# Verified Verifying

*SMT-LIB for Strings in Isabelle*

Lotz, Kevin; Kulczynski, Mitja ; Nowotka, Dirk; Poulsen, Danny Bøgsted; Schlichtkrull, Anders

*Published in:*
Implementation and Application of Automata

*DOI (link to publication from Publisher):*
10.1007/978-3-031-40247-0_15

*Publication date:*
2023

*Document Version*
Accepted author manuscript, peer reviewed version

# Verified Verifying: SMT-LIB for Strings in Isabelle

Kevin Lotz [1], Mitja Kulczynski [1], Dirk Nowotka[1], Danny Bøgsted Poulsen[2], and Anders Schlichtkrull[2]

[1] Department of Computer Science, Kiel University, Kiel, Germany
{kel,mku,dn}@informatik.uni-kiel.de
[2] Department of Computer Science, Aalborg University, Aalborg, Denmark
{dannybpoulsen,andsch}@cs.aau.dk

**Abstract.** The prevalence of string solvers in formal program analysis has led to an increasing demand for more effective and dependable solving techniques. However, solving the satisfiability problem of string constraints, which is a generally undecidable problem, requires a deep understanding of the structure of the constraints. To address this challenge, the community has relied on SMT solvers to tackle the quantifier-free first-order logic fragment of string constraints, usually stated in SMT-LIB format. In 2020, the SMT-LIB Initiative issued the first official standard for string constraints. However, SMT-LIB states the semantics in a semi-formal manner, lacking a level of formality that is desirable for validating SMT solvers. In response, we formalize the SMT-LIB theory of strings using Isabelle, an interactive theorem prover known for its ability to formalize and verify mathematical and logical theorems. We demonstrate the usefulness of having a formally defined theory by deriving, to the best of our knowledge, the first automated verified model verification method for SMT-LIB string constraints and highlight potential future applications.

## 1 Introduction

Satisfiability Modulo Theories (SMT) solvers [6] have been instrumental in determining the satisfiability of first-order logic formulae within a given theory, driving much of the development of software verification [24,18] and formal verification applications. They have also seen widespread use in areas such as security [41,48] and program analysis [9,39].

A particular area where SMT solving is frequently applied is the verification of string-heavy programs. This can be partly attributed to the fact that strings are the most commonly used data type for processing user data and, consequently, mishandling of strings can pose significant security risks. Indeed, third-ranked security risk on the Open Web Application Security Project's (OWASP) top ten security risks in 2023 are injection attacks, which are fundamentally caused by inadequate string handling. While string reasoning is most commonly associated with web security applications[41,47], it is also applied in other areas

such as model checking [25] and cloud security [2,40]. Verification and automated reasoning tools typically discharge the heavy lifting of string constraint solving to dedicated SMT solvers. Over the past years, this led to the development of numerous solvers specialized in string reasoning (e.g.,[1,15,36,16,30]) and to widely adapted SMT solvers, such as CVC5 [3] and Z3 [17], adding support for strings. In 2020, efforts converged and were incorporated into the SMT-LIB 2.6 standard [44]. With the addition of the theory of strings, the SMT-LIB 2.6 standard strives to establish a common language and consistent semantics for solving problems related to strings.

Given their extensive usage, it is crucial to ensure that SMT solvers behave correctly and that implementation errors are detected as early as possible. A central question in that regard: *Can we trust an SMT solver's result*? If a solver determines that an input formula is satisfiable, it usually provides evidence of its decisions in the form of a *model*, i.e., an assignment of constants to the variables which satisfies the formula. A common practice to assess the soundness of a solver is to use another solver as an oracle to check whether the produced assignment indeed satisfies the formula at hand. However, that shifts the trust problem from one solver to another and poses a high risk of implementation errors carrying over to new solvers.

