Exploring the Capabilities and Limitations of Algorithm Verification in Vampire

Case Studies in Verifying the Correctness of Selection Sort and of a Key-Value Store

Mohammed Balfakeih[1]
Supervisors: Dr. Benedikt Ahrens[1]**, Kobe Wullaert**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
22nd June 2025

Name of the student: Mohammed Balfakeih
Final project course: CSE3000 Research Project
Thesis committee: Dr. Benedikt Ahrens, Kobe Wullaert, Dr. Maliheh Izadi

Abstract

Formal software verification is an important task for ensuring software correctness, especially in safety-critical systems. One method of formal verification is automated theorem proving, where one defines their program as a set of axioms and its correctness criteria as conjectures. We investigated the suitability of Vampire, an automated theorem prover, for formal verification by analyzing the logical foundations and performing case studies. In these case studies, we implemented Selection Sort and a Key-Value Store to observe what could be represented and proven. We were unable to prove the correctness of Selection Sort, but were able to prove most properties of a Key-Value Store. We discuss the capabilities and limitations of Vampire for formal software verification, namely that it is able to prove many properties of the functions we defined, unless they are heavily reliant on inductive definitions or too general with respect to the axioms. We believe that Vampire's inductive capabilities are currently insufficient for our use case. We reflect on the experience of using Vampire from the perspective of the average computer scientist and give suggestions for further documentation. We then discuss related work, including using Isabelle, another ATP and proof assistant, to reason about sorting algorithms and a Key-Value Store. Finally, we give suggestions for future work, such as rewriting the Key-Value Store in SMT-LIB, a language used as input for SMT solvers.

Keywords: Vampire, Automated Theorem Proving, Formal Verification, Selection Sort, Sorting Algorithms, Key-Value Store

1 Introduction

Software bugs can lead to disastrous consequences on public health and safety. A notable example of this is the Air France Flight 447 crash in June 2009. This accident was caused due to sensors inaccurately measuring the speed of the aircraft, leading it to stall and crash, killing all 228 people on board [1]. Since hardware and software systems have the ability to harm people if programmed incorrectly, ensuring their correctness is an important task. To address this need, many techniques have been invented for effective software testing [2]. However, none of them can guarantee the absence of bugs [3]. One class of techniques that can guarantee correctness is formal methods, which are mathematical techniques to specify and verify software [4]. Among these techniques is automated theorem proving, which involves modeling a software system as a set of logical statements, namely axioms and conjectures. Here, axioms represent the semantics of the system, and conjectures represent what it means for the system to be correct. Since Automated Theorem Provers (ATPs) do not require human assistance, they can be especially useful for formal verification [5]. Ideally, a developer should be able to write a formal specification of their code and the correctness criteria, and the ATP would tell them if the correctness criteria hold.

One of these ATPs is Vampire, which is one of the fastest ATP systems [6]. This is evident from its performance in the CADE ATP System Competition (CASC), a competition comparing the performance of different ATPs on different forms of logic [7]. In CASC, especially the most recent CASC-J12, Vampire won in many of the divisions, solving problems faster than any of the other competing ATPs [8]. Furthermore, there has been previous work on performing formal verification in Vampire, namely in proving the correctness of Insertion Sort, Merge Sort, and Quick Sort [9].

This research aims to explore the algorithm verification capabilities and limitations of Vampire, to investigate if its effectiveness as an ATP can be leveraged for formal verification. This was done by researching the logical foundations of Vampire, described in Section 2. We then performed case studies in proving the correctness of Selection Sort and of a Key-Value store to analyze its capabilities. Their formal problem descriptions are given in Section 3, and their implementations are described in Section 4, along with a discussion of the results. We then discuss the experience of using Vampire, TPTP, and SMT-LIB in Section 5, and highlight a few issues found. These issues include difficulty proving multiple conjectures and issues with the LaTeX output of proofs. We ensured throughout the project that our research was performed responsibly, which is explained in Section 6. Furthermore, we discuss how other ATPs and SMT Solvers, such as cvc5 and Isabelle, were used to verify sorting algorithms or Key-Value Stores in Section 7. Finally, we conclude in Section 8 that Vampire is a capable ATP for formal verification. However, we note that it struggles with inductive definitions of algorithms, as evidenced by its inability to prove the correctness of Selection Sort, or prove the conjectures of the Key-Value Store that used inductively defined functions. Furthermore, it has difficulty with proving more general statements, such as the conjecture that keys are unique in a Key-Value Store. We give possibilities for future work, including rewriting the Key-Value Store in SMT-LIB.

2 Background

2.1 Formal Verification and Automated Theorem Proving

Automated theorem proving can be an effective method for formal verification. It allows one to model a software system as a set of logical statements, then to verify if specified properties of the system hold. More generally, formal verification is the field of using formal methods to verify the correctness of software and hardware systems. Formal methods are mathematical techniques used to perform this verification and analysis [4]. One of these formal verification methods is theorem proving, which is the process of proving whether a logical statement, known as a conjecture, follows from a set of logical statements, known as axioms [5]. Theorem provers can either be interactive or automated, which refers to whether they may require human assistance to prove theorems [10]. These systems take in statements in different forms of logic, such as propositional logic or predicate first-order logic [11]. We explain how Vampire performs automated theorem proving in subsection 2.4.

2.2 An Introduction to Typed First-Order Logic

We will now briefly explain typed first-order logic, as we make heavy use of it throughout this paper. Typed first-order logic, sometimes referred as many-sorted first-order logic, is an extension of first-order logic which gives each variable a type. For example, if we have the type of integers represented as $\mathbb{Z}$, then an example of a tautology in typed first-order logic would be:

Problem 1

Axioms:

$$(\forall x : \mathbb{Z})(Even(x) \Leftrightarrow (x \bmod 2 \equiv 0))$$
$$(\forall x : \mathbb{Z})\ (Odd(x) \Leftrightarrow (x \bmod 2 \equiv 1))$$

Conjecture:

$$(\forall x : \mathbb{Z})(Even(x) \lor Odd(x))$$

In Problem 1, x is a typed variable, *Even* and *Odd* are predicates, and their definitions are assumed when proving the conjecture. Typed first-order logic can be reduced to untyped first-order logic [12]. This is often done by ATPs, including Vampire [13].

2.3 What is Vampire?

Vampire is an ATP which supports reasoning about classical typed first-order logic. Vampire takes in a set of axioms and a conjecture, and attempts to prove that the conjecture follows from the axioms.

It supports theories, which are sets of logical formulae that define how a type works, such as integer arithmetic [13, 14]. Moreover, Vampire can efficiently reason about programming constructs such as `if-then-else` statements, `let` expressions, and arrays, allowing for more direct reasoning regarding program correctness [15, 16]. Furthermore, it is one of the fastest ATPs, having won many divisions in the CADE ATP System Competition (CASC) over the years [7], especially in the most recent CASC-J12 [8].[1] Certain versions of Vampire support higher-order logic, that is, logic that allows quantification over predicates such as the statement $\forall x \; \exists P \; P(x)$ [17]. The version of Vampire used in this paper does not support this. Vampire can take in problems in two languages, TPTP and SMT-LIB 2, which are designed for ATPs and SMT solvers, respectively [13, 18, 19]. All of these features together make Vampire a capable and advanced ATP.

2.4 The Theory of Vampire

To prove theorems, Vampire uses an inference system. An inference system is a set of inference rules, which tells Vampire that given a certain set of logical statements, it can derive a new logical statement. For example, Reger et al. [20] define the following inference system, which consists of two inference rules in propositional logic:

$$\frac{p \vee C_1 \quad \neg p \vee C_2}{C_1 \vee C_2}\ \text{(BR)} \qquad\qquad \frac{L \vee L \vee C}{L \vee C}\ \text{(F)}$$

where p, C, C_1, C_2, and L are statements in propositional logic (specific letters denote specific types of statements, described by Kovács and Voronkov [13], but this is not relevant for this example). They call the rule on the left "binary resolution", and the one on the right "factoring" [20]. The statements above each rule are the premises, and the one on the bottom is the conclusion. Therefore, if the premises exist in the current set of formulae (e.g. in the axioms), then Vampire can derive the conclusion [13]. If a set of clauses is unsatisfiable, for example the formula:

$$(p \vee q) \wedge (p \vee \neg q) \wedge \neg p,$$

then we can write the following derivation:

$$\cfrac{\cfrac{\cfrac{p \vee q \quad p \vee \neg q}{p \vee p}\ \text{(BR)}}{p}\ \text{(F)} \qquad \neg p}{\square}\ \text{(BR)}$$

where $\square$ denotes the empty clause. The empty clause is false. This shows that the set of clauses is unsatisfiable, and that C, C_1, or C_2 may be the empty clause in the inference rules. This example is simplified from Reger et al. [20].

Vampire's inference system is a sound refutationally complete superposition inference system. Soundness is defined as the implication: If the empty clause is derivable in the inference system, then the set of formulae is unsatisfiable. Refutationally complete is the converse of this: If the set of formulae is unsatisfiable, then the inference system will derive the empty clause. Superposition refers to Vampire working on formulae defined in clausal normal form. Formulas are conjunctions of clauses, while clauses are a disjunction of literals (a single variable which may be negated), i.e. $L_1 \vee L_2 \vee ... \vee L_n$. Therefore, to prove that a conjecture holds given a set of axioms, Vampire first negates the conjecture, then tries to derive unsatisfiability [13]. This is called a proof by refutation, which makes use of double negation elimination from classical logic $\neg\neg p = p$ [21]. A theoretical consequence of this inference

[1] See `https://tptp.org/CASC/`

system is that if a conjecture follows from the axioms, then Vampire will prove it given enough time and resources. Furthermore, this implies it is impossible to tell before termination if a conjecture does not hold, or if the proof search is diverging.

