Lean on Vampire Proofs

Jonas Bodingbauer
TU Wien, Austria

Márton Hajdu
TU Wien, Austria

Laura Kovács
TU Wien, Austria

Axel Polaczek
TU Wien, Austria

Michael Rawson
University of Southampton, UK

Abstract

Vampire proves theorems completely automatically in first- and higher-order logic extended with theories. Proof checking is increasingly demanded to consolidate user trust in Vampire's output. We describe ongoing efforts in reconstructing Vampire proofs as trusted proofs in Lean. Our experiments showcase feasibility of generating trusted Vampire proofs that are validated in Lean.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Theory of computation → Type theory; Theory of computation → Higher order logic; Computing methodologies → Theorem proving algorithms

Keywords and phrases Automated Reasoning, Interactive Theorem Provers, Automated Theorem Provers, Lean, Vampire, Proof Reconstruction

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.36

Category Short Paper

Related Version *Previous Version*: `https://arxiv.org/abs/2603.26342`

Supplementary Material *Software*: `https://github.com/vprover/vampire/tree/leancheck`
archived at `swh:1:rev:ca13c0a9d5c3a301c7403026a2c0d57faf882e7b`
Software: `https://github.com/vprover/vamplean`
archived at `swh:1:rev:d41e04f26cdc9173648a4b2e54753b59030484c4`

Funding This research was funded in whole or in part by the ERC Consolidator Grant ARTIST 101002685, the ERC Proof of Concept Grant LEARN 101213411, and by the SBA Research COMET Center SBA-K1 NGC managed by the FFG.

Acknowledgements We thank Henrik Böving for providing helpful insights on Lean.

Declaration about GenAI/LLM use Generative AI tools were not used in the writing of this paper. GitHub Copilot suggestions (that mainly rely on GPT5-mini) in VS Code have been used in our implementation.

1 Introduction

Computer-assisted proofs are indispensable in applications of mathematics [23, 34, 10], cybersecurity [31, 17, 21], software analysis [7, 1, 16], neural networks [5, 15], and many other areas. Automated reasoners, such as automated theorem provers (ATPs) and satisfiability

1.	$\mathsf{mul}(x, \mathsf{mul}(y, z)) = \mathsf{mul}(\mathsf{mul}(x, y), z)$	Input
2.	$\mathsf{mul}(\mathsf{id}, x) = x$	Input
4.	$\mathsf{id} = \mathsf{mul}(x, x)$	Input
6.	$\neg\forall x, y : \mathsf{mul}(x, y) = \mathsf{mul}(y, x)$	Neg. Goal
7.	$\exists x, y : \mathsf{mul}(x, y) \neq \mathsf{mul}(y, x)$	ENNF 6
8.	$\mathsf{mul}(\sigma_0, \sigma_1) \neq \mathsf{mul}(\sigma_1, \sigma_0)$	Skolem 7
15.	$\mathsf{mul}(x, \mathsf{mul}(x, y)) = \mathsf{mul}(\mathsf{id}, y)$	Sup 1,4
19.	$\mathsf{id} = \mathsf{mul}(x, \mathsf{mul}(y, \mathsf{mul}(x, y)))$	Sup 4,1
22.	$\mathsf{mul}(x, \mathsf{mul}(x, y)) = y$	FwDem 15,2
25.	$\mathsf{mul}(x, \mathsf{id}) = x$	Sup 22,4
80.	$\mathsf{mul}(y, \mathsf{mul}(x, y)) = \mathsf{mul}(x, \mathsf{id})$	Sup 22,19
85.	$\mathsf{mul}(y, \mathsf{mul}(x, y)) = x$	FwDem 80,25
160.	$\mathsf{mul}(x, y) = \mathsf{mul}(y, x)$	Sup 22,85
249.	$\mathsf{mul}(\sigma_0, \sigma_1) \neq \mathsf{mul}(\sigma_0, \sigma_1)$	Sup 8,160
250.	**false**	Resolve 249

