Towards Practical Verification of Large-Scale Numerical Software Libraries

Joshua M. Cohen Andrew W. Appel

February 21, 2024

Abstract

Formal verification has been successful in developing new provably correct systems, including compilers, OS kernels, and cryptographic functions. But these efforts seldom verify existing, real-world implementations, and those that do focus on small components. Verification of real-world software libraries at scale introduces new challenges and considerations. In this paper, we explore these challenges in the context of PETSc, a widely-used, state-of-the-art numerical linear algebra library. In particular, we discuss how to evaluate the feasibility of a library for verification, how to identify a verifiable subset and minimize its set of dependencies, and how to keep such a subset up-to-date with an actively developed library. We create several small tools automate these tasks and apply them to PETSc, producing a verifiable (though not yet verified) subset usable to external clients with no additional work required. We hope that our methodology can help to enable feasible and scalable verification of large-scale software, especially numerical and other mathematically intensive libraries.

1 Introduction

As software grows more and more complex, the potential for bugs increases. *Formal verification* uses the tools of mathematical logic to prove the absence of bugs in a program, and a variety of automated and interactive methods have been developed for this purpose. Such techniques have been applied at scale, for instance to develop the CompCert verified C compiler [22], which is accompanied by a formal proof that the compilation preserves the semantics of the input program.

Existing efforts to verify real-world programs typically take one of two forms: either they involve writing a new program and proving it correct, carefully designing the program to facilitate reasoning (as in CompCert or Fiat crypto [11]); or they involve verifying a piece of already-existing code, such as a cryptographic primitive [1]. However, the latter efforts involve verification of a single version of a program. Especially for programs whose correctness requires significant mathematical reasoning, it can be extremely difficult to evolve the specification and proofs as the software itself changes. As a result, users often cannot choose to link their code against verified versions of existing software, as these are out-of-date and no longer reflective of the current state of the program. These issues are only avoided by considering programs that change very infrequently.

In this paper we consider techniques and principles for both analyzing the feasibility of verifying real-world, large-scale software libraries and for making it practical to keep such verification efforts up-to-date and thereby allow users to use verified versions of such libraries. We use as a case study PETSc (Portable, Extensible Toolkit for Scientific Computation) [3], a state-of-the-art library for numerical linear algebra. While we do not verify PETSc, we describe the process by which it could be done and discuss how one might systematically approach such a verification effort, with the goal of allowing external users to link against verified components of PETSc with no additional effort.

Library verification is a distinct task from ordinary program verification, with its own set of considerations and challenges. In this paper, we address the following questions:

1. How can we evaluate whether a library is amenable for verification? Considerations include the use and amount of encapsulation, abstraction, and modularity in the library design, the ability to identify library features in isolation (for example, by separating logging and error handling capabilities), and the amount of system-level infrastructure the library contains. We analyze these aspects, among others, in our discussion on the feasibility of PETSc as a verification target.
2. How can we transform a large-scale, state-of-the-art library into a smaller subset that minimizes the number of dependencies (so as to require verification of only the minimal subset of library components) yet, for a particular client program, still runs *exactly the same code* as the existing, unverified library? We design a methodology and a tool to address this problem for C libraries.
3. How can we keep a verified library in sync with, or at least knowledgeable about the differences between, the current version of an actively-developed library, and incorporate this information into continuous integration (CI)? We again discuss design considerations and produce a small tool to help with this.

2 Background

2.1 Verification and VST

Most program verification tools capable of proving full functional correctness (i.e. proving that a function has the correct behavior on any possible input) can broadly be split into two categories. *Automated* tools use SMT solvers, model checkers, and other methods to prove programs correct with minimal effort needed by the user – the user typically annotates their program with a specification and some hints to help the solver, and the solver takes care of the rest. Examples include Dafny [21], a language with built-in verification capabilities, and Frama-C [10] and VeriFast [17] for C programs.

Interactive methods require the user to manually construct a proof of the program's correctness in a proof assistant, with only user-defined automation. Such tools are much more powerful than automated ones and are necessary for proving program properties that require sophisticated mathematical reasoning, but require significantly more expertise. Examples include VST [2] for verifying C programs and CakeML [20] for verifying ML programs.