Vampire uses the inference system in its saturation algorithm. A saturation takes a set of clauses, and applies an inference rule to derive a new clause, which is added to the set of clauses. This repeats until Vampire derives the empty clause, or inference rules can no longer be applied in a helpful manner. There are three possible outcomes for a saturation algorithm:

- It generates the empty clause, meaning the problem is unsatisfiable.
- It terminates and does not generate the empty clause, meaning the problem is satisfiable.
- It times out or runs out of resources, in which case the satisfiability of the problem is unknown.

Vampire supports multiple saturation algorithms, each of which apply different techniques to make Vampire as fast as possible for different benchmarks. These are used in Vampire's different strategies, which are sets of parameters for proof search. Vampire also has schedules, which are lists of different strategies that can be run either sequentially or in parallel [13]. A more complete guide on Vampire can be found in [22], and we provide a short guide on how to use Vampire in Appendix A.

2.5 What is TPTP?

The Thousands of Problems for Theorem Provers (TPTP) World is an infrastructure for ATP research, including a language to represent theorem proving problems. The TPTP World comes with the TPTP Problem Library, a wide selection of problems in many forms of logic, spanning different domains, used to benchmark ATPs [23]. The TPTP language supports different forms of logic, including typed first-order logic with arithmetic [24]. It closely follows the structure of formal logic problems. Note that the term "TPTP" may be used to refer to only the language, including in this paper.

We now describe the TPTP type system, as it is necessary for understanding this paper. A further description of TPTP can be found in Appendix B and on the TPTP website.[2]

The TPTP type system has five predefined types, shown below. Note that predefined types and functions are prefixed with `$` in TPTP.

- `$i`: individuals, a type without its own axioms
- `$o`: booleans
- `$int`: integers
- `$rat`: rationals
- `$real`: reals

Types are non-empty and pairwise disjoint domains (a type cannot be a subtype of another) [25]. One can also define their own types in TPTP if needed, which we make use of in this paper.

2.6 Using Vampire for Formal Verification

We explored Vampire's capabilities in automated formal verification. In particular, we analyzed its capabilities and limitations in proving the correctness of Selection Sort and of a Key-Value Store, by representing these in SMT-LIB and TPTP, respectively. Then, we had Vampire try to prove their properties of correctness. These programs were chosen as they are both of critical importance in a wide variety of applications [26, 27].

Given the variety of theories in Vampire, its speed, and its existing applications, we expected it would be fairly capable in proving these properties. We noted that there has already been work in

[2]See `https://tptp.org/UserDocs/TPTPLanguage/TPTPLanguage.shtml` for further explanation of TPTP.

using Vampire to prove the correctness of sorting algorithms. Namely, Georgiou et al. use Vampire to prove the correctness of Insertion Sort, Merge Sort, and Quick Sort. Furthermore, they provide a general framework for proving the correctness of sorting algorithms [9]. We used this framework to attempt to prove Selection Sort. To the best of our knowledge, there has been no work in verifying a Key-Value Store in Vampire. We now describe these problems formally.

3 Formal Problem Descriptions

We formally defined Selection Sort and the Key-Value Store by defining the axioms we would assume, and the properties we wanted to prove. Selection Sort is formally defined in Section 3.1, while the Key-Value store is defined Section 3.2.

3.1 Selection Sort

We formally define a sorting algorithm as an algorithm that takes in a list of elements with a linear ordering $\leq$. A linear ordering is one which is reflexive, antisymmetric, transitive, and total. The algorithm must place the elements in the ordering (note that we ignore the property of stability). Furthermore, the resulting list must be a permutation of the original list, with no new elements being added or any being removed [28].

We formally define the semantics of Selection Sort through functional pseudocode. Furthermore, we define the conjectures that define correctness below.

3.1.1 Pseudocode

We use the list constructors and functions: nil, $cons(x, xs)$, $head(xs)$, $tail(xs)$, $min(xs)$, $remove(x, xs)$, where x is an integer and xs is a list. $remove(x, xs)$ only removes the first occurrence of x from xs, if it exists. Furthermore, we use the **match** and **case** keywords to denote the use of pattern matching.

Algorithm 1 Functional Selection Sort

```
function SELECT($xs$)
    match $xs$
      case $nil$
        $nil$
      case $cons(y, ys)$
        let $z \leftarrow min(cons(y, ys))$
        in $cons(z, remove(z, cons(y, ys)))$
end function

function SELECTION-SORT($xs$)
    match $xs$
      case $nil$
        $nil$
      case $cons(y, ys)$
        let $l \leftarrow$ SELECT($cons(x, xs)$)
        in $cons(head(l)$, SELECTION-SORT($tail(l)$))
end function
```

3.1.2 Conjectures

Permutation Equivalence: The multiset of elements is the same before and after sorting. This is defined as by Georgiou et al. [9]:

$$(\forall x \in \mathbb{Z})(\forall xs \in L)(filter_{eq}(x, xs) = filter_{eq}(x, \textsc{Selection-Sort}(xs))),$$

where L represents the set of integer lists, and $filter_{eq}(x, xs)$ filters the list xs and returns all elements that are equal to x.

Sortedness: The output list is sorted. This is defined as by Georgiou et al. [9]:

$$sorted(nil) = true$$

$$(\forall x \in \mathbb{Z})(\forall xs \in L)(sorted(cons(x, xs)) = (elem_{\leq}list(x, xs) \wedge sorted(xs))),$$

where $elem_{\leq}list(x, xs)$ is true iff x is less than or equal to all the elements in xs.

3.2 Key-Value Store

We formally define a Key-Value Store as a data structure that stores key-value pairs, where keys are unique. We require that the following functions exist:

- A function to retrieve a value given a key (typically called get, select, read, etc.).
- A function that adds a key-value pair, possibly replacing another key-value pair if the key already exists in the Key-Value Store (typically called put, store, write, etc.).
- A function that removes a key-value pair given a key, if it exists (typically called remove, delete, etc.).

For our Key-Value Store, we define five main functions. Note that we use KVS and $\mathcal{I}$ to refer to the set of all Key-Value Stores and the type of individuals, respectively:

$$get(a, k) \rightarrow \mathcal{I}, \quad remove(a, k) \rightarrow KVS, \quad put(a, k, v) \rightarrow KVS,$$

$$contains(a, v) \rightarrow \{false, true\}, \quad \text{and} \quad size(a) \rightarrow \mathbb{Z},$$

where a is a Key-Value Store, k is a key and v is a value. We define keys as nonnegative integers, and values as individuals. Furthermore, we define a simple hash function $h(k) = k \bmod n$, where n is the length of the Key-Value Store. We use separate chaining for collision handling, meaning that each position in the Key-Value Store contains a linked list. Note that the terms list and chain are used interchangeably in the context of the Key-Value Store in this paper.

3.2.1 Axioms and Conjectures

We list the axioms defined and what we believe are the most important conjectures to prove. The other conjectures are defined in Appendix C. Moreover, we use either natural language or typed first-order logic to describe logical statements depending on which we believe to be clearer. Note that the last three axioms are based on the axioms for arrays defined in the work of Kotelnikov et al. [15]:

Axioms

Definitions of *get*, *put* and *remove* functions: These functions are defined as retrieving the separate chain associated with the key, then calling the respective linked list function (*get*, *append*, or *remove*, note these are linked list functions, not the Key-Value Store functions) on that list. For *put* and *remove*, the chain is placed back into the Key-Value Store, and the new Key-Value Store is returned. Note that if the key passed into *get* is not present in the Key-Value Store, it returns *null*.

Definition of the *contains* function: This function returns true iff there exists a key that can be used to retrieve that value.

$$(\forall a \in KVS)(\forall v \in \mathcal{I})(contains(a, v) \Leftrightarrow ((\exists k \in \mathbb{Z}^{\geq 0})\ get(a, k) = v))$$

Definition of the *size* function: $size(a)$ is defined recursively, i.e. at position 0, the size is the size of the linked list at position 0. At position 1, it is the size of the linked list at position 1 plus the size at 0, etc.

Read over write one: If one gets the value at a key where they put a value, they should retrieve the same value.

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(\forall v \in \mathcal{I})(get(put(a, k, v), k) = v)$$

Read over write two: If one gets the value at a key different from the one they placed a value at, it is the same as if they got the key at the original Key-Value Store.

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(\forall l \in \mathbb{Z}^{\geq 0})(\forall v \in \mathcal{I})((k \neq l) \Rightarrow (get(put(a, k, v), l) = get(a, l)))$$

Extensionality: Key-Value Stores are defined as being equal if they contain the same key-value pairs.

$$(\forall a \in KVS)(\forall b \in KVS)(((\forall k \in \mathbb{Z}^{\geq 0})\ get(a, k) = get(b, k)) \Rightarrow a = b)$$

Conjectures

1. **Putting different values:** Putting two different values at the same key results in two different Key-Value Stores.

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(\forall v \in \mathcal{I})(\forall w \in \mathcal{I})(v \neq w \Rightarrow put(a, k, v) \neq put(a, k, w))$$

2. **Put over put:** Putting two values at the same key is equivalent to only putting the last value.

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(\forall v \in \mathcal{I})(\forall w \in \mathcal{I})(put(put(a, k, v), k, w) = put(a, k, v))$$

3. **Put is contained:** If one puts a value, then that value is in the Key-Value Store.