To address this problem, we present a novel approach for validating models produced by SMT solvers using Isabelle/HOL, an interactive theorem prover that provides a high-level specification language for expressing logical problems and powerful automation capabilities for proving properties [45]. In particular, our contributions are the following. We formalise the semantics of the SMT-LIB 2.6 theory of strings in Isabelle/HOL and provide an implementation of the standard model that is provably correct. The formalisation proved itself useful as we found inconsistencies in the standard, e.g., in the str. indexof operator, as we highlight in Section 3. The formalisation enables us to assess the soundness of SMT solvers by proving that an assignment produced by a solver is indeed a model of the input formula. Unlike using existing SMT solvers as test oracles, this provides a very strong guarantee that a model is indeed correct. We outline our efforts in building a model verification framework in Section 4 and show its usefulness on soundness issues reported in the literature.

*Related Work.* If we look at the Boolean satisfiability problem (SAT) rather than SMT then there are a number of works that use interactive theorem provers to construct verified SAT solvers. These were developed in Isabelle by Maric et al. [35,33,34], also in Isabelle by Fleury et al. [20,23,19,22,10,21,12,11], in Coq by Lescuyer [32] and in PVC by Shankar and Vaucher [43]. Additionally, there is the verified SAT solver in GURU by Oe et al. [38] which ensures model soundness at run time but is not proven to terminate.

SMT has also been combined with interactive theorem proving. In Isabelle, the *smt* tactic will run an SMT solver on a proof goal given in Isabelle, and then the tactic will try to reconstruct the proof in Isabelle's logic. The tactic supports Z3 with the theories of equality, uninterpreted functions, arrays, linear integer arithmetics and real arithmetics [13]. In order to be able to reconstruct the

proofs generated by the SMT solver, the SMT solver needs to be able to return a representation of the proof it generated. A recent work in this direction is by Schurr et al. who are building a common format, Alethe, to be used to reconstruct proofs made by SMT solvers [42]. Another recent work is by Barbosa et al. [4] who generate proofs for cvc5 and are also working on exporting them to the Alethe format. The above work concerns proof production from SMT solvers where the interest is in proving that a formula is satisfied by all models. The focus of our work goes in a different direction, namely that of checking in the String theory whether the model given by an SMT solver actually satisfies the input formula.

In the context of string solving, Kan et al. [27] developed a solver for a fragment of string constraints that is completely verified in Isabelle/HOL. On the theoretical side, implementations of regular expressions [29] and general theorems on combinatorics on words [26] were made in Isabelle/HOL.

## 2 Preliminaries

*SMT.* SMT extends SAT solving to many-sorted first-order logic within a given logical background theory $T$ that fixes the domain and the interpretation of the non-logical symbols. Solving an SMT formula $\varphi$ in the theory $T$ involves determining whether a model $M$ exists in $T$ that satisfies $\varphi$. The increasing interest in SMT led to the development of the SMT-LIB standard [5], which provides a modeling language for many-sorted first-order logic and specifies various background theories. SMT solvers ingest formulae expressed in the language of SMT-LIB and check their satisfiability according to a theory or a combination of theories specified in the SMT-LIB standard. Theories in SMT-LIB are described in terms of syntactical elements, i.e., the non-logical symbols. The intended semantics of theories are defined informally in natural language [5]. Examples of available theories include fixed-size bitvectors, integers, reals, and strings.

*Theory of strings.* We briefly summarise the theory of strings, $T_S$. An outline of the syntax is depicted in Figure 1. The theory deals with Boolean combinations of atomic formula including string equalities and inequalities, regular expression membership, and extended string predicates like containment and prefix relations.