VST in particular is crucial to our efforts. It consists of a program logic for C in the Coq proof assistant, proved sound with respect to the CompCert C compiler. This means that we can verify real C programs (CompCert supports a significant subset of C99) and compile with CompCert, resulting in a formal, machine-checked proof that the assembly code satisfies the specification we proved. VST is very expressive, supporting concurrency and higher-order, object-oriented patterns. Its assertion language is that of Coq, a dependently typed, higher-order logic. The upshot is that one can prove sophisticated, real-world, and unmodified C programs correct using VST and connect these proofs seamlessly to higher-level mathematical proofs about the underlying algorithms.

Such an approach has been used to verify many different C programs, including an HTTP key-value server [28], OpenSSL SHA256 [1], a packet error-correction system based on Reed-Solomon erasure coding [9], and a variety of numerical programs including a leapfrog integrator for a harmonic oscillator [18] and an implementation of Jacobi iteration to solve linear systems [27]. The latter 4 efforts rely on nontrivial mathematical reasoning in the application domain of interest: cryptography, error-correcting codes (linear algebra over finite fields), and numerical analysis, respectively; Coq and associated mathematical libraries like Mathematical Components [23] are well-equipped for this.

However, all of these existing projects either implement their own C program (HTTP, integrator, Jacobi) or verify a single version of an existing C program (SHA256, erasure correction). In particular, existing proof assistant-based verified numerical programs rely on a newly written program, in C or otherwise. This does not align with how developers write code in practice, which is to use existing, mature software libraries. We would like VST-style verification for such libraries (we are particularly focused in the numerical domain, but the same ideas apply more broadly) so that users with no verification expertise or knowledge can use verified code with no additional effort. We focus on the PETSc library.

2.2 PETSc

PETSc is a widely used, state-of-the art library for numerical linear algebra. It includes data structures such as sequential and parallel vectors and a variety of (parallel and sequential) sparse matrix formats, as well as dozens of algorithms for solving linear and nonlinear systems and differential equations. PETSc is written in C with bindings for Python and Fortran, but it uses an object-oriented programming style (with an ad-hoc implementation of objects using function pointers). It includes a very large “standard library” of data structures, string operations, and methods to handle memory allocation and deallocation. It also contains error handling and logging capabilities, as well as the ability to modify the behavior of code at runtime via (configurable) command-line options.

PETSc is an intriguing target for VST-style interactive verification for several reasons. It is widely used in dozens of higher-level projects and tools, as well as directly for a variety of numerical simulation problems, which means that verification can have large benefits and bugs are potentially very costly. It implements numerous sophisticated numerical algorithms and efficient data structures; this introduces the possibility of bugs both in the mathematics (while the underlying algorithms are well-understood, some proofs do not rigorously handle the semantics of floating point numbers) as well as the possibility for low-level implementation bugs (dereferencing null pointers, overflow, crashes, etc), particularly as C is a memory-unsafe language. To verify such a library, we need a tool capable of reasoning at both of these levels – VST is a particularly good fit, and it is equipped to handle the higher-order, object-oriented design patterns used extensively in PETSc. Automated tools would not be able to handle the sophisticated real and numerical analysis needed to prove correctness of the underlying algorithms, and we need a powerful program logic to handle the inherently higher-order nature of the library.

Finally, PETSc is attractive for verification because it fills a gap in the existing verified pipeline. Numerous projects have formalized properties of numerical algorithms in Coq and other proof assistants [27, 18, 16, 6], about which there are now intricate proofs of correctness and precise bounds on error and/or iteration time. However, these proofs are connected to simple implementations written for verification purposes. A user who wishes to solve a linear system with Jacobi iteration, for instance, cannot make use of this code. This is not an inherent limitation; indeed, an advantage of the standard Coq+VST workflow is that the mathematical proofs about the algorithm are completely separate from the C-language verification conditions; this means that the higher-level proofs can be reused for any equivalent implementation. In practice, of course, one wishes such an implementation to be efficient; libraries like PETSc have undergone significant optimization and careful engineering to improve speed and accuracy. Therefore, connecting existing proofs about numerical algorithms with their PETSc implementations can make it possible for users to run verified code without suffering any performance penalty or requiring any additional effort.

