

Fusion: A Fold and Unfold Annotation Synthesis Tool for Viper-Based Verification

Joonghoon Lee
Korea University
Seoul, Republic of Korea
joonghoonlee@korea.ac.kr

Nicolas Klose
ETH Zurich
Zurich, Switzerland
nicolas.klose@inf.ethz.ch

Seokhyun Lee
Korea University
Seoul, Republic of Korea
seokhyunlee@korea.ac.kr

Peter Müller
ETH Zurich
Zurich, Switzerland
peter.mueller@inf.ethz.ch

Hakjoo Oh[✉]
Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

Abstract

Viper is a verification infrastructure for permission-based reasoning and serves as a common back-end for verifiers targeting multiple programming languages. In Viper, predicates package heap resources into reusable abstractions. To prevent SMT solvers from unfolding recursive predicates indefinitely, Viper treats predicates isorecursively: a predicate instance is not automatically interchangeable with the resources described by its body. Consequently, users must provide explicit fold/unfold annotations to transition between abstraction levels.

To address this issue, we present FUSION, a Viper-based tool that synthesizes missing fold/unfold annotations, built around a novel graph-based synthesis method. Given a verification failure due to a missing resource, FUSION constructs a predicate dependency graph from Viper predicate definitions and finds potential ways to obtain the missing resource from available resources, thereby formulating annotation synthesis as a pathfinding problem over the graph. It uses the resulting paths to synthesize candidate sequences of fold/unfold annotations that *potentially* provide the missing resource, and uses the verifier to validate the annotations. Our evaluation on a large real-world verification project shows that FUSION successfully synthesizes annotations for 93 of 149 benchmark functions (62.4%) within a 20-minute per-function budget, demonstrating its potential to reduce the annotation burden in practical verification workflows.

CCS Concepts: • Software and its engineering → Formal software verification.

Keywords: Formal verification, Separation logic, Viper, Annotation synthesis, Fold/unfold



This work is licensed under a Creative Commons Attribution 4.0 International License.

SPLASH Companion '26, Oakland, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2910-2/2026/10

<https://doi.org/10.1145/3837729.3840481>

ACM Reference Format:

Joonghoon Lee, Nicolas Klose, Seokhyun Lee, Peter Müller, and Hakjoo Oh. 2026. Fusion: A Fold and Unfold Annotation Synthesis Tool for Viper-Based Verification. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26)*, October 4–9, 2026, Oakland, CA, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3837729.3840481>

1 Introduction

Viper is a verification infrastructure for permission-based heap reasoning that serves as an intermediate verification language [6]. It is used as a back-end for multiple verifier front-ends for different programming languages, such as Gobra for Go, Nagini for Python, VerCors for Java/C, and Prusti for Rust [1, 3, 4, 11]. They translate annotated source programs into Viper, where an automated SMT-based back-end verifier checks the resulting proof obligations. This shared verification layer provides a common representation of heap permissions and verifier states, thereby enabling the development of one core proof engine that can be applied across multiple front-ends and source languages. Built around this common layer, the Viper ecosystem has supported substantial verification efforts, including real-world verification projects such as network protocol verification [2, 7] and industrial concurrent software verification [5].

Viper expects programs to be annotated with preconditions, postconditions, and loop invariants, as well as auxiliary statements that guide SMT-based verification. In particular, Viper uses separation-logic predicates to package heap resources into reusable abstractions. To avoid unbounded unfolding of recursive predicates by SMT solvers, Viper treats predicates isorecursively: a predicate instance is not automatically interchangeable with the resources described by its body in the symbolic heap [6, 9]. Consequently, users must provide explicit **fold** and **unfold** annotations whenever a proof requires transitioning between these abstraction levels.

```

1 type S struct { f, g int }
2 //@ pred FieldsPerm(x *S) { acc(&x.f) && acc(&x.g) }
3
4 //@ requires acc(&a.f) && acc(&a.g)
5 //@ ensures acc(FieldsPerm(a))
6 func foo(a *S) {
7     //@ fold acc(FieldsPerm(a))
8     return
9 }

```

Figure 1. Example function with a `fold` annotation.