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(\forall v \in \mathcal{I})(contains(put(a, k, v), v))$$

4. **Key uniqueness:** Keys must be unique.
5. **Equality implication:** Two key-value stores being equal implies they contain the same values.
6. **Put changes size:** Putting a key-value pair increases size by 1 or does not change it.
7. **Remove changes size:** Removing a key-value pair decreases size by 1 or does not change it.
8. **Put check:** If you put a value v in a at key k and it results in a Key-Value Store b, then for all keys x that are not k, $get(a, x) = get(b, x)$. Otherwise, $get(b, x) = v$.
9. **Remove check:** If you remove the value at key k from a and it results in a Key-Value Store b, then for all keys x that are not k, $get(a, x) = get(b, x)$ otherwise, $get(b, x) = null$.

4 Implementation and Discussion of Results

We describe our implementation of Selection Sort and the Key-Value Store, and discuss our results. To prove the conjectures, we mainly used Vampire's CASC mode, which uses a number of different proof searching strategies [6]. Note that this mode is specifically designed for the rules of the CASC, though the competition contains a variety of problems. If the CASC mode was unable to prove something within the time limit, we also attempted it with the parameters `--mode portfolio -sched induction` or `--mode portfolio -sched struct_induction`, which uses proof strategies targeted for induction and structural induction, respectively [29]. This was used since some functions are defined inductively.

We used Vampire 4.9 (commit `5ad494e78`)[3] linked with Z3 4.12.3.0 on a machine with an AMD Ryzen 7 4800H with Radeon Graphics @ 2.900GHz CPU, with 16 virtual cores, and 16GB of RAM. Furthermore, we used the maximum number of cores available on our machine by setting `--cores 0`.

We set a wall clock time limit of 60 seconds for Vampire to prove a conjecture. If it timed out, we assumed Vampire could not prove the conjecture in a reasonable amount of time. The choice of 60 seconds was made based on the results of the CASC-J12 [8], where Vampire had an average CPU time of under 30 seconds (CASC-J12 used two octa-core Intel(R) Xeon(R) E5-2667 v4 CPUs, running at 2.10 GHz [8]) in the category "TFA (typed-first order logic with arithmetic) using Integers".[4] Therefore, we decided that 60 seconds when using multiple cores was a sufficient wall clock time limit for our use case.

We give function signatures in a TPTP-like syntax, namely: `function_name(Param_1: type, Param_2: type, ..., Param_n: type)` $\rightarrow$ `return_type`. When giving snippets of implementations, variables are implicitly universally quantified unless otherwise stated. Moreover, functions are total (every element of their domain is mapped to an element in their codomain) unless otherwise stated.

We now explain the axioms defined, and the most important conjectures that could or could not be proven. The full code can be found on the GitHub repository.[5]

4.1 Common List Functions

Firstly, we give a general definition of lists, as they are used in Selection Sort and in the Key-Value Store (for separate chaining). These are based on an implementation in the TPTP problem library.[6] Functions and variables common to both implementations are defined here, while functions and variables that are only used for one problem are in their respective subsections. We use `placeholder` as a placeholder type for the elements of the list.

- `nil` is of type `list` and represents the empty `list`.
- `cons(P: placeholder, L: list)` $\rightarrow$ `list`, constructs a non-empty `list`.
- `head(L: list)` $\rightarrow$ `placeholder`, retrieves the first element of `L`. This is a partial function and is not defined for `nil`.
- `tail(L: list)` $\rightarrow$ `list`, returns `L` excluding the first element. This is a partial function and is not defined for `nil`.

[3]https://github.com/vprover/vampire/releases/tag/v4.9casc2024
[4]See https://tptp.org/CASC/J12/WWWFiles/ResultsSummary.html
[5]https://github.com/mbalfakeih/Vampire-Case-Study
[6]https://tptp.org/cgi-bin/SeeTPTP?Category=Problems&Domain=DAT&File=DAT081_1.p

4.2 Selection Sort

To implement Selection Sort, we implemented the functional pseudocode from Algorithm 1 in SMT-LIB 2.7, due to it having support for the theory of data types, which TPTP does not [30]. The theory of data types provides helpful axioms for our proof search, including those related to induction [31]. Furthermore, we decided to sort integers, as certain Vampire strategies do not work with polymorphic types.[7] With these decisions in mind, we defined the following functions:

- `remove(I : $int, L : list)` $\rightarrow$ `list`, removes the first occurrence of `I` from `L`, if it exists.
- `min(L : list)` $\rightarrow$ `$int`, retrieves the minimum of `L`. For a `list` with a single element, it returns that element, otherwise it returns whichever is smaller, the first element or the minimum of the tail of `L`. This is a partial function and is not defined for `nil`.
- `select(L : list)` $\rightarrow$ `list`, returns `nil` if the list is `nil`, otherwise it uses a `let` expression to assign the minimum of the list to `z`, then returns `cons(z, remove(z,L))`.
- `selection_sort(L : list)` $\rightarrow$ `list`, returns `nil` if the list is `nil`, otherwise it uses a `let` expression to assign `select(L)` to `l`, then returns `cons(head(l),selection_sort(tail(l)))`.

Furthermore, we defined the functions `sorted(L : list)` $\rightarrow$ `$o` and `filter_eq(I : $int, L : list)` $\rightarrow$ `list` to prove sortedness and permutation equivalence respectively.

Unfortunately, we were unable to prove either of the two properties of Selection Sort. The only sub-lemma we were able to prove is that for a non-empty list, the minimum of a list is contained in the list. Moreover, we were not able to reproduce all of the results in the work of Georgiou et al. [9] from their GitHub repository using Vampire 4.9.[8] A further investigation of this can be found in Appendix D. We believe this to be due to changes in Vampire that would have occurred between the version Georgiou et al. [9] were using, and the release of Vampire 4.9.

We believe the reason why Vampire could not prove the correctness of Selection Sort is due to its inductive capabilities. Induction is a notoriously difficult task in automated theorem proving. Einarsdóttir et al. explain that this is a challenge for a few reasons. Firstly, there can be many possible inductive schemes (also known as inductive hypotheses) to choose from, which makes the search space much larger. Secondly, conjectures may need to be generalized in order to be proven, hence requiring the ATP to use a more general conjecture in its saturation, and it may not be immediately clear that this is necessary. Finally, inductive proofs may require the proofs of additional lemmas, which may also need to be proven via induction [32].

Because of this, there has been work in the past few years in improving Vampire's inductive capabilities [29, 32–34]. In particular, we note the work of Einarsdóttir et al. [32], where they use a tool called QuickSpec to assist Vampire in induction. QuickSpec generates lemmas that may assist Vampire in proving its conjecture. They tested their method on the Tons of Inductive Problems (TIP) benchmark [35]. This is especially interesting for us, as a subset of this benchmark named TIP2015 has 126 out of 326 conjectures related to sorting algorithms. Vampire, without any additional help, was reported to only prove 39 of the 326 conjectures, while their best method solved 134 conjectures. From this, we gather that Vampire, especially without the use of other tools, is not yet capable of effectively proving conjectures relating to larger inductive algorithms, such as sorting algorithms.

[7]See the comment from Márton Hajdu here: `https://github.com/mina1604/sorting_wo_sorts/issues/1`
[8]`https://github.com/mina1604/sorting_wo_sorts`

4.3 Key-Value Store

To implement the Key-Value Store, we defined a set of axioms for key-value pairs, the elements stored in the Key-Value Store. Furthermore, we defined additional functions for lists, and defined a set of axioms for the Key-Value Store itself. We describe how we defined these axioms in TPTP, then discuss which conjectures could be proven from this.

4.3.1 Key-Value Pairs

We defined `pair` as a type containing an integer and an individual. With the following functions and variables:

- `null` is of type `pair` and represents an empty `pair`
- `pair_cons(I: $int, J: $i)` $\rightarrow$ `pair`, to construct a `pair`.
- `get_left(P: pair)` $\rightarrow$ `$int`, to get the integer. This is not defined for `null`.
- `get_right(P: pair)` $\rightarrow$ `$i`, to get the individual. This is not defined for `null`.

The left integer is used for the key, while the right individual is used for the value.

4.3.2 Lists

The type of `list` elements is `pair`. Note that we suffix some function names with `_l` to avoid confusion with other function names. The functions defined are:

- `length(L : list)` $\rightarrow$ `$int` returns the length of `L`. This is defined inductively.
- `remove_l(L: list, I : $int)` $\rightarrow$ `list` removes the `pair` with a key equal to `I`, if it exists, and returns the remaining `list`.
- `get_l(L: list, I: $int)` $\rightarrow$ `pair` retrieves the key-value `pair` with a key equal to `I`, if it exists, otherwise, it returns `null`.

Unfortunately, Vampire was unable to prove that the result of `get_l(L, I)` would contain the same key as `I`. We believe this to be due the fact that TPTP does not have a theory for data types, and hence it does not have the necessary axioms to prove this. Because of this, we assumed this property as a lemma (note that in TPTP, lemmas are effectively the same as axioms, this was done to make this property distinct to whoever reads the code).

4.3.3 Key-Value Store

We define the Key-Value Store as `$array($int, list)`. Arrays are defined as mappings from indices to elements, in this case from `$int` to `list`. This implies that the array is infinite, however, we only consider the range $[0, \mathtt{n})$, where `n` is the length of the Key-Value Store, and impose this restriction when necessary onto the axioms. Note that we assume `$greater(n, 0)`. We use the built-in functions `$select` and `$store`, which retrieve an element from the array and place an element in the array respectively. They are defined axiomatically in the work of Kotelnikov et al. [15]. From this point onward, the definitions of functions are fairly direct translations of the Formal Problem Description in Section 3.2, hence we leave a description of the implementation in Appendix E.