3 Analyzing Suitability for Verification

PETSc is an intriguing target for verification, but is such verification feasible? There is a good reason why many existing verification projects consider their own code: formal verification is difficult, and verifying code that was not written with verification in mind is even more so.

PETSc is a well-designed library; accordingly, it makes crucial design decisions that make verification easier in principle. In particular, it makes extensive use of abstraction and modularity. Figure 1 shows this

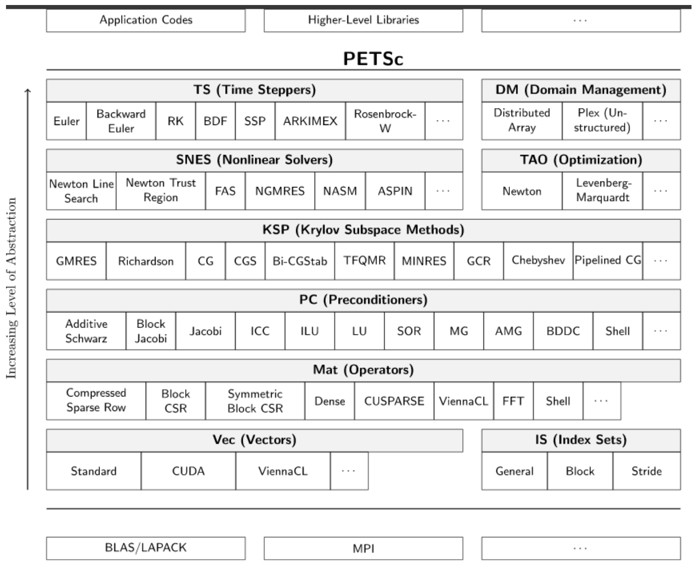

Figure 1: Structure of the PETSc library [1]

structure, in which each object type (Vector, Matrix, Krylov Subspace Solver, etc) implements a well-defined interface, and only interacts with lower layers through this interface. In principle, one could design specification boundaries accordingly; for example, the specification for matrix multiplication would be defined on the abstraction, and each matrix implementation should implement multiplication in such a way that we can prove it correct against this same specification. Then, at higher layers in the proof, one does not need to know *which* implementation was used, only that *some* implementation satisfying the spec was. Also particularly helpful for verification is PETSc's extensive use of abstraction, largely avoiding code duplication and grouping common functionalities.

Of course, verifying an entire library the size of PETSc (roughly 800K LOC) is not feasible. Instead, we want to identify a small subset that nevertheless allows one to write real programs. With the modular nature of PETSc, even this small subset is extremely valuable, as it would be relatively easy to incrementally verify more components and plug them into the existing verified pipeline. Therefore, such an initial subset should comprise a vector implementation, a matrix implementation, a linear solver, a preconditioner (used to adjust the matrix to make iterative linear solvers converge faster), and enough of the base PETSc system so that the resulting system compiles and runs exactly the same as with the full PETSc.

A natural choice for such a subset is that required to implement Jacobi iteration (which in PETSc consists of Richardson iteration with a Jacobi preconditioner). Then, we can link the PETSc proofs with the existing

[1] https://petsc.org/release/overview/nutshell/

numerical and convergence proofs about this algorithm in Coq [27], giving us an end-to-end correctness theorem about an efficient, real-world implementation.

We note that PETSc, as with many large software libraries, has a variety of flags and configuration options. For instance, one can enable or disable logging, ways of drawing and viewing data, and certain types of error handling. For our verification purposes, we would like to consider a *single but realistic* configuration of PETSc – that is, we identify a single setting of flags and options under which we will verify the system. Then, we can identify which flags must be set if a user wishes to stay within the verifiable subset.

Ideally, this set of flags would minimize the amount of code that we need to verify – both by disabling features like error handling which are not needed because we *prove* that no errors are encountered and by disabling irrelevant features like logging. However, this is only partially possible: PETSc includes many features which are both difficult to verify and largely irrelevant for verification purposes but which are tightly integrated into the library and which cannot be disabled. Such features include dynamic options and libraries and use of the Message Passing Interface (MPI) for parallelism even in purely sequential code.