Figure 1 shows an example Go function with Gobra annotations that requires a `fold` annotation to verify. The precondition of `foo` provides the individual field permissions `acc(&a.f)` and `acc(&a.g)`, whereas its postcondition requires the packaged predicate access `acc(FieldsPerm(a))`. Here, `acc(e)` denotes permission to the resource denoted by `e`, such as a field or a predicate instance. Although the predicate body describes exactly the permissions available at the end of the function, Viper does not automatically replace them with the corresponding predicate instance. As a consequence, without the `fold` annotation, the verifier cannot establish the postcondition, and verification fails. To make the verification succeed, the user must annotate the function with `fold acc(FieldsPerm(a))`, which transforms the verifier’s symbolic heap by consuming the individual field permissions and producing `acc(FieldsPerm(a))`. The resulting predicate access allows the postcondition to be established. Conversely, an `unfold` annotation consumes a predicate instance and exposes the resources described by its body.

Writing fold/unfold annotations requires substantial manual effort and expertise in real-world verification projects. For example, in VerifiedSCION [7], a large Gobra verification project for the SCION Internet architecture [12], our analysis shows that 55.1% of annotated functions contain at least one `fold` or `unfold` annotation. For these functions, fold/unfold annotations account for 17.8% of all annotation lines, 4.5 times as many as loop invariant lines. This burden reflects a broader challenge in SMT-based separation-logic verification: users need to guide transitions between abstract predicate instances and the concrete resources they describe. In current Viper-based verification workflows, this guidance remains manual: when verification fails because resources are represented at the wrong abstraction level, users must inspect the failure, determine which predicate to `fold` or `unfold`, write the appropriate annotation, and re-run the verifier to check whether further annotations are needed.

To reduce this annotation burden, we present FUSION, a fold/unfold annotation synthesis tool for Viper-based verification. FUSION is built around a novel graph-based synthesis method that formulates annotation synthesis as a path-search problem over a predicate dependency graph, which captures how Viper predicates expose lower-level predicates

and heap resources when unfolded. Given a verification failure, the tool synthesizes candidate annotation sequences by searching for paths in this graph that connect the available resources to the missing access reported by the verifier.

We evaluate FUSION on 149 functions from the Verified-SCION project [7], a large real-world codebase verified using Gobra. FUSION synthesizes annotations that make 93 of 149 benchmark functions verify, achieving a success rate of 62.4% within a 20-minute per-function budget. These results show that synthesis driven by Viper verification failures and symbolic states can automate a substantial portion of the fold/unfold annotation work in a real-world verification project.

FUSION is publicly available as an open-source tool, with repository documentation describing how to install and run it:

- GitHub: <https://github.com/kupl/Fusion>
- YouTube: <https://youtu.be/RZuHIFN6HQQ>
- Zenodo: <https://doi.org/10.5281/zenodo.20936789>

2 FUSION Overview

FUSION is a graph-based fold/unfold annotation synthesis tool for Viper-based verification. Its input is a target function with functional specifications and other proof annotations already written, but possibly missing the source-level `fold` and `unfold` annotations needed for verification. FUSION aims to synthesize the missing source-level `fold` and `unfold` annotations that make the function verify. The output is either an annotated version of the target function that verifies, or failure after a user-configured synthesis budget is exhausted. Operationally, FUSION organizes synthesis as a worklist-based search over annotated versions of the target function. Figure 2 shows one iteration of this search.

Each iteration starts by removing one annotated version from the worklist, which initially contains the original target function, and running the verifier on it (steps (1)–(2)). If the verifier succeeds, FUSION returns the verified annotated version. Otherwise, FUSION extracts failure-local information needed for synthesis (step (3)). The extracted information includes the program location where verification failed, the resource access that the verifier could not establish, the resources currently available in the symbolic heap, and the concrete receiver and argument expressions appearing in these accesses.

