Bridging Automated and Foundational Verification: Towards Translation Validation for CN Separation Logic Proofs Working Paper

Vadim Zaliva, Christopher Pulte, Valerii Huhnin

Bridging Automated and Foundational Verification: Towards Translation Validation for CN Separation Logic Proofs

Working Paper

March 25, 2025

Vadim Zaliva
University of Cambridge, Department of Computer Science and Technology, UK

Christopher Pulte
University of Cambridge, Department of Computer Science and Technology, UK

Valerii Huhnin
Digamma.ai, Ukraine

Abstract

Separation logic frameworks and tools can broadly be classified into two categories, each with distinct trade-offs. Foundational frameworks, such as Iris and VST, offer rigorous machine-checked proofs in which all reasoning is verified by a small, independently trusted proof assistant kernel (e.g. Rocq's type checker). This leads to high confidence in the soundness of the reasoning. However, despite significant progress in separation logic proof automation, scaling verification with these approaches remains a challenge. In contrast, non-foundational tools, such as VeriFast, Viper, and Gillian, achieve greater scalability by incorporating custom separation logic implementations and SMT-based proof automation. However, because they are standalone tools, their reasoning lacks separately verifiable, mechanised soundness guarantees. In this work, we seek to bridge the gap between the two sides, combining the proof automation of standalone verifiers with the strong soundness guarantees of foundational logics, using a translation validation approach. We explore this in the context of CN, a standalone tool for semi-automatic verification of C language programs based on separation-logic refinement types. To increase confidence in CN's verification results, we aim to (1) formalise CN's separation logic and intermediate language in Rocq, (2) extend CN to generate proof certificates, and (3) implement in Rocq an independent, automatic proof checker for CN's proof certificates. We present our work in progress, discussing the approach taken, challenges encountered, and the remaining steps toward foundationally verified CN proofs. The current development validates resource-inference proof-log entries against formal correctness predicates, but does not yet establish soundness against a semantics of CN's separation logic. This preprint reports work in progress and should be read as a working-paper version rather than a peer-reviewed publication.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Software and its engineering → Correctness; Software and its engineering → Software verification; Theory of computation → Linear logic; Theory of computation → Separation logic

Keywords and phrases verification, separation logic, refinement types, translation validation, proof automation

Funding This material is based upon work supported by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agencies (DARPA) under Contract No. FA8750-24-C-B044. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the AFRL and DARPA.

1 Introduction

Verifying system software at scale has long been an elusive goal. Vast existing codebases of mission-critical software, including hypervisors, operating systems, vehicle control, and other

critical systems are written in memory-unsafe languages such as C and C++ that significantly complicate formal reasoning (and rewriting such code in safer languages is not always feasible). Safer languages such as Rust enforce code restrictions that guarantee memory safety and facilitate formal reasoning, but consequently also require parts of systems software to be written in unsafe code where the implementation relies on idioms incompatible with these restrictions (e.g. implementations of aliasing data structures or concurrency primitives that break safe Rust's aliasing-XOR-mutability restriction [16]).

Separation logic [15, 12] has proved to be particularly well-suited for reasoning about the safety and correctness of such low-level pointer-manipulating code. Tools for mechanised software verification building on separation logic broadly fall into two categories, with different trade-offs. Foundational separation logic frameworks, such as VST [1] and Iris [5] provide a high degree of confidence in the verification results: embedded in proof assistants such as Rocq, their soundness relies only on a small, independently trusted proof assistant kernel with well-established logical foundation, as well as the semantics of the programming language to be analysed. However, verification using these tools typically requires significant manual effort from expert users. Tools such as Refined C [17] and Diaframe [10] significantly improve proof automation but may rely on typing rules or proof hints provided by expert users.

In contrast, non-foundational verifiers such as VeriFast [4], Viper [11], Gillian [18, 7], and CN [14] typically provide a higher degree of proof automation, but as standalone verifiers do not offer the same degree of assurance as foundational tools: their separation logics do not come with mechanised soundness proofs against the programming language semantics, and they typically incorporate custom implementations of separation-logic and SMT-based proof automation that is not independently checked in a proof assistant kernel. (They are typically also significantly less expressive than foundational tools.)

