Fifteen Years of Viper

Marco Eilers[1][0000−0003−4891−6950], Malte Schwerhoff[1][0000−0003−2569−9121], Alexander J. Summers[2][0000−0001−5554−9381], and Peter Müller[1][0000−0001−7001−2566]

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland
{marco.eilers, malte.schwerhoff, peter.mueller}@inf.ethz.ch
[2] Department of Computer Science, University of British Columbia, Canada
alex.summers@ubc.ca

Abstract. Viper is a verification infrastructure that facilitates the development of automated verifiers based on separation logic. Viper consists of the Viper intermediate language and two backend verifiers based on symbolic execution and verification condition generation, respectively. It has been used to build over a dozen program verifiers that translate verification problems in Go, Java, Python, Rust, and many others, into the Viper language and automate verification using the Viper backends. In this paper, we describe the original design goals for Viper's language, verification logic, and tool architecture, summarize our experiences, and explain our principles for evolving the system.

Keywords: Intermediate verification language, separation logic, verification structure, automated reasoning

1 Introduction

Automated program verifiers are often organized into a frontend, which translates verification problems into an intermediate verification language (IVL), and a backend, which extracts proof obligations from the IVL program and uses an SMT solver to discharge them. For instance, Corral [33], Dafny [35], Spec# [6], and SymDiff [32] are based on the Boogie IVL [34], whereas Frama-C [12] and Creusot [17] use WhyML [27]. This architecture allows backends to be reused across many different verifiers, such that sophisticated proof search algorithms, inference, error reporting, etc. do not have to be re-implemented for each tool.

The main goal of the Viper [41] project is to bring these benefits to the realm of separation logic verifiers, for which Boogie and WhyML provide no dedicated support. Fifteen years after starting the Viper project, we have by and large achieved this goal; over a dozen frontends have been built on top of Viper [3,9,10,13,21,23,28,49,50,57,59,60,61], several complex verification projects apply Viper-based tools [2,47], and Viper has been used successfully in teaching.

To make Viper an attractive IVL for separation logic verifiers, we have prioritized the following three design goals:

- *Expressiveness:* Viper can capture a wide range of programs and properties, as well as the proof principles of a wide variety of different separation logics.
- *Soundness:* Viper is designed to be sound, that is, successful verification implies that the input program always satisfies its specifications. Viper does not aim to be complete in general (as is standard for SMT-based verifiers), but does aim for a clear definition of which questions it *should* be expected to answer automatically (and conversely, where annotations are expected).
- *Usability:* Viper aims to offer a smooth user experience, most importantly, by (1) effectively automating separation logic proofs with modest user annotations, (2) being user-friendly for humans who manipulate Viper code directly without going through a frontend, and (3) providing helpful error messages when verification fails.

In this paper, we describe how these design goals are reflected in Viper's language, verification logic, and tool architecture, summarize our experiences, and explain our principles for evolving the system. We believe our observations can be transferred to other formal methods tools and will be useful for developers and maintainers of such tools.

2 The Viper Language

In this section, we discuss how the three design goals from the introduction are reflected in Viper's language design, and we explain our principles for extending the language.

2.1 Language Design

Expressiveness. To support the translation of diverse source languages, the Viper language offers a very permissive type system, imperative statements with structured and unstructured control flow (e.g. to encode languages with abrupt termination), as well as features to encode language features not directly supported by Viper, such as arrays.

Viper's assertion language supports a rich first-order separation logic (see Sec. 3.1 for details) including universal and existential quantification. Moreover, it allows defining custom datatypes, either as algebraic datatypes or as (interpreted or uninterpreted) sorts and functions.

A core virtue of Viper is its ability to encode the proof rules of a wide range of separation logics. To this end, Viper offers `inhale` and `exhale` statements. Inhaling an assertion P adds the separation logic resources described by P to the current state and assumes the value constraints in P. Conversely, exhaling P checks that its resources are available in the current state and removes them; moreover it asserts all value constraints. `inhale` and `exhale` are separation logic's analogs of `assume` and `assert`, and equally versatile in encoding

proof rules, as demonstrated by frontends for complex logics such as RSL [57] and TaDA [60].

Human-Friendliness. When designing an IVL, there is an inherent trade-off between keeping the language minimal (which simplifies both tool development and formalization) and adding richer language constructs to make the language more human-friendly. Following the examples of Boogie and WhyML, we designed Viper to be well-suited as both a target language for verification frontends *and* a language directly usable by humans. The latter has proven invaluable for prototyping new encoding schemes that a frontend might eventually automate, for debugging frontend-generated code, and for having raw access to Viper's features, for instance, in teaching and during verification competitions.