Using the failure information, FUSION maps the available resources and the missing access to nodes in the *predicate dependency graph*, which is built from the predicate definitions in the program. It then searches the graph for ways to transform the available resources into the missing access reported by the verifier (step (4)); Section 3 describes this procedure in detail. Each discovered path yields a candidate sequence of `fold` and `unfold` operations. FUSION then instantiates the sequence with the concrete expressions collected

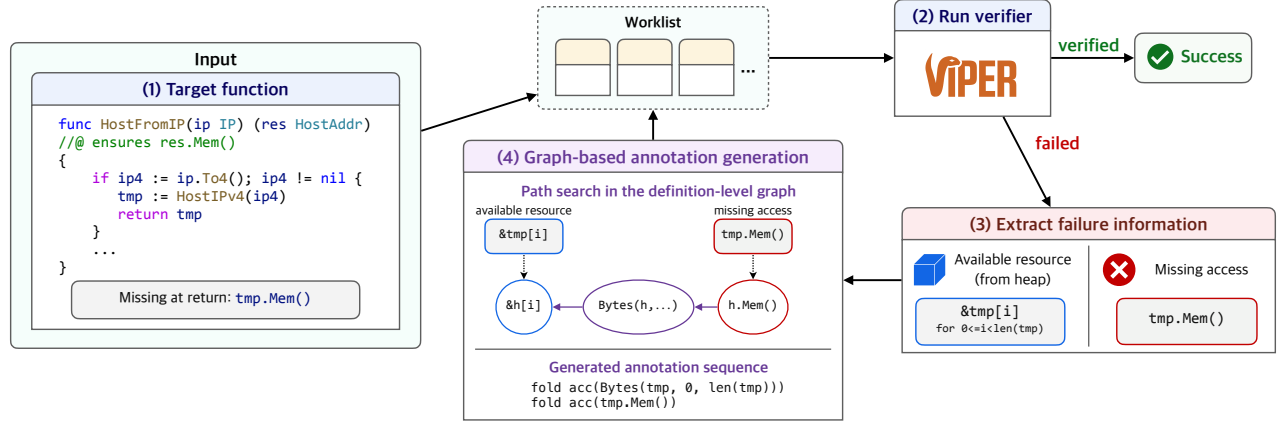


Figure 2. One iteration of FUSION's fold/unfold synthesis workflow.

from the failure, inserts the resulting source-level annotations immediately before the reported failure location in a fresh copy of the current annotated version, and adds the new version to the worklist.

FUSION repeats this process until some annotated version verifies or the attempt budget is exhausted. Because generated candidates are validated by re-running the verifier, the search can tolerate infeasible candidates produced by the graph abstraction: such candidates simply fail verification and lead to further worklist iterations.

We implement FUSION by directly extending the Viper engine with failure extraction, graph-based sequence generation. Although source-level annotation generation is currently implemented only for Gobra, FUSION can be easily extended to other front-ends by adding a source-level translation layer. A Python orchestration layer manages the worklist, invokes the verifier on candidate versions, and enforces the attempt budget. Users can run FUSION with the following command:

```
$ python fusion_gobra.py --target
FILE@LINE --gobra-config CONFIGJSON
```

3 Fold/Unfold Annotation Synthesis

This section describes the core graph-based synthesis method of FUSION. The key idea is to view predicate definitions as a directed graph used for search: unfolding a predicate follows graph edges, while folding a predicate traverses those edges in reverse. A verification failure provides the concrete resources in the symbolic heap and the missing access that must be established; FUSION maps these accesses onto the graph and searches for paths that suggest candidate fold/unfold annotations.

3.1 Running Example

Figure 3 shows a simplified VerifiedSCION function and the relevant predicate definitions, which converts an IP value

```
1 type IP []byte
2 type HostIPv4 IP
3 HostIPv4 implements HostAddr
4 /*@
5 pred (h HostIPv4) Mem() { acc(Bytes(h, 0, len(h))) }
6 pred Bytes(h []byte, start int, end int) {
7     0 <= start && forall i int ::
8         start <= i && i < end ==> acc(&h[i])
9 }
10 @*/
11 func HostFromIP(ip IP) (res HostAddr)
12     //@ requires acc(ip)
13     //@ ensures res.Mem()
14 {
15     if ip4 := ip.To4(); ip4 != nil {
16         // ip4: IP, sliced to a 4-byte representation
17         tmp := HostIPv4(ip4)
18         // tmp: HostIPv4, converted from ip4
19         //@ fold acc(Bytes(tmp, 0, len(tmp)))
20         //@ fold acc(tmp.Mem())
21         return tmp
22     }
23 }
```