A string term is a finite, ordered sequence of characters drawn from a finite alphabet, like ASCII or Unicode. String concatenation is denoted by $t_{str} \cdot t_{str}$. The length of a string term $w$, denoted by $str.len(w)$, is the number of characters. We also use $|w|$ to refer to the length of $w$ for readability. An empty string is represented by $\epsilon$ and has a length of 0. Operations referring to the index of a character or a sub-string within a string utilise zero-based indexing, that is, the first character has an index of zero. The term `str.to_int` treats a string as a non-negative base-10 integer, possibly with leading zeros. If the string is negative or contains non-digit characters, the value is -1. The term `str.from_int` converts a non-negative integer to the shortest possible string representing it in base 10. If the integer is negative, the value is an empty string. The atoms in $A_{re}$ correspond to regular membership constraints of a string term in a regular

$$
\begin{array}{lll}
F & ::= & Atom \mid F \wedge F \mid F \vee F \mid \neg F \\
Atom & ::= & t_{str} = t_{str} \mid A_{int} \mid A_{ext} \mid A_{re} \\
A_{re} & ::= & t_{str} \in RE \\
A_{int} & ::= & t_{int} = t_{int} \mid t_{int} < t_{int} \\
A_{ext} & ::= & str.contains(t_{str}, t_{str}) \mid str.prefixof(t_{str}, t_{str}) \mid str.suffixof(t_{str}, t_{str}) \\
t_{int} & ::= & m \mid v \mid str.len(t_{str}) \mid t_{int} + t_{int} \mid m \cdot t_{int} \mid str.indexof(t_{str}, t_{str}, t_{int}) \mid \\
 & & str.to\_int(t_{str}) \text{ where } m \in Con_{int} \ \& \ v \in Var_{int} \\
t_{str} & ::= & s \mid v \mid t_{str} \cdot t_{str} \mid str.from\_int(t_{int}) \mid str.replace(t_{str}, t_{str}, t_{str}) \mid \\
 & & str.replace\_all(t_{str}, t_{str}, t_{str}) \mid str.at(t_{str}, t_{int}) \mid str.substr(t_{str}, t_{int}, t_{int}) \\
 & & \text{where } s \in Con_{str} \text{ and } v \in Var_{str}
\end{array}
$$

Fig. 1: The syntax of the theory of strings $T_S$.

expression $RE$, constructed using accustomed regular operators. The satisfiability problem for the quantifier-free theory $T_S$ involves determining whether there exists an assignment of some constant in $Con_{str}$ to every string variable in $Var_{str}$ and some constant in $Con_{int}$ to every integer variable in $Var_{int}$, such that the formula evaluates to true under the semantics given in the SMT-LIB standard. If such an assignment exists, the formula is satisfiable, and if not, it is unsatisfiable.

For more information on the syntax and semantics of the theory of strings, we recommend referring to the SMT-LIB standard for the theory of strings [44].

*Isabelle.* Isabelle [37,45,46] is a generic proof assistant. Proof assistants are computer programs in which users can define objects from mathematics, logic and computer science, and prove lemmas and theorems about them. The proof assistant checks that the proofs are correct, and it can also perform parts of (or whole) proofs automatically. The value of this is that users get a very strong guarantee that their proofs are correct. Isabelle achieves this by having a small kernel that implements a relatively simple logical derivation system. That Isabelle is generic means that it supports several logics, the most prominent being Isabelle/HOL which is Isabelle's implementation of higher-order logic (HOL). This is also the logic that we are using in the present paper. Popularly speaking, Isabelle/HOL combines typed functional programming and logic.

## 3 Formalising the SMT Theory of Strings

The SMT-LIB theory of strings allows reasoning over finite sequences of characters and regular languages. The signature of the theory contains three sorts String, RegLan, and Int and consists of the various function symbols shown in Figure 1, such as str. substr, str. at. The SMT-LIB standard not only specifies syntax but also provides corresponding semantics. That is, the symbols are not intended to be interpreted arbitrarily but rather in a standard model in which the domains of the sorts are fixed and the interpretation of the function symbols is predefined. In particular, the domain of `String` is fixed to the set of all

finite words in $UC^*$, where $UC$ is the alphabet of 196607 Unicode characters, the domain of `RegLan` is the set of all regular languages over String, i.e., the set $2^{UC^*}$, and `Int` refers to the standard set of integers. The intended semantics of the function symbols within the respective domains are expressed semi-formally, in natural language. For example, the standard defines the semantics of the function as follows.