Figure 1 Simplified Vampire proof showing that if every element of a group has order two, the group is commutative. Here, mul denotes the group operator and id the group's identity element. The variables x, y are implicitly universally quantified, whereas σ_0 and σ_1 are Skolem constants. The proof steps Sup, FwDem respectively denote superposition and its variant forward demodulation.

modulo theory (SMT) solvers, provide push-button solutions to machine-based proof generation. Proofs returned by automated reasoners are can be checked by e.g. using an external ATP/SMT solver to reprove each step of the generated proof [29, 20, 2]. Alternatively, one can rely on interactive theorem proving (ITP) to replay the proof inference-by-inference [24, 25].

Our contributions. This paper addresses proof checking for the Vampire theorem prover [6], making proof checking feasible and adjustable. We enhance Vampire with *trusted proofs*: proofs generated by Vampire are reconstructed in, and checked by, the Lean interactive theorem prover [13]. We describe our ongoing efforts in extending Vampire with Lean-based proof formats (Section 3), allowing us to use Lean for increasing user trust in Vampire proofs. Our experiments showcase scalability of using Lean for trusting Vampire (Section 4).

Related methods. have emerged from both ATP and ITP communities. We complement the interfacing efforts of Lean-auto [28] by providing a method for validating Vampire proofs in Lean. Compared to the ground truth approach of [29], trust in Vampire proofs is handed off to Lean, rather than an SMT solver, significantly reducing the trusted computing base. Methods are availiable for verifying SMT proofs in Lean [25] and Isabelle [9]. TESC [4] checks first-order proofs in a prover-independent manner using a formally verified proof-checker written in Agda, proof details being reconstructed during proof translation into an intermediate proof format.

Our work makes the first step towards a long-term vision turning Vampire into a hammer for Lean, complementing existing efforts in ATP/ITP integration [37, 18] and continuing a long tradition of Vampire service in hammer systems [14].

2 Illustrative Example and Scope

Saturation and Vampire. Proof search in Vampire is performed via saturation [19]: given a conjecture F in (extensions of) first-order logic (FOL), Vampire negates and clausifies $\neg F$ and tries to derive falsum from the resulting clause set by saturating the set with respect to the the superposition calculus [3, 26]. During proof search, clauses may be removed if they are determined to be *redundant*. Whenever the empty clause **false** is derived as a consequence of $\neg F$, (classical) validity of F is established and a proof is generated as part of the Vampire output. Figure 1 shows a sample proof in which Vampire applies several inferences described in more detail in Section 3.3. Put simply, inference rules of the superposition calculus rewrite terms by equal terms, embedding equality reasoning in a first-order calculus.

Checking Vampire proofs for soundness. To improve efficiency of proof search Vampire implements dedicated inference rules for equality reasoning, redundancy checking, arithmetic, and induction [6], among others. These advances increase the number of inference rules in Vampire to around 200. Our work identifies a core set of inference rules (Section 3) from which trusted proofs of Vampire are generated. Trust in Vampire proofs is achieved by the help of Lean: (i) we extend Vampire to output proofs as Lean theorems proven using lemmas corresponding to inferences. Once Vampire finds a proof, a Lean input file is generated where tactics are used to (ii) *reconstruct Vampire inferences in Lean*, demonstrating soundness for each step. A final combination of these steps results in a full Vampire proof from assumptions to conclusions trusted by Lean. For example, each Vampire rule from Figure 1 is converted to a Lean theorem and proven by a collection of dedicated tactics, such as the superposition in Listing 3.

3 Lean checked Vampire Proofs

We produce Lean-checkable proofs for problems solved by Vampire (Section 3.1). We report on our efforts in reconstructing the preprocessing steps (Section 3.2) and the most important inference rules (Section 3.3) of Vampire in Lean. While this paper focuses on standard first-order logic with equality, extending our work to multi-sorted first-order logic is an immediate step for future work.

