Accelerating Automated Program Verifiers by Automatic Proof Localization

Kiran Gopinathan[1⋆], Dionysios Spiliopoulos[2], Vikram Goyal[3], Peter Müller[2], Markus Püschel[2], and Ilya Sergey[3]

[1] University of Illinois Urbana-Champaign, USA
[2] Department of Computer Science, ETH Zurich, Switzerland
[3] National University of Singapore, Singapore

Abstract. Automated program verifiers such as Dafny, F⋆, Verus, and Viper are now routinely used to verify real-world software. Unfortunately, the performance of the SMT solvers employed by these tools is not always able to keep up with the increasing size and complexity of verification problems, resulting in long verification times and verification failures due to time-outs. This performance degradation occurs because large SMT queries increase the search space for the SMT solver, in particular, the number of possible quantifier instantiations. Most existing attempts to mitigate this problem require substantial manual effort to reduce the size of the search space, for instance, by decomposing proofs.

In this paper, we present an automatic technique to significantly improve the performance of SMT-based program proofs by drastically reducing the proof search space for each assertion, in particular, the performed quantifier instantiations. Starting from a successful verification, we automatically extract for each assertion the quantified axioms used by the SMT solver to show that the assertion is valid. Crucially, these include *lurking axioms*, which are logically irrelevant, but needed to trigger the instantiation of other, relevant axioms. We describe a novel *proof localization* algorithm that implements a semantics-preserving source-to-source translation of a program such that re-verifying an assertion in the optimized program uses only the axioms in its proof essence. This rewriting greatly reduces the possible quantifier instantiations and, thereby, the search space for the SMT solver, such that all future runs of the verifier, for instance as part of continuous integration, are substantially faster. We implemented our algorithm for the Boogie verifier and demonstrated its effectiveness on examples from Dafny and Viper. Specifically, for files with verification times over a minute, we show significant speedups of up to 100–1000 times and no slowdowns. We also provide some evidence that these improvements persist as projects evolve.

1 Introduction

Automated program verification has become an increasingly popular approach to certifying real world systems, with the most recent developments tackling larger and more ambitious targets such as distributed systems [18], cryptographic

⋆ Work done while at National University of Singapore.

libraries [28,1], file systems [7], and access protocols [9]. By delegating proof obligations to Satisfiability Modulo Theories (SMT) solvers, such as Z3 [11] or cvc5 [2], automated verification tools such as Dafny [21], F* [30], Verus [20], and verifiers built on top of Viper [25] or Why3 [15] substantially reduce the proof burden for users and provide a lighter alternative to interactive proof assistants.

However, with the growing ambition of verification projects, automated verifiers increasingly face performance-related maintenance problems. Complex proof obligations substantially increase the proof search space, which may lead to long verification times and failures due to time-outs [18,27]. This performance degradation is primarily caused by an explosion of the number of quantifier instantiations performed by the SMT solver. Most program verifiers use the SMT solver's e-matching algorithm [12], which associates each universal quantifier with a syntactic matching pattern (often called *trigger*) and instantiates the quantifier when a ground term in the proof context matches the pattern. Large queries contain thousands of quantifiers and many ground terms that trigger instantiations, introducing additional ground terms and, thus, even more instantiations.

The prevalent solution to date is to address maintenance concerns by manually refactoring verified programs in order to reduce the number of quantifiers as well as the general size of the proof context. For instance, Dafny and Verus allow programmers to control which function definitions (encoded as quantified axioms) are available in a proof. Dafny and Gobra [31] also allow programmers to break up proofs into smaller pieces, which reduces the size of the proof context for each piece. Both approaches are helpful, but require substantial manual effort and insight to decide when to refactor. Moreover, they are difficult to apply, even for expert users, because the effect of a code refactoring on verification time is rather unpredictable due to the SMT solver's complex heuristics. Bordis and Leino use a syntactic analysis to automatically pre-instantiate some quantified axioms [5], which improves verification times, but may lead to spurious errors if the analysis cannot determine all necessary instantiations.

This work. We present *proof localization*, an automatic procedure to systematically minimize the proof search space for the SMT solver and, thereby, improve verification time. Given an initial successful verification of an assertion, we extract from the SMT solver the *essence of the proof*, which includes the information needed for the solver to efficiently re-prove the assertion on all future runs of the verifier. The proof essence contains the $\forall$-quantified axioms used by the SMT solver to prove that the assertion is valid. Moreover, it includes additional axioms that are logically irrelevant for the proof, but needed by the SMT solver to trigger the instantiation of other, relevant axioms—we refer to them as *lurking axioms*.

Once we have identified all axioms needed to prove an assertion, we automatically rewrite the input program in a semantics-preserving way, such that computing standard verification conditions over the rewritten program includes, for each assertion, only those axioms strictly needed by the SMT solver (rather than all available in the context). The result is a drastically reduced search space.

Since proof localization requires the existence of a proof, it does not improve performance of the initial verification of each assertion. Nevertheless, speeding

up future runs of the verifier is of great practical value. In an active project, each assertion is typically verified a countless number of times. For example, when the code evolves, when iteratively verifying increasingly-strong properties, when code is reused in different contexts, when upgrading the program verifier or the underlying SMT solver, or when verification is integrated into continuous integration pipelines. Caching of verification results is helpful in only a few of these cases, when a proof obligation remains entirely unchanged compared to the cached result; however, even a small, irrelevant change requires re-verification.

Our implementation performs a source-to-source transformation on programs in the Boogie intermediate verification language [3]. It could easily be integrated into any Boogie-based source verifier by performing a corresponding transformation on the source program, such that the unmodified translation into Boogie produce the optimized program. Alternatively, the proof essence could be stored separately from the source program and used by an adapted Boogie translation.

We have implemented our technique in a tool called Axolocl and evaluated it on Boogie programs generated by Dafny and Viper, including various benchmarks from industrial verification projects. Our evaluation shows significant speedups for files with verification times over a minute. Using semantics-preserving mutations of SMT queries, we also provide some evidence that the positive effects of our transformations persist as verification projects evolve.

Contributions. We make the following specific technical contributions:

- We present a systematic approach to extracting the proof essence for a proof obligation, that is, the axioms needed for the proof. We identify the phenomenon of *lurking axioms*, which must be included in the essence to ensure that an SMT solver can reconstruct the proof from the essence.
- We introduce *proof localization*, a simple semantics-preserving translation that embeds the proof essence for each assertion *directly into the verified program*, such that verification conditions over the resulting program include, for each assertion, only the axioms strictly needed by the SMT solver. Our technique is applicable with all SMT-based verifiers.
- We have implemented our technique in a tool called Axolocl [16] for the Boogie intermediate language, which enables the optimization of all verifiers built on top of Boogie, including Dafny and Viper.
- We present an extensive evaluation of Axolocl on a diverse set of benchmarks from eight verification projects. Specifically, for long-running files, we show significant speedups of up to 100–1000 times and no slowdowns.