Figure 3. Simplified VerifiedSCION example.

into a HostAddr. In the successful IPv4 branch, `ip.To4()` returns either the original slice or a four-byte subslice of it, so the result refers to the same underlying byte storage. The subsequent conversion to HostIPv4 changes only the slice type. Consequently, the verifier continues to track permissions to the byte locations referenced by the original input slice.

The two `fold` annotations in the branch are the annotations that FUSION should synthesize. Immediately before the `return`, the symbolic heap contains permissions to the byte elements of `tmp`, represented as `&tmp[i]`, but the verifier fails to establish the required predicate access `tmp.Mem()`. This failure-local information gives FUSION a concrete synthesis problem: derive the missing predicate access from the available byte-element permissions.

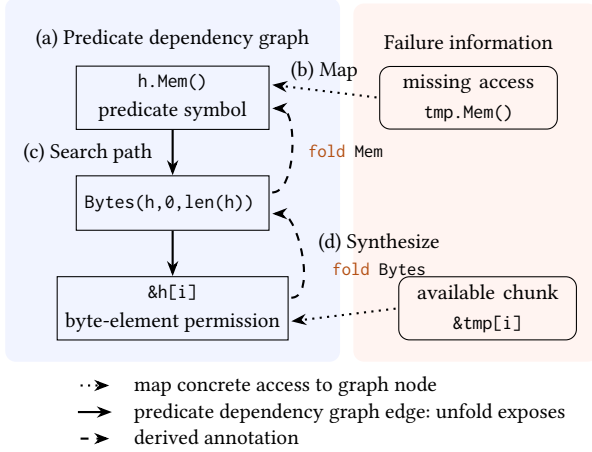


Figure 4. Graph-based sequence generation for the running example in Figure 3.

3.2 Predicate Dependency Graph

FUSION builds a *predicate dependency graph* from the predicate definitions in the Viper program. Each node represents a definition-level resource access, either the predicate access corresponding to a predicate definition or an access that appears in a predicate body. An edge $p \rightarrow r$ records that unfolding the predicate access p may expose the resource access r . Thus, graph edges are stored in the direction of unfolding.

Figure 4(a) shows the predicate dependency graph for predicates defined in the running example (lines 5–9). Its nodes correspond to the predicate accesses and byte-element permission shown in the figure. The body of `h.Mem()` contains the nested predicate access, and the body of `Bytes` contains a quantified accessibility expression for the byte elements. These body references induce the two graph edges shown in the figure. The graph is a conservative over-approximation: it records resource-access patterns in predicate bodies but omits pure logical conditions, such as range checks and quantified guards. Thus, a graph path represents a *potential* way to obtain one resource access from another.

3.3 Graph-Guided Synthesis

FUSION uses the failure-local information to turn annotation synthesis into a pathfinding problem over the predicate dependency graph. Because the graph is built over definition-level accesses, FUSION first maps the concrete accesses in the failed verifier state to corresponding graph nodes based on their types. In the running example, the missing access `tmp.Mem()` maps to the graph node `h.Mem()`, and the available byte-element permissions `&tmp[i]` map to the graph node `&h[i]` (Figure 4(b)). The node matched by the missing access becomes the search target, and nodes matched by available resources become search sources.

FUSION then searches for paths from available-resource nodes to the missing-access node (Figure 4(c)). Although graph edges are stored in the unfolding direction, the search may traverse them in either direction. Traversing an edge $p \rightarrow r$ forward yields an *unfold* operation for p , whereas traversing it in reverse yields a *fold* operation for p . In the running example, the search finds the reverse path `&h[i] → Bytes(h, 0, len(h)) → h.Mem()` (Figure 4(c)). To avoid nontermination in the presence of cycles, such as recursive predicates, FUSION bounds the maximum path length and does not expand paths beyond this bound. It then converts each path into a candidate fold/unfold sequence; because the example path traverses both graph edges in reverse, it yields two fold annotations (Figure 4(d)):

```
1 //@ fold acc(Bytes(tmp, 0, len(tmp)))
2 //@ fold acc(tmp.Mem())
```

Here, concrete receiver and argument expressions come from the mapping established earlier. For example, substituting `tmp` for `h` instantiates the intermediate `Bytes` predicate in the first generated annotation above.