> If $0 \leq m < |w|$ and $0 < n$ then str.substr$(w, m, n)$ is the unique word $w_2$ such that $w = w_1 w_2 w_3$, $|w_3| = m$, and $|w_2| = min(n, |w| - m)$ for some words $w_1$ and $w_3$. Otherwise str.substr$(w, m, n) = \varepsilon$.

We formalise the standard model in Isabelle/HOL. This formalisation comprises an interpretation of all symbols as functions in Isabelle, a translation of the semantics of the standard model into higher-order logic sentences, and proofs that our interpretation satisfies them.

For example, we implement a function `substr::"uc_string ⇒ int ⇒int ⇒ uc_string"`, where `uc_string` denotes the type of Unicode strings, and `int` is the type of integers, and prove the following two lemmas:

```
assumes "0≤m ∧ m<|w| ∧ 0<n" shows "∃!v. substr w m n = v
    ∧ ∃x y. w=x·v·y ∧ |x|=m ∧ |v|=min n (|w|-m)"
assumes "0>m ∨ m≥|w| ∨ 0≥n" shows "substr w m n = ε"
```

These lemmas formalise the meaning of str.substr as detailed above. Our formalisation[3] currently encompasses all function symbols except for str.replace_all, str.replace_re, and str.replace_re_all.

*Formalisation.* We implement SMT-LIB functions based on a formalisation of the notions of words and regular expressions in Isabelle/HOL. To represent words over an arbitrary alphabet, we introduce the type `'a word` which is a synonym for lists of arbitrary type, i.e., `'a word ≡ 'a list`. The type `'a word` allows instantiation with arbitrary types as the underlying alphabet. The SMT-LIB standard specifies the alphabet to be Unicode characters. Therefore, we introduce a new type `UC` defined as the subset $\{0, \ldots, 196,607\}$ of integers, such that each term of type `UC` corresponds to a unique code point in the Unicode alphabet. By using `UC` as the type of the alphabet, the resulting type `UC word` is inhabited by all words over the Unicode alphabet and is thus a formalisation of the String sort defined by the SMT-LIB standard.

We implement the SMT-LIB string functions in terms of `UC word` and show that the implementation satisfies the properties of the standard model, as exemplified above. In doing so, we rely on functions on lists, as well as associated lemmas, that are already present in the Isabelle core libraries. For instance, the str.++ function directly corresponds to the Isabelle list `append` function. More complex SMT-LIB functions additionally require reasoning about the factors of a word, which we handle by implementing a function that projects a word $w$ onto the factor $w[i; j]$ between two indices $i, j \in \mathbb{N}$ with $0 \leq i \leq j$. We implement

[3] Available at `https://github.com/formalsmt/isabelle_smt`

SMT-LIB str.substr and str.at in terms of this function. Moreover, the projection to factors allows searching and replacing occurrences and factors with other words, as required by the SMT-LIB functions str.indexof, str.replace, respectively. However, while Isabelle uses natural numbers to represent the lengths of lists, the SMT-LIB standard defines the length of a string, including the indices of substrings, as integers. Therefore, we require additional conversions between integers and natural numbers and must handle edge cases. For example, in the str.substr implementation, we first check whether the given indices are valid, i.e., whether they are within the bounds of the string, then convert them to natural numbers, and finally project the string onto the corresponding factor:

```
str_substr w m n = if n ≥ 0 ∧ 0 ≤ m ∧ ((nat m) ≤ |w|-1)
    then w[(nat m);(nat (m+n))] else ε
```