3.1 Implementation

Listing 1 Third part of a Lean input file, representing a trusted Vampire proof corresponding to Figure 1. It combines the previously proven lemmas corresponding to inference steps for the full proof of Figure 1.

```
theorem fullProof : (∀ v0 v1 v2 : ι, ((mul (mul v0 v1) v2)=(mul v0 (mul v1 v2)))) →
(∀ v0 : ι, ((mul id v0)=v0)) → (∀ v0 : ι, (id=(mul v0 v0))) →
(∀ v0 v1 : ι, ((mul v0 v1)=(mul v1 v0))) := by
  intros step1 step2 step3
  apply Classical.byContradiction; intro step5
  ...
  have step160 := inf_s160 step22 step85
  have step249 := inf_s249 step8 step160
```

Our approach for generating trusted Vampire proofs is implemented as a shared effort between Vampire and Lean: once a Vampire proof is generated, we output a Lean file in which we construct proof terms in Lean using tactics provided by Lean and mathlib [22]. Generating proof terms can be assisted by additional information gained through inference replay in Vampire, reconstructing most general unifiers (mgu) in Section 3.3.

General structure of trusted Vampire proofs. A Lean file representing a trusted Vampire proof is structured in three parts: after the preamble that declares all required symbols, the second (typically largest) section includes all individual inference steps as theorems. Each theorem is proven using a collection of Lean tactics, or in very simple cases directly by providing suitable proof terms. Tactics are often needed, because Vampire inferences do not retain the ordering of literals in clauses, nor the orientation of equalities and hence require suitable applications of associativity and commutativity (AC) and symmetry. The third part is a proof of the input conjecture, which shows that the set of axioms imply the conclusion by using proven lemmas to reconstruct the proof found by Vampire. It includes symbol introduction, such as Skolemization, which are part of Vampire's preprocessing (see Figure 1). Formulas are encoded in a natural way in Lean: the primitives of first-order formulas are used as defined by Lean core and mathlib [22].

3.2 Preprocessing

Several preprocessing steps are used in Vampire. To transform a first-order formula F (i.e. conjecture) into clausal normal form (CNF), the formula F is (typically) rectified, simplified, flattened, transformed to equivalence negation normal form (ENNF),[1] named, converted to NNF, skolemized, and finally transformed to conjunctive normal form [30]. For ENNF transformation as well as flattening and simplification, we use the Lean `simp` tactic, with suitable theorems.[2] For the transformation from ENNF to CNF, our implementation currently uses custom tactics in combination with `grind`. Furthermore, some preprocessing steps introduce new symbols, which are defined globally in the Lean file and used via suitable equivalence theorems.

Skolemization. After the input conjecture F has been transformed to NNF, the existential quantifiers are replaced with Skolem functions of suitable arity. The default implementation of Vampire replaces existentially-quantified variables by functions that are dependent on free variables of the immediate subformulas. Automating this behavior in proof reconstruction is challenging; we therefore implemented an option to change Vampire skolemization to use all currently bound variables (which is always a superset of the previously bound variables). For showing correctness of the employed skolemization mechanism, we prenex the existential quantifiers using the theorem `Classical.skolem` of type: $(\forall(x:\alpha),\exists(y:b\ x),p\ x\ y) \leftrightarrow \exists(f:(x:\alpha\to b\ x)),\forall(x:\alpha),p\ x\ (f\ x)$. Additional theorems that allow prenexing an existential from both $\vee$ and $\wedge$ are also used. For the formula in existential prenex form, each of the quantifiers can then be assigned a new function name. Special care needs to be taken to match the ordering of Skolem functions assigned to variables as well as the orientation of equalities which is handled by custom tactics. The skolemization step 8 of the running example shown in Figure 1 is given in Listing 2.