The fold/unfold annotations obtained from a path edge are only candidate operations. The resulting annotations are inserted immediately before the reported failure location and validated by re-running the verifier: it checks whether the current symbolic state satisfies the full predicate body and its pure conditions. This validation step filters out infeasible annotations that may arise from the graph’s over-approximation.

4 Evaluation

Our evaluation asks whether FUSION can synthesize fold/unfold annotations efficiently on realistic verification tasks. We evaluate efficiency by measuring how many benchmark functions FUSION can synthesize within a fixed time budget.

We use 149 functions from VerifiedSCION [7], a large real-world verification project using Gobra [11], as our benchmark. We select functions that originally contained at least one *fold* or *unfold* annotation, excluding those that fail verification even with the original annotations. VerifiedSCION comprises 135.1 kLoC, and the selected functions contain an average of 4.28 fold/unfold annotations each. For each function, we remove all existing fold/unfold annotations and run FUSION to synthesize the missing annotations.

For comparison, we also implement ENUM, a simple enumerative search strategy. At each synthesis step, ENUM enumerates applicable *fold* and *unfold* annotations and produces new annotated functions by inserting one annotation at a time.

Each benchmark function is given a 20-minute synthesis budget. A synthesis run succeeds if FUSION produces annotations that make the function verify within this budget. All experiments were conducted on an Ubuntu 22.04 machine

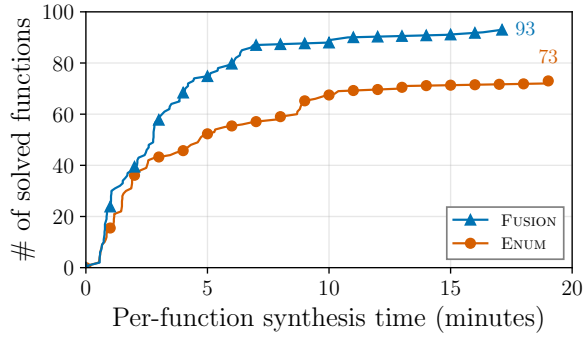


Figure 5. Fold/unfold annotation synthesis success over time.

with two Intel(R) Xeon(R) Silver 4214 CPUs and 128 GB of RAM.

4.1 Results

Figure 5 shows the cumulative number of solved benchmark functions over per-function repair time. The x-axis shows repair time in minutes, up to the 20-minute budget, and the y-axis shows the cumulative number of solved functions. The blue line with triangle markers represents FUSION, and the orange line with circle markers represents ENUM. Within the 20-minute budget, FUSION synthesizes annotations for 93 of 149 functions (62.4%), while ENUM solves 73 functions (49.0%). This result shows that FUSION can synthesize valid fold/unfold annotations for a substantial fraction of realistic verification tasks within a practical per-function budget. Notably, FUSION solves all 73 benchmarks solved by ENUM plus 20 more, while its graph-search overhead is neglectable, averaging only 1.5 seconds compared with 3.3 minutes of end-to-end synthesis time across solved benchmarks.

4.2 Discussion

The results suggest that FUSION is effective in practice because many missing fold/unfold annotations can be synthesized as short local sequences. FUSION uses predicate dependencies to generate such sequences, allowing the search to make useful progress within a small time budget.

The remaining failures highlight two sources of imprecision. First, FUSION’s predicate dependency graph conservatively over-approximates predicate definitions: it records resource-access patterns but not the pure conditions under which those resources are available. As a result, FUSION may spend time validating fold/unfold sequences that are invalid in the current verification context. Second, FUSION inserts synthesized annotations immediately before the reported failure location, which can miss cases where the required annotation must be located at an earlier program point, such as inside a conditional statement. Future work could improve context-sensitive dependency reasoning and support more

flexible insertion-location choices to address these limitations.