For these features, our specifications will be largely trivial, but verification is still nontrivial: we need to ensure that these components do not crash or use memory incorrectly. However, we do not want to change the code or remove these features, inconvenient as they may be, as we are guided by the following principles:

- The subset of PETSc we choose must be achievable with *some* possible set of configuration flags and options. Then, we can inform the user to choose such a configuration if they wish to use the verifiable subset of PETSc.
- To write a client-level program that uses the verifiable subset of PETSc, assuming the configuration options are set correctly, the user should need to do nothing other than link against the verified PETSc binaries.
- When a user writes a program and links against the verifiable subset, the code that runs must be *exactly the same code* as that of linking against PETSc (except of course for any bugs found during the verification process, which would hopefully be fixed in PETSc as well).
- A user linking against the verifiable subset need not be responsible for knowing exactly which parts of PETSc are in this subset; at runtime their code should throw an error if they attempt to call functions outside of the subset.

Our verifiable subset will include large parts of the PETSc system and libraries which are largely unnecessary for our efforts yet are still executed, and thus must be considered. These include the aforementioned dynamic options, functions for string processing, some error handling and printing (we would prove that these errors are not triggered), MPI infrastructure (though the subset we wish to verify is sequential), and functions to register citation metadata.

4 From Large-Scale to Small(ish)-Scale

One might suppose, then, that the task is rather simple: take the minimal set of dependencies needed to verify Jacobi iteration, which will hopefully be small enough to be feasible, while allowing for further incremental and modular verification of future components as described above. Yet the concept of the "minimal set" of dependencies is more complicated than it may seem.

Appendix A shows the PETSc program we are interested in, which computes Jacobi iteration on a matrix stored in Compressed Sparse Row (CSR) format (called MATAIJ in PETSc). The example matrix is not too relevant (although it has a property called diagonal row dominance under which Jacobi iteration is provably guaranteed to converge); verifying the underlying algorithms would allow us to prove correct any such example. The naive approach is to start with this program, identify all the functions called (PetscInitialize, MatCreate, etc), identify all of their dependencies, and so on until we have a PETSc subset that compiles.

This approach results in an enormous number of dependencies, for 2 main reasons. First, PetscInitialize, the first function call in any PETSc program, initializes a variety of global data structures to handle the dynamic options database, MPI, error handling, and so on. These structures and associated functions rely on lower-level functions for handling memory allocation and generic PETSc objects as well as data structures including hash tables and circular buffers. Secondly, PETSc handles dynamic libraries by registering (in global data structures) all known classes (Vec, Mat, KSP, etc), all known types in this class (sequential vectors, compressed sparse row matrices, etc), and a constructor for each type. Thus, even though we are only interested in a single implementation, the dependencies include *every* vector, matrix, preconditioner, and solver implementation (or at least their constructors, but their constructors initialize their function pointers to their other methods, so in practice, all such implementations become dependencies). Thus, under such an approach, the required dependencies include a substantial fraction of PETSc.

One obvious reduction in the number of dependencies is to ignore functions stored in function pointers yet not called – e.g. in all matrix implementations other than CSR, only the constructors are relevant dependencies, and the rest of the functions can be replaced with trivial stubs (and specifications asserting that such functions are never called). This helps, but still leaves at least 30K LOC, which is still infeasible.

Instead, we will recall the principles of §3 and note that we only need to preserve PETSc functions which are *actually run*. To illustrate this, consider the following toy example:

```
int foo() {              int a() {               int b() {
  if (true) a();           c();                    e();
  else b();                d();                    f();
}                        }                       }
```

In the naive approach, a(), b(), c(), d(), e(), and f() are all dependencies, and may have further dependencies. However, we really only care about a(), c(), and d() (and their dependencies); i.e. we would like the following:

```
int foo() {
  if (true) a();
  else b();
}
```

```
int a() {
  c();
  d();
}
```

```
int b() {
  assert(false);
}
```

And we note that the actual code run when foo is executed is *exactly the same* in both cases. Using this as motivation, we aim to only include the dependencies of our Jacobi iteration implementation that are actually run. Thus, any functions in PetscInitialize that are only run if certain options are enabled (say, for logging), any functions registered as constructors but never called, any function pointers of the data structures we verify which are not called, and any code paths not executed should have their resulting dependencies turned into stubs. Moreover, as the assert(false); above demonstrates, we should enforce that such code is not used; if the user desires to link their program against the verifiable subset of PETSc, they should have a way to know that they have left the subset.