We describe work-in-progress exploring a new approach to the design of separation logic verification tools that seeks to combine the proof automation of standalone verifiers with the strong soundness guarantees of foundational tools using a variant of *translation validation*, in which a standalone verifier generates *proof certificates* that are independently certified in a foundational *proof checker*. We build on CN, a separation-logic refinement type system for C systems software based on decidable SMT solving, whose verification case studies include industry hypervisor code [14]. While the CN paper proved soundness of type *checking* for a core calculus with explicit resource passing [13], this did not cover CN's proof automation, including resource inference. Moreover the proof was pen-and-paper only, not mechanised in a proof assistant. To increase confidence in CN's analysis, our overall goal in this line of work is to extend CN to produce proof certificates — programs fully annotated with CN-inferred separation-logic assertions — and certify these in a separate proof checker in Rocq. Our work to-date has focused on the key aspect of this problem not covered by the existing paper formalisation: the certification of CN's proof automation. We describe the architecture of the envisioned tool, our progress towards the certification of CN's separation logic resource inference, and details of the current Rocq implementation, which handles CN's folding and unfolding of separation-logic predicates for C structures but not yet user-defined predicates.

2 Roadmap to Foundationally Verified CN

The CN verifier builds on Cerberus, an existing well-validated semantics for ISO C [9, 8]. Cerberus defines the C semantics by elaboration into a functional language, called Core: for a given C program Cerberus generates Core code that makes explicit the subtleties of C, while Core itself has a straightforward semantics. CN exploits this, using Cerberus to

elaborate C to Core, which it then rewrites into CN's MuCore dialect for verification. CN is set up as a separation-logic refinement type system for MuCore, using linear separation logic types for tracking memory ownership and refinement types for reasoning about properties of the values.

CN's first-order specification language has separation logic types Owned_τ, points-to indexed by C-types τ, and user-defined resource predicates. Verification is largely automated using an off-the-shelf SMT solver for proof of pure/non-ownership assertions and custom resource inference for reasoning about separation logic ownership. CN's resource inference is inspired by a similar scheme in refinement types [19]:

- Resources in the proof context are *eagerly unfolded* when possible: Owned_τ resources, for complex C structures τ, are unfolded into Owned resources for the members (and padding bytes); applications of user-defined predicates are unfolded when this is possible without introducing case distinctions;
- When a resource is required for a memory access or user-provided specification but not available in the proof context, CN attempts to satisfy the "request" by *lazy folding*: satisfying the request for an Owned_τ resource, of struct type τ, by consuming a collection of member Owned resources, or satisfying the request for an instance of a user-defined predicate by unfolding its definition and issuing recursive requests.

While the original CN work included a paper formalisation and soundness proof of a kernel type system [13] this did not cover the resource inference that is the focus of this work, discussed in detail in Section 3.

For the overall verification strategy, we are considering two approaches: one is based on the earlier formulation of CN as a type system, essentially implementing a verified CN type checker in Rocq. The second, more conventional, approach is to use program logic. This approach is well-suited for formalisation in Rocq, with prior work to draw upon or reuse (e.g., we plan to use the Iris framework to implement CN's flavour of separation logic[5]).

The list of CN components that need to be formalised and verified (directly or via translation validation) below presents a high-level outline of the work to be done.

1. MuCore syntax – *completed.*
2. Resource inference – *partially completed.* (See Section 3.)
3. MuCore operational semantics – *future work.*
4. Memory model – *future work.* (There is an OCaml prototype [6] and a formalisation example [20] in Rocq.)
5. Linear separation logic – *future work.* (Potentially based on the Iris framework.)
6. Refinement type system – *future work.*
7. SMT witness checking – *future work.* (Using SMTCoq [2] or similar.)

As we progress along this roadmap, the current preliminary definition of the CN *proof certificate* introduced in Section 3 will be gradually expanded to include additional information necessary to verify these steps, and proof automation will be updated.

3 Verification of Resource Inference

The resource inference is a key component of CN. Building our verification chain from bottom up, we began by verifying low-level operations used in CN's linear separation logic:

Resource Unfolding - recursively transforms composite resources in the set into atomic ones. For instance, a resource corresponding to a C `struct` is unfolded into resources representing its fields and padding bytes.

Predicate Check - verifies whether a predicate matches one or more resources in a given set, consuming matched resources.