To support direct use, Viper's core language provides various constructs that do not increase expressiveness, but make the code more accessible. Examples include method calls (which, given Viper's modular verification could be encoded using `inhale` and `exhale`) as well as structured control flow with conditional statements and loops (which could be encoded using `goto`). Similarly, Viper's assertion language includes some technically-redundant connectives (e.g. includes disjunction as well as implication) to make formulas easier to understand. Moreover, macros greatly increase the readability of Viper code by providing concise notations for complex statements or assertions. These features not only improve usability, but also enable potential future Viper functionality, such as method inlining and specification inference techniques (e.g. for explicit loops). We have found that these benefits clearly outweigh their very modest development effort.

Error Reporting. Even though error reporting mostly concerns the verification logic and tool architecture, it also represents an important trade-off in terms of language design. In general, a tool that checks more properties (particularly well-definedness of expressions) by default is able to provide more-precise error messages, at the cost of increased verification effort. For instance, SMT solvers are based on a logic with total functions, where operations that are not meaningful, such as division by zero or out-of-bounds sequence access, yield unspecified results [7]. Even though Viper expressions and assertions are encoded into SMT, Viper instead uses a partial interpretation of these operations and imposes proof obligations to ensure that they are always applied within their domain. This choice enables more precise error reporting. For instance, for the assertion $x/y > 0$, Viper reports different error messages when y might be zero vs. when y is non-zero but the division might not yield a positive result. We have found that Viper's well-definedness checks for assertions have proven very useful both during the development of frontends and for direct uses of Viper, especially in teaching. However, they slightly increase verification time, and it would sometimes be useful to (selectively) disable well-definedness checks, for instance, when a frontend guarantees that they will always succeed.

2.2 Growing the Language

Increasing adoption of Viper as the IVL for diverse verifiers and as a teaching language continuously leads to new requirements. We add a new language feature only if it does not compromise our overarching goals (e.g. we know how to soundly automate verification) and at least one of the following criteria applies: (1) The new feature simplifies the development of multiple frontends, (2) it enhances the experience of manually writing Viper programs, or (3) it enables more efficient verification. Features in the first two categories are often implemented as plugins (cf. Sec. 4) that extend Viper's surface language but desugar into core Viper, so that backends need not be adapted. Features in the third category require changes to the backends to realize the intended performance improvement.

As an example for criterion (1), a common demand from frontends to support the verification of hyperproperties motivated us to implement a product program construction [24], initially needed by the Python frontend Nagini, in Viper [22]. It was subsequently used by other frontends [21,58]. However, there are also concepts that occur in many frontends, but are have not yet found their way into Viper. For instance, each occurrence of ghost code in a frontend has specific characteristics, and we were not yet able to devise unified support in the IVL.

As an example for criterion (2), we added (optional) support for proving termination. This feature is not required by frontends because termination checks can easily be encoded. However, such an encoding is tedious when using Viper directly because programmers not only have to insert the necessary assertions (which is error-prone) but also need to define appropriate well-founded orders. Built-in support for termination proofs is especially useful for teaching.

As an example for criterion (3), we added support for havocking parts of the heap, which can be handled much more efficiently than encodings that achieve the same result. This feature reduced the verification time of many Voila [60] examples by an order of magnitude.

When there has been tension between requirements for using Viper as an IVL and using it manually, we have tended to resolve it in favor of IVL usage. As a result, the Viper language does not directly offer some features (e.g. arrays, lemmas, and the aforementioned ghost code) that exist in verification languages for manual use, such as Dafny. In Viper, those features have to be encoded.

Possible future extensions include extended support for generic types, and modularization features that allow one to control dependencies between modules to improve verification time and facilitate the maintenance of verified code.

3 Verification Logic

As a deductive program verifier, Viper automates proof search in a program logic for the Viper language (by generating and checking logical conditions). In this section, we reflect on our choice of program logic, summarize our efforts to formalize it, and discuss our approach to increasing its expressiveness over time.

3.1 Implicit Dynamic Frames

The main application domain for Viper is the verification of imperative, concurrent programs, which makes separation logic [51,42] a fairly obvious choice for Viper's program logic. However, even at the time when the work on Viper started, there were dozens of different separation logics to choose from, with different trade-offs in expressiveness, simplicity, and potential for automation.