2 Overview

The positive effect of proof localization is best observed at scale, *i.e.*, on large programs with complex specifications. For the sake of clarity, in this section, we provide the intuition for the most likely sources of long proof runtimes and the ways we mitigate them via proof localization, by means of a simple example.

```
1  type Seq;
2  const Emp: Seq;
3  function Length(Seq): int;
4  function Append(Seq, Seq): Seq;
5  function NonEmpty(Seq): bool;
6
7  axiom (∀ l: Seq, r: Seq • {NonEmpty(Append(l, r))}
8    {Length(l), Length(r)}
9    NonEmpty(Append(l, r)) = (NonEmpty(l) ∨ NonEmpty(r)));
10
11 axiom (∀ s: Seq • {NonEmpty(s)} NonEmpty(s)=(1≤Length(s)));
12
13 procedure test(x: Seq, y: Seq)
14   returns (res: Seq)
15   requires NonEmpty(x) ∧ NonEmpty(y);
16   ensures NonEmpty(res); {
17   res := Append(x,y);
18   assert NonEmpty(res);
19 }
```

Fig. 1: An example Boogie program. The final assert statement is redundant and used for illustration purposes. The color-highlighted lines specify different matching patterns (triggers) for the ∀-quantifiers.

2.1 A Primer on SMT-based Verification

Our approach for proof localization works by performing a source-to-source transformation on verified programs written in the Boogie intermediate verification language [3]. Fig. 1 presents an example program written in Boogie. Let us ignore the highlighted parts, which will be explained later. A Boogie program can be broadly considered as being composed of three main components: firstly, a declaration context, consisting of a series of uninterpreted types (line 1), constants (line 2), and functions (lines 3–5); secondly, a series of axioms (lines 7–12) that provide an interpretation for the prior declarations; and finally a series of verified procedures (lines 14–19) each specified in terms of the previous definitions using a pre- (line 15) and postcondition (line 16), and whose body (lines 17–18) is written in imperative style. Our example procedure concatenates the input sequences `x` and `y` and returns the result. Its specification expresses that, provided that the input sequences are non-empty (as required by the precondition), the result will also be non-empty. The latter property is expressed as a postcondition and, for illustration purposes, also as an assertion in line 18. The postcondition follows from the precondition and the axiom in line 7. When run on this program, Boogie checks its correctness by computing a verification condition (VC) using weakest preconditions calculus and proves its validity using an SMT solver.

Let us delve a little into the internals of how Boogie's SMT-based verification works. In this discussion, we ignore the postcondition in line 16 because the same check is already expressed by the assertion in line 18. Ignoring various optimizations, Boogie computes a VC expressing that the assertion on line 18 holds for the result of the program, given the assumptions in the precondition:

```
BG ∧ NonEmpty(x) ∧ NonEmpty(y) ⇒ NonEmpty(Append(x,y))
```

In this VC, `BG` is the so-called background theory, which includes the axioms declared in our program and various built-in axioms, *e.g.*, about lists. To prove that this VC is valid, Boogie uses an SMT solver to show that its negation is unsatisfiable, that is, that there are no counterexamples to the VC's validity. To achieve this, the underlying SMT solver must instantiate the axiom in line 7. Boogie, like most program verifiers, rely on e-matching [12] to instantiate quantifiers. Each quantifier is associated with a *matching pattern* (often called *trigger*), a term using variables bound by the quantifier. When the SMT solver encounters a term that matches the trigger, it uses unification to instantiate the quantifier.

In our running example, Fig. 1, the blue areas highlight possible triggers for the two axioms. These are used by the SMT solver to guide instantiation based on the terms available in the context. In particular, the conclusion of the implication `NonEmpty(Append(x,y))` in the VC matches the trigger of the first axiom. This enables the SMT solver to instantiate the axiom with `x` for `l` and `y` for `r`, after which the proof can be completed using the obtained equality. The terms `NonEmpty(x)`, `NonEmpty(y)`, and `NonEmpty(Append(x,y))` in the VC all also match the trigger of the second axiom in line 11. However, *none* of these three instantiations are useful for the proof and correspond to redundant instantiations.

Triggers give program verifiers such as Boogie fine-grained control over the heuristics used by the underlying SMT solver to instantiate quantifiers. However, even with carefully-chosen triggers, the SMT solver will still often consider several redundant instantiations, and the search space remains vast. Verifying realistic VCs often involves hundreds or thousands of quantifier instantiations. Whether such SMT queries are solved efficiently, in particular, before a time-out occurs, largely depends on how effectively the SMT solver's heuristics navigate the search space of possible instantiations. Therefore, *reducing the search space of quantifier instantiations is key to improving the efficiency of SMT-based verification.* In the rest of this section, we will present the key ideas for achieving this goal.

2.2 Capturing the Essence of a Proof

To improve the efficiency of the prover, our main objective is to reduce the space of possible quantifier instantiations it has to make. We achieve this in two steps. First, we extract from the SMT solver which axioms it used to prove the validity of an assertion. Second, we rewrite the input program such that the VC for this assertion contains *exactly* the relevant axioms rather than the entire background theory. In the following, we explain these steps in the context of Boogie and its underlying SMT solver Z3, but our approach can be implemented in any verifier whose SMT solver supports triggers and UNSAT cores.

Extracting relevant axioms. To determine which axioms were used to verify an assertion, we instruct Boogie to compute a separate VC for each assertion (rather than a single VC for the entire procedure). Moreover, we assign a unique label to each axiom in the background theory. Recall that an SMT solver proves validity of a formula by showing that its negation is unsatisfiable. Consequently, when verification of an assertion succeeds, we can obtain the UNSAT core from the

SMT solver, which includes the facts used to prove validity, and, in particular, the (labels of) the axioms that were used in the proof. In our example from Fig. 1, the UNSAT core contains the first axiom. The second one might have been instantiated during the proof search. However, since these instantiations were not helpful for the proof, the axiom does not show up in the UNSAT core.

Axiom guarding. Once we have obtained the relevant axioms for each assertion, we transform the input program to ensure that future attempts to re-prove the assertion consider only the relevant axioms and, thus, have to explore a drastically reduced search space. We call this transformation *axiom guarding*.

As explained in Sec. 2.1, the VC for each assertion contains a background theory with all axioms. Conceptually, we would like to customize the background theory for each assertion to include only the relevant axiom. However, different background theories per assertion lead to a substantial overhead for the SMT solver (for instance, for parsing the different background theories). Therefore, instead, we use the same background theory for all assertions, but *guard* each axiom in a way that lets us enable and disable individual axioms for a proof.