These operations are applied repeatedly during CN's resource inference. To verify their correctness, we extended CN to record them in Rocq syntax during `cn verify` command execution, generating a *proof log*. This log records successful resource inference steps with their arguments and contexts (before and after)[1] A resource inference step is represented in one of the following forms:

$$\text{PredicateRequest}(\Gamma_{\text{in}}, \phi_{\text{in}}, (\phi_{\text{out}}, \omega), h, \Gamma_{\text{out}})$$
$$\text{UnfoldResources}(\Gamma_{\text{in}}, h, \Gamma_{\text{out}})$$

The input and output contexts, Γ_{in} and Γ_{out}, are CN contexts of the form $\langle \mathcal{C}, \mathcal{L}, \mathcal{R}, \Phi, \mathcal{G} \rangle$. The components are as follows: the typing context $\mathcal{C}$, which provides type mappings for computational variables derived from the Core program; $\mathcal{L}$, which maps purely logical variables mentioned in specifications to their respective types; $\mathcal{R}$, the set of available resources, which is pairwise disjoint according to the *separating conjunction* in separation logic; Φ, which denotes a set of non-quantified SMT constraints; and $\mathcal{G}$, which contains global definitions of functions and types. In the first form, ϕ_{in} is the predicate to be resolved, and ϕ_{out} and ω are the output predicate and value, respectively. Additionally, both forms include a hints object h, discussed in Section 3.3.

To verify the proof log, we define correctness predicates for all operations and apply them to each log entry. We also want to demonstrate that these operations can be sequenced such that the output resources of one operation serve as input for the next.

Our correctness predicates' formulation is informed by the translation validation approach as we do not attempt to replicate the proof search performed by CN, but rather verify the results knowing that the proof search succeeded. Whenever possible, correctness conditions are formulated as inductive predicates. This allows us to exploit the bidirectionality of the relations. For example, function application `let y := f x` could be expressed as `exists y, f_rel x y` where `Inductive f_rel x y : Prop` is a relation that holds if `f x = y`. Then, as we have concrete values for x and y, recorded in the proof log, we can use proof automation to instantiate the existentials with their values and automatically prove the correctness condition.

3.1 Resource unfolding

In CN, a resource unfolding operation splits composite resources into atomic ones. For instance, a resource corresponding to a C `struct` type is unfolded into a set of separate resources, each corresponding to a field of the struct. Although CN implements resource unfolding as a function, for verification purposes we formalise it as a relation between a resource corresponding to a struct and a set of resources corresponding to its fields, defined in Rocq as an inductive predicate, parameterised by the global environment used to look up struct fields definitions:

```
Inductive unfold_one (globals:Global.t): Resource.t → ResSet.t → Prop.
```

Or more formally, omitting the global environment parameter for brevity: Let R be a set of resources and let unfold_one $\subseteq R \times \mathcal{P}(R)$ be a relation such that $(r, U) \in$ unfold_one means that resource r unfolds to the set U.

[1] It also contains the MuCore AST, which will be necessary for verifying other reasoning steps later.

An unfolding step of CN resource inference recorded in the proof log documents a fixed point of unfolding of resources in the input set. We can define the fixed point of unfolding as a binary relation unfold_all $\subseteq \mathcal{P}(R) \times \mathcal{P}(R)$ by induction as follows:

Step case:

$$\forall\, X, Y \subseteq R:$$
$$\big(\exists\, r \in X,\ \exists U \subseteq R:\ (r, U) \in \text{unfold\_one} \wedge \text{unfold\_all}\big((X \setminus \{r\}) \cup U,\ Y\big)\big)$$
$$\implies \text{unfold\_all}(X, Y).$$

Fixpoint case:

$$\forall\, X \subseteq R:$$
$$\big(\forall\, r \in X,\ \forall U \subseteq R:\ (r, U) \notin \text{unfold\_one}\big) \implies \text{unfold\_all}(X, X).$$

We then formulate the correctness predicate for the unfolding step:

$$\forall\, \Gamma_{\text{in}},\ \Gamma_{\text{out}},\ h:$$
$$\text{unfold\_all}(\Gamma_{\text{in}}.\mathcal{R},\ \Gamma_{\text{out}}.\mathcal{R}) \implies$$
$$\text{log\_entry\_valid}\ \big(\ \text{UnfoldResources}(\Gamma_{\text{in}}, h, \Gamma_{\text{out}})\big)$$

3.2 Predicate check

The predicate check step attempts to obtain a resource matching a given predicate. A typical use is to ensure ownership of a memory pointed by a pointer. The predicate is matched against a set of available resources, and if a match is found, the matched resources are consumed (removed from the resource context).