4.3.4 Proven Conjectures

A summary of the results can be found in Table 1. From the conjectures listed in Section 3.2.1, we were able to prove conjectures 1, 2, 3, 5, and 8. Additional proven conjectures can be found in the GitHub repository. This shows that Vampire is able to reason from our definitions about how `put` interacts with the Key-Value Store, as many of these properties rely on this definition. Furthermore, it is able to reason about the notion of equality defined by the extensionality axiom, as it is able to prove that equal Key-Value Stores contain the same values. Finally, we note that all of these properties were able to be proven in around a second or less on our machine.

4.3.5 Unproven Conjectures

We were unable to prove conjectures 4, 6, 7, and 9. Additional unproven conjectures can be found in the GitHub repository.

For conjectures 6 and 7, our issue was that Vampire was unable to prove any properties that involved the `size` function. To confirm, we defined a lemma on a concrete instance of the Key-Value Store with `n = 10`, and with all chains defined to be `nil`. We conjectured that `size(put(kvs, 0, a)) = 1`, where `kvs` is the concrete instance of the Key-Value Store and `a` is a value with type `$i`. Vampire was unable to prove this before timing out, regardless of the schedule. We believe this to be due to `size` relying on two inductive definitions, one across lists, `length_l`, and one across the Key-Value Store, `size` itself. Because of this, we omitted writing out properties that involved `size`, as we assumed Vampire was unable to prove them.

Table 1: Summary of Results on the Key-Value Store

Conjecture		Proven?
Putting different values	(1)	✓
Put over put	(2)	✓
Put is contained	(3)	✓
Key uniqueness	(4)	✗
Equality Implication	(5)	✓
Put changes size	(6)	✗
Remove changes size	(7)	✗
Put check	(8)	✓
Remove check	(9)	✗

For conjecture 4, we claimed that if there were two different pairs that existed in the Key-Value Store, then they must have different keys. We believe Vampire could not prove this since the fact that keys are unique is only evident from our definition of `put`, namely that we `remove` the key-value pair with the same key as the one we are putting, before adding the new key-value pair. The conjecture does not use `put`, as the pairs already exist in the Key-Value Store. We believe this led to Vampire being unable to prove this conjecture. We also attempted to define key uniqueness in terms of `put`, stating that if two key-value pairs with the same key were placed one after another, then the size of the Key-Value Store should not change. However, this also could not be proven, likely due to the aforementioned issues with the `size` function. We were only able to prove the necessary condition that every key in the Key-Value Store must have exactly one value associated with it, but this is not sufficient since it does not ensure that there is at most one of these key-value pairs in the Key-Value Store. Similarly, conjecture 9 only holds if keys are unique, and hence could not be proven.

4.4 Summary of Results

To summarize, we were unable to prove the correctness properties of Selection Sort, which we believe to be due to the inductive capabilities of Vampire. Furthermore, we were able to prove most correctness properties of the Key-Value Store, with the exception of those related to `size` or key uniqueness. We believe the former to be due to struggles with inductive definitions, while the latter is due to uniqueness not being directly encoded into the axioms. Generally, we believe this shows that Vampire can only prove conjectures that explicitly rely on function output, and not more general conjectures such as key uniqueness, which hold across data structures.

5 Discussion on using Vampire

We now discuss the experience of using Vampire, TPTP, and SMT-LIB generally, and list a few issues found with using Vampire and these languages. We understand that not all of these can be solved, possibly due to theoretical reasons. However, we feel they are important to note for users of Vampire.

Firstly, we note that SMT-LIB was not initially designed to be written in, rather to be generated by other programs. This is evident from the fact that you cannot include an SMT-LIB file in another. Therefore, to prove multiple conjectures, one must copy their axioms into every file. This makes maintainability of problems difficult, as changing an axiom requires changing them in every file.

This is slightly better in TPTP, as files can be included in one another. However, since one cannot have multiple conjectures in a file, one must run Vampire individually on each conjecture. We note that TIP does support multiple proof goals, showing that this is viable in these standards.[9] Alternatively, Vampire could have this feature integrated, for example by specifying multiple files or a folder containing conjectures to prove.

Moreover, TPTP uses the SZS Ontology, which defines the different status values that can be outputted by an ATP [36]. One of these is ContradictoryAxioms (CAX), which is supposed to be returned when the user defines axioms which contradict one another. However, we experienced a few situations in which Vampire would be able to show that a conjecture followed from the axioms, but returned CAX when trying to prove a different conjecture followed from those same set of axioms. We hypothesize that Vampire simply returns whatever it finds first. It is likely impossible to resolve this issue, as it is impossible for a set of axioms containing arithmetic to prove its own consistency [37]. Furthermore, it is possible for one to define a set of axioms that causes the principle of explosion [38], and Vampire may not output CAX, which occured in our experience.[10]

Vampire provides the user with the ability to output their proofs in LaTeX. This is useful for debugging as one can read the proof in formal logic rather than TPTP. However, there are a few issues with the output. Firstly, the output is created after Vampire preprocesses the problem into clausal normal form [13]. Therefore, the LaTeX output does not contain the types of variables, making the reasoning more difficult to follow. These could be derived from the original typed problem and added to the LaTeX output. Furthermore, there remain bugs in the LaTeX output, such as it rendering $<$ as $\nless$ and vice versa. It may also possibly crash, despite Vampire being able to prove a conjecture when it is not outputting LaTeX. We have reported these on the Vampire GitHub repository.[11]

Finally, we note that Vampire's proof capabilities do not strictly increase between versions, i.e. a conjecture that can was proven in a previous version of Vampire is not guaranteed to be proven in a later version. This is evident from our investigation in Appendix D. This problem would be an incredibly difficult, and perhaps impossible, guarantee to make while still generally improving Vampire's proof capabilities. However, it is something users should keep in mind if they use different versions of Vampire.

Since it is expected that users of Vampire are those within the ATP community, it is understandable that these limitations are present in the tool. With further documentation and guides on some of the issues mentioned here, we believe Vampire could become a more accessible tool for the average computer scientist.

[9] See `https://tip-org.github.io/format.html`

[10] See `https://github.com/vprover/vampire/issues/722`

[11] See `https://github.com/vprover/vampire/issues/721` and the above footnote.

6 Responsible Research

We note that our research would have a positive ethical impact on software development and formal verification, since proper and widespread use of automated software verification tools would result in a decrease in buggy code, which is especially relevant in safety-critical systems. Our research demonstrates the capabilities and limitations of Vampire, which may be helpful for further development and use of Vampire in formal verification.

Furthermore, we aimed to ensure that these results are reproducible by documenting our decisions thoroughly. This was of great importance to us, as we noticed throughout the research process that many tools or results using Vampire were unmaintained and that we could not run them (Rapid [39] and SATVIS [40] are examples of this).

To ensure our work was reproducible, we created a public GitHub repository with our results,[12] with frequent commits (at least once per day that we worked on it). This allows anyone interested to follow the development of this research, and possibly take it in a different direction. Furthermore, this GitHub repository also contains Bash scripts with the commands to run Vampire that include the specific parameters used. This allows anyone with GNU Bash [41] and Vampire 4.9 to be able to reproduce our results on their machine.

Moreover, our `README` details how the repository is organized, requirements for the Bash scripts, versions of tools, and the machine specifications that the repository was ran on. We also kept properties that Vampire was unable to prove in our repository, such that if we made a mistake, or a new version of Vampire releases that is able to prove these properties, then the repository can be updated.

Finally, we also kept a logbook of our research throughout the project, an edited and abridged version can be made available upon request.

No generative AI was used during this project.

7 Related Work

Different ATPs and SMT solvers have been used for formal software verification to varying degrees. For example, cvc5 (and its predecessor CVC4) is an SMT solver [42] that has competed in CASC [7]. It has native support for induction, which has been shown to be effective against numerous benchmarks [43]. Furthermore, it has been used to reason about SQL queries [44]. This shows that SMT solvers such as cvc5 also have promising applicability in software verification.

There exists a decent amount of literature in reasoning about sorting algorithms using ATPs. One example of this is Isabelle, an ATP/proof assistant that works with higher-order logic, and makes use of other ATPs in its backend [45]. Isabelle has been used to reason about the number of comparisons done in Randomized Quick Sort, and has reasoned about Randomized Quick Sort's performance in comparison to other variations of Quick Sort [46]. Furthermore, Isabelle has been used for the verification of different implementations of Insertion Sort. Specifically, an implementation which directly swaps the values, and another which uses an auxiliary data structure called an index list. The index list maps each element to an index, then sorts the indices [47]. From this, it is clear that Isabelle and higher-order logic are capable of verifying the correctness of more complex sorting algorithms.

There appears to be less work on the verification of a Key-Value Store than sorting algorithms. An instance of a Key-Value Store being verified is the use of Isabelle to verify a keystore used in the TRENTOS operating system. This keystore is written in C and is used to store cryptographic keys. Therefore, they have additional properties to check such as memory safety. However, the authors claim that their work can easily be generalized to a generic Key-Value Store [48].

[12] `https://github.com/mbalfakeih/Vampire-Case-Study`

8 Conclusion and Future Work

We aimed to research the capabilities and limitations of algorithm verification in Vampire. We investigated the logical foundations of Vampire and performed case studies in proving the correctness of Selection Sort and of a Key-Value Store. We did this by defining the algorithms and correctness criteria in typed first-order logic, and implemented them in SMT-LIB and TPTP to prove them with Vampire. We reflected on these results and the user experience of Vampire.