Listing 2 Skolemization of the two existentially quantified variables in step 8. This snippet is from the third part of a trusted proof and assembles all inferences. `symm_match` and `exists_prenex` are custom tactics which take care of the orientation of equalities and prenexing.

```
exists_prenex at step7 -- step8 skolemisation
let ⟨sK0,sK1,step8'⟩ := step7
have step8 : (¬((mult sK0 sK1)=(mult sK1 sK0))) := by symm_match using step8'
```

3.3 Inference Rules within Saturation

Saturation-based proof search in Vampire implements the superposition calculus and uses the following core rules: resolution, factoring, paramodulation, equality resolution, and equality factoring. (Equality) resolution and (equality) factoring are based on the first-order resolution calculus adapted for FOL with equality. The paramodulation rule, shown in Listing 3(a), is used to rewrite terms by an equal term, where syntactic equality between these terms is achieved via unification producing the most general unifier (mgu). For efficient proof search, the paramodulation rule is further restricted to be applied only to clauses/literals that respect conditions on some term ordering $\succ$; for example, rewriting is applied only over terms that are maximal wrt $\succ$. This ordered version of paramodulation is referred to as superposition [27]. Another ordered variant of paramodulation is demodulation [36, 19], also

[1] Equivalence NNF allows $\leftrightarrow$ and $\oplus$, with negations in front of atoms.
[2] In (E)NNF transformation, Vampire by default uses non-confluent rewrite rules, however its outermost reduction strategy can be replicated using the $\downarrow$ annotation.

called rewriting with unit equality [32]: in this rule, one premise is a unit equality (i.e. C is empty) and additional (redundancy) constraints are enforced in order to simplify clauses in other premises (i.e. D). Importantly, from a proof-checking perspective, all of these inferences may be regarded as instances of paramodulation.
Listing 3 demonstrates superposition used by Vampire at step 160 of Figure 1. It is translated as a Lean theorem that takes two premises and gives a conclusion. At the beginning of the theorem, all premises are introduced as well as the variables of the conclusion. The most general unifier (mgu) found by Vampire is instantiated via `have` statements. After that, the `grind` tactic is called, which is well-suited for ground reasoning and can effectively reconstruct proof terms.[3]

Other inference rules of Vampire follow the same steps: introduce premises and conclusion variables, instantiate them according to the unifier found by Vampire, and then dispatch that to `grind`. When it occurs that two variables are unified, the unifier can contain a variable that is not present in the conclusion [29]. In this case, the default element of the inhabited first order domain is used. In the future, more specialized, custom, tactics could be employed that can use additional information provided by Vampire such as the positions where rewrites occurred.[4]

Listing 3 (a) Lean theorem representing the superposition step 160 of Figure 1. (b) The instantiation of paramodulation (i.e. more general superposition) for (a). Clauses C, D are empty for the instance of Figure 1.

(a)

```
theorem inf_s160 :
 (∀ x₀ y₀ : ι, ((mul y₀ (mul x₀ y₀))=x₀)) →
 (∀ x₁ y₁ : ι, ((mul x₁ (mul x₁ y₁))=y₁)) →
 (∀ x  y  : ι, ((mul x y)=(mul y x))) := by
 intros h0 h1 x y
 have i0 := h0 x y
 have i1 := h1 y (mul x y)
 grind only [cases Or]
```

(b) General paramodulation rule.

$$\frac{l = r \vee C \qquad L[k] \vee D}{(L[r] \vee C \vee D)\sigma}\ \sigma = \mathrm{mgu}(l, k)$$

l	$\mathsf{mul}(y_0, \mathsf{mul}(x_0, y_0))$
r	x_0
$L[k]$	$\mathsf{mul}(x_1, \mathsf{mul}(x_1, y_1)) = y_1$
σ	$\{x_0 \mapsto x, y_0 \mapsto y,$ $x_1 \mapsto y, y_1 \mapsto \mathsf{mul}(x, y)\}$