For illustration, let us consider the simple case where the predicate matches exactly one resource. In Rocq, the correctness of this step is defined as an inductive predicate and can be expressed by the following formula:

$$\begin{aligned}
&\forall\, \Gamma_{\text{in}},\ \Gamma_{\text{out}},\ \phi_{\text{in}},\ \phi_{\text{out}},\ \omega,\ h:\\
&\quad\Big[\ \phi_{\text{in}} = \phi_{\text{out}}\\
&\qquad\wedge\ \Gamma_{\text{in}}.\mathcal{C} = \Gamma_{\text{out}}.\mathcal{C} \wedge \Gamma_{\text{in}}.\mathcal{G} = \Gamma_{\text{out}}.\mathcal{G} \wedge\ \Gamma_{\text{in}}.\mathcal{L} = \Gamma_{\text{out}}.\mathcal{L} \wedge \Gamma_{\text{in}}.\Phi = \Gamma_{\text{out}}.\Phi\\
&\qquad\wedge\ \exists\, \phi':\ \Big(\Gamma_{\text{in}}.\mathcal{R} = \Gamma_{\text{out}}.\mathcal{R} \cup \{(\phi',\ \omega)\}\\
&\qquad\quad\wedge\ \text{subsumed}(\phi_{\text{in}}.\text{name},\ \phi'.\text{name})\\
&\qquad\quad\wedge\ \text{alloc_id_eq}(\Gamma_{\text{in}}.\mathcal{G},\ \Gamma_{\text{in}}.\Phi,\ \phi_{\text{in}}.\text{pointer},\ \phi'.\text{pointer})\\
&\qquad\quad\wedge\ \text{args_eq}(\Gamma_{\text{in}}.\mathcal{G},\ \Gamma_{\text{in}}.\Phi,\ \phi_{\text{in}}.\text{pointer},\ \phi'.\text{pointer},\ \phi_{\text{in}}.\text{iargs},\ \phi'.\text{iargs})\Big)\\
&\quad\Big]\ \implies\\
&\quad\text{log_entry_valid}\big(\text{PredicateRequest}(\Gamma_{\text{in}}, \phi_{\text{in}}, (\phi_{\text{out}},\ \omega), h, \Gamma_{\text{out}})\big)
\end{aligned}$$

A log entry corresponding to this resource inference step is correct only if specific conditions are met. Firstly, the returned predicate must exactly match the requested one ($\phi_{\text{in}} = \phi_{\text{out}}$). Additionally, the computational ($\mathcal{C}$), global ($\mathcal{G}$), logical ($\mathcal{L}$), and constraints (Φ) components of both the input and output contexts must remain unchanged. Furthermore, there must be precisely one resource $(\text{P}(\phi'),\ \omega)$ present in Γ_{in} but absent in Γ_{out}. For this resource, the predicate name in ϕ_{in} must be subsumed by that in ϕ'.

During CN proof search, there are some additional constraints verified by the SMT solver. For successful steps, it ensures, using the global ($\mathcal{G}$) and constraints (Φ) components of the input context, that the allocation id of Γ_{in}.pointer matches that of ϕ'.pointer, the address of Γ_{in}.pointer is equivalent to that of ϕ'.pointer, and there is a pairwise equality between the `iargs` components of ϕ_{in} and ϕ'. In our Rocq formalisation, there is a placeholder predicate `provable`, which will be implemented in the future using a witness recorded in the proof log using SMTCoq [2] or a custom plugin.

A more general case is where a predicate request for a C `struct` must match not a single resource, but resources corresponding to all its fields. The formalisation of this case is expressed using the `unfold_all` relation described in the previous section, applied to $(\phi_{\text{in}}, \omega)$.

3.3 Proof Automation

We aim to automatically verify a proof log without human intervention. To this end, we employ a combination of custom Ltac2 proof automation together with Rocq standard library tactics implementing decision procedures for finite sets (`FSetDecide` library).