Based on our prior experience with Chalice [36,37], we decided to use Implicit Dynamic Frames (IDF) [56], a dialect of separation logic whose assertions express constraints on ownership and values separately. For instance, instead of separation logic's points-to predicate `x.f` $\mapsto$ `v`, IDF uses an accessibility predicate `acc(x.f)` to express ownership of the heap location, and an expression `x.f == v` to express a constraint on the value stored in the location. This separation is lifted to entire data structures, where inductive predicates [44] abstract over permissions, and side-effect free functions over values; notably, these functions may read from heap locations [29]. For instance, the IDF assertion `list(l) && length(l) > 0` expresses that the data structure stored in `l` is a non-empty list. Suitable well-formedness checks ensure that assertions constrain the value of a heap location only in contexts where that location is owned.

Choosing IDF over a more standard separation logic has the potential drawback of being less known in the community, thus potentially making Viper more difficult for others to adopt. Overall, however, IDF proved to be an excellent foundation for an IVL such as Viper for the following main reasons: (1) Source programs typically contain deterministic, side-effect free methods such as getters, comparison functions, and other operations to inspect data structures. These can be encoded as Viper functions and used directly in specifications. For instance, in the verification of a large Go codebase [11], 324 out of 823 (non-ghost and ghost) source methods fell into this category (ca. 40%). (2) Separating permission specifications from value specifications facilitates incremental verification. Verification typically starts by proving memory safety, which requires mostly permission specifications. With IDF, functional specifications can later define various abstractions of data structures without modifying the predicates used for proving memory safety and, thus, without adjusting the features of the original proof. In contrast, standard separation logics must express data abstractions as part of the predicate, such that the abstractions need to be fixed upfront when the predicates are declared. For instance, a binomial heap implementation in Viper [39] uses three predicates to prove memory safety and then introduces ten functions to express various layers of invariants and functional properties. This separation of concerns greatly simplified its verification. (3) Separating permission from value constraints enables the development of tools that handle these two aspects differently. For instance, specification inference techniques may target only one kind of constraint [20]. Moreover, Viper frontends for languages with ownership type systems, most prominently Prusti for Rust [3], automatically extract permission specifications from type information and then simply conjoin user-provided value constraints. Finally, IDF proved to be useful for gradual

verification [62], because it allows programmers to provide partial specifications that contain relevant value constraints, but to defer permission specifications.

In our experience, these advantages clearly outweigh the potential drawbacks of IDF in the context of an intermediate verification language.

3.2 Formalization and Soundness

Viper has been designed to be sound, that is, if verification succeeds for a Viper program then it is actually correct. However, like other SMT-based verifiers [34,27], Viper does not yet provide end-to-end formal soundness guarantees. Users have to trust the definition of Viper's program logic, the correctness of its implementation in the Viper backends, and the underlying SMT solver (plus the correctness of the Viper frontend). On the other hand, verification in interactive theorem provers typically focuses on formal soundness proofs first and considers automation as an afterthought.

A fundamental question is thus how best to design verification tools that provide good automation *and* formal soundness guarantees. In Viper, we have chosen to focus on practical applications first: Once a feature is sufficiently automated and has proved useful, we formalize its semantics and prove soundness. We found that this "practice drives theory" approach is a viable alternative to the prevalent theory-first approach and has led to alternative mathematical directions and the discovery and solution of interesting theoretical problems [15,14].

There have been recent efforts to provide a formal semantics for parts of the Viper language [16,45,62]. In particular, Dardinier et al. [16] formalize an operational and an axiomatic Viper semantics in Isabelle and prove soundness and completeness between them. Their supported subset contains core features such as fractional permissions and Viper's `exhale` and `inhale` statements. The formal treatment of other important features such as inductive predicates and abstraction functions are planned for future work. Dardinier et al. use their semantics to formally connect a frontend based on concurrent separation logic to two Viper backends. In contrast to this once-and-for-all soundness proof, Parthasarathy et al. [45] extended Viper's verification condition generator to produce a certificate for each successful verification. This certificate is a formal proof in Isabelle that shows that correctness of the Boogie program produced by Viper's verification condition generator implies correctness of the Viper program; this result can be combined with an existing certification for Boogie [46]. In contrast to a soundness proof for the logic, certification covers also the *implementation* of the logic in the backend. These works suggest avenues for obtaining foundational guarantees for an automated verifier such as Viper. Extending them to a larger Viper subset and combining them with certification techniques for SMT solvers to obtain end-to-end guarantees is promising future work.

3.3 Growing the Logic

Since the original design of Viper, the expressiveness of state-of-the-art separation logics has grown tremendously, for instance, by supporting higher-order

programs and specifications, as well as various forms of concurrency reasoning. Viper's philosophy for growing its own supported logic has been rather conservative. We consider adding a feature if (1) an encoding into existing features is not possible or leads to bad performance, (2) there is a demand from several frontends for this feature, and (3) the feature can be automated reliably. The third criterion is especially important to preserve Viper's usability.