3.4 AVATAR

A feature that contributes to Vampire's success is Avatar [35], which enables efficient reasoning over the propositional/quantifier-free structure of clauses. Avatar splits clauses into variable-disjoint sub-clauses and defines propositional labels for each of the sub-clauses, as follows: $\dfrac{\neg Q(z) \vee P(x, f(z)) \vee R(y)}{\mathrm{sp}_1 \vee \mathrm{sp}_2}$ using $\begin{array}{l}\mathrm{sp}_1 \equiv \neg Q(z) \vee P(x, f(z)) \\ \mathrm{sp}_2 \equiv R(y)\end{array}$
A SAT solver in Avatar is used to reason about the propositional structure $\mathrm{sp}_1 \vee \mathrm{sp}_2$, while Vampire handles Avatar components of the form $R(y) \leftarrow \mathrm{sp}_2$, which should be understood as "$R(y)$ if sp_2". If the SAT solver finds that the propositional structure is unsatisfiable, a refutation is produced. When the SAT solver finds a model, Vampire continues reasoning using the learned knowledge from the propositional model and derives new contradictions using first-order reasoning, that can in turn be used to define new propositional clauses to be passed to the SAT solver. When a Vampire proof is found with Avatar, the final contradiction (**false**) is derived by the SAT solver.

[3] This approach is inspired by a previous approach where ground instances are passed to an SMT solver [29]
[4] Vampire does not preserve literal ordering, therefore a tactic that takes into account AC of $\vee$ is required.

Avatar in Lean. To reconstruct and check Avatar steps in Lean, the propositional names are declared and defined in the first and last parts of Vampire proofs (Section 3.1). When a proof step involves an Avatar component, the propositional Avatar labels for the conclusion (e.g. sp_1) are introduced before introducing the conjectures variables; these labels are used to instantiate premises, as shown in Listing 4. The other rules are treated as additional inference rules, where the splitting of clauses is treated using rewrites as well as `grind`, which only needs to handle AC reordering as well as removing potentially duplicate literals. The final SAT refutation is handled using the `bv_decide` tactic, which calls a SAT solver and then verifies the received proof within Lean [11].

Listing 4 Variant of the inference rule of Listing 3 if it were part of an Avatar proof. The final SAT proof, which is reconstructed using `bv_decide`, is also listed.

```
theorem inf_s160_avatar : (sP1→(∀ x₀ y₀ : ι, ((mul y₀ (mul x₀ y₀))=x₀))) →
 (∀ x₁ y₁ : ι, ((mul x₁ (mul x₁ y₁))=y₁)) →
 (sP1→(∀ x  y  : ι, ((mul x y)=(mul y x)))) := by
 intros h0 h1 sp1 x y
 have i0 := h0 sp1 x y
 ...
theorem avatarRefutation' (sP1' sP2' sP3' : Bool) : (sP1' || sP2' || sP3') → (!sP2') →
     (!sP3') → (!sP1') → False := by bv_decide
```

4 Experiments

Our experiments used benchmarks from version 9.2.1 of the TPTP library [33] which is a well known problem set for ATPs with a large variety of application domains and sources. We relied on BenchExec [8] for conducting experiments, using an AMD Epyc 7502 2.5GHz processor and 1TB RAM. Each benchmark ran with a single core and 16GB of memory. Our benchmark set consists of all problems in clause normal form (CNF) and first oder form (FOF) from TPTP, overall 17,603 problem instances. Our Vampire runs used the base version 5.0.1 with the Discount saturation loop and a 100-giga-instruction (≈ 30 s) resource limit (`-sa discount -i 100000`); we refer to this Vampire version as baseline Vampire. For Lean, we used a 1000-second timeout and 64 GB of RAM and the Version 4.29.0.