We found Ltac2 proof automation suitable for simple goals, but it is cumbersome due to the deep embedding of Rocq terms, requiring complex introspection and pattern matching. To address this we decided to switch to a *computational reflection* [3] approach. This involves defining computable predicate functions in Gallina and proving their logical equivalence to inductive predicates, allowing evaluation within the proof assistant to fulfil proof obligations. For instance, to verify the unfold_all predicate from Section 3.1, we can ensure the recursive boolean function returns `true` using the `vm_compute` tactic. Although it is still a work in progress, it has shown promising results and demonstrates good performance.

We also use *hints*, which are records of CN's proof search steps. While not part of high-level proof log correctness predicates, hints can drive proof automation. We define a new predicate using hints and prove it implies the original one. For instance, for the unfold_all predicate, hints may include individual unfold_one steps. Then, for computational reflection, we define a function that recurses over hints, applying unfolding steps repeatedly. This provides us with the advantage of a clear mathematical specification of correctness, along with the benefits of simple proof automation and good performance.

One of the challenges we encountered was the performance of our automated proof log checking in Rocq. The proof log files even for small programs are quite large (tens of megabytes) and on bigger ones Rocq runs out of memory. We have a few ideas on how to improve this which we plan to implement in the future.

4 Conclusions and Future Work

This is a work in progress. Thus far, we have achieved partial verification of resource inference with proof automation. We have also laid down the foundations for the remaining parts: formalised the MuCore syntax and core types for CN logic (contexts, resources, predicates, etc.) in Rocq, and instrumented CN OCaml code to produce proof logs for resource inference steps. With the core of the proof and automation framework in place, completing the remaining parts of resource inference verification appears straightforward.

Given that resource inference is a critical component of CN, our verification, although partial, and the generated proof certificates for independent validation, could already enhance users' confidence in CN's positive verification outcomes.

We plan to extend our verification to other components with the ultimate goal of achieving the complete foundational verification of CN proofs, as outlined in the tentative roadmap in Section 2. Upon achieving this milestone, several more ambitious directions open up. One is to extract some of the CN reasoning logic from Rocq to OCaml as a verified implementation and integrate it into the tool. Furthermore, we could extend the CN specification language and logic with more powerful constructs, which are optional but would require interactive manual verification in Rocq, for which we can provide some notations, lemmas, and tactics.

References

1 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. doi:10.1017/CBO9781107256552.

2 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011. doi:10.1007/978-3-642-25379-9_12.

3 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997. URL: https://doi.org/10.1007/BFb0014565, doi:10.1007/BFB0014565.

4 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

5 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

6 Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. VIP: verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.*, 6(POPL):1–32, 2022. doi:10.1145/3498681.

7 Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: real-world verification for javascript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 827–850. Springer, 2021. doi:10.1007/978-3-030-81688-9_38.

8 Kayvan Memarian. The Cerberus C semantics. Technical Report UCAM-CL-TR-981, University of Cambridge, Computer Laboratory, May 2023. PhD thesis. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-981.pdf, doi:10.48456/tr-981.

9 Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 1–15. ACM, 2016. doi:10.1145/2908080.2908081.

10 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: automated verification of fine-grained concurrent programs in iris. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and*

Implementation, San Diego, CA, USA, June 13 - 17, 2022, pages 809–824. ACM, 2022. doi:10.1145/3519939.3523432.

11 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.

12 Peter W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019. doi:10.1145/3211968.

13 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. Cn type system formalisation for "cn: Verifying systems c code with separation-logic refinement types", November 2022. doi:10.5281/zenodo.7323297.

14 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: verifying systems C code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL):1–32, 2023. doi:10.1145/3571194.

15 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

16 Rust Project Developers. The Rustonomicon. https://doc.rust-lang.org/nomicon/, 2023. Accessed: 24 March 2025.

17 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: automating the foundational verification of C code with refined ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 158–174. ACM, 2021. doi:10.1145/3453483.3454036.

18 José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: a multi-language platform for symbolic execution. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 927–942. ACM, 2020. doi:10.1145/3385412.3386014.

19 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018. doi:10.1145/3158141.

20 Vadim Zaliva, Kayvan Memarian, Brian Campbell, Ricardo Almeida, Nathaniel Wesley Filardo, Ian Stark, and Peter Sewell. A CHERI C memory model for verified temporal safety. In Kathrin Stark, Amin Timany, Sandrine Blazy, and Nicolas Tabareau, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, pages 112–126. ACM, 2025. doi:10.1145/3703595.3705878.