In summary, we now have 3 different kinds of functions: those functions appearing in the verifiable subset (i.e. those actually executed by the program in Appendix A), functions needed for compilation but which are not run and should be stubified, and functions not needed at all, which can be removed. Manually identifying which function falls into each category and stubifying functions is tedious, error-prone, and would need to be repeated with each new version of PETSc. Instead we create a tool to handle this largely automatically.

4.1 Stubify: A Tool for Producing Minimal Dependencies

Our tool, Stubify[2], takes in a preprocessed C program and a list of functions to keep, and it stubs out everything else, returning an updated preprocessed C file. Specifically, it is a modified C lexer/parser that replaces all function bodies of functions not in the input list with equivalent whitespace, preserving preprocessor line numbers and other metadata. The subset of C accepted is that of CompCert (we modify CompCert's lexer/parser), which is the same subset supported by VST.

There is some subtlety in ensuring that the resulting code is still valid C. There are two issues: first, to ensure compilation, the function must still return something of the correct type. Second, recall that we want the client at runtime to encounter an error if they try to use a stubified function. Achieving this in general (for any C file) is not trivial. The assert(false) in the previous example was an oversimplification; we cannot assume that the C files we are stubifying have included assert.h. Similarly, we cannot directly print an error message (stdio.h may not be included) or exit with exit(1) (stdlib.h may not be included).

To deal with the first issue, we can end the function with an infinite loop (**while**(1)). For the second issue, we use the fact that C allows **extern** declarations within function bodies. Thus, the tool adds the following C

[2] github.com/joscoh/stubify

code immediately before the closing curly brace of the stubified function (where curr_name stores the name of the function it is stubifying):

```
extern void stub_error(char *); stub_error(curr_name); while(1);
```

Then, we link the resulting stubs with a file stub_error.c, which can be user-configurable but in our case is simple:

```
#include <stdio.h>
#include <stdlib.h>

void stub_error(char * str) {
  fprintf(stderr,
    "ERROR: Called %s, which is not in the verified subset of PETSc\n", str);
  exit(1);
}
```

Therefore, calling a stubified function results in an error with an informative error message about which non-verifiable function was called. In the VST specifications, we would give stub_error a precondition of false – this would ensure that if any verifiable function calls a stubbed-out function on a code path not ruled out by the function's preconditions, the verification will fail. In other words, these functions will *provably* correspond to the functions that are not actually run from within the verifiable subset.

5 Verifying Actively-Developed Libraries

PETSc, like any large-scale software library, is actively developed and changes frequently. Formal verification is difficult and time-consuming, particularly when we are interested in full functional correctness. Accordingly, it is very difficult to keep proofs up-to-date with changing code. We do not directly address that problem here, although we speculate that the lower-level parts of PETSc, comprising foundational data structure and algorithms that the rest of the library builds on, should not change very often.

Instead, we consider two simpler problems: how can we keep our verifiable subset up to date with the library, and how can we detect changes in the verifiable subset? The stubify tool is extremely helpful for both of these purposes.

5.1 Keeping the Verifiable Subset Up-to-Date

To address the first issue, we build up a small framework for using our Stubify tool on PETSc.[3] In particular, we mirror the directory structure of the subset of PETSc we are interested in; for each corresponding C

[3]The scripts and files mentioned in this section can be found at https://github.com/joscoh/petsc/tree/main/verified. Subsequent footnotes in this section contain files relative to this repository. The repository includes a README explaining how to run the relevant scripts.

file, we include the list of functions in the verifiable subset.[4] This approach makes it clear exactly what files and functions are included in the verifiable subset, and allows us to deal with static functions from header files. Some C files have associated function lists which are empty; this means that (some of) the functions are needed for compilation but none are actually verified, so all should be stubified. Other files do not have any function list; we can ignore these files when compiling the verifiable subset since no verified code uses them. Otherwise, we proceed with the Stubify tool, keeping the listed functions and stubifying the others. This process is entirely automated, with several scripts, so that generating the preprocessed, stubified C files and compiling (with CompCert) requires only a single command.[5]

5.2 Compiling the Verifiable Subset

With this, we can compile our Jacobi program (Appendix A) with CompCert against this PETSc subset. We do require a few very minor changes to allow CompCert to compile this (PETSc supports gcc and clang):

1. We provide a configuration flag PETSC_VERIFIED and, if enabled, ensure that the other configuration flags are set appropriately (for example, to disable logging, disable some use of builtin functions that CompCert cannot handle, and so on). This does not change the code run, only the specific (valid) PETSc configuration used.