In order to *guard* an axiom, we add another trigger to the axiom such that the axiom is instantiated *only* if a VC includes a term that matches this dedicated trigger. Fig. 2 shows the guarded version of the first axiom from Fig. 1. To obtain a trigger that is specific to this axiom, we first define a unique uninterpreted *guard function*, `ax1` (line 2) and then wrap the axiom in a second ∀-quantifier with the guard function as trigger (line 3). Now we can selectively enable the axiom by including the term `ax1(true)` in a VC. This will allow the SMT solver to instantiate the outer quantifier and get access to the original axiom. If no such term is present, the axiom is disabled.

```
1 // dedicated guard for axiom 1
2 function ax1(bool): bool;
3 axiom (∀ b • {ax1(b)}
4  (∀ l,r • {NonEmpty(Append(l,r))}
5   NonEmpty(Append(l, r)) =
6   (NonEmpty(l) ∨ NonEmpty(r))));
```

Fig. 2: Example guarded axiom.

To enable axioms selectively per assertion, we rewrite each assertion into a non-deterministic `if`-statement, where one branch proves the assertion and then stops verification, and the other branch simply assumes the asserted property and then proceeds to the subsequent statement. This allows us to enable axioms selectively in the first branch without polluting the proof of the subsequent code with unnecessary facts. Since a verifier considers both branches of an `if`-statement, this encoding both proves the assertion and verifies the subsequent code; verifying the first branch justifies the assumption in the second one.

```
1 if (*) {
2   assume ax1(true);
3   assert NonEmpty(res);
4   assume false;
5 }
6 assume NonEmpty(res)
```

Fig. 3: A transformed assertion.

Fig. 3 illustrates this translation for the assertion from the line 18 of Fig. 1. The first branch of the non-deterministic `if`-statement enables the axiom relevant for the assertion (that is, our first axiom), by mentioning the dedicated trigger `ax1` in an assume statement (line 2). After proving the original assertion (line 3), we

```
1  function ax1(bool): bool;
2  function ax2(bool): bool;
3
4  axiom (∀ b: bool • {ax1(b)}
5   (∀ l: Seq, r: Seq
6      • {NonEmpty(Append(l,r))}
7     NonEmpty(Append(l, r)) =
8     (NonEmpty(l)∨NonEmpty(r))));
9
10 axiom (∀ b: bool • {ax2(b)}
11  (∀ s: Seq • {NonEmpty(s)}
12   NonEmpty(s) = (1≤Length(s)));
13
14 procedure test(x: Seq, y: Seq)
15 returns (res: Seq)
16 requires NonEmpty(x) ∧
17          NonEmpty(y);
18 ensures NonEmpty(res); {
19  res := Append(x,y);
20  if (*) {
21    assume ax1(true);
22    assert NonEmpty(res);
23    assume false;
24  }
25  assume NonEmpty(res);
26 }
```

(a) An initial transformed version.

```
1  function ax1(bool): bool;
2  function ax2(bool): bool;
3
4  axiom (∀ b: bool • {ax1(b)}
5   (∀ l: Seq, r: Seq
6      • {Length(l), Length(r)}
7     NonEmpty(Append(l, r)) =
8     (NonEmpty(l)∨NonEmpty(r))));
9
10 axiom (∀ b: bool • {ax2(b)}
11  (∀ s: Seq • {NonEmpty(s)}
12   NonEmpty(s) = (1≤Length(s)));
13
14 procedure test(x: Seq, y: Seq)
15 returns (res: Seq)
16 requires NonEmpty(x) ∧
17          NonEmpty(y);
18 ensures NonEmpty(res); {
19  res := Append(x,y);
20  if (*) {
21    assume ax1(true)∧ax2(true);
22    assert NonEmpty(res);
23    assume false;
24  }
25  assume NonEmpty(res);
26 }
```

(b) The final version.

Fig. 4: Proof-localized example of the Boogie program from Fig. 1; the left version enables UNSAT core axioms only, the right one also includes lurking axioms.

kill off this branch by assuming false (line 4), which ensures that the subsequent code verifies trivially. In the second branch, we assume that the property holds (line 6) without introducing any axioms into the context and continue with the rest of the verification.

Proof localization. We apply the procedure described above to each proof obligation in a Boogie program. These include assert statements, procedure preconditions (at call sites), procedure postconditions (at the end of the procedure body), and loop invariants (before the loop and at the end of the loop body). Boogie internally makes all of these proof obligations explicit as assert statements (just like our assertion in line 18 makes the proof obligation for the postcondition explicit), which we then transform. Fig. 4a presents a transformed version of the example from Fig. 1 (with the blue triggers) using the localization procedure described so far. Our transformation effectively removes the second axiom from the search space for the assertion. Note that our proof-localizing transformation is able to remove this axiom even though it syntactically seems relevant since both the trigger and the quantified assertion share terms with the assertion. This *fine-grained* pruning of the search space is important for realistic examples, where many axioms are syntactically related.

2.3 The Missing Piece: Lurking Axioms

Unfortunately, the story does not yet end here. To see why, consider the variation of the example from Fig. 1, but this time with the orange trigger for the first axiom instead of the blue (and still the blue trigger for the second axiom). The orange trigger is contrived in this example, but illustrates a problem that occurs frequently, as we demonstrate in our evaluation.

The program with the orange trigger from Fig. 1 verifies successfully. However, when we apply the localization procedure described so far, the resulting program fails to verify! As it turns out, successful verification requires an entire class of additional axioms that are not included in the UNSAT core and, thus, are not enabled in our transformation from Sec. 2.2. Next, we explain how such hidden dependencies can arise, and present a systematic solution for identifying them.

The transformed program enables the first axiom, which is *logically* sufficient to prove the assertion. However, with the orange trigger, the SMT solver will not instantiate this axiom during the proof search because it does not encounter any term that matches the orange trigger (`Length` does not occur in the proof obligation). Consequently, the proof fails. In contrast, the non-transformed program verifies because the SMT solver will instantiate the second axiom with both `x` and `y`, which produces the `Length` terms to instantiate the first axiom. This shows that even though the second axiom is logically irrelevant and, thus, not included in the UNSAT core, it is *essential* for the SMT solver to find the proof. We call such axioms *lurking axioms*. Therefore, we must extend our proof essence to include not only the axioms from the UNSAT core, but also these lurking axioms. We identify them by extracting from the debug output of Z3 an *instantiation graph* that reflects the dependencies between quantifier instantiations [4].