From our case studies, we were unable to prove the correctness properties of Selection Sort. We believe this to be due to changes in Vampire since the work of Georgiou et al. [9] and now, since we were unable to completely reproduce their results. Furthermore, we discussed Vampire's inductive capabilities, and how they are insufficient for our purposes.

Moreover, we showed that Vampire was able to prove most correctness properties of a Key-Value Store, including properties relating to the behavior of the `put` function and the notion of equality. However, even with the use of inductive strategies, we were unable to prove properties that heavily relied on inductive definitions, such as conjectures related to the size of the Key-Value Store. Furthermore, we were unable to prove properties that were not directly encoded into the axioms, such as key uniqueness. We believe that Vampire is not able to prove more general conjectures across data structures, rather only conjectures directly related to function output.

We reflected on the user experience of Vampire and its input languages, and noted a few issues with them, including that there is no straightforward way of using Vampire to prove multiple conjectures, and bugs with the LaTeX output that Vampire provides.

In conclusion, we believe Vampire is a capable ATP for formal verification, as it supports many theories from TPTP and SMT-LIB and is incredibly fast. This was evident in the number of properties we could prove. However, despite the recent advancements in inductive reasoning, we believe there is still more work to be done to be able to perform reasoning on inductive algorithms. Moreover, further documentation on Vampire would be useful in making it more accessible for those outside of the ATP community.

Future work would involve rewriting the Key-Value Store in SMT-LIB rather than TPTP. Rewriting would allow for the linked lists used for separate chaining to be data types, which may allow for additional conjectures to be proven. Moreover, a $containsKey(a, k)$ function could be added, which would allow one to define and prove more conjectures. Furthermore, future work would involve improvement to Vampire's structural induction capabilities, such that the results of the work of Georgiou et al. [9] could be completely reproduced and other inductive algorithms could be proven. From this, one could prove Selection Sort, and generalize it to work on elements with a linear ordering, rather than only integers.

Acknowledgments

I am incredibly grateful to have had the chance to work on this project, and working with all the people involved greatly contribute to this. I would like to thank my supervisors Kobe Wullaert and Benedikt Ahrens, your guidance and feedback throughout this project was incredibly helpful, and I am grateful to have had the opportunity to work with you.

Furthermore, I would like to thank my peer group, Kaj Neumann, Tejas Kochar, Dinu Blanovschi, and Jeroen Koelewijn for discussing my project with me and providing assistance and advice when possible. Our weekly meetings, were always enjoyable and it was interesting hearing about how our different tools worked.

I would also like to thank Martin Damyanov and Maarten Sijm for proofreading my paper and discussing my project with me. Your feedback, as pedantic and nitpicky as it may have been, was very helpful.

Moreover, I would like to thank Laura Kovács, Márton Hajdu, and Martin Suda for providing your expertise on Vampire and answering my questions.

Finally, I would like to thank Betti for helping me make the poster associated with this paper, and all my friends who listened to me talk for way too long about this project.

Appendix A How to use Vampire

To run Vampire on a TPTP problem, one must install a Vampire executable, then use it in the command line as `vampire problem_name.p`. The file extension of the problem is not relevant. Vampire provides many different options for running it, such as performing different pre-processing steps, using different proof search strategies, or setting time limits. If Vampire proves that the conjecture follows from the axioms, it will provide the user with an output similar to:

```
% Refutation found. Thanks to Tanya!
% SZS status Theorem for problem_name
% SZS output start Proof for problem_name
[Beginning of proof removed for the sake of readability]
tff(f27,plain,(
  ( ! [X0 : $int,X1 : $int] : (~$less(X0,X1) | ~$less(X1,$sum(X0,1))) )),
  introduced(theory_axiom_169,[])).
% SZS output end Proof for problem_name
% ------------------------------
% Version: Vampire 4.9 (commit 5ad494e78 on 2024-06-14 14:05:27 +0100)
% Linked with Z3 4.12.3.0 79bbbf76d[truncated] z3-4.8.4-7980-g79bbbf76d
% Termination reason: Refutation

% Memory used [KB]: 907
% Time elapsed: 0.048 s
% Instructions burned: 147 (million)
% ------------------------------
% ------------------------------
```

This provides the user with runtime statistics, along with the proof itself. Note that comments in `[]` are added by us.

Appendix B Description of the TPTP Language

TPTP statements are called annotated formulae. They begin with a string denoting the form of logic being used, e.g. `tff` (Typed First-order Logic). Annotated formulae have a name, role, and formula. Roles are described in Table 2. Note that there can only be one conjecture per TPTP file. Therefore, to prove multiple conjectures, one can define a file in TPTP containing the axioms, and `include` it in separate files containing conjectures.

Table 2: Description of the basic roles for TPTP annotated formulae

Role	Description
`type`	Used to define a new type, this includes functions.
axiom-like formluae such as `axiom`	Logical statements assumed to be true
`conjecture`	Logical statement that should follow from the axioms.

Furthermore, formulas may use quantifiers and connectives, TPTP defines the following:

- `!`: Universal Quantifier $\forall$
- `?`: Existential Quantifier $\exists$
- `~`: Negation $\neg$
- `|`: Disjunction $\vee$
- `&`: Conjunction $\wedge$
- `=>`,`<=`: Implication $\Rightarrow, \Leftarrow$
- `<=>`: Bi-Implication $\Leftrightarrow$
- `<~>`: Exclusive Or $\oplus$

Quantifying variables in a formula is done as: `quantifier [Var_1 : type, Var_2 : type, ..., Var_n : type] : (formula)`. Variable names must be capitalized.

Function types are defined as follows: `(input_1 * input_2 * ... * input_n) > output`. Note that predicates are equivalent to functions that return booleans [16]. Moreover, predicates can be defined as a bi-implication statement, functions in general can be defined using `=`.

Finally, TPTP defines an arithmetic system with the standard arithmetic functions and comparisons[13]. We now give a translation of Problem 1 to TPTP, where the arithmetic function `$remainder_e` is used (the `e` refers to division being defined by the euclidean quotient).

```
tff(even, type, even: $int > $o ).
tff(odd, type, odd: $int > $o ).
tff(even_def,axiom,
    ! [X: $int] : ( even(X) <=> ( $remainder_e(X,2) = 0 ) ) ).
tff(odd_def,axiom,
    ! [X: $int] : ( odd(X) <=> ( $remainder_e(X,2) = 1 ) ) ).
tff(tautology,conjecture,
    ! [X: $int] : ( even(X) | odd(X) ) ).
```

[13]See `https://tptp.org/UserDocs/TPTPLanguage/TPTPLanguage.shtml` for a further explanation of TPTP

Appendix C Additional Conjectures for the Key-Value Store

- The converse of extensionality, namely:

$$(\forall a \in KVS)(\forall b \in KVS)(a = b \rightarrow ((\forall k \in \mathbb{Z}^{\geq 0})\ get(a, k) = get(b, k)))$$

- Removing a key-value pair at a key k, then putting another value at k and getting that, should give one the key-value pair they just put:

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(\forall v \in \mathcal{I})(get(put(remove(a, k), k, v), k) = v)$$

- The result of *get* either returns a value or *null*:

$$(\forall a \in KVS)(\forall k \in \mathbb{Z}^{\geq 0})(((\exists v \in \mathcal{I})(get(a, k) = v)) \vee get(a, k) = null)$$

- If ones gets a value with a certain key k, the key-value pair returned should have that same key k.

Appendix D Reproducing the Results of Georgiou et al.

We wanted to ensure the results of the work of Georgiou et al. [9] were reproducible. We initially hypothesized that some of the results were not reproducible in Vampire 4.9 based on initial testing, despite the fact that Vampire 4.9 supported the necessary induction implementation. Because of this, we decided to check this rigorously.

To perform this check, we first built the version of Vampire we assumed the authors used. Their commits on their GitHub repository were done on August 16th 2023.[14] Moreover, they link to a branch in the Vampire repository in their `README`, which has now been merged to their `master` branch. The most recent commit on this branch before August 16th 2023 was on June 17th 2023, with hash `1b51a80`. We assumed this to be the version of Vampire they used, and therefore built it at that commit using gcc 11.5.0.[15]

Furthermore, we followed the advice of Hajdu (found in Appendix F) to prove the problems. Specifically, we use three proof searching strategies: the decoded strategy given in the benchmark files (D), the structural induction schedule (S), and the induction schedule (I), each of which are ran on the release version of Vampire 4.9 (commit `5ad494e78`), and the aforementioned version of Vampire 4.7 (commit `1b51a80`). Note that these schedules have slightly changed between versions. Each of these was ran with a time limit of two minutes, which is larger than the largest time limit the authors had given for their strategies. We used the maximum number of cores available on our machine for the schedules (S and I). We note that both versions of Vampire include Z3. Results are shown below and can be found on our fork of their GitHub repository.[16]

Table 3: Reproducing the results of Georgiou et al. [9] on the version of Vampire they likely used (4.7) and the current release version (4.9). D, S, and I refer to the decoded strategy, the structural induction schedule, and the induction schedule, respectively. We use the same problem naming as in their paper, namely IS, MS, and QS, refer to Insertion Sort, Merge Sort, and Quick Sort, respectively. S and PE refer to sortedness and permutation equivalence, respectively. Li refers to the ith lemma, if this is ommitted, this is the conjecture. Finally N/A, T, and P stand for Not Applicable, Timeout and Proven, respectively.