2. There are 3 included header files that CompCert cannot compile – 2 from Valgrind and quadmath.h. The **#include** directives are already conditional; we add the additional condition that PETSC_VERIFIED is not defined. None of these files contain code that is run in the verifiable subset.

3. We refactor one function for convenience; the function first tests if the input is null, and if not, calls many other functions. In our case, the value will always be null; instead of including the roughly 25 additional (stubified) dependencies from this function, we refactor it to put the post-null-return code in a separate, now-stubified function. Again, this does not change the code run.

4. The final change is more interesting; it is due to the following macro, which is used in a **sizeof**:

   ```
   #define PetscSafePointerPlusOffset(ptr, offset) ((ptr) ? (ptr) + (offset) : NULL)
   ```

   PETSc relies on behavior of gcc and clang that violates the C standard: defining **sizeof(void)** to be 1. CompCert rejects this as undefined behavior, and we replace this (when PETSC_VERIFIED is defined) with the macro:

   ```
   #define PetscSafePointerPlusOffset(ptr, offset) ((ptr) + (offset))
   ```

[4] verified/src

[5] Such commands are automated by our Makefile, which calls cpp_stub.h, a wrapper around stubify which creates the stubified files based on the provided function lists.

> This is not ideal, as we are changing the code that is executed. However, in any verification effort, we would *prove* that the input pointer exists; thus the first case is always hit and so these two are equivalent – when the verifiable subset is used as specified.

In total, these changes comprise 29 insertions and 18 deletions compared to the main PETSc branch, with #3 above excluded, there are 8 additions and 4 deletions. In principle, if these changes were added to PETSc via a pull request, the entire generation and compilation of the verifiable subset would be completely automatic.

We also note that the verifiable subset can change after commits; for example due to refactoring. This requires changes to the lists of non-stubified functions. However, there is an easy way to identify such functions: run the Jacobi program. If any newly created functions are executed, our stub_error call will alert us. As one data point, we note that pulling about 8 months of commits (about 2000 commits) from PETSc, re-running the scripts, and adding refactored functions so that the program still compiled required about a half hour.

5.3 Monitoring the Verifiable Subset: Continuous Integration

In practice, libraries like PETSc are continually developed, and the majority of such developers are likely completely unconcerned with verification. The tools, scripts, and workflows we have presented produce the verifiable subset of PETSc largely automatically, and thus can be run each time the code changes (say, at each commit). However, such commits may alter the subset, and we would like a way to detect this. To that end, we have written a script using Bash and Python which compares the generated stubified files between the current directory and the last git commit and determines if these files have changed.[6] We do not consider changes in whitespace, and some care must be taken to ignore irrelevant debug information (e.g. the directory in which the program is stored). However, this script successfully returns a zero error code if and only if the two verifiable subsets are identical (and otherwise, prints the file names that differ). With this information, the mantainers of the library could integrate this into continuous integration and could determine what action to take in case of changes to the verifiable subset.

6 Related Work

Verification of Real-World Software As discussed in §2.1, most proof-assistant-verified software is built from the ground up for verification purposes, and only rarely is existing, real-world software verified. Real-world verified software includes 2 functions from OpenSSL: the SHA-256 cryptographic hash function [1], and HMAC message authentication [5]. Additionally, Cohen, et. al. verify a real-world packet error correction system [9]. Beyond C programs, Breitner et. al. apply the hs-to-coq tool

[6] diff.sh and compare.py

to verify parts of Haskell's containers library in Coq [7]; they produce a verified (though unmaintained) subset of this library.[7]

Beyond these efforts, many verification projects that are writing new code, rather than verifying existing code, nevertheless deal with real systems and standards. These include the CompCert verified compiler for C [22], the sel4 verified operating system kernel [19], the CertiKOS concurrent operating system kernel [14], the GoJournal verified file system [8], a verified POSIX shell [13], and verified cryptography in EverCrypt [24] and Fiat [11], which both generate correct-by-construction C code from F* and Coq specifications, respectively. Fiat in particular has become real-world code, as it is included in Chrome.