We compare (i) baseline Vampire without proof output (`--proof off`) and (ii) our work (Section 3.1) enhancing Vampire (Tag: `Leancheck2026`) with output of proofs checkable by Lean, using the options (`--skolemization syntactic --proof_extra lean --proof leancheck -om lean -sa discount -ppr off -i 100000`). Table 1 summarizes our results, showcasing 99% success rate of proof generation and checking on both CNF and FOF problems within the chosen timeouts. We note that CNF proofs greatly reduce the load of reconstructing preprocessing steps in Lean, since these problems are already in the form required for the saturation loop and do not need (many) preprocessing steps. For problems whose Vampire proofs cannot yet be checked by Lean, the reasons stem from large Lean files (columns 4, 5) or hard to reconstruct inferences, as well as a too small thread stack (column 6). Figure 2**(a)** indicates that proof replay and the two new options

Table 1 Number of refutation proofs found by Vampire with our new skolemization mechanism on and pure predicate removal (ppr) off (column 2) vs. Lean proof reconstructions (columns 3–6), sorted by success/failure reasons.

TPTP	Vampire	Lean			
		Success	Timeout	Out of Memory	Stack overflow
CNF	4118	4062	31	24	1
FOF	4216	4175	35	3	3

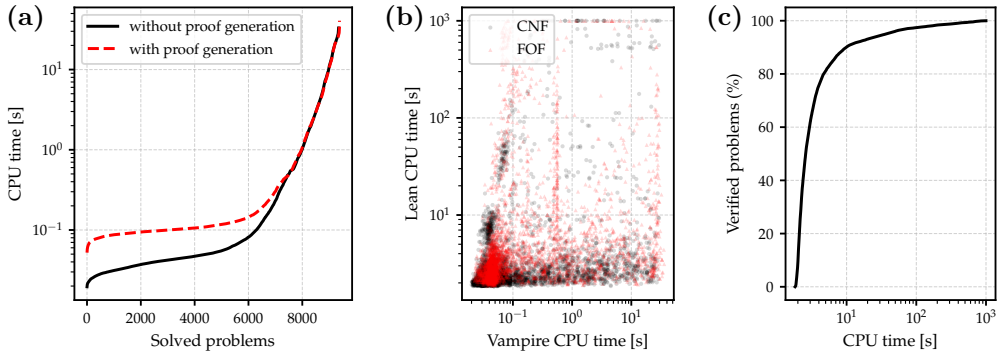

Figure 2 **(a)** Comparing baseline Vampire and our Vampire version outputting a Lean file; here, we include only CNF and FOF problems where baseline Vampire found a proof by contradiction. This plot only accounts for the overhead in Vampire, not for Lean proof reconstruction and checking time. Notice the logarithmic y-axis. **(b)** A scatter plot comparing the Lean proof reconstruction and checking runtime vs the Vampire runtime. **(c)** Percentage of successfully checked problems by Lean for problems that are solved by the custom output producing Vampire in ≤ 1 s.

have a non-negligible influence on proofs that have been found (comparatively) quickly. This may be explained by proof replay taking time when there are lots of possible inferences from two premises. However, in total, 26 more proofs are found due to the efficacy of the modified skolemization tactic compared to baseline Vampire. Figure 2**(b)** shows that there is only minimal correlation between Vampire proof search and Lean proof reconstruction and checking. Furthermore, the time spent in Lean is typically up to two magnitudes longer, which is also due to a long startup time of approx. 1.8 s (loading dependencies). Finally, Figure 2**(c)** shows that 81% of problems that can be solved by Vampire within 1 s are checkable in Lean within 5 s, which is amounts to a tolerable total delay of 6 s for proof search in interactive proofs, showing a potential path towards a hammer using Vampire.