	4.9 + D	4.9 + S	4.9 + I	4.7 + D	4.7 + S	4.7 + I
IS-PE	N/A	P	P	P	P	P
IS-PE-L1	N/A	P	P	P	P	P
IS-S	N/A	P	P	P	P	P
IS-S-L1	N/A	T	T	T	T	T
MS-PE	N/A	T	T	P	P	P
MS-PE-L1	N/A	P	P	N/A	P	P
MS-PE-L2	N/A	T	T	P	P	P
MS-PE-L3	N/A	P	P	P	P	P
MS-S	N/A	T	T	P	P	P
MS-S-L1	N/A	T	T	N/A	T	T
MS-S-L2	N/A	P	P	T	P	P
QS-PE	N/A	T	T	P	P	P
QS-PE-L1	P	P	P	P	P	P
QS-PE-L2	P	P	P	P	P	P

[14]`https://github.com/mina1604/sorting_wo_sorts`
[15]`https://gcc.gnu.org/`
[16]`https://github.com/mbalfakeih/sorting_wo_sorts`

QS-S	N/A	T	T	P	P	P
QS-S-L1	N/A	T	T	T	P	P
QS-S-L2	P	P	P	P	P	P
QS-S-L3	T	T	T	T	T	T
QS-S-L4	N/A	T	T	T	P	P
QS-S-L5	T	P	P	T	T	T
QS-S-L6	N/A	P	P	P	P	P
QS-S-L7	T	P	P	T	P	P

There are a few conclusions that we can derive from thesse results. Firstly, as initially expected, there are a few problems that can be solved in Vampire 4.7, but can no longer be solved in Vampire 4.9, in particular, these are MS-PE, MS-PE-L2, MS-S, QS-PE, QS-S, QS-S-L1, QS-S-L4. This may be due to a change in the schedules used, or there may have been a more general change (such as in the saturation algorithms), that caused this.

Moreover, many of the strategies used in the work of Georgiou et al. [9] no longer work on Vampire 4.9, evident by the number of N/As in column 4.9 + D. This is due to Vampire 4.9 imposing different restrictions on strategies. The exact error messages can be found on the GitHub repository.

Furthermore, it appears that some of the strategies they defined could not be used with the version of Vampire we selected, in particular for MS-S-L1 (MS-PE-L1 does not have a strategy defined). This may indicate that we did not select the correct version of Vampire, and it was perhaps an earlier version. Given the results we derived, we assume the difference between the version of Vampire they used and the one we chose is not significant. This may also be the reason why some of the defined strategies time out in Vampire 4.7 in particular for MS-S-L2, QS-S-L1, QS-S-L4, and QS-S-L7.

Interestingly, QS-S-L5 is the only problem that was proven in Vampire 4.9, but not in 4.7. Once again, this is likely due to changes in the schedules used, or a more general change in Vampire.

In conclusion, not all of the results in the work of Georgiou et al. [9] can be reproduced in Vampire 4.9, due to changes in the schedules and general changes in proof strategies. We may not have used the exact same version of Vampire used by the original authors, however we believe our results are close enough to what they originally had in their paper that the difference is not significant.

Appendix E Key-Value Store Implementation

To implement the Key-Value Store, we made significant use of the TPTP arithmetic syntax [14]. We defined three helper functions, along with the five functions in Section 3.2 and their associated axioms, these are:

E.1 Helper Functions

`h(I : $int)` $\rightarrow$ `$int` is the hash function, and is implemented as `$remainder_e(I, n)`. This takes the remainder of `I` divided by `n`, when defining division as the euclidean quotient.

`get_chain(A: $array($int, list), I: $int)` $\rightarrow$ `list` retrieves a list in position `h(I)` of the Key-Value Store, and is implemented as `$select(A, h(I))`.

`size_up_to_index(A: $array($int, list), I: $int)` $\rightarrow$ `$int` retrieves the size of the Key-Value Store up to `I`, and is implemented using two axioms: a base case, where we compute `length_l` of the list at index 0, and a recursive case where we compute `length_l` of the list at the current index, and add that to the size at the previous index.

E.2 Main Functions

`get(A: $array($int, list), K: $int)` $\rightarrow$ `pair` retrieves the key-value pair with key `K` if it exists, otherwise it returns `null`. This is implemented as `get_l(get_chain(A, K), K)`.

`remove(A: $array($int, list), K: $int)` $\rightarrow$ `$array($int, list)` removes the element with key `K`, if it exists, and returns the new Key-Value Store. This is implemented as `$store(A, h(K), remove_l(get_chain(A, K), K))`.

`put(A: $array($int, list), K: $int, V: $i)` $\rightarrow$ `$array($int, list)` places the key-value pair `(K, V)` into the Key-Value Store and returns the new Key-Value Store. This is implemented in the same way as `remove`, but the call to `remove_l` is replaced with `cons(pair_cons(K, V), get_chain(remove(A, K), K))`.

`contains(A: $array($int, list), V: $i)` $\rightarrow$ `$o` checks if a value is contained in the Key-Value Store, this is defined as being true iff there exists a `K` such that `get(A, K) = pair_cons(K, V)`.

`size(A: $array($int, list))` $\rightarrow$ `$int` returns the number of key-value pairs values in the Key-Value Store, and is implemented as calling `size_up_to_index` with the index `n-1`.

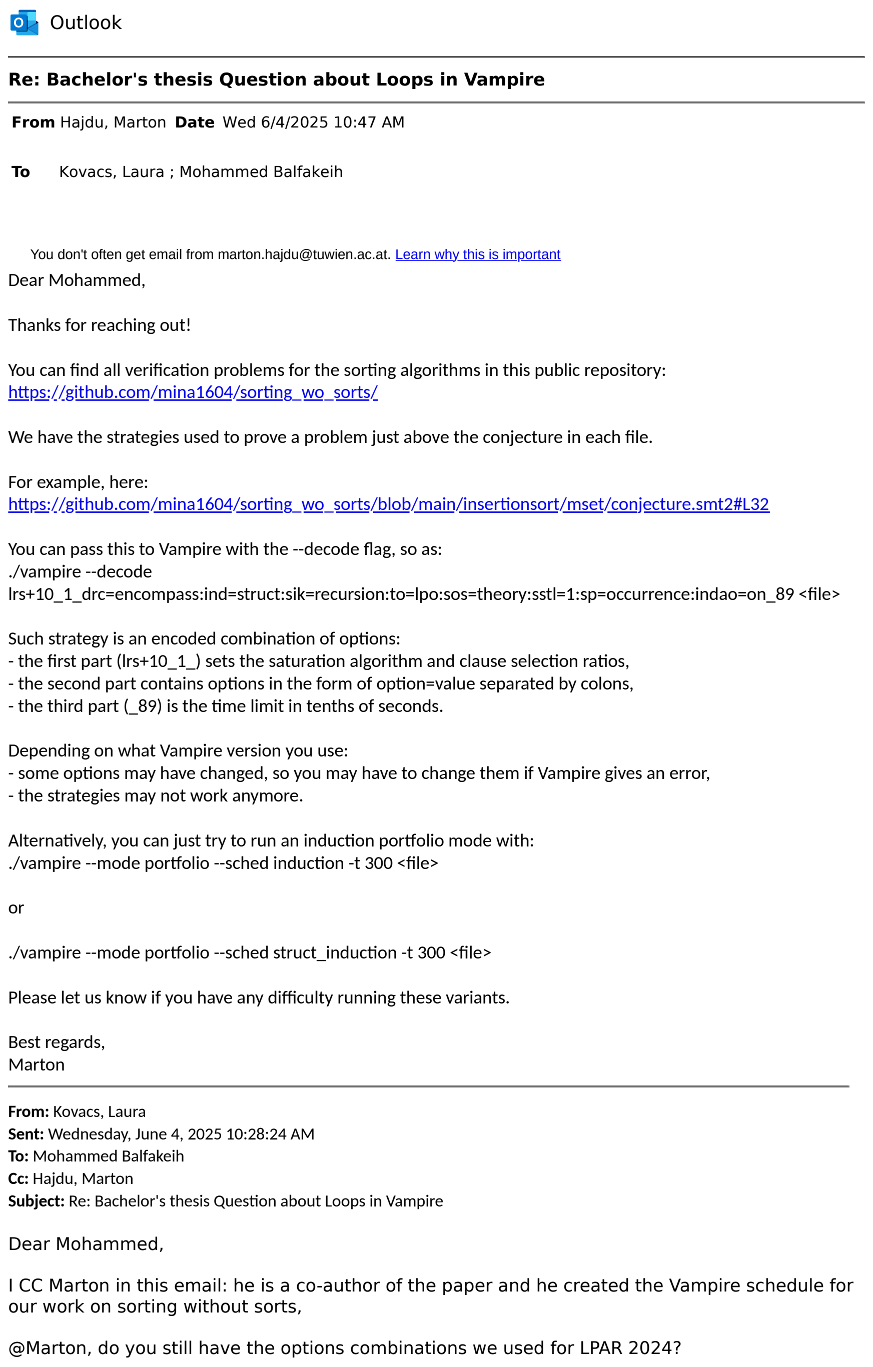
E.3 Additional axioms

Read over write one: `get(put(A, K, V), K) = pair_cons(K, V)` ensures that if a value is `put` at a certain key, then it can be retrieved.

Read over write two: `(K != L) => get(put(A, K, V), L) = get(A, L)` ensures that placing a value at one key does not affect other key-value pairs.

Extensionality: If for all `K`: `get(A, K) = get(B, K)` then `A = B` defines equality between Key-Value Stores.

Appendix F Email regarding Sorting from Vampire Developers



Outlook

Re: Bachelor's thesis Question about Loops in Vampire

From Hajdu, Marton **Date** Wed 6/4/2025 10:47 AM

To Kovacs, Laura ; Mohammed Balfakeih

You don't often get email from marton.hajdu@tuwien.ac.at. Learn why this is important

Dear Mohammed,

Thanks for reaching out!