The two most significant extensions to Viper's logic have been magic wands [55] and iterated separating conjunctions [40]. Magic wands allow one to specify partial data structures and were initially added to express invariants for iterative traversals: in particular, the permissions to the part of the data structure already visited can be conveniently expressed with a wand. Later, magic wand support turned out to be essential for the encoding of borrowing in the Rust verifier Prusti [3]. Iterated separating conjunctions are especially useful for specifying random-access data structures such as arrays, as well as data structures with complex sharing such as arbitrary graphs. In the latter case, specifications can maintain a set of nodes and express ownership of all nodes in the graph by quantifying over the set members using an iterated separating conjunction. This use of a set results in specifications akin to those in Dafny [35], but with support for concurrent programs. Adding these features required developing novel verification algorithms and, for magic wands, also novel theoretical foundations [15].

A recent development has been to start fine tuning the degree of automation Viper provides, another key trade-off in verifier design: Verifiers can generally choose to require more user input for their proofs and thereby obtain simpler, fast proof automation, or alternatively provide more automation at the expense of a more complex proof search. Viper generally aims for a high degree of automation (higher than, for example, VeriFast); this choice increases usability, but can negatively impact verification performance for complex projects. Thus, we have started adding features that enable users to control the degree of automation (e.g. the automatic unfolding of function definitions); more such extensions are necessary to improve scalability further. Another avenue for future work is Viper's fixed permission model. It is able to encode many program logics, but such encodings can lack automation. Therefore, inspired by Gillian [53], we plan to add support for custom permission models and separation algebras.

4 Tool Architecture

Over the years, many new frontends have been added to Viper's ecosystem, but the core architecture (Fig. 1) has remained unchanged. Its central component is the Viper IVL, targeted by frontends for real-world programming languages [3,49,59,9,23,10,28,50,61] and specialized program logics [57,60,21,13].

Viper programs, no matter whether they are generated by a frontend or written manually, are verified by one of Viper's two backends. The symbolic execution (SE) backend [54] operates in the style of Smallfoot [8] and VeriFast [30], but employs different algorithms for Viper's more advanced features. It maintains a symbolic heap to track separation logic resources and interacts directly with

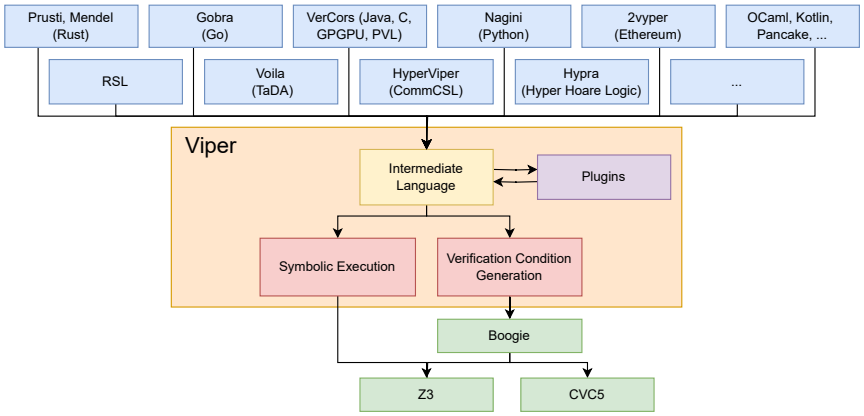

Fig. 1. Viper's architecture, with frontends (blue), backends (red), dependencies (green), and plugin infrastructure (purple).

an SMT solver to discharge verification conditions. The verification condition generation (VCG) backend encodes Viper programs into Boogie [5], that is, it is based on another, lower-level IVL. Viper mainly uses the Z3 SMT solver [38], but also has support for others, e.g. CVC5 [4].

A core design principle behind Viper's architecture, which is essential for its overall usability, is to shield Viper users from the details of the backends and the underlying SMT solver. All interactions happen through the Viper language, and any kind of feedback from the tool can be understood at the level of the language. For instance, Viper's error messages and counterexamples do not refer to details of the backends, and it remains transparent whether a Viper feature is implemented natively or via a plugin. We successfully maintain this abstraction with very few exceptions; for instance, even though Viper offers trigger inference for the SMT solver's quantifier instantiation algorithm, users sometimes have to specify triggers manually (although nonetheless in terms of Viper-level features, not their SMT representations). Most Viper frontends provide a similar abstraction, that is, they in turn shield their users from Viper, such that they can work entirely on the level of the source language.