To formalise the regular membership predicate, i.e., str.in_re, which evaluates to true whenever a given string term is a member of a given regular language, we first introduce an algebraic data type `'a regex` characterizing regular expressions. The type contains a constructor for each regular operation defined by the SMT-LIB standard, including `'a word` as one of the base cases. Likewise to the type `'a word`, we instantiate `'a regex` with the type `UC` to obtain the type `UC regex`, which is inhabited by all regular expressions over the Unicode alphabet. To establish a connection between regular expressions and regular languages as defined by the standard, we additionally define the function `lang::"'a regex ⇒ 'a word set` that maps a regular expression to its language using accustomed semantics. Hence, for any `UC regex` term $r$, the set `lang` $r$ is a (regular) subset of the set of all `UC word` terms, which means that `UC regex` formalises the sort RegLan. We prove that the regular expression type, equipped with the `lang` function, satisfies all properties that the SMT-LIB theory of strings requires. For example, we show that regular concatenation, `re_concat`, expresses the exact language specified by the standard, by proving the following lemma:

```
lang (re_concat r e) = {x·y | x y. x ∈ lang r ∧ y ∈ lang e}
```

Finally, we implement the str.in_re predicate in terms of Brzozowski derivatives [14]. We follow the approach outlined in [28], but adapt it to account for the full set of regular operations defined by the SMT-LIB theory of strings. That is, we define a function `deriv::"'a regex ⇒ 'a ⇒ 'a regex` that computes the derivative of a regular expression w.r.t. a single character, and its extensions to words `derivw::"'a regex ⇒ 'a word ⇒ 'a regex`. We then prove that for a term $w$ of type `'a word` and a term $r$ of type `'a regex`, $w$ is contained in the set `lang` $r$ if and only if the derivative `derivw` $r$ $w$ contains the empty word.

Using Brzozowski derivatives, testing regular membership amounts to executing a finite number of deterministic derivations steps. This approach is preferable to testing whether a words is contained in a (possibly infinite) set using the `lang` function, as Isabelle can perform the finitely many derivation steps automatically. This is especially important for the automated model verification as described in Section 4.

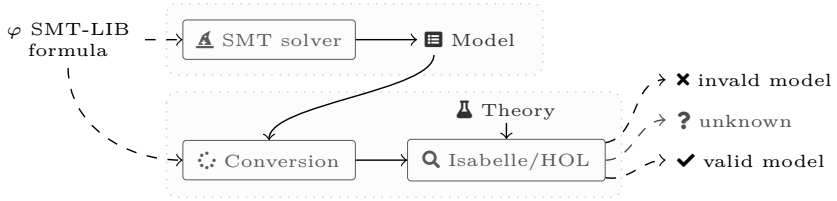


Fig. 2: SMT model verification process overview.

*Inconsistencies.* During our formalisation, we discovered several inconsistencies in the standard. Foremost, we found that the function str. indexof is not well-defined for all inputs. The standard requires that

> if $\text{str.contains}(w, w_2) = \text{true}$ and $i \geq 0$ then $\text{str.indexof}(w, w_2, i)$ is the smallest $n$ such that $w = w_1 w_2 w_3$ for some words $w_1$, $w_3$ with $i \leq n = |w_1|$. Otherwise, $\text{str.indexof}(w, w_2, i)$ is $-1$.

However, if either $i \geq |w|$ or $\text{str.contains}(\text{substr}(w, i, |w|), w_2) = \text{false}$, then such an $n$ cannot exist. For instance, $\text{str.indexof}(\text{“ab”}, \varepsilon, 3)$ is a counterexample concerning the first case. We have $3 \geq 1$ and $\text{str.contains}(\text{“ab”}, \varepsilon)$ by the definition of str. contains, but there are no words $w_1, w_3$ with $\text{“ab”} = w_1 \cdot \varepsilon \cdot w_3 = w_1 \cdot w_2$ and $3 \leq |w_1|$. For the second case, consider $\text{str.indexof}(\text{“ab”}, \text{“a”}, 1)$. Again $\text{str.contains}(\text{“ab”}, \text{“a”})$ and $1 \geq 0$, but no words $w_1$ $w_2$ with $\text{“ab”} = w_1 \cdot \text{“a”} \cdot w_2$ and $1 \leq |w_1|$ exist. In order to establish well-defined semantics for str. contains, we suggest modifying the premises such that $\text{str.contains}(\text{substr}(w, i, |w|), w_2) = \text{true}$ instead of $\text{str.contains}(w, w_2) = \text{true}$ and additionally ensuring that $i \leq |w|$ is satisfied.

