatic program verifier for functional correctness.
In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Yogyakarta, Indonesia.
25. Heiser G, Elphinstone K. 2016 L4 microkernels: The lessons from 20 years of research and deployment.
ACM Transactions on Computer Systems **34**, 1:1–1:29.
26. Liedtke J. 1996 Towards real microkernels.

Communications of the ACM **39**, 70–77.

27. Andronick J, Jeffery R, Klein G, Kolanski R, Staples M, Zhang HJ, Zhu L. 2012 Large-scale formal verification in practice: A process perspective.
In *International Conference on Software Engineering*, pp. 1002–1011. Zurich, Switzerland.
28. Blackham B, Shi Y, Chattopadhyay S, Roychoudhury A, Heiser G. 2011 Timing analysis of a protected operating system kernel.
In *IEEE Real-Time Systems Symposium*, pp. 339–348. Vienna, Austria.
29. Blackham B, Shi Y, Heiser G. 2012 Improving interrupt response time in a verifiable protected microkernel.
In *EuroSys Conference*, pp. 323–336. Bern, Switzerland.
30. Blackham B, Liffiton M, Heiser G. 2014 Trickle: automated infeasible path detection using all minimal unsatisfiable subsets.
In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 169–178. Berlin, Germany.
31. Sewell T, Kam F, Heiser G. 2016 Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis.
In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Vienna, Austria.
32. Li X, Liang Y, Mitra T, Roychoudhury A. 2007 Chronos: A timing analyzer for embedded software.
Science of Computer Programming, Special issue on Experimental Software and Toolkit **69**, 56–67.
33. Sewell T, Myreen M, Klein G. 2013 Translation validation for a verified OS kernel.
In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 471–481. Seattle, Washington, USA.
34. de Moura LM, Bjørner N. 2008 Z3: An efficient SMT solver.
In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Budapest, Hungary.
35. Peleska J, Vorobev E, Lapschies F. 2011 Automated test case generation with SMT-solving and abstract interpretation.
In *NASA Formal Methods Symposium*, pp. 298–312. Pasadena, CA, USA.
36. Murray T, Matichuk D, Brassil M, Gammie P, Bourke T, Seefried S, Lewis C, Gao X, Klein G. 2013 seL4: from general purpose to a proof of information flow enforcement.
In *IEEE Symposium on Security and Privacy*, pp. 415–429. San Francisco, CA.
37. Sewell T, Winwood S, Gammie P, Murray T, Andronick J, Klein G. 2011 seL4 enforces integrity.
In *International Conference on Interactive Theorem Proving*, pp. 325–340. Nijmegen, The Netherlands.
38. Alkassar E. 2009 *OS Verification Extended – On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*.
Ph.D. thesis, Saarland University, Computer Science Department.
39. Alkassar E, Hillebrand MA. 2008 Formal functional verification of device drivers.
In *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pp. 225–239. Toronto, Canada.
40. Ni Z, Yu D, Shao Z. 2007 Using XCAP to certify realistic system code: Machine context management.
In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pp. 189–206. Kaiserslautern, Germany.
41. Boyton A, Andronick J, Bannister C, Fernandez M, Gao X, Greenaway D, Klein G, Lewis C, Sewell T. 2013 Formally verified system initialisation.
In *International Conference on Formal Engineering Methods*, pp. 70–85. Queenstown, New Zealand.
42. Boyton A. 2014 *Secure architectures on a verified microkernel*.
PhD thesis, CSE, UNSW, Sydney, Australia.
43. Fernandez M. 2016 *Formal Verification of a Component Platform*.
PhD thesis, UNSW Computer Science & Engineering, Sydney, Australia.
44. Fernandez M, Andronick J, Klein G, Kuz I. 2015 Automated verification of RPC stub code.
In *International Symposium on Formal Methods*, pp. 273–290. Oslo, Norway.
45. Klein G, Nipkow T, Paulson L. 2004.
The archive of formal proofs.
https://isa-afp.org.