4.1 Project Management

Developing a tool infrastructure over 15 years in an academic research group is a major challenge because the development spans several generations of PhD students and postdocs, and because the incentives to implement features, fix bugs, and optimize performance are not aligned well with the realities of academia.

We have addressed the first challenge by having team members from several generations work on each Viper component, to ensure continuity and to retain expertise in the group as members graduate and leave. We were also fortunate

that some key team members decided to stay at ETH Zurich beyond their graduation or to continue contributing to Viper in their next job.

We have addressed the second challenge by having an owner for each Viper component (including frontends and IDE), who is responsible for bug fixing and maintenance. This allows us to spread this effort over many people, making it bearable for each of them. Moreover, new PhD projects such as the development of frontends or major case studies are usually a good motivation for PhD students to extend Viper to better support their work.

Over the last 15 years, we usually had between 10 and 20 people working on Viper and its frontends. We coordinate them in monthly project meetings, and (since 2021) maintain a biannual release schedule to drive the project and ensure steady improvements. Finally, we have found occasional hackathons and regular participation in verification competitions to be great ways to obtain experience with our own tools, to find new areas for improvement and to boost team spirit.

4.2 Backends

A key feature that distinguishes Viper from many (intermediate) verification languages is its support for two different backend verifiers that implement two technically very different verification techniques. The decision to support multiple backends instead of a single one represents a major trade-off with far-reaching consequences.

The main advantages are the backends' varied performance and completeness characteristics for different classes of programs (as we explored previously [25]), which enabled frontends and verification efforts that would not have been possible with only one backend.

Another major advantage is that the two backends enable differential testing: any case where the two backends disagree signals either a bug, an incompleteness, or possibly unclear semantics of a Viper feature. Such testing has been one of the main ways to detect such issues, and also helps users triage their own issues.

On the other hand, supporting different backends limits extensions of the language and logic to features that can be automated well in both: e.g. SE-based verifiers can easily support pattern matching for separation-logic resources, whereas a VCG-based backend would have to produce verification conditions containing (both universal and) existential quantifiers, which are not supported well by SMT solvers. Hence, Viper does not have this feature (which is rarely needed with IDF). Similarly, since Viper's VCG backend goes through Boogie, it can only use SMT-level features (e.g. natively-supported types) that are exposed by Boogie. Finally, counterexamples returned by Viper must be generated from two substantially different sources, which limits the information that can be presented. Despite such individual cases, this limitation has not proved to be a major obstacle in practice. As a mitigation, we have recently introduced annotations to express backend-specific instructions in Viper programs, although we expect these to be used only rarely.

Moreover, maintaining two backends incurs substantial engineering costs: language or logic extensions have to be implemented for both backends, which

can slow down development and lead to situations where a new language feature is temporarily supported by only one backend. To mitigate this issue, Viper uses plugins, which enable modular extensions of Viper's syntax that are desugared into the core IVL *before* reaching the backends. This allows us to extend the language and logic *without* impacting the backends. For example, user-defined ADTs and termination checks are implemented as plugins. Facilitating extensions via plugins has substantially contributed to Viper's agility and expressiveness.

Overall, we have enjoyed significant benefits from Viper's two-backend strategy in terms of our ability to use Viper for challenging verification projects, and to advance the state of the art in SE- and VCG-based verification. However, the implementation and maintenance effort required to do this, especially in a research group, should not be underestimated.

4.3 Growing the Architecture

The most significant additions to the Viper ecosystem since its inception have been its frontends, particularly its four most mature frontends for real-world languages (Prusti for Rust, Gobra for Go, VerCors for Java and CUDA/OpenCL, and Nagini for Python). Each is able to verify realistic code; for instance, we have verified an entire system (of over 4,000 lines of production code) only with Gobra [47]. VerCors has been used for multiple realistic Java and CUDA case studies [1,52,43]. Nagini targets statically-typed Python and has support for some of Python's more dynamic features (such as dynamic addition of fields). In addition to these main frontends, a substantial number of research prototypes for automating advanced program logics has been developed, and there is ongoing work on further frontends.

Each new frontend has significantly benefitted from the shared infrastructure built around the Viper language, but also challenged Viper and motivated improvements that ultimately benefit the whole ecosystem. An interesting special case is the gradual verification tool Gradual C0 [19], which is not implemented as a frontend, but as a fork of Viper and its SE backend.