Besides that, we found a minor inconsistency in the signatures of str. replace_re and str. replace_re_all. The standard defines them first as functions of type *String* → *RegLan* → *String* → *String* but later as functions of type *String* → *String* → *String* → *String*. The former was clearly intended.

## 4 Model Verification using Isabelle

We present SMTMV[4], an automated tool for SMT model verification that leverages our formalisation to check the accuracy of models generated by SMT solvers. The verification process involves a sequence of steps, which are summarized in Figure 2. For a satisfiable formula $\varphi$, an SMT solver is able to produce a model $M$, i.e., a variable assignment that satisfies the formula. SMTMV takes this potential model and converts it into Isabelle/HOL by mapping SMT-LIB functions to corresponding counterparts in the presented formalisation, and logical connectives to equivalent Isabelle primitives. The result is a shallow embedding $\varphi_I$ in Isabelle that is equivalent to $\varphi$ within the standard model of the theory

[4] Available at `https://github.com/formalsmt/SMTmv`

Table 1: Results using SMTMV to replicate and verify soundness issues highlighted in [8]. Here, "t.o." stands for "timeout" (60 seconds) and "⌀-time" measures the average time it took SMTMV to verify a model, not including timeouts.

| Solver | Tested | Solver Result | | | SMTMV | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **sat** | unsat | t.o. | **invalid** | valid | unknown | error | t.o. | ⌀-time |
| Z3TRAU | 5325 | 2940 | 2126 | 259 | 1846 | 0 | 1 | 0 | 1093 | 33.36s |
| OSTRICH | 28 | 21 | 7 | 0 | 20 | 0 | 0 | 1 | 0 | 4.88s |
| Z3STR3 | 13 | 0 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | - |

of strings. Afterwards, SMTMV represents the assignment $M$ equivalently as a conjunction of equalities $M_I$. Thus, within the SMT-LIB theory of strings, $M$ is a model to $\varphi$ if and only if $M_I \models \varphi_I$. SMTMV expresses $M_I \models \varphi_I$ as a lemma in Isabelle/HOL and queries the system to search for a proof. If Isabelle finds a proof, the assignment produced by the SMT solver is provably a model to the formula, and consequently, SMTMV returns *valid*. If Isabelle/HOL instead finds that the model doesn't satisfy the formula, it returns *invalid*. In cases where Isabelle/HOL can neither find a proof nor a counterexample, SMTMV returns *unknown*.

*Analysis.* To showcase the effectiveness of SMTMV, we utilise it to verify the soundness problems highlighted in [8]. The accompanying artifact offers a database of SMT-LIB instances and their reported outcomes. We re-ran the – at the time – unsound solvers Z3TRAU [1], OSTRICH [15], and Z3STR3 [7] to replicate the soundness issues and used SMTMV to verify the produced models. Note that we intentionally used outdated solver versions on which the errors were reported to demonstrate the effectiveness of SMTMV and that all solvers might have since been fixed. All experiments were run with a timeout of 60 seconds per instance for each, the solver and SMTMV. Our findings are summarised in Table 1. According to Berzish et al., Z3TRAU had 5,325 soundness issues, out of which SMTMV was able to identify 1846 provably invalid models. On 1,093 instances SMTMV timed out. However, Z3TRAU persistently produced models with a total length of well over 40,000 characters, which Isabelle was unable to handle within the set time limit. OSTRICH supposedly had 28 soundness issues. SMTMV found that 20 of them are due to invalid models. In one case OSTRICH returned *sat* on a formula that contains a unary disjunction, which is not valid syntax according to the SMT-LIB standard. Isabelle rejected this formula and SMTMV reported *error*. For Z3str3 [7], the authors reported 13 soundness issues, none of which were due to an invalid model.