Reasoning about Proof Changes Real-world software libraries are continually evolving, but there has been little work on dealing with these issues in the context of verification. SymDiff [15] is a tool for comparing two versions of a program to prove properties such as relative termination; it operates not at the source level but at the level of the Boogie [4] intermediate language. Regression Verification [12] attempts a similar task but broadens to full functional correctness; it uses CBMC, the C Bounded Model Checker, which is much less powerful and scalable but more automated than a tool like Coq/VST.

There is more recent work in the area of *proof repair*, which develops techniques for automatically adapting proofs to changing programs. Such work includes PUMPKIN PATCH [26], which searches for patches to Coq scripts automatically based on the history of changes. This was extended to deal with transformations across general type equivalences [25] and a focus on integrating this into real Coq workflows.

7 Conclusion

In this work, we examine the issues present in verifying a large, mathematically sophisticated, evolving, real-world software library, taking the PETSc numerical linear algebra library as a case study. We examine the features of such libraries that make them more and less amenable to verification, propose methods to identify minimal dependency subsets in such libraries, examine the problem of keeping such subsets up-to-date, and incorporate these analyses into a small set of tools to enable verification of the smallest amount of code possible while allowing a client to link against the verified code with no additional effort.

One key aspect to making verification of such large-scale libraries feasible is the use of automated tools. Since we need to deal with sophisticated mathematics, we need the power of a proof assistant; however, the main PETSc system, comprising the vast majority of the code in the verifiable subset, could in principle be verified with an automated or semi-automated solver such as VeriFast [17], leaving only the mathematically interesting parts to VST and Coq. Further work is needed to study how

[7]https://hackage.haskell.org/package/containers-verified

to combine such tools in a sound and principled way that do not significantly compromise the guarantees that the VST-CompCert pipeline provide. Nevertheless, we hope that verification of real-world libraries at scale can become a future reality, and we believe that the issues we have considered will be crucial to making such efforts viable.

References

[1] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), apr 2015.

[2] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[3] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang. PETSc Web page, 2023.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., Aug. 2015. USENIX Association.

[6] S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Formal proof of a wave equation resolution scheme: The method error. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, pages 147–162, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] J. Breitner, A. Spector-zabusky, Y. Li, C. Rizkallah, J. Wiegley, J. Cohen, and S. Weirich. Ready, set, verify! applying hs-to-coq to real-world haskell code. *Journal of Functional Programming*, 31:e5, 2021.

[8] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 423–439. USENIX Association, July 2021.

[9] J. M. Cohen, Q. Wang, and A. W. Appel. Verified erasure correction in Coq with MathComp and VST. In S. Shoham and Y. Vizel, editors, *Computer Aided Verification*, pages 272–292, Cham, 2022. Springer International Publishing.

[10] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, pages 233–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[12] B. Godlin and O. Strichman. Regression verification. In *2009 46th ACM/IEEE Design Automation Conference*, pages 466–471, 2009.

[13] M. Greenberg and A. J. Blatt. Executable formal semantics for the posix shell. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[14] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, Nov. 2016. USENIX Association.

[15] C. Hawblitzel, S. Lahiri, M. Kawaguchi, and H. Rebelo. Towards modularly comparing programs using automated theorem provers. In *International Conference on Automated Deduction (CADE '13)*. Springer, June 2013.

[16] F. Immler and J. Hölzl. Numerical analysis of ordinary differential equations in isabelle/hol. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving*, pages 377–392, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[17] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[18] A. E. Kellison and A. W. Appel. Verified numerical methods for ordinary differential equations. In O. Isac, R. Ivanov, G. Katz, N. Narodytska, and L. Nenzi, editors, *Software Verification and Formal Methods for ML-Enabled Autonomous Systems*, pages 147–163, Cham, 2022. Springer International Publishing.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[20] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[21] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[23] A. Mahboubi and E. Tassi. *Mathematical Components.* Zenodo, Sept. 2022.