Fig. 5 shows the instantiation graph for our example. The white boxes are terms from the solver's context, such as `NonEmpty(x)` or `NonEmpty(y)`. The rounded colored boxes depict axiom instantiations, *e.g.*, `ax2 @ x` is the instantiation of the second axiom with the expression `x`. We use blue for axioms in the UNSAT core and red for all other axioms. Arrows between the nodes capture dependencies between terms and axioms: the line between `NonEmpty(x)` and `ax2 @ x` means that during the proof search, the solver used the term `NonEmpty(x)` to match the trigger for `ax2` and instantiate it. By examining this graph, we can see that for the solver to instantiate the necessary axiom (`ax1` with `x` and `y`) in the actual proof search, it first instantiated `ax2` with both `x` and `y` to obtain the terms `Length(x)` and `Length(y)`, thereby triggering `ax1`.

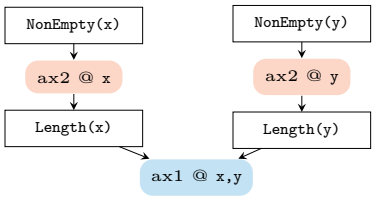

Fig. 5: Instantiation graph for Fig. 1 (with the orange trigger for axiom 1).

We identify lurking axioms by inspecting each path from a root of the instantiation graph to an axiom in the UNSAT core. All axioms on this path that are not in the UNSAT core are considered lurking axioms. The *proof essence* for an assertion consists of all axioms in the UNSAT core plus all lurking axioms,

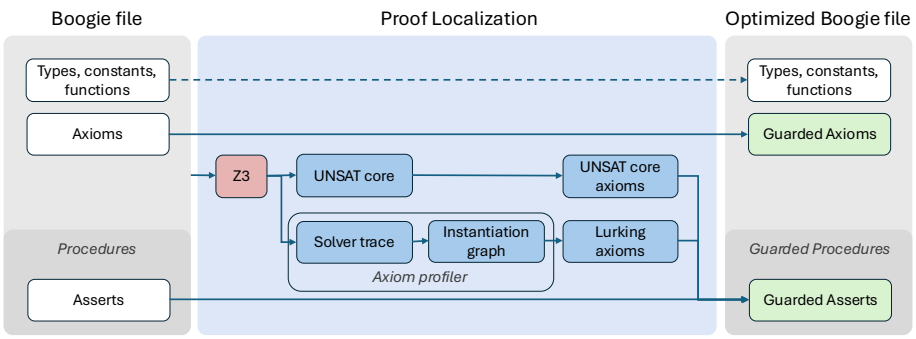

Fig. 6: Overview of the approach. Boxes with white background are user-provided components; the gray background indicates Boogie source files; boxes with green backgrounds are generated components; blue boxes depict intermediate artifacts; the red box indicates the external executable Z3. The dashed arrow indicates that the components are copied verbatim from the old to the new program.

and we use all of them in our proof localization. In our example, the proof essence contains both axioms, the first one because it is in the UNSAT core and the second because it is a lurking axiom. Consequently, the proof-localized program, shown in Fig. 4b, enables both axioms and thus verifies successfully. So for the orange trigger, proof localization does not actually reduce the search space for the SMT solver. However, as we discussed above, this trigger is contrived; in a realistic example, the transformation effectively reduces the search space and, thereby, proof search runtime, as we will show in our evaluation.

2.4 Putting It All Together

Fig. 6 presents the high-level overview of our approach, which integrates the proof localization process presented throughout this section behind a push-button interface. Our tool, Axolocl, takes as input any standard Boogie file, and produces as output a proof-localized version where all the axioms and assertions have been guarded in order to improve its stability. Axolocl does so in four steps: (1) It copies the declaration context, consisting of the types, constants, and functions, over to the new file. (2) It performs a straightforward syntactic translation to wrap each axiom into a quantifier with a unique trigger. (3) It verifies each assertion in the original program, and obtains from the Z3 SMT solver both the UNSAT core and a solver trace, from which it extracts the instantiation graph for the proof search. Combining these two components, for each assertion, provides the proof essence. (4) It rewrites each of the assertions in the original program to enable exactly the axioms in its proof essence. The resulting proof-localized program is semantically equivalent to the input program, contains all information to verify successfully, and its proof is typically substantially faster.

ALGORITHM 3.1: Proof Localization

```
Procedure Localize-Proofs(P)
    Input: Original Boogie program P
    Output: Transformed Boogie program with localized proofs
    C ← ExtractDeclContext(P)
    A ← BuildGuardedAxioms(P)
    I ← [ ]
    for proc in P do
        essence ← [ ]
        for a in SplitVCByAsserts(proc) do
            q ← ConstructSMTQuery(a)
            c, t ← VerifyWithZ3(q)
            g ← ExtractInstGraph(t)
            c′ ← ComputeEssence(c, g)
            essence[a] ← c′
        proc′ ← BuildGuardedProc(proc, essence)
        I ← I + [proc′]
    P′ ← BuildBoogieProgram(C, A, I)
    return P′
```

3 Algorithms and Encoding

Our tool, Axolocl, implements proof localization in a fork of Boogie version 3.0.12.0, vendoring a modified version of Becker *et al.*'s Axiom Profiler [4] to generate instantiation graphs for inferring lurking axioms. In this section, we present Axolocl's main algorithms and various alternative encoding schemes.

3.1 Localizing Boogie Proofs

Algorithm 3.1 is the core of Axolocl's proof localization. It takes as input an arbitrary Boogie program P and produces an optimized version as depicted in Fig. 6. The algorithm operates in three steps: (1) it retrieves the declaration context (`ExtractDeclContext`) composed of all the types, constants, and functions shared between the original and optimized program; (2) it then constructs guarded forms of each axiom in P (`BuildGuardedAxioms`), generating a unique guard function and rewriting the axiom body to wrap it with a trigger-based guard; (3) it performs proof localization on each of the procedures in P.