## 5 Conclusion and Further Work

We presented a formalisation of the SMT-LIB theory of strings in Isabelle/HOL. Through this formalisation, we have identified inconsistencies in the SMT-LIB theory of strings and proposed rectifications for them. Additionally, we have

introduced a tool, named SMTmv, that automates the validation of SMT models and successfully identified invalid model production as the cause of known soundness issues in several solvers. We believe SMTmv will be valuable for both SMT solver developers and practitioners in identifying and rectifying soundness errors, e.g. integrated into the benchmarking tool ZaligVinder [31]. Our formalisation in Isabelle/HOL lays the groundwork for future research, such as extending the expressiveness to support other SMT-LIB theories beyond strings or providing a deep embedding of the SMT-LIB logic into Isabelle.

## References


1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–5. IEEE (2018)
2. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8602994
3. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I. pp. 415–442. Springer (2022)
4. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.W.: Flexible proof production in an industrial-strength SMT solver. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 15–35. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_3, https://doi.org/10.1007/978-3-031-10769-6_3
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
6. Barrett, C., Tinelli, C.: Satisfiability modulo theories. Springer (2018)
7. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59. IEEE (2017)
8. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: An SMT solver for regular expressions and linear arithmetic over string length. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II. pp. 289–312. Springer (2021)
9. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 307–321. Springer (2009)
10. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. J. Autom. Reason.

**61**(1-4), 333–365 (2018). https://doi.org/10.1007/s10817-018-9455-7, https://doi.org/10.1007/s10817-018-9455-7

11. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Olivetti, N., Tiwari, A. (eds.) Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9706, pp. 25–44. Springer (2016). https://doi.org/10.1007/978-3-319-40229-1_4, https://doi.org/10.1007/978-3-319-40229-1_4
12. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 4786–4790. ijcai.org (2017). https://doi.org/10.24963/ijcai.2017/667, https://doi.org/10.24963/ijcai.2017/667
13. Böhme, S., Weber, T.: Fast LCF-Style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 179–194. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_14, https://doi.org/10.1007/978-3-642-14052-5_14
14. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM (JACM) **11**(4), 481–494 (1964)
15. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
16. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using sat. In: Reachability Problems: 13th International Conference, RP 2019, Brussels, Belgium, September 11–13, 2019, Proceedings 13. pp. 93–106. Springer (2019)
17. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340. Springer (2008)
18. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 1–24 (2014)
19. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11460, pp. 148–165. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_10, https://doi.org/10.1007/978-3-030-20652-9_10
20. Fleury, M.: Formalization of logical calculi in Isabelle/HOL. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2020), https://tel.archives-ouvertes.fr/tel-02963301
21. Fleury, M., Blanchette, J.C., Lammich, P.: A verified SAT solver with watched literals using imperative HOL. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 158–171. ACM (2018). https://doi.org/10.1145/3167080, https://doi.org/10.1145/3167080
22. Fleury, M., Schurr, H.: Reconstructing veriT proofs in Isabelle/HOL. In: Reis, G., Barbosa, H. (eds.) Proceedings Sixth Workshop on Proof eXchange for Theorem