You can find all verification problems for the sorting algorithms in this public repository:
https://github.com/mina1604/sorting_wo_sorts/

We have the strategies used to prove a problem just above the conjecture in each file.

For example, here:
https://github.com/mina1604/sorting_wo_sorts/blob/main/insertionsort/mset/conjecture.smt2#L32

You can pass this to Vampire with the --decode flag, so as:

```
./vampire --decode
lrs+10_1_drc=encompass:ind=struct:sik=recursion:to=lpo:sos=theory:sstl=1:sp=occurrence:indao=on_89 <file>
```

Such strategy is an encoded combination of options:
- the first part (lrs+10_1_) sets the saturation algorithm and clause selection ratios,
- the second part contains options in the form of option=value separated by colons,
- the third part (_89) is the time limit in tenths of seconds.

Depending on what Vampire version you use:
- some options may have changed, so you may have to change them if Vampire gives an error,
- the strategies may not work anymore.

Alternatively, you can just try to run an induction portfolio mode with:

```
./vampire --mode portfolio --sched induction -t 300 <file>
```

or

```
./vampire --mode portfolio --sched struct_induction -t 300 <file>
```

Please let us know if you have any difficulty running these variants.

Best regards,
Marton

From: Kovacs, Laura
Sent: Wednesday, June 4, 2025 10:28:24 AM
To: Mohammed Balfakeih
Cc: Hajdu, Marton
Subject: Re: Bachelor's thesis Question about Loops in Vampire

Dear Mohammed,

I CC Marton in this email: he is a co-author of the paper and he created the Vampire schedule for our work on sorting without sorts,

@Marton, do you still have the options combinations we used for LPAR 2024?

References

[1] "Final Report On the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris," English, BEA, Tech. Rep., Jul. 2012. [Online]. Available: `https://www.aaiu.ie/sites/default/files/FRA/BEA%20France%20Accident%20Airbus%20A330-203%20F-GZCP%20AF447%20Atlantic%20Ocean%2001-06-2012_opt.pdf`.

[2] M. Aniche, A. Deursen and S. Freeman, *Effective software testing: a developer's guide*, eng, First edition. Shelter Island, NY: Manning Publications Co, 2022, ISBN: 978-1-63343-993-1 978-1-63835-058-3.

[3] E. W. Dijkstra, *Notes on Structured Programming*, en, Apr. 1970. DOI: `10.26153/TSW/53177`. [Online]. Available: `https://repositories.lib.utexas.edu/handle/2152/126640` (visited on 30/05/2025).

[4] J. Woodcock, P. G. Larsen, J. Bicarregui and J. Fitzgerald, "Formal methods: Practice and experience," en, *ACM Computing Surveys*, vol. 41, no. 4, pp. 1–36, Oct. 2009, ISSN: 0360-0300, 1557-7341. DOI: `10.1145/1592434.1592436`. [Online]. Available: `https://dl.acm.org/doi/10.1145/1592434.1592436` (visited on 30/05/2025).

[5] M. S. Nawaz, M. Malik, Y. Li, M. Sun and M. I. U. Lali, *A Survey on Theorem Provers in Formal Methods*, arXiv:1912.03028 [cs], Dec. 2019. DOI: `10.48550/arXiv.1912.03028`. [Online]. Available: `http://arxiv.org/abs/1912.03028` (visited on 29/05/2025).

[6] A. Riazanov and A. Voronkov, "The design and implementation of VAMPIRE," *AI Communications*, vol. 15, no. 2, pp. 91–110, 2002. [Online]. Available: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-0036327027&partnerID=40&md5=e49c5a61a7465dd21eff9a7ba3e2d715`.

[7] G. Sutcliffe, "The CADE ATP System Competition - CASC," *AI Magazine*, vol. 37, no. 2, pp. 99–101, 2016.

[8] G. Sutcliffe, "The 12th IJCAR Automated Theorem Proving System Competition—CASC-J12," en, *The European Journal on Artificial Intelligence*, vol. 38, no. 1, pp. 3–20, Feb. 2025, ISSN: 3050-4554, 3050-4546. DOI: `10.1177/30504554241305110`. [Online]. Available: `https://journals.sagepub.com/doi/10.1177/30504554241305110` (visited on 21/06/2025).

[9] P. Georgiou, M. Hajdu and L. Kovács, "Saturating Sorting without Sorts," pp. 88–69. DOI: `10.29007/rg9z`. [Online]. Available: `https://easychair.org/publications/paper/qbDc` (visited on 13/05/2025).

[10] Ahrendt *et al.*, "Integrating Automated and Interactive Theorem Proving," in *Automated Deduction — A Basis for Applications*, D. M. Gabbay, J. Barwise, W. Bibel and P. H. Schmitt, Eds., vol. 9, Series Title: Applied Logic Series, Dordrecht: Springer Netherlands, 1998, pp. 97–116, ISBN: 978-90-481-5051-9 978-94-017-0435-9. DOI: `10.1007/978-94-017-0435-9_4`. [Online]. Available: `http://link.springer.com/10.1007/978-94-017-0435-9_4` (visited on 21/06/2025).

[11] D. A. Plaisted, "Automated theorem proving," en, *WIREs Cognitive Science*, vol. 5, no. 2, pp. 115–128, Mar. 2014, ISSN: 1939-5078, 1939-5086. DOI: `10.1002/wcs.1269`. [Online]. Available: `https://wires.onlinelibrary.wiley.com/doi/10.1002/wcs.1269` (visited on 27/05/2025).

[12] M. Manzano and V. Aranda, "Many-Sorted Logic," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta and U. Nodelman, Eds., Winter 2022, Metaphysics Research Lab, Stanford University, 2022. [Online]. Available: `https://plato-stanford-edu.tudelft.idm.oclc.org/archives/win2022/entries/logic-many-sorted/`.

[13] L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire," in *Computer Aided Verification*, D. Hutchison *et al.*, Eds., vol. 8044, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35, ISBN: 978-3-642-39798-1 978-3-642-39799-8. DOI: `10.1007/978-3-642-39799-8_1`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-39799-8_1` (visited on 21/05/2025).

[14] G. Sutcliffe, S. Schulz, K. Claessen and P. Baumgartner, "The TPTP Typed First-Order Form with Arithmetic," in *Logic for Programming, Artificial Intelligence, and Reasoning*, D. Hutchison *et al.*, Eds., vol. 7180, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 406–419, ISBN: 978-3-642-28716-9 978-3-642-28717-6. DOI: `10.1007/978-3-642-28717-6_32`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-28717-6_32` (visited on 29/04/2025).

[15] E. Kotelnikov, L. Kovács, G. Reger and A. Voronkov, "The Vampire and the FOOL," in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, arXiv:1510.04821 [cs], Jan. 2016, pp. 37–48. DOI: `10.1145/2854065.2854071`. [Online]. Available: `http://arxiv.org/abs/1510.04821` (visited on 25/04/2025).

[16] E. Kotelnikov, L. Kovács and A. Voronkov, "A First Class Boolean Sort in First-Order Theorem Proving and TPTP," in vol. 9150, arXiv:1505.01682 [cs], 2015, pp. 71–86. DOI: `10.1007/978-3-319-20615-8_5`. [Online]. Available: `http://arxiv.org/abs/1505.01682` (visited on 25/04/2025).

[17] A. Bhayat and M. Suda, "A Higher-Order Vampire (Short Paper)," en, in *Automated Reasoning*, C. Benzmüller, M. J. Heule and R. A. Schmidt, Eds., vol. 14739, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2024, pp. 75–85, ISBN: 978-3-031-63497-0 978-3-031-63498-7. DOI: `10.1007/978-3-031-63498-7_5`. [Online]. Available: `https://link.springer.com/10.1007/978-3-031-63498-7_5` (visited on 10/06/2025).

[18] G. Sutcliffe, "The TPTP Problem Library and Associated Infrastructure: From CNF to TH0, TPTP v6.4.0," en, *Journal of Automated Reasoning*, vol. 59, no. 4, pp. 483–502, Dec. 2017, ISSN: 0168-7433, 1573-0670. DOI: `10.1007/s10817-017-9407-7`. [Online]. Available: `http://link.springer.com/10.1007/s10817-017-9407-7` (visited on 23/04/2025).

[19] C. Barrett, P. Fontaine and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, Published: \tt www.SMT-LIB.org, 2016.

[20] G. Reger, M. Suda, L. Kovács and A. Voronkov, "First-Order Theorem Proving and Vampire The Theory Bit," en, [Online]. Available: `https://github.com/vprover/ase17tutorial/blob/master/theory.pdf`.

[21] M. Beeson, R. Veroff and L. Wos, "Double-Negation Elimination in Some Propositional Logics," en, *Studia Logica*, vol. 80, no. 2-3, pp. 195–234, Aug. 2005, ISSN: 0039-3215, 1572-8730. DOI: `10.1007/s11225-005-8469-4`. [Online]. Available: `http://link.springer.com/10.1007/s11225-005-8469-4` (visited on 13/06/2025).

[22] F. Bártek *et al.*, *The Vampire Diary*, Version Number: 1, 2025. DOI: `10.48550/ARXIV.2506.03030`. [Online]. Available: `https://arxiv.org/abs/2506.03030` (visited on 13/06/2025).