For the latter, the algorithm partitions each procedure into the verification conditions for each assertion in the code (`SplitVCByAsserts`). For each of these verification conditions, the algorithm constructs an SMT query (`ConstructSMTQuery`) including appropriate annotations to allow extracting the axiom dependencies. It is submitted to the Z3 SMT solver (`VerifyWithZ3`), to obtain both an UNSAT core c and a solver debug trace t, which contains a log of each of the axiom instantiations that the solver performed during the proof search. This solver trace is then used to construct an instantiation graph g (`ExtractInstGraph`), capturing the dependencies between each of the axioms in the query. The algorithm uses

ALGORITHM 3.2: Compute Proof Essence

```
Procedure ComputeEssence(C, G)
    Input: UNSAT Core C and Instantiation Graph G
    Output: Set of Required Axioms
    C' ← [ ]
    for a in C do
        deps ← {a}
        s ← [a]
        while s ≠ [ ] do
            n, s ← TakeFirst(s)
            deps ← deps ∪ {n}
            for n' ← GetAxiomDependencies(G,n) do
                if n' ∉ deps then
                    s ← Append(s, n')
        C' ← C' + deps
    return C'
```

the UNSAT core c and the instantiation graph g to infer any missing lurking axioms and constructs the minimal set of required axioms c' to complete the proof (`ComputeEssence`). Once this proof essence has been constructed for all verification conditions, the algorithm optimizes the original Boogie procedure (`BuildGuardedProc`), replacing each assertion with a guarded version that explicitly enables the axioms in the proof essence. The last step of the algorithm concatenates the declaration context C, the guarded axioms A, and the optimized procedures I to produce the final optimized Boogie program.

3.2 Identifying Lurking Axioms

Algorithm 3.2 determines the necessary set of axioms for a verification query. It takes as input an UNSAT core C and the instantiation graph G generated by the solver for a given SMT query. It then iterates through each of the axioms in the UNSAT core, and, for each axiom a, performs a basic breadth first search over the instantiation graph rooted at a to retrieve the set of dependencies of a. We allocate a list of nodes s to visit, initially populated by a, and a set *deps* to track the set of lurking axioms required for a. While s is non-empty, the algorithm repeatedly takes the first element n, adds it to the list of dependencies and uses the instantiation graph to retrieve the list n' of axioms that were required for n to be instantiated. If any of these dependencies are not present in *deps* then they are added to s to be visited. Finally, the algorithm takes the union of all the axiom dependencies for each axiom in the UNSAT core (including the core axiom itself) to produce the final proof essence.

3.3 Encoding Schemes

After extracting the proof essence for each assertion, Axolocl rewrites the program into a guarded form to improve verification times. One subtle aspect of our design

```
function
   ax(bool): bool;
axiom
(∀ b: bool • {ax(b)} A);

procedure foo(x) {
 if(*) {
  assume ax(true);
  assert P;
  assume false;
 }
 assume P;
 ...
}
```

(a) Trigger-based (fine)

```
function
   ax(bool): bool;
axiom
(∀ b: bool • {ax(b)} A);

procedure foo(x) {
 assume ax;
 ...
 assert P;
 ...


}
```

(b) Trigger-based (coarse)

```
procedure foo(x) {
 if(*) {
  axiom A;
  assert P;
  assume false;
 }
 assume P;
 . ..
}
```

(c) Inlining-based

Fig. 7: Comparison of different guarded proof encoding schemes.

is in the particular trigger-based encoding that we use to localize proofs. Given a proof essence, there are several possible encodings that we considered in order to enable these axioms in the source program. As we will show in the evaluation (Sec. 3.3), the choice of encoding has a major effect on verification times.

Fig. 7 presents three different encoding schemes that we considered for guarding axioms and restricting the proof context at the source level. Fig. 7a presents the trigger-based encoding used by our implementation that constrains the proof context at a fine-grained per-assertion level. We introduce a unique guard function `ax` for each axiom `A`, and move the axiom under a universal quantifier over a Boolean variable `b` with a trigger using that guard `{ax(b)}`. At each assertion, we introduce branching; one branch selectively enables the relevant axioms by assuming their guards `ax(true)` and then verifies the assertion. The other branch continues with the rest of the verification assuming the assertion. This encoding allows rewriting programs with minimal changes and achieves the largest performance improvements in our evaluation.

Fig. 7b presents an alternative, coarse-grained trigger-based encoding. It performs the same transformation on each of the axioms, but instead of rewriting each assertion individually, it triggers all axioms needed for the entire procedure at the beginning of the procedure body. In other words, this encoding localizes proofs at the procedure (rather than assertion) level. Compared to the previous fine-grained encoding, the coarse-grained encoding leads to simpler verification conditions because it does not complicate the control-flow of the programs. It also restricts the proof search space in comparison to the original program, but not as effectively as the fine-grained encoding. As we will see in the evaluation, the benefit of fine-grained proof localization outweighs the increased complexity of the verification conditions.

Fig. 7c presents an alternative encoding based on inlining axioms rather than disabling them via triggers. With this encoding, we remove all axioms from the global context, and then instead, at each assertion, we selectively *assume* the (quantified) body of each axiom in the proof essence. This encoding is as

precise as our first encoding. It avoids the overhead of guarding each axiom, but duplicates axiom bodies at each assertion. For most real-world Boogie programs that we tried it on, it produced results that are too large to even fit into memory.

For our trigger-based encodings, we also considered adding the guard as an additional pattern to the triggers of the existing axioms, rather than wrapping them in an additional universal quantifier:

```
axiom (∀ s: Seq, b: bool • {Length(s), ax(b)}  1 ≤ Length(s) );
```

This encoding reduces the total number of universal quantifiers, but, as it turns out, does not improve verification times.

4 Evaluation

We evaluate Axolocl on a broad range of benchmarks. Most important is its effectiveness in reducing verification time, but we also assess whether the achieved runtime improvements can be expected to be maintained across changes as the verification project. Measuring the latter on real projects would require detailed information about revisions, manual performance optimizations, and updates of the verifier and SMT solver. Since we do not have access to this information, we use robustness against semantics-preserving mutations as proxy; prior work has shown that such mutations can already perturb verification times substantially [13]. Finally, we also compare the different proof localization strategies from Sec. 3.3. Specifically, we answer the following research questions:

- **RQ1**: Is Axolocl effective at reducing verification times?
- **RQ2**: Are these improvements robust to mutations of the input?
- **RQ3**: Is Axolocl's encoding for localizing axioms effective?

4.1 Evaluation Setup

We explain the benchmark set used and the evaluation methodology.