Proving, PxTP 2019, Natal, Brazil, August 26, 2019. EPTCS, vol. 301, pp. 36–50 (2019). https://doi.org/10.4204/EPTCS.301.6, https://doi.org/10.4204/EPTCS.301.6
23. Fleury, M., Weidenbach, C.: A verified SAT solver framework including optimization and partial valuations. In: Albert, E., Kovács, L. (eds.) LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020. EPiC Series in Computing, vol. 73, pp. 212–229. EasyChair (2020). https://doi.org/10.29007/96wb, https://doi.org/10.29007/96wb
24. Grimm, T., Lettnin, D., Hübner, M.: A survey on formal verification techniques for safety-critical systems-on-chip. Electronics **7**(6), 81 (2018)
25. Hojjat, H., Rümmer, P., Shamakhi, A.: On strings in software model checking. In: Lin, A.W. (ed.) Programming Languages and Systems. pp. 19–30. Springer International Publishing, Cham (2019)
26. Holub, v., Starosta, v.: Formalization of Basic Combinatorics on Words. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 193, pp. 22:1–22:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.22, https://drops.dagstuhl.de/opus/volltexte/2021/13917
27. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: a certified string solver. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 210–224 (2022)
28. Krauss, A., Nipkow, T.: Regular sets and expressions. Archive of Formal Proofs (May 2010), https://isa-afp.org/entries/Regular-Sets.html, Formal proof development
29. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. Journal of Automated Reasoning **49**(1), 95–106 (Mar 2011). https://doi.org/10.1007/s10817-011-9223-4, https://doi.org/10.1007/s10817-011-9223-4
30. Kulczynski, M., Lotz, K., Nowotka, D., Poulsen, D.B.: Solving string theories involving regular membership predicates using sat. In: Model Checking Software: 28th International Symposium, SPIN 2022, Virtual Event, May 21, 2022, Proceedings. pp. 134–151. Springer (2022)
31. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: The power of string solving: simplicity of comparison. In: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test. pp. 85–88 (2020)
32. Lescuyer, S.: Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq). Ph.D. thesis, University of Paris-Sud, Orsay, France (2011), https://tel.archives-ouvertes.fr/tel-00713668
33. Maric, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. **411**(50), 4333–4356 (2010). https://doi.org/10.1016/j.tcs.2010.09.014, https://doi.org/10.1016/j.tcs.2010.09.014
34. Maric, F., Janicic, P.: Formalization of abstract state transition systems for SAT. Log. Methods Comput. Sci. **7**(3) (2011). https://doi.org/10.2168/LMCS-7(3:19)2011, https://doi.org/10.2168/LMCS-7(3:19)2011
35. Marić, F.: Formal verification of modern sat solvers. Archive of Formal Proofs (July 2008), https://isa-afp.org/entries/SATSolverVerification.html, Formal proof development

36. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In: International Symposium on Formal Methods. pp. 389–406. Springer (2021)
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
38. Oe, D., Stump, A., Oliver, C., Clancy, K.: versat: A verified modern SAT solver. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7148, pp. 363–378. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_24, https://doi.org/10.1007/978-3-642-27940-9_24
39. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference. pp. 139–148. SAICSIT '12 (2012)
40. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 3–18. Springer International Publishing, Cham (2022)
41. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: 2010 IEEE Symposium on Security and Privacy. pp. 513–528. IEEE (2010)
42. Schurr, H., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a generic SMT proof format (extended abstract). In: Keller, C., Fleury, M. (eds.) Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021. EPTCS, vol. 336, pp. 49–54 (2021). https://doi.org/10.4204/EPTCS.336.6, https://doi.org/10.4204/EPTCS.336.6
43. Shankar, N., Vaucher, M.: The mechanical verification of a dpll-based satisfiability solver. In: Haeusler, E.H., del Cerro, L.F. (eds.) Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop, LSFA 2010, Natal, Brazil, August 31, 2010. Electronic Notes in Theoretical Computer Science, vol. 269, pp. 3–17. Elsevier (2010). https://doi.org/10.1016/j.entcs.2011.03.002, https://doi.org/10.1016/j.entcs.2011.03.002
44. Tinelli, C., Barrett, C., Fontaine, P.: Smt: Theory of strings. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml, accessed: 2022-03-03
45. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings 21. pp. 33–38. Springer (2008)
46. Wenzel, M., et al.: The Isabelle/Isar reference manual (2004)
47. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. Formal Methods Syst. Des. **44**(1), 44–70 (2014). https://doi.org/10.1007/s10703-013-0189-1, https://doi.org/10.1007/s10703-013-0189-1
48. Zbrzezny, A.M., Szymoniak, S., Kurkowski, M.: Practical approach in verification of security systems using satisfiability modulo theories. Logic Journal of the IGPL **30**(2), 289–300 (2022)