Most changes to the Viper backends are triggered by extensions of the Viper language and logic (see Sec. 2.2 and Sec. 3.3). In addition to those, we have extended both backends to explore different core algorithms, combinations of heap models and proof search algorithms. These exhibit different performance and completeness tradeoffs [25], and users can choose the algorithm that best suits the verification problem domain at hand. As it has become difficult even for experts to predict which specific algorithm will perform best on a given example, automating this process is ongoing work.

Finally, one of the most important additions to the Viper architecture has been its IDE integration in the form of a mature plugin for VSCode, which significantly simplifies the process of installing and running Viper, and additionally enhances the user experience through features such as verification result caching, intuitive error reporting, and many standard IDE features. We are convinced that Viper's availability in VSCode has played a central role in its adoption for teaching at multiple universities.

5 Related Work

There are a number of other verification tools for separation logics, such as VeriFast [30], GRASShopper [48], SecC [26], the Gillian verifiers [53], and Caper [18], as well as other similar tools not based on separation logic, in particular, Dafny [35], which uses dynamic frames [31] to reason about heap-manipulating programs. Moreover, there are other intermediate verification languages (IVLs) with corresponding backends, most notably Boogie [34], Why3 [27], and GIL [53].

Compared to most other separation logic based tools, Viper supports a richer set of core features in its separation logic (offering, e.g., magic wands, iterated separating conjunction, and permission introspection), along with appropriate proof search algorithms, which facilitates the encoding of a wide range of verification problems. For example, SecC and Caper directly implement more specific logics (for information flow security and fine-grained concurrency, respectively), while those use cases can both be supported in Viper via a plugin and a frontend, respectively. Moreover, Viper supports specific assertions and statements (e.g., `inhale` and `exhale`) whose purpose is to *encode* separation logic proof rules in Viper; no other tool has those. Compared to VeriFast, Viper requires less user input and provides more automation at the cost of a more complex proof search that can potentially lead to worse performance.

Compared to Dafny, Viper's logical foundation allows one to encode verification problems for concurrent programs, whereas Dafny is limited to sequential code. On the other hand, Dafny offers advanced features for proof authoring and specification inference, whereas Viper leaves such tasks to frontends.

Compared to other IVLs, Viper supports a fixed heap model and logic (implicit dynamic frames), whereas GIL is parametric in its memory model, and Boogie and Why3 require frontends to encode the memory model. Viper's approach allows it to provide predictable automated proof search algorithms tailored to its heap model and logic.

Viper is also unique in that it is (to our knowledge) the only deductive verification framework supporting more than one independent verification backend.

6 Discussion

Revisiting our original design goals, we believe we have achieved the goal of *expressiveness*, as demonstrated by Viper's ability to verify a wide range of frontend languages, and advanced properties expressed in complex program logics. Making Viper parametric in its permission model and adding support for higher-order code and assertions would further increase expressiveness.

Viper is designed for *soundness* above all else; we leverage differential testing to effectively identify implementation bugs in the backends, and extensions such as termination checks help detect inconsistent encodings. Significant progress has been made in formalizing Viper's semantics and generating proof certificates, with ongoing efforts to enable independent proof validation.

As for *usability*, we find it very encouraging that Viper is increasingly used by people with little or no connection to the Viper team, for both teaching (e.g.,

in Copenhagen, Nancy, and Oldenburg) and tooling (e.g., frontends for Java, Kotlin, OCaml, Pancake; Gradual Viper). It has also won awards in several categories of the VerifyThis verification competition (best team, best student team, tool used by most teams), demonstrating its effectiveness. Ongoing work on specification inference and verification debugging will further increase usability.

Overall, we see the verification of a next-generation internet router [47] as evidence for Viper's usability and expressiveness. This is a substantial Go codebase under active development, not written with verification in mind, optimized for performance, and one of the largest verification efforts ever undertaken with automated separation logic verifiers.

Such ambitious projects demonstrate the capabilities of the Viper infrastructure, but also relentlessly expose aspects that need improving. In fact, many of the directions for future work mentioned throughout this paper are motivated by large case studies on correctness and security verification. They will fuel our research agenda for the next 15 years.

Acknowledgments. Numerous people have contributed to the development of the Viper infrastructure and its frontends, both in our team and elsewhere. We are very grateful for all of these contributions.