Benchmarks. To answer the above research questions, we apply Axolocl to a suite of Boogie programs, all extracted from existing verification projects. Specifically, we consider eight benchmark sets, seven of which are publicly-available open-source Dafny projects. The last one is based on the Viper test suite. We chose these benchmarks for their size and to assess the effectiveness of Axolocl on a diverse set of verified systems. The resulting benchmark suite consists of:

- **Cedar**: The Dafny formalization of the Cedar authorization language [9]
- **AWS**: The AWS Cryptographic Material providers library [1]
- **Daisy-NFSD**: The Daisy-NFSD verified crash-safe file system [7]
- **EVM**: An EVM disassembler in Dafny [6]
- **Dafny-VMC**: A verified library for Monte Carlo algorithms [32]
- **Komodo**: An implementation of an SGX-like enclave protection model in a formally verified privileged software stack for ARMv7 TrustZone [17]
- **VeriBetrKV**: A formally verified key-value store based on a B^eTree [14]
- **Viper**: Various test cases from the Viper program verifier's test suite

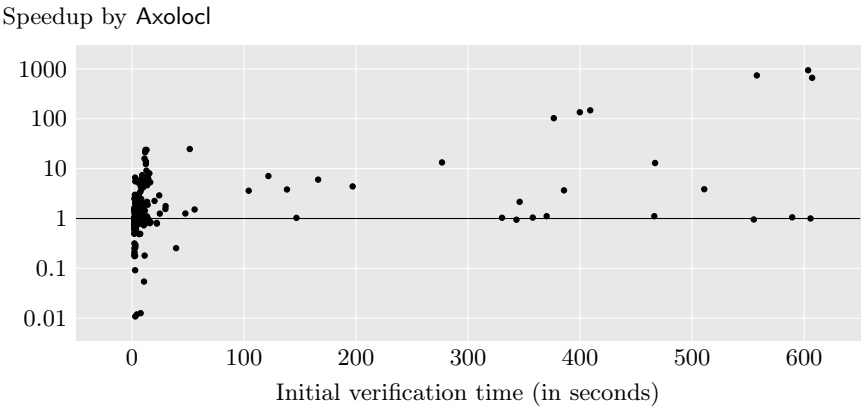

Fig. 8: Average speedup per Axolocl-optimized file as a function of the initial verification time over our 365 benchmark files. The y axis is logarithmic.

Methodology. We use Axolocl to optimize each Boogie file from the benchmark suite and compare the verification times before and after proof localization. All experiments are conducted on a 2-vCPU Intel Xeon Platinum 8000 series processor (up to 3.6 GHz) running Ubuntu 22.04. We use Z3 version 4.8.7.

Since verification times are dominated by SMT solving, we measure the runtime at the SMT level rather than at the Boogie level. For each Boogie file, we measure the wall-clock time spent in the SMT solver, averaged over 50 runs, in which we alter the Z3 random seed to account for different Z3-heuristic behaviors. The timeout for each verification run is set to ten minutes.

Axolocl is designed to accelerate verification on slow inputs. Therefore, we discard Boogie files for which the mean verification time is at most two seconds. These are fast already, such that further optimization is not justified. This leaves us with 365 Boogie files across our eight projects; see Table 1 for more details.

4.2 Runtime Improvement (RQ1)

Our first research question investigates whether our tool is effective at reducing verification times. Fig. 8 plots the speedup (verification time before divided by verification time after) achieved by Axolocl versus the original verification time for each file. A speedup of less than one is thus a slowdown. We first observe that Axolocl achieves a speedup in the majority of cases. Importantly, slowdown occur only for files whose verification is very fast to begin with, with one exception that takes around 35 seconds and suffers a 4× slowdown. Conversely, for slow files beyond that, Axolocl yields very significant speedup of up to a 1000× and, equally important, never incurs a slowdown.

Tab. 1 presents the aggregate effect of Axolocl on the verification times across the eight projects in our benchmark suite. It shows the number of Boogie files for each project, the total number of lines of code in the Boogie files, the mean verification time for the original files, as well as the total verification time for all

Table 1: Effect of Axolocl on verification times. All times in seconds.

Suite	Files	Total LoC	Mean time	Total Time		Speedup
				Before	After	
AWS	23	1,264,497	16	370	380	1.0
Dafny-VMC	1	10,916	2	2	2	1.0
Cedar	15	1,198,291	64	967	349	2.8
Daisy-NFSD	8	109,801	7	69	50	1.4
EVM	16	548,790	9	141	131	1.1
Viper	17	63,790	181	3,070	711	4.3
Komodo	5	26,371	9	45	15	3.0
VeriBetrKV	280	6,718,495	23	6,569	5,757	1.1

Table 2: Effect of Axolocl on quantifier instantiations (QIs).

Suite	Mean QIs	Total QIs		Reduction factor
		Before	After	
AWS	41,930	2,138,465	664,442	3.2
Dafny-VMC	4,796	47,960	21,623	2.2
Cedar	812,507	26,000,237	3,792,461	6.9
Daisy-NFSD	36,956	628,261	203,276	3.1
EVM	45,121	1,308,522	185,303	7.1
Viper	540,591	6,487,097	3,243,952	2.0
Komodo	677	74,567	13,406	5.6
VeriBetrKV	6,716	4,996,978	1,272,005	3.9

files in the project, before and after using Axolocl. The speedup obtained for this total time is shown in the last column and reflects what would be experienced by a developer working on the project or in CI. Except for AWS, Axolocl reduces the overall verification time by up to 4.3 times for the Viper test suite. Among the considered projects, the AWS codebase is the only one that is actively maintained and has verification integrated as part of its CI pipeline; therefore, we hypothesize that it already underwent considerable manual optimization.

Tab. 2 shows the impact of Axolocl on the number of quantifier instantiations (QIs) across the same set of projects. For each test case, we measured the number of QIs over multiple runs and computed the average to mitigate the effects of proof instability. The table reports the mean number of QIs across the entire test suite before applying proof localization. Unlike earlier experiments, we include all files in this analysis, regardless of whether their verification time was below two seconds. The table also reports the total number of QIs before and after proof localization, as well as the reduction factor for each project. Proof localization significantly reduces the number of QIs, providing evidence that the observed performance improvements indeed stem from a decrease in quantifier instantiations.

We conclude that Axolocl is a viable solution to improve verification runtimes for slow files and for entire large-scale projects.

4.3 Robustness to Mutation (RQ2)

Improving verification times alone does not tell the complete picture: if a program (as well as the employed verifier and SMT solver) remained entirely unchanged, then a developer could improve verification times by simple caching. Our second research question investigates whether the performance improvements achieved by Axolocl are likely to persist across changes in the project. We focus on the simplest kind of such changes, semantic-preserving re-orderings and renamings of variables and declarations, and investigate how the generated files' verification times behave with respect to them. Such simple mutations can have a substantial effect on verification times; therefore, they are a good proxy for other changes.

Concretely, we measure the verification times of each Boogie file before and after proof localization, averaged across several runs with random mutations applied at the SMT level. We consider two kinds of mutations: renaming user-defined symbols (*e.g.*, functions, types) and reordering assertions. We then calculate the mean verification time over 50 measurements while considering both mutations and variations in the random seed. The results of this evaluation are presented in Fig. 9, which plots the speedups against the initial mean verification time over 50 mutations.

We can see from this diagram that mutations have a significantly greater impact on verification time than changes in the verification seed as considered in Sec. 4.2. For reference, mutations increase the mean verification time of 16 out of the 20 files in the Viper test suite. On average, each Viper file takes 42.59 seconds longer to verify under mutation.