5 Related Work

Viper is a common back-end for permission-based verifiers for Go, Python, Java, C, and Rust, which have supported substantial verification efforts such as VerifiedSCION [1, 2, 4–7, 11]. The closest related work is Prusti [3], a Viper-based verifier for Rust, which infers auxiliary fold/unfold operations by analyzing an intermediate verification representation derived from Rust’s Mid-level Intermediate Representation (MIR). FUSION instead constructs a predicate-dependency graph from verifier-reported available and missing resources and Viper predicate definitions, and derives candidate fold/unfold sequences from paths in this graph.

The burden of fold/unfold annotations also arises outside the Viper ecosystem, such as in other separation-logic verification tools. CN, a recent separation-logic-based verifier for C systems software, requires users to manually provide fold/unfold annotations for user-defined predicates [8]. Work on VeriFast has studied automation of fold/unfold annotations [10], but largely relies on enumerative search over candidate annotations.

6 Conclusion

We presented FUSION, a tool that synthesizes missing fold/unfold annotations for Viper-based verification. FUSION uses verifier failure information and a graph derived from Viper predicate definitions to generate candidate fold/unfold sequences. It validates these sequences with the verifier and translates valid sequences into source-level annotations. Our evaluation shows that FUSION can synthesize fold/unfold annotations for a large real-world verification project. Future work includes improving context-sensitive dependency reasoning to support more flexible insertion-location choices, and extending the source-language integration layer to additional Viper front-ends.

Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, 50%) and by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2026-RS-2020-II201819, 5%). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No.2021R1A5A1021944(10%), RS-2026-25470524(30%)).

References

- [1] Lukas Armbrorst, Pieter Bos, Lars B van den Haak, Marieke Huisman, Robert Rubbens, Ömer Şakar, and Philip Tasche. 2024. The VerCors verifier: a progress report. In *International Conference on Computer Aided Verification*. Springer, 3–18. doi:10.1007/978-3-031-65630-9_1
- [2] Linard Arquent, Malte Schwerhoff, Vaibhav Mehta, and Peter Müller. 2023. A generic methodology for the modular verification of security protocol implementations. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1377–1391. doi:10.1145/3576915.3623105
- [3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30. doi:10.1145/3360573
- [4] Marco Eilers and Peter Müller. 2018. Nagini: a static verifier for Python. In *International Conference on Computer Aided Verification*. Springer, 596–603. doi:10.1007/978-3-319-96145-3_33
- [5] Raúl E Monti, Robert Rubbens, and Marieke Huisman. 2022. On deductive verification of an industrial concurrent software component with VerCors. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 517–534. doi:10.1007/978-3-031-19849-6_29
- [6] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2015. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*. Springer, 41–62. doi:10.1007/978-3-662-49122-5_2
- [7] João Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, et al. 2025. Protocols to Code: Formal Verification of a Secure Next-Generation Internet Router. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1469–1483. doi:10.1145/3719027.3765104
- [8] Christopher Pulte, Dhruv C Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying systems C code with separation-logic refinement types. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1–32. doi:10.1145/3571194
- [9] Alexander J Summers and Sophia Drossopoulou. 2013. A formal semantics for isorecursive and equirecursive state abstractions. In *European Conference on Object-Oriented Programming*. Springer, 129–153. doi:10.1007/978-3-642-39038-8_6
- [10] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. 2011. Annotation inference for separation logic based verifiers. In *Proceedings of the Joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems (FMODS'11/FORTE'11)*. Springer-Verlag, 319–333. doi:10.1007/978-3-642-21461-5_21
- [11] Felix A Wolf, Linard Arquent, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. 2021. Gobra: Modular specification and verification of go programs. In *International Conference on Computer Aided Verification*. Springer, 367–379. doi:10.1007/978-3-030-81685-8_17
- [12] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G Andersen. 2011. SCION: Scalability, control, and isolation on next-generation networks. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 212–227. doi:10.1109/SP.2011.45

Received 2026-06-27; accepted 2026-07-25