SpaceSearch: A Library for Building and Verifying Solver-Aided Tools

KONSTANTIN WEITZ, University of Washington, USA
STEVEN LYUBOMIRSKY, University of Washington, USA
STEFAN HEULE, Stanford University, USA
EMINA TORLAK, University of Washington, USA
MICHAEL D. ERNST, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

Many verification tools build on automated solvers. These tools reduce problems in a specific application domain (e.g., compiler optimization validation) to queries that can be discharged with a highly optimized solver. But the correctness of the reductions themselves is rarely verified in practice, limiting the confidence that the solver's output establishes the desired domain-level property.

This paper presents SpaceSearch, a new library for developing solver-aided tools within a proof assistant. A user builds their solver-aided tool in Coq against the SpaceSearch interface, and the user then verifies that the results provided by the interface are sufficient to establish the tool's desired high-level properties. Once verified, the tool can be extracted to an implementation in a solver-aided language (e.g., Rosette), where SpaceSearch provides an efficient instantiation of the SpaceSearch interface with calls to an underlying SMT solver. This combines the strong correctness guarantees of developing a tool in a proof assistant with the high performance of modern SMT solvers. This paper also introduces new optimizations for such verified solver-aided tools, including parallelization and incrementalization.

We evaluate SpaceSearch by building and verifying two solver-aided tools. The first, SaltShaker, checks that RockSalt's x86 semantics agrees with STOKE's x86 semantics for a given instruction instantiation. When run on 15,255 instruction instantiations, SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE. After these systems were patched by their developers, SaltShaker verified the semantics' agreement on these instruction instantiations in under 2h. The second tool is a verified version of Bagpipe, a Border Gateway Protocol (BGP) router configuration checker. Like the previous, unverified, version, our new Bagpipe implementation scales to checking industrial configurations spanning over 240 KLOC, identifying 19 configuration inconsistencies with no false positives. Furthermore, in the process of verifying Bagpipe, we identified and fixed 2 bugs from the unverified implementation. These results demonstrate that SpaceSearch is a practical approach to developing efficient, verified solver-aided tools.

CCS Concepts: • **Theory of computation → Program verification**; **Constraint and logic programming**;

Additional Key Words and Phrases: Coq, SMT solver-aided tools, Bagpipe, SaltShaker

ACM Reference Format:
Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. 2017. SpaceSearch: A Library for Building and Verifying Solver-Aided Tools. *Proc. ACM Program. Lang.* 1, ICFP, Article 25 (September 2017), 28 pages.
https://doi.org/10.1145/3110269

2475-1421/2017/9-ART25
https://doi.org/10.1145/3110269

1 INTRODUCTION

Solver-aided tools are used in a variety of domains including data-race detection [Lahiri et al. 2009; Li and Gopalakrishnan 2010; Said et al. 2011], memory-model checking [Torlak et al. 2010], and compiler optimization validation [Kundu et al. 2009; Lopes et al. 2015]. Such tools reduce complex properties in their application domain to simpler queries that can be checked by a high-performance automated solver. In practice, the correctness of these reductions is rarely formally verified, limiting confidence in the soundness of the tool.

Past work has helped mitigate this problem by providing solver-aided host languages, such as Smten [Uhler and Dave 2014] and Rosette [Torlak and Bodik 2013, 2014]. These languages modify an existing language's runtime to provide a high-level interface to the underlying solver. Compared to a solver's low-level interface (e.g., SMTLib's [Barrett et al. 2016a] data types and commands), the high-level interface reduces the effort required to build a solver-aided tool by orders of magnitude [Uhler and Dave 2014] and, because the implementation is simpler, improves confidence in the tool's correctness.

However, even solver-aided tools written in a solver-aided host language still require complex reductions, which are difficult to get right, as they typically depend on sophisticated domain knowledge [Kundu et al. 2009; Sigurbjarnarson et al. 2016a; Torlak et al. 2010; Weitz et al. 2016]. As existing solver-aided host languages do not support formal reasoning about the meaning of solver calls in terms of the tool's application domain, it is difficult to ensure that such reductions maintain the soundness of the tool. For example, PEC is a solver-aided tool for verifying compiler loop optimizations [Kundu et al. 2009]; it decomposes the problem of proving equivalence between the original and optimized code (its *application domain property*) by splitting the code at its branching points, and proving the equivalence of the resulting original and optimized straight-line code fragments using an SMT solver. Proving that the equivalence of these straight-line code fragments can always be "stitched together", to ensure the equivalence of the entire original and the optimized code, requires reasoning in a higher-order logic [Tatlock and Lerner 2010], and thus cannot be done using existing solver-aided host languages.

Even after a problem has been reduced to a solver query, various optimizations are typically required to achieve good performance, including selecting the right solver data types [Panchekha and Torlak 2016; Sigurbjarnarson et al. 2016b; Singh et al. 2014], query incrementalization [Phothilimthana et al. 2014], and query parallelization [Jeon et al. 2015]. Without the ability to formally reason about solver results, it is difficult to ensure that such optimizations maintain the soundness of the tool.

This paper presents SpaceSearch, the first library to provide a high-level interface for building and formally verifying solver-aided tools. SpaceSearch exposes it's high-level interface to programs written in a proof assistant. SpaceSearch is thus the first solver-aided host language for proof assistants.

SpaceSearch's high-level interface is inspired by Smten [Uhler and Dave 2014], extended with a machine-checkable denotational semantics of the interface, to support reasoning about solver-aided tools in proof assistants. Programmers use the programming language of the proof assistant to build their solver-aided tool against SpaceSearch's interface, and use the expressive logic of the proof assistant to formally verify that the results of SpaceSearch's interface operations are sufficient to establish the tool's desired application domain properties.