Due to the increased average verification time, more files *excluded* in Sec. 4.2 for being verified in less than 2 seconds are *included* in this experiment. Since proof-localization is generally less effective for fast queries with few quantifier instantiations, the inclusion of these queries leads to a lower average speedup across all files.

Despite the additional variability introduced by mutation, the general trends from Fig. 8 still persist, especially among the slowest files.

We conclude that Axolocl achieves substantial performance improvements for files with long verification times, even under mutation.

4.4 Encoding Effectiveness (RQ3)

In the final experiment, we assess the benefit of the chosen fine-grained trigger-based encoding in Axolocl, compared to the alternatives explained in Sec. 3.3:

- **Trigger (fine)**: Axolocl's default encoding strategy (see Fig. 7a).
- **Trigger (coarse)**: Enabling all axioms at a procedure level (see Fig. 7b).
- **Trigger (embedded)**: Embedding the axiom guard within existing triggers for the axiom.

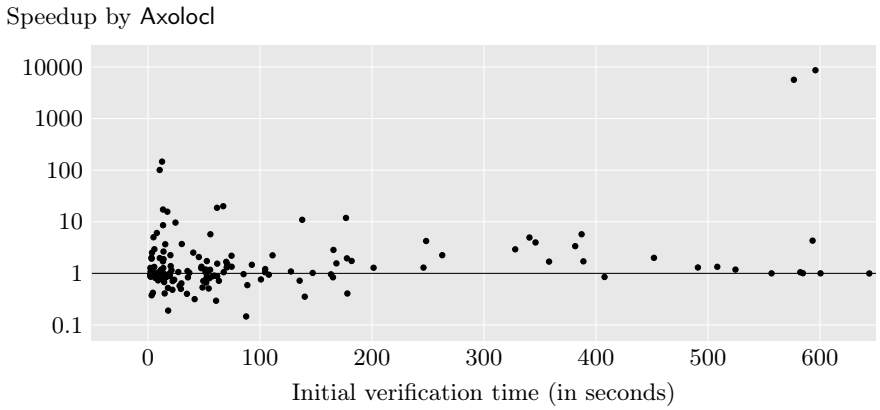

Fig. 9: Average speedup of Axolocl-optimized files under mutations as a function of the initial verification time, over 365 benchmark files. The y axis is logarithmic.

Table 3: Comparison of different encoding strategies.

Encoding	Speedup
Trigger (fine)	4.8
Trigger (coarse)	0.6
Trigger (nested)	0.4
Inline	0.4

– **Inline**: Inlining axiom bodies at each assertion (see Fig. 7c).

For this research question, we focus on the Viper suite of benchmarks specifically as it contains the files with the largest verification times, and on which Axolocl exhibits the greatest speedups (see Tab. 1).

We run Axolocl to generate optimized versions of each of the Viper files in the benchmark suite for each encoding scheme, and then measure the verification times averaged over 25 runs. Tab. 3 presents total speedups achieved for each different encoding scheme as before. It shows that our chosen fine-grained encoding outperforms all other encodings by far, and is the only scheme which produces improvements in verification times. The slowdown caused by the coarse-grained trigger-based encoding demonstrates the importance of constraining axioms at a per-assertion level. The speedup achieved for Viper is slightly different from the speedup reported in Tab. 1 primarily because one of the files couldn't be transformed for the inlining encoding. As a result, the speedups were calculated only for the remaining files.

5 Related Work

Performance problems in SMT-based deductive verifiers are typically addressed by manual code refactorings using three different strategies. (1) Modularization by breaking up the verification of a method into several smaller proofs reduces the number of terms in the proof context and, thereby, quantifier instantiations, leading to better performance. Some verifiers offer language support for this strategy, for instance, Dafny's `opaque` blocks, Gobra's `outline` statement [31] allow one to specify code blocks with pre- and postconditions and verify them separately from the enclosing code, as if they had been extracted into separate methods. Similarly, Ivy's [26] `isolate`s achieve the same goal in the context of automatically verifying safety properties of state-transition systems, providing a scoping mechanism to confine a logical fragment or theory that a prover can handle reliably and efficiently.[4] (2) Carefully controlling the availability of quantified assertions also reduces the number of instantiations. For instance, Dafny and Verus allow for hiding the quantified axiom by defining a specification function and revealing it selectively. (3) Several verifiers allow the addition of proof hints to assertions, for instance to invoke lemmas or reveal definitions. Both Dafny and Verus offer assert-by statements, whose encoding ensures that the provided proof hints do not pollute the proof context of other assertions. All three approaches require substantial manual effort. Moreover, they are difficult to apply, even for expert users, because the effect of a code refactoring on verification times is unpredictable due to the SMT solver's complex heuristics and thus its inherent black-box character. In contrast, our proof localization technique is completely automatic and requires no expertise in SMT solving.

Bordis and Leino [5] aim at reducing the number of quantified axioms and, thereby, the number of quantifier instantiations by automatically pre-instantiating some quantified axioms based on syntactic clues. However, their syntactic analysis cannot reliably identify all instantiations needed for a proof. In contrast, our technique starts from an existing proof, from which we can extract all necessary axioms, including lurking axioms.

The Axiom Profiler [4] supports developers and users of automated verifiers in understanding and debugging performance and completeness issues. It offers visualizations and analyses of SMT traces to identify issues with quantifier instantiations such as matching loops. This tool is useful to improve the matching patterns of quantifiers, but does not address the general problem of improving verification performance. We use it in our technique to identify lurking axioms.

Qed [8] simplifies verification conditions before handing them to the SMT solver, which reduces the proof context and, therefore, might have a positive effect on verification time, but this is not assessed in their evaluation.

A lot of work on the performance of SMT-based verifiers targets proof instability (also called *brittleness*) [13], which occurs when small changes in the SMT query have large effects on the verification time or even verification success. Most research into a systematic solution focuses on techniques to characterize

[4] *Cf.* https://microsoft.github.io/ivy/proving.html.

particular instances of instability [34,24] or develops strategies for mitigating the instability in specific cases [22,10,19,23,29]. The recent Shake tool [33] offers an approach to mitigating proof instability. For an already verified project, it intercepts SMT queries emitted by verification tools and dynamically rewrites them to simplify and constrain the verification context, omitting irrelevant axioms and assumptions that are not required for the solver to prove correctness. Similarly to our algorithm, Shake aims at reducing the size of the proof context. Unlike our algorithm, it does not identify lurking axioms and instead relies on a *syntactic* dependency analysis to determine which axioms *might* be relevant. This coarse analysis may include axioms that are not actually needed, making the context (and, thus, the search space for the SMT solver) larger than necessary. On the other hand, it may also miss lurking axioms; as a result, the SMT solver may fail to prove the rewritten queries, leading to spurious verification errors requiring expensive post-hoc repair. Zhou *et al.* demonstrate Shake's effectiveness in reducing proof instability, but do not demonstrate performance improvements. Moreover, Shake does not offer a way to *persist* the rewritten queries, so that the tool needs to be rerun even at the slightest change in the file being verified. By contrast, our proof localization determines precisely which axioms are needed to rerun a proof and can record this information for future runs.