[23] G. Sutcliffe, "TPTP, TSTP, CASC, etc.," en, in *Computer Science – Theory and Applications*, V. Diekert, M. V. Volkov and A. Voronkov, Eds., vol. 4649, ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 6–22, ISBN: 978-3-540-74509-9 978-3-540-74510-5. DOI: `10.1007/978-3-540-74510-5_4`. [Online]. Available: `http://link.springer.com/10.1007/978-3-540-74510-5_4` (visited on 28/05/2025).

[24] G. Sutcliffe, "Stepping Stones in the TPTP World," in *Proceedings of the 12th International Joint Conference on Automated Reasoning*, C. Benzmüller, M. Heule and R. Schmidt, Eds., ser. Lecture Notes in Artificial Intelligence, Issue: 14739, 2024, pp. 30–50.

[25] G. Sutcliffe and E. Kotelnikov, "TFX: The TPTP extended typed first-order form," in *CEUR Workshop Proceedings*, Type: Conference paper, vol. 2162, 2018, pp. 72–87. [Online]. Available: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-85052731565&partnerID=40&md5=274ddee6a9c6116ce52e58b288461718`.

[26] R. Wang, "The investigation and discussion of the progress related to sorting algorithms," Shanghai, China, 2024, p. 040 014. DOI: `10.1063/5.0222533`. [Online]. Available: `https://pubs.aip.org/aip/acp/article-lookup/doi/10.1063/5.0222533` (visited on 10/06/2025).

[27] T. Nguyen Trung and M. Nguyen Hieu, "ZDB-high performance key-value store," in *2013 Third World Congress on Information and Communication Technologies (WICT 2013)*, Hanoi, Vietnam: IEEE, Dec. 2013, pp. 311–316, ISBN: 978-1-4799-3230-6. DOI: `10.1109/WICT.2013.7113154`. [Online]. Available: `http://ieeexplore.ieee.org/document/7113154/` (visited on 10/06/2025).

[28] T. Nipkow *et al.*, *Functional Algorithms, Verified!* English, 2021.

[29] M. Hajdu, P. Hozzová, L. Kovács, G. Reger and A. Voronkov, "Getting Saturated with Induction," en, in *Principles of Systems Design*, J.-F. Raskin, K. Chatterjee, L. Doyen and R. Majumdar, Eds., vol. 13660, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2022, pp. 306–322, ISBN: 978-3-031-22336-5 978-3-031-22337-2. DOI: `10.1007/978-3-031-22337-2_15`. [Online]. Available: `https://link.springer.com/10.1007/978-3-031-22337-2_15` (visited on 24/05/2025).

[30] C. Barrett, P. Fontaine and C. Tinelli, "The SMT-LIB Standard: Version 2.7," Department of Computer Science, The University of Iowa, Tech. Rep., 2025.

[31] C. Barrett, I. Shikanian and C. Tinelli, "An Abstract Decision Procedure for a Theory of Inductive Data Types," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, no. 1-2, B. Cook and R. Sebastiani, Eds., pp. 21–46, Jul. 2007, ISSN: 15740617. DOI: `10.3233/SAT190028`. [Online]. Available: `https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SAT190028` (visited on 08/06/2025).

[32] S. H. Einarsdóttir, M. Hajdu, M. Johansson, N. Smallbone and M. Suda, "Lemma Discovery and Strategies for Automated Induction," en, in *Automated Reasoning*, C. Benzmüller, M. J. Heule and R. A. Schmidt, Eds., vol. 14739, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2024, pp. 214–232, ISBN: 978-3-031-63497-0 978-3-031-63498-7. DOI: `10.1007/978-3-031-63498-7_13`. [Online]. Available: `https://link.springer.com/10.1007/978-3-031-63498-7_13` (visited on 20/05/2025).

[33] M. Hajdu, L. Kovács and M. Rawson, "Rewriting and Inductive Reasoning," pp. 278–260. DOI: `10.29007/vbfp`. [Online]. Available: `https://easychair.org/publications/paper/T1mjX` (visited on 10/06/2025).

[34] L. Kovács, P. Hozzová, M. Hajdu and A. Voronkov, "Induction in Saturation," en, in *Automated Reasoning*, C. Benzmüller, M. J. Heule and R. A. Schmidt, Eds., vol. 14739, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2024, pp. 21–29, ISBN: 978-3-031-63497-0 978-3-031-63498-7. DOI: 10.1007/978-3-031-63498-7_2. [Online]. Available: https://link.springer.com/10.1007/978-3-031-63498-7_2 (visited on 17/06/2025).

[35] K. Claessen, M. Johansson, D. Rosén and N. Smallbone, "TIP: Tons of Inductive Problems," en, in *Intelligent Computer Mathematics*, M. Kerber, J. Carette, C. Kaliszyk, F. Rabe and V. Sorge, Eds., vol. 9150, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 333–337, ISBN: 978-3-319-20614-1 978-3-319-20615-8. DOI: 10.1007/978-3-319-20615-8_23. [Online]. Available: https://link.springer.com/10.1007/978-3-319-20615-8_23 (visited on 17/06/2025).

[36] G. Sutcliffe, "The SZS ontologies for automated reasoning software," in *CEUR Workshop Proceedings*, Type: Conference paper, vol. 418, 2008, pp. 38–49. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84885656172&partnerID=40&md5=c27375bacd0f9a2e6ad43363324d7010.

[37] P. Raatikainen, "Gödel's Incompleteness Theorems," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta and U. Nodelman, Eds., Summer 2025, Metaphysics Research Lab, Stanford University, 2025. [Online]. Available: https://plato-stanford-edu.tudelft.idm.oclc.org/archives/sum2025/entries/goedel-incompleteness/.

[38] C. Başkent, "Some topological properties of paraconsistent models," en, *Synthese*, vol. 190, no. 18, pp. 4023–4040, Dec. 2013, ISSN: 0039-7857, 1573-0964. DOI: 10.1007/s11229-013-0246-8. [Online]. Available: http://link.springer.com/10.1007/s11229-013-0246-8 (visited on 14/06/2025).

[39] P. Georgiou, B. Gleiss, A. Bhayat, M. Rawson, L. Kovács and G. Reger, *The RAPID Software Verification Framework*, en, A. Griggio and N. Rungta, Eds., Oct. 2022. DOI: 10.34727/2022/ISBN.978-3-85448-053-2_32. [Online]. Available: https://repositum.tuwien.at/handle/20.500.12708/81379 (visited on 20/06/2025).

[40] B. Gleiss, L. Kovács and L. Schnedlitz, "Interactive Visualization of Saturation Attempts in Vampire," en, in *Integrated Formal Methods*, W. Ahrendt and S. L. Tapia Tarifa, Eds., vol. 11918, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 504–513, ISBN: 978-3-030-34967-7 978-3-030-34968-4. DOI: 10.1007/978-3-030-34968-4_28. [Online]. Available: http://link.springer.com/10.1007/978-3-030-34968-4_28 (visited on 30/05/2025).

[41] P. GNU, *Free Software Foundation. Bash (3.2. 48)[Unix shell program]*, 2007.

[42] H. Barbosa *et al.*, "Cvc5: A Versatile and Industrial-Strength SMT Solver," en, in *Tools and Algorithms for the Construction and Analysis of Systems*, ISSN: 1611-3349, Springer, Cham, 2022, pp. 415–442, ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-99524-9_24 (visited on 26/04/2025).

[43] A. Reynolds and V. Kuncak, "Induction for SMT Solvers," in *Verification, Model Checking, and Abstract Interpretation*, D. D'Souza, A. Lal and K. G. Larsen, Eds., vol. 8931, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 80–98, ISBN: 978-3-662-46080-1 978-3-662-46081-8. DOI: 10.1007/978-3-662-46081-8_5. [Online]. Available: http://link.springer.com/10.1007/978-3-662-46081-8_5 (visited on 02/06/2025).

[44] M. M. Y. Mohamed, A. Reynolds, C. Tinelli and C. Barrett, "Verifying SQL queries using theories of tables and relations," pp. 445–425. DOI: 10.29007/rlt7. [Online]. Available: https://easychair.org/publications/paper/MWDj (visited on 02/06/2025).

[45] T. Nipkow, M. Wenzel, L. C. Paulson, G. Goos, J. Hartmanis and J. Van Leeuwen, Eds., *Isabelle/HOL* (Lecture Notes in Computer Science). Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, vol. 2283, ISBN: 978-3-540-43376-7 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9. [Online]. Available: http://link.springer.com/10.1007/3-540-45949-9 (visited on 02/06/2025).

[46] M. Eberl, M. W. Haslbeck and T. Nipkow, "Verified Analysis of Random Binary Tree Structures," en, *Journal of Automated Reasoning*, vol. 64, no. 5, pp. 879–910, Jun. 2020, ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-020-09545-0. [Online]. Available: http://link.springer.com/10.1007/s10817-020-09545-0 (visited on 02/06/2025).

[47] D. Jiang and M. Zhou, "A comparative study of insertion sorting algorithm verification," in *2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Chengdu: IEEE, Dec. 2017, pp. 321–325, ISBN: 978-1-5090-6414-4. DOI: 10.1109/ITNEC.2017.8284998. [Online]. Available: http://ieeexplore.ieee.org/document/8284998/ (visited on 02/06/2025).

[48] J. Boender and G. Badevic, "Formal Verification of a Keystore," en, in *Theoretical Aspects of Software Engineering*, Y. Aït-Ameur and F. Crăciun, Eds., vol. 13299, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 49–64, ISBN: 978-3-031-10362-9 978-3-031-10363-6. DOI: 10.1007/978-3-031-10363-6_4. [Online]. Available: https://link.springer.com/10.1007/978-3-031-10363-6_4 (visited on 02/06/2025).