Once a solver-aided tool is implemented against SpaceSearch's interface, it can be extracted (translated) to a target solver-aided host language (e.g., Rosette) where SpaceSearch provides an efficient instantiation of the SpaceSearch interface with calls to an SMT solver. This combines the strong correctness guarantees of developing a tool in a proof assistant with the high performance of SMT solvers. A user of SpaceSearch trusts the implementation of the proof assistant (including

its extraction mechanism), the instantiation of the SpaceSearch interface in the target solver-aided host language, the implementation of the target solver-aided host language (e.g., Rosette), and the implementation of the SMT solver[1].

To enable construction of efficient tools, SpaceSearch employs a modular design that factors its interface into multiple abstract data types (ADTs). Thanks to this design, SpaceSearch can be easily extended with new backends and optimizations, including incrementalization and parallelization.

We evaluated SpaceSearch on two solver-aided tools. First, we built and verified SaltShaker, a solver-aided tool to check the correctness of RockSalt's Coq x86 semantics [Morrisett et al. 2012] with respect to STOKE's SMT x86 semantics [Schkufza et al. 2013]. SaltShaker checks, for all possible machine states, that a given x86 instruction's RockSalt semantics is equivalent to the instruction's STOKE [Schkufza et al. 2013] semantics. The verification of SaltShaker required showing that searching a smaller space of machine states is sound, and that the search problem is incrementalizable. Once built and verified, we ran SaltShaker on 15,255 instruction instantiations. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE. We reported these bugs, and they were subsequently fixed by the respective developers. Once fixed, SaltShaker verified the semantics' agreement on these instruction instantiations in under 2h.

Second, we used SpaceSearch to revise and verify a Bagpipe prototype [Weitz et al. 2016]. Bagpipe is a solver-aided tool written in Rosette that checks Border Gateway Protocol (BGP) configurations. Bagpipe's main algorithm relies on a sophisticated reduction from its domain-specific BGP problem to a set of SMT queries. The revised Bagpipe runs on industrial configurations with over 240 KLOC, found 19 inconsistencies, and provides the same performance as the unverified Bagpipe prototype. In the process of verifying Bagpipe, we found 2 bugs in the Bagpipe prototype.

SpaceSearch presents the first general approach to formally verifying solver-aided tools that were written in a solver-aided host language. This required several technical innovations, including:

- Formalizing the Smten interface to support reasoning about solver-aided tools in proof assistants (Section 2.1). Our denotational semantics enables both ease of reasoning and extensibility as shown by our case studies and libraries (Sections 3, 5 and 6).
- A translation from Smten's interface to Rosette's symbolic execution API (Section 4.2), thus enabling any Smten-based tool to potentially be ported to Rosette.
- An application of proof assistant extraction mechanisms to support solver-aided tool development (Section 4.2) without modifying the host language interpreter (as done in Haskell for Smten, and in Racket for Rosette).
- Extensions to the Smten interface to support new optimizations including parallelization and incrementalization (Sections 3 and 4). These extensions could be back-ported to existing solver-aided frameworks and are not proof-assistant-specific. These general mechanisms are novel and were essential for building effective tools in our case studies.
- Designing a set of libraries that users can employ to build and reason about solver-aided tools. The design tradeoffs for such libraries (discussed in Section 3) will be valuable for users applying and extending SpaceSearch.

This paper is organized as follows. Section 2 provides an overview of the paper. Section 3 describes the SpaceSearch library, which exposes an interface for constructing solver-aided tools in proof assistants, as well as formal denotational semantics to reason about this interface. Section 4 describes various backends for SpaceSearch's high-level interface that enable solving search problems using brute force, parallel, incremental, and SAT/SMT search, as well as a discussion of when each

[1]If used with an SMT solver that generates witnesses (e.g. [Armand et al. 2011]), and a solver-aided host language that generates witnesses (we are not aware that such a language currently exists), then we expect that SpaceSearch could generate and verify these witnesses, thus removing the SMT solver and solver-aided host language from the TCB.

$$Space : Type \to Type$$
$$[\![\_]\!] : Space(A) \to \mathcal{P}(A)$$
$$empty_A : Space(A)$$
$$single_A : A \to Space(A)$$
$$union_A : Space(A) \to Space(A) \to Space(A)$$
$$bind_{A,B} : Space(A) \to (A \to Space(B)) \to Space(B)$$

$$[\![empty]\!] = \emptyset$$
$$[\![single(x)]\!] = \{x\}$$
$$[\![union(s,t)]\!] = [\![s]\!] \cup [\![t]\!]$$
$$[\![bind(s,f)]\!] = \bigcup_{a \in [\![s]\!]} [\![f(a)]\!]$$
$$search_A : Space(A) \to option(A)$$
$$search(s) = None \implies [\![s]\!] = \emptyset$$
$$search(s) = Some(a) \implies a \in [\![s]\!]$$

Fig. 1. *Basic SpaceSearch ADT.* The Basic SpaceSearch ADT provides the type of search problems *Space*, operations to construct empty and singleton search problems (*empty* and *single* respectively), an operation to combine search problems (*union*), an operation to bind functions over search problems (*bind*), and an operation to find solutions in a search problem (*search*). A search problem s is denoted to a set using the $[\![s]\!]$ function.

variant is most appropriate. Section 5 and Section 6 describe an evaluation of SpaceSearch via the construction of two solver-aided tools: SaltShaker which checks the correctness of x86 instruction semantics in RockSalt, and Bagpipe which checks the correctness of Border Gateway Protocol (BGP) configurations. Section 7 describes related work, and Section 8 concludes.

2 OVERVIEW

SpaceSearch provides a high-level interface to solver operations and their semantics