5 Conclusions and Future Work

We describe ongoing efforts on reconstructing Vampire proofs in Lean, allowing us to trust Vampire proofs thanks to Lean's kernel (alongside a trustworthy SAT proof-checker [11]). Our experiments show scalability of generating trusted Vampire proofs. We handle only a small fraction of the inference steps implemented by Vampire so far. To remedy this, the proof-producing Duper ATP [12] may be invoked if no specialized set of tactics is available. We did not discover bugs in Vampire, which can be explained by the generally extremely well-tested problem set as well as the longstanding support for first-order logic in Vampire. Nevertheless, our approach enables (for the first time) fully checking Vampire proofs by a trusted kernel, greatly increasing the trustworthiness of Vampire. Further work plans addressing various efficiency limits of proof reconstruction in Lean as well as supporting theories. More work towards specialized tactics for proof reconstruction for specific inferences may lead to additional speedup. The general proof format and structure may also be further optimized, while also reducing the load on Lean elaboration by explicitly stating dependencies. The evaluation on TPTP did not yet clearly show the efficacy of our approach for use as a hammer, as translations from dependent type theory to first-order logic, like Lean-auto, may result in different needs for efficient proof reconstruction.

References

1 Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY Approach: Integrating Object Oriented Design and Formal Verification. In *JELIA*, 2000. doi:10.1007/3-540-40006-0_3.

2 Michael 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 *CPP*, 2011. doi:10.1007/978-3-642-25379-9_12.

3 Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*. North-Holland, 2001. doi:10.1016/B978-044450813-3/50004-7.

4 Seulkee Baek. A Formally Verified Checker for First-Order Proofs. In *ITP*, Leibniz International Proceedings in Informatics (LIPIcs), pages 6:1–6:13, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2021.6.

5 Clark W. Barrett, Thomas A. Henzinger, and Sanjit A. Seshia. Certificates in AI: Learn but Verify. *Communications of ACM*, 2026. doi:10.1145/3737447.

6 Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdú, Matthias Hetzenberger, Petra Hozzová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, Martin Suda, Johannes Schoisswohl, and Andrei Voronkov. The Vampire Diary. In *CAV*, 2025. doi:10.1007/978-3-031-98682-6_4.

7 Raven Beutner and Bernd Finkbeiner. Checking Satisfiability of Hyperproperties Using First-Order Logic. In *ATVA*, 2024. doi:10.1007/978-3-031-78750-8_10.

8 Dirk Beyer. Reliable and Reproducible Competition Results with BenchExec and Witnesses Report on SV-COMP 2016. In *TACAS*, 2016. doi:10.1007/978-3-662-49674-9_55.

9 Sascha Böhme. Proof reconstruction for Z3 in Isabelle/HOL. In *SMT Workshop*, 2009.

10 Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, Floris van Doorn, Martin Dvorak, Andrés Goens, Aaron Hill, Harald Husum, Hernán Ibarra Mejia, Zoltan A. Kocsis, Bruno Le Floch, Amir Livne Bar-on, Lorenzo Luccioli, Douglas McNeil, Alex Meiburg, Pietro Monticone, Pace P. Nielsen, Emmanuel Osalotioman Osazuwa, Giovanni Paolini, Marco Petracci, Bernhard Reinke, David Renshaw, Marcus Rossel, Cody Roux, Jérémy Scanvic, Shreyas Srinivas, Anand Rao Tadipatri, Terence Tao, Vlad Tsyrklevich, Fernando Vaquerizo-Villar, Daniel Weber, and Fan Zheng. The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale, 2025. doi:10.48550/arXiv.2512.07087.

11 Henrik Böving, Siddharth Bhat, Luisa Cicolini, Alex Keizer, Léon Frenot, Abdalrhman Mohamed, Léo Stefanesco, Harun Khan, Joshua Clune, Clark Barrett, and Tobias Grosser. Interactive Bitvector Reasoning using Verified Bit-Blasting. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2025. doi:10.1145/3763167.

12 Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad. Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory. In *ITP*, 2024. doi:10.4230/LIPIcs.ITP.2024.10.

13 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *CADE*, 2021. doi:10.1007/978-3-030-79876-5_37.

14 Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen Provers Under the Hammer. In *ITP*, 2022. doi:10.4230/LIPIcs.ITP.2022.8.