6 Conclusion

In this work we have presented Axolocl, a tool for the systematic performance optimizations of SMT-based program verifiers. Given an initial successful verification, it computes for each assertion the proof essence and encodes it into the input program, such that the assertion can be proved more efficiently in the future, for instance, during code maintenance or as part of continuous integration.

As future work, we plan to integrate proof localization in source verifiers, for instance, using Dafny's or Verus' assert-by statements. We also plan to explore whether proof localization can be applied also to address proof instability.

Acknowledgements. We thank CAV'25 reviewers for their detailed and insightful comments. This work has been supported by a Singapore Ministry of Education (MoE) Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems" and Singapore MoE Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

Disclosure of Interests The authors have no further competing interests to declare that are relevant to the content of this article.

References

1. AWS: AWS Cryptographic Material Providers Library (2024), https://github.com/aws/aws-cryptographic-material-providers-library, [Online; accessed 8. Sept. 2024]

2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: TACAS. LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: FMCO. LNCS, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
4. Becker, N., Müller, P., Summers, A.J.: The axiom profiler: Understanding and debugging SMT quantifier instantiations. In: TACAS. LNCS, vol. 11427, pp. 99–116. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_6
5. Bordis, T., Leino, K.R.M.: Free facts: An alternative to inefficient axioms in dafny. In: FM. LNCS, vol. 14933, pp. 151–169. Springer (2024). https://doi.org/10.1007/978-3-031-71162-6_8
6. Cassez, F.: evm-dis: An EVM bytecode disassembler/assembler (Nov 2024), https://github.com/franck44/evm-dis, [Online; accessed 1. Nov. 2024]
7. Chajed, T., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Verifying concurrent, crash-safe systems with Perennial. In: SOSP. pp. 243–258. ACM (2019). https://doi.org/10.1145/3341301.3359632
8. Correnson, L.: Qed. computing what remains to be proved. In: NASA Formal Methods. LNCS, vol. 8430, pp. 215–229. Springer (2014). https://doi.org/10.1007/978-3-319-06200-6_17
9. Cutler, J.W., Disselkoen, C., Eline, A., He, S., Headley, K., Hicks, M., Hietala, K., Ioannidis, E., Kastner, J.H., Mamat, A., McAdams, D., McCutchen, M., Rungta, N., Torlak, E., Wells, A.: Cedar: A new language for expressive, fast, safe, and analyzable authorization. Proc. ACM Program. Lang. **8**(OOPSLA), 670–697 (2024). https://doi.org/10.1145/3649835
10. Cutler, J.W., Torlak, E., Hicks, M.: Improving the stability of type soundness proofs in Dafny. In: Proceedings of the First Workshop on Dafny (2024)
11. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102
13. Dodds, M.: Formally verifying industry cryptography. IEEE Security & Privacy **20**(3), 65–70 (2022). https://doi.org/10.1109/MSEC.2022.3153035
14. Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B.: Komodo: Using verification to disentangle secure-enclave hardware from software. In: SOSP. pp. 287–305. ACM (2017). https://doi.org/10.1145/3132747.3132782
15. Filliâtre, J., Paskevich, A.: Why3 - Where Programs Meet Provers. In: ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
16. Gopinathan, K., Spiliopoulos, D., Goyal, V., Müller, P., Püschel, M., Sergey, I.: Axolocl: Accelerating Automated Program Verifiers by Automatic Proof Localization. Software Artifact. (Apr 2025). https://doi.org/10.5281/zenodo.15201479
17. Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., Parno, B.: Storage systems are distributed systems (so verify them that way!). In: OSDI. pp. 99–115. USENIX Association (2020), https://www.usenix.org/conference/osdi20/presentation/hance

18. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: SOSP. pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428
19. Ho, S., Pit-Claudel, C.: Incremental Proof Development in Dafny with Module-Based Induction. In: Proceedings of the First Workshop on Dafny (January 2024)
20. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A Practical Foundation for Systems Verification. In: SOSP. pp. 438–454. ACM (2024). https://doi.org/10.1145/3694715.3695952
21. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR. LNCS, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
22. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: CAV. LNCS, vol. 9779, pp. 361–381. Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_20
23. McLaughlin, S., Jaloyan, G.A., Xiang, T., Rabe, F.: Enhancing Proof Stability. In: Proceedings of the First Workshop on Dafny (January 2024)
24. Mugnier, E., McLaughlin, S., Tomb, A.: Portfolio Solving for Dafny. In: Proceedings of the First Workshop on Dafny (January 2024)
25. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: VMCAI. LNCS, vol. 9583, pp. 41–62. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
26. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI. pp. 614–630. ACM (2016). https://doi.org/10.1145/2908080.2908118
27. Pereira, J.C., Klenze, T., Giampietro, S., Limbeck, M., Spiliopoulos, D., Wolf, F.A., Eilers, M., Sprenger, C., Basin, D., Müller, P., Perrig, A.: Protocols to code: Formal verification of a next-generation internet router (2024), https://arxiv.org/abs/2405.06074
28. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., Kulatova, N., Ramananandro, T., Rastogi, A., Swamy, N., Wintersteiger, C.M., Béguelin, S.Z.: EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In: S&P. pp. 983–1002. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00114
29. Srinivasan, P., Padon, O., Howell, J., Lattuada, A.: Domesticating automation. In: Proceedings of the First Workshop on Dafny (January 2024)
30. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F*. In: POPL. pp. 256–270. ACM (2016). https://doi.org/10.1145/2837614.2837655
31. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular Specification and Verification of Go Programs. In: CAV. LNCS, vol. 12759, pp. 367–379. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_17
32. Zaiser, F., Zetzsche, S., Tristan, J.B.: VMC: a Dafny Library for Verified Monte Carlo Algorithms. In: Proceedings of the First Workshop on Dafny (January 2024)
33. Zhou, Y., Bosamiya, J., Li, J., Heule, M.J., Parno, B.: Context Pruning for More Robust SMT-based Program Verification. In: FMCAD. pp. 59–69. TU Wien Academic Press, IEEE (2024)
34. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: FMCAD 2023. pp. 178–188. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_26