46. Alkassar E, Hillebrand M, Leinenbach D, Schirmer N, Starostin A. 2008 The Verisoft approach to systems verification.
In *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pp. 209–224.
47. Alkassar E, Paul W, Starostin A, Tsyban A. 2010 Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices.
In *Proceedings of Verified Software: Theories, Tools and Experiments 2010*, volume 6217 of *Lecture Notes in Computer Science*, pp. 71–85. Edinburgh, UK.
48. Bourke T, Daum M, Klein G, Kolanski R. 2012 Challenges and experiences in managing large-scale proofs.
In *Conferences on Intelligent Computer Mathematics (CICM) / Mathematical Knowledge Management*, pp. 32–48. Bremen, Germany.
49. Greenaway D, Lim J, Andronick J, Klein G. 2014 Don't sweat the small stuff: Formal verification of C code without the pain.
In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 429–439. Edinburgh, UK.
50. Greenaway D. 2015 *Automated proof-producing abstraction of C code*.
PhD thesis, CSE, UNSW, Sydney, Australia.
51. Staples M, Kolanski R, Klein G, Lewis C, Andronick J, Murray T, Jeffery R, Bass L. 2013 Formal specifications better than function points for code sizing.
In *International Conference on Software Engineering*, pp. 1257–1260. San Francisco, USA.
52. The COSMIC Consortium. 2009 *The COSMIC Functional Size Measurement Method Version 3.0.1: Measurement Manual*.
53. Staples M, Jeffery R, Andronick J, Murray T, Klein G, Kolanski R. 2014 Productivity for proof engineering.
In *Empirical Software Engineering and Measurement*, p. 15. Turin, Italy.
54. Matichuk D, Murray T, Andronick J, Jeffery R, Klein G, Staples M. 2015 Empirical study towards a leading indicator for cost of formal software verification.
In *International Conference on Software Engineering*, p. 11. Firenze, Italy.
55. Lochbihler A. 2007 Jinja with threads.
Archive of Formal Proofs . https://isa-afp.org/entries/JinjaThreads.shtml, Formal proof development
56. Maric F. 2008 Formal verification of modern SAT solvers.
Archive of Formal Proofs . https://isa-afp.org/entries/SATSolverVerification.shtml, Formal proof development
57. Blanchette J, Haslbeck M, Matichuk D, Nipkow T. 2015 Mining the archive of formal proofs.
In *Conference on Intelligent Computer Mathematics*, pp. 3–17. Washington DC, USA.
58. Amani S, Hixon A, Chen Z, Rizkallah C, Chubb P, O'Connor L, Beeren J, Nagashima Y, Lim J, Sewell T, Tuong J, Keller G, Murray T, Klein G, Heiser G. 2016 Cogent: Verifying high-assurance file system implementations.
In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 175–188. Atlanta, GA, USA.
59. Rizkallah C, Lim J, Nagashima Y, Sewell T, Chen Z, O'Connor L, Murray T, Keller G, Klein G. 2016 A framework for the automatic formal verification of refinement from Cogent to C.
In *International Conference on Interactive Theorem Proving*. Nancy, France.
60. O'Connor L, Chen Z, Rizkallah C, Amani S, Lim J, Murray T, Nagashima Y, Sewell T, Klein G. 2016 Refinement through restraint: Bringing down the cost of verification.
In *International Conference on Functional Programming*. Nara, Japan.
61. Palix N, Thomas G, Saha S, Calvès C, Lawall J, Muller G. 2011 Faults in Linux: ten years later.
In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 305–318. Newport Beach, CA, US.
62. 2014.
The Rust programming language.
http://rustlang.org.
Accessed March 2015
63. Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang Y. 2002 Cyclone: A safe dialect of C.
In *Proceedings of the 2002 USENIX Annual Technical Conference*, pp. 275–288. Monterey, CA, USA.

64. HASP project. 2010 The Habit programming language: The revised preliminary report. Technical Report http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf, Department of Computer Science, Portland State University, Portland, OR, USA.
65. Hunt G, Aiken M, Fähndrich M, Hawblitzel C, Hodson O, Larus J, Levi S, Steensgaard B, Tarditi D, Wobber T. 2007 Sealing OS processes to improve dependability and safety. In *Proceedings of the 2nd EuroSys Conference*, pp. 341–354. Lisbon, Portugal.
66. Amani S, Murray T. 2015 Specifying a realistic file system. In *Workshop on Models for Formal Analysis of Real Systems*, pp. 1–9. Suva, Fiji.
67. Reynolds JC. 2002 Separation logic: A logic for mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74. Copenhagen, Denmark.
68. Globalscale Technologies Inc. 2015. MiraBox development kit. https://www.globalscaletechnologies.com/p-58-mirabox-development-kit.aspx. Accessed Nov 2015
69. The Go Project. 2015. The Go programming language. https://go.googlesource.com/go. Accessed Nov 2015
70. Filebench Project. Filebench file system benchmark. http://sourceforge.net/projects/filebench. Accessed Nov 2015
71. Fisher K, Walker D. 2011 The PADS project: An overview. In *International Conference on Database Theory*, pp. 11–17.
72. Cogent Project. Cogent: code and proof co-generation. https://github.com/NICTA/cogent.