15 Remi Desmartin, Omri Isac, Grant O. Passmore, Ekaterina Komendantskaya, Kathrin Stark, and Guy Katz. A Certified Proof Checker for Deep Neural Network Verification in Imandra. In *ITP*, 2025. doi:10.4230/LIPIcs.ITP.2025.1.

16 Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, and Giles Reger. The Rapid Software Verification Framework. In *FMCAD*, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_32.

17 Simon Jeanteur, Laura Kovács, Matteo Maffei, and Michael Rawson. CryptoVampire: Automated Reasoning for the Complete Symbolic Attacker Cryptographic Model. In *IEEE SP*, 2024. doi:10.1109/SP54263.2024.00246.

18 Anja Petkovic Komel, Michael Rawson, and Martin Suda. Case Study: Verified Vampire Proofs in the LambdaPi-calculus Modulo. *CoRR*, 2025. doi:10.48550/arXiv.2503.15541.

19 Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *CAV*, 2013. doi:10.1007/978-3-642-39799-8_1.

20 Hanna Lachnitt, Mathias Fleury, Leni Aniva, Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. IsaRare: Automatic Verification of SMT Rewrites in Isabelle/HOL. In *TACAS*, 2024. doi:10.1007/978-3-031-57246-3_17.

21 Evan Laufer, Alex Ozdemir, and Dan Boneh. zkPi: Proving Lean Theorems in Zero-Knowledge. In *CCS*, 2024. doi:10.1145/3658644.3670322.

22 The mathlib Community. The lean mathematical library. In *CPP*, 2020. doi:10.1145/3372885.3373824.

23 William McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 1997. doi:10.1023/A:1005843212881.

24 Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *Journal of Automated Reasoning*, 2008. doi:10.1007/s10817-007-9085-y.

25 Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark W. Barrett. lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. In *CAV*, 2025. doi:10.1007/978-3-031-98682-6_11.

26 Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In *IJCAR*, 2001. doi:10.1007/3-540-45744-5_19.

27 Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. Elsevier and MIT Press, 2001. doi:10.1016/B978-044450813-3/50009-6.

28 Yicheng Qian, Joshua Clune, Clark W. Barrett, and Jeremy Avigad. Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers. In *CAV*, 2025. doi:10.1007/978-3-031-98682-6_10.

29 Michael Rawson, Andrei Voronkov, Johannes Schoisswohl, and Anja Petkovic Komel. Ground Truth: Checking Vampire Proofs via Satisfiability Modulo Theories. In *CADE*, 2025. doi:10.1007/978-3-031-99984-0_8.

30 Giles Reger, Martin Suda, and Andrei Voronkov. New Techniques in Clausal Form Generation. In *GCAI*, 2016. doi:10.29007/dzfz.

31 Neha Rungta. A Billion SMT Queries a Day (Invited Paper). In *CAV*, 2022. doi:10.1007/978-3-031-13185-1_1.

32 Stephan Schulz. Empirical properties of term orderings for superposition. In Boris Konev, Claudia Schon, and Alexander Steen, editors, *PAAR*, CEUR Workshop Proceedings. CEUR-WS.org, 2022. URL: https://ceur-ws.org/Vol-3201/paper4.pdf.

33 Geoff Sutcliffe. Stepping Stones in the TPTP World. In *IJCAR*, 2024. doi:10.1007/978-3-031-63498-7_3.

34 Josef Urban and Geoff Sutcliffe. Automated Reasoning and Presentation Support for Formalizing Mathematics in Mizar. In *AISC*, 2010. doi:10.1007/978-3-642-14128-7_12.

35 Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *CAV*, 2014. doi:10.1007/978-3-319-08867-9_46.

36 Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The Concept of Demodulation in Theorem Proving. *J. ACM*, 14(4):698–709, October 1967. doi:10.1145/321420.321429.

37 Thomas Zhu, Joshua Clune, Jeremy Avigad, Albert Qiaochu Jiang, and Sean Welleck. Premise Selection for a Lean Hammer, 2025. doi:10.48550/arXiv.2506.07477.