[24] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy.* IEEE, May 2020.

[25] T. Ringer, R. Porter, N. Yazdani, J. Leo, and D. Grossman. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 112–127, New York, NY, USA, 2021. Association for Computing Machinery.

[26] T. Ringer, N. Yazdani, J. Leo, and D. Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 115–129, New York, NY, USA, 2018. Association for Computing Machinery.

[27] M. Tekriwal, A. W. Appel, A. E. Kellison, D. Bindel, and J.-B. Jeannin. Verified correctness, accuracy, and convergence of a stationary iterative linear solver: Jacobi method. In C. Dubois and M. Kerber, editors, *Intelligent Computer Mathematics*, pages 206–221, Cham, 2023. Springer Nature Switzerland.

[28] H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L.-Y. Xia, L. Beringer, W. Mansky, B. Pierce, and S. Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

A PETSc Code for Jacobi Iteration

This program is a slight modification to ex1.c in PETSc's KSP examples.

```
#include <petscksp.h>
int main(int argc, char **args)
{
  Vec x, b, u; /* approx solution, RHS, exact solution */
  Mat A; /* linear system matrix */
  KSP ksp; /* linear solver context */
```

```
PC pc; /* preconditioner context */
PetscReal norm; /* norm of solution error */
PetscInt i, n = 5, its;
PetscMPIInt size;
//Values in each row of matrix
PetscScalar value[5][3] = {{1, 0.25, 0.25}, {1, 0.5, 0.125}, {0.5, 1, 0.25}, {0.25, 1, 0.5}, {0.125, 0.5, 1}};
//Columns filled in each row
PetscInt col[5][3] = {{0, 1, 2}, {1, 3, 4}, {0, 2, 4}, {1, 3, 4}, {2, 3, 4}};

// Initialize PETSc library
PetscFunctionBeginUser;
PetscCall(PetscInitialize(&argc, &args, (char *)0, help));

// Create and initialize x, b, u as vectors of size n
PetscCall(VecCreate(PETSC_COMM_SELF, &x));
PetscCall(VecSetSizes(x, PETSC_DECIDE, n));
PetscCall(VecSetType(x, VECSEQ));
PetscCall(VecDuplicate(x, &b));
PetscCall(VecDuplicate(x, &u));

// Create matrix A in compressed sparse row format and populate values
PetscCall(MatCreateSeqAIJ(PETSC_COMM_SELF, n, n, 3, NULL, &A));
for(i = 0; i < n; i++) {
  PetscCall(MatSetValues(A, 1, &i, 3, col[i], value[i], INSERT_VALUES));
}
PetscCall(MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY));
PetscCall(MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY));

// Create the linear system Ax = b, with expected solution x = (1,1,1,1,1)
PetscCall(VecSet(u, 1.0));
PetscCall(MatMult(A, u, b));

// Create a Richardon iteration solver with Jacobi preconditioner
PetscCall(KSPCreate(PETSC_COMM_SELF, &ksp));
PetscCall(KSPSetType(ksp, KSPRICHARDSON));
PetscCall(KSPSetOperators(ksp, A, A));
PetscCall(KSPGetPC(ksp, &pc));
PetscCall(PCSetType(pc, PCJACOBI));
PetscCall(KSPSetTolerances(ksp, 1.e-5, PETSC_DEFAULT, PETSC_DEFAULT, PETSC_DEFAULT));

// Solve the system and check the result
PetscCall(KSPSolve(ksp, b, x));
PetscCall(VecAXPY(x, -1.0, u));
PetscCall(VecNorm(x, NORM_2, &norm));
PetscCall(KSPGetIterationNumber(ksp, &its));

printf("Norm of error %g, Iterations %" PetscInt_FMT "\n", (double)norm, its);

// Destroy the created solver, vectors, and matrix
PetscCall(KSPDestroy(&ksp));
```

```
  PetscCall(VecDestroy(&x));
  PetscCall(VecDestroy(&u));
  PetscCall(VecDestroy(&b));
  PetscCall(MatDestroy(&A));

  // Destroy all library data structures and exit
  PetscCall(PetscFinalize());
  return 0;
}
```