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

References

1. Armborst, L., Huisman, M.: Permission-based verification of red-black trees and their merging. In: FormaliSE@ICSE. pp. 111–123. IEEE (2021)
2. Arquint, L., Schwerhoff, M., Mehta, V., Müller, P.: A generic methodology for the modular verification of security protocol implementations. In: CCS. pp. 1377–1391. ACM (2023)
3. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. Proc. ACM Program. Lang. **3**(OOPSLA), 147:1–147:30 (2019)
4. 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 (1). Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022)
5. 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)
6. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM **54**(6), 81–91 (2011)
7. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
8. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005)
9. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: FM. Lecture Notes in Computer Science, vol. 8442, pp. 127–131. Springer (2014)
10. Bräm, C., Eilers, M., Müller, P., Sierra, R., Summers, A.J.: Rich specifications for Ethereum smart contract verification. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021)
11. Chuat, L., Legner, M., Basin, D.A., Hausheer, D., Hitz, S., Müller, P., Perrig, A.: The Complete Guide to SCION - From Design Principles to Formal Verification. Information Security and Cryptography, Springer (2022)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C – A software analysis perspective. In: SEFM. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer (2012)
13. Dardinier, T., Li, A., Müller, P.: Hypra: A deductive program verifier for hyper hoare logic. Proc. ACM Program. Lang. **8**(OOPSLA2), 1279–1308 (2024). `https://doi.org/10.1145/3689756`, `https://doi.org/10.1145/3689756`
14. Dardinier, T., Müller, P., Summers, A.J.: Fractional resources in unbounded separation logic. Proc. ACM Program. Lang. **6**(OOPSLA2), 1066–1092 (2022). `https://doi.org/10.1145/3563326`, `https://doi.org/10.1145/3563326`
15. Dardinier, T., Parthasarathy, G., Weeks, N., Müller, P., Summers, A.J.: Sound automation of magic wands. In: CAV (2). Lecture Notes in Computer Science, vol. 13372, pp. 130–151. Springer (2022)
16. Dardinier, T., Sammler, M., Parthasarathy, G., Summers, A.J., Müller, P.: Formal foundations for translational separation logic verifiers. Proc. ACM Program. Lang. **9**(POPL) (Jan 2025)

17. Denis, X., Jourdan, J., Marché, C.: Creusot: A foundry for the deductive verification of Rust programs. In: ICFEM. Lecture Notes in Computer Science, vol. 13478, pp. 90–105. Springer (2022)
18. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J., Birkedal, L.: Caper - automatic verification for fine-grained concurrency. In: ESOP. Lecture Notes in Computer Science, vol. 10201, pp. 420–447. Springer (2017)
19. DiVincenzo, J., McCormack, I., Zimmerman, C., Gouni, H., Gorenburg, J., Ramos-Dávila, J., Zhang, M., Sunshine, J., Tanter, É., Aldrich, J.: Gradual C0: Symbolic execution for gradual verification. ACM Trans. Program. Lang. Syst. **46**(4) (Jan 2025). https://doi.org/10.1145/3704808, https://doi.org/10.1145/3704808
20. Dohrau, J., Summers, A.J., Urban, C., Münger, S., Müller, P.: Permission inference for array programs. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 55–74. Springer (2018). https://doi.org/10.1007/978-3-319-96142-2_7, https://doi.org/10.1007/978-3-319-96142-2_7
21. Eilers, M., Dardinier, T., Müller, P.: CommCSL: Proving information flow security for concurrent programs using abstract commutativity. Proc. ACM Program. Lang. **7**(PLDI), 1682–1707 (2023). https://doi.org/10.1145/3591289, https://doi.org/10.1145/3591289
22. Eilers, M., Meier, S., Müller, P.: Product programs in the wild: Retrofitting program verifiers to check information flow security. In: CAV (1). Lecture Notes in Computer Science, vol. 12759, pp. 718–741. Springer (2021)
23. Eilers, M., Müller, P.: Nagini: A static verifier for Python. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 596–603. Springer (2018)
24. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. **42**(1), 3:1–3:37 (2020)
25. Eilers, M., Schwerhoff, M., Müller, P.: Verification algorithms for automated separation logic verifiers. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14681, pp. 362–386. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_18, https://doi.org/10.1007/978-3-031-65627-9_18
26. Ernst, G., Murray, T.: SecCSL: Security concurrent separation logic. In: CAV (2). Lecture Notes in Computer Science, vol. 11562, pp. 208–230. Springer (2019)
27. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
28. Gros, C., Pereira, M.: Le chameau et le serpent rentrent dans un bar : vérification quasi-automatique de code OCaml en logique de séparation (2024), https://arxiv.org/abs/2412.14894
29. Heule, S., Kassios, I.T., Müller, P., Summers, A.J.: Verification condition generation for permission logics with abstract predicates and abstraction functions. In: ECOOP. Lecture Notes in Computer Science, vol. 7920, pp. 451–476. Springer (2013)
30. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011)

31. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4085, pp. 268–283. Springer (2006). https://doi.org/10.1007/11813040_19, https://doi.org/10.1007/11813040_19
32. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 712–717. Springer (2012)
33. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 427–443. Springer (2012)
34. Leino, K.R.M.: This is Boogie 2 (June 2008), https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
35. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (Dakar). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
36. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: ESOP. Lecture Notes in Computer Science, vol. 5502, pp. 378–393. Springer (2009)
37. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: FOSAD. Lecture Notes in Computer Science, vol. 5705, pp. 195–222. Springer (2009)
38. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
39. Müller, P.: The binomial heap verification challenge in Viper. In: Müller, P., Schaefer, I. (eds.) Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday. pp. 203–219. Springer (2018). https://doi.org/10.1007/978-3-319-98047-8_13, https://doi.org/10.1007/978-3-319-98047-8_13
40. Müller, P., Schwerhoff, M., Summers, A.J.: Automatic verification of iterated separating conjunctions using symbolic execution. In: CAV (1). Lecture Notes in Computer Science, vol. 9779, pp. 405–425. Springer (2016)
41. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016)
42. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3170, pp. 49–67. Springer (2004). https://doi.org/10.1007/978-3-540-28644-8_4, https://doi.org/10.1007/978-3-540-28644-8_4
43. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Ahrendt, W., Tarifa, S.L.T. (eds.) Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11918, pp. 418–436. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_23, https://doi.org/10.1007/978-3-030-34968-4_23
44. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL. pp. 247–258. ACM (2005)
45. Parthasarathy, G., Dardinier, T., Bonneau, B., Müller, P., Summers, A.J.: Towards trustworthy automated program verifiers: Formally validating translations into an intermediate verification language. Proc. ACM Program. Lang.

8(PLDI), 1510–1534 (2024). https://doi.org/10.1145/3656438, https://doi.org/10.1145/3656438
46. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 704–727. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_33, https://doi.org/10.1007/978-3-030-81688-9_33
47. 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
48. Piskac, R., Wies, T., Zufferey, D.: GRASShopper - complete heap verification with mixed specifications. In: TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 124–139. Springer (2014)
49. Poli, F., Denis, X., Müller, P., Summers, A.J.: Reasoning about interior mutability in Rust using library-defined capabilities (2024)
50. Protopapa, F.: Verifying Kotlin Code with Viper by Controlling Aliasing. Master's thesis, University of Padua (2024)
51. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
52. Safari, M., Huisman, M.: Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. Theor. Comput. Sci. **912**, 81–98 (2022)
53. Santos, J.F., Maksimovic, P., Ayoun, S., Gardner, P.: Gillian, part I: a multi-language platform for symbolic execution. In: PLDI. pp. 927–942. ACM (2020)
54. Schwerhoff, M.: Advancing Automated, Permission-Based Program Verification Using Symbolic Execution. Ph.D. thesis, ETH Zurich, Zürich, Switzerland (2016)
55. Schwerhoff, M., Summers, A.J.: Lightweight support for magic wands in an automatic verifier. In: ECOOP. LIPIcs, vol. 37, pp. 614–638. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
56. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP. Lecture Notes in Computer Science, vol. 5653, pp. 148–172. Springer (2009)
57. Summers, A.J., Müller, P.: Automating deductive verification for weak-memory programs. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 190–209. Springer (2018)
58. Wolf, F.A., Müller, P.: Verifiable security policies for distributed systems. In: Computer and Communications Security (CCS). pp. 4–18. CCS '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3658644.3690303, https://doi.org/10.1145/3658644.3690303
59. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular specification and verification of Go programs. In: CAV (1). Lecture Notes in Computer Science, vol. 12759, pp. 367–379. Springer (2021)
60. Wolf, F.A., Schwerhoff, M., Müller, P.: Concise outlines for a complex logic: A proof outline checker for TaDA **13047**, 407–426 (2021)
61. Zhao, J., Legnani, A., Ung, T.T., Truong, H., Sau, T.W., Tanaka, M., Åman Pohjola, J., Sewell, T., Sison, R., Syeda, H., Myreen, M., Norrish, M., Heiser, G.: Verifying device drivers with Pancake (2025), https://arxiv.org/abs/2501.08249
62. Zimmerman, C., DiVincenzo, J., Aldrich, J.: Sound gradual verification with symbolic execution. Proc. ACM Program. Lang. **8**(POPL), 2547–2576 (2024). https://doi.org/10.1145/3632927, https://doi.org/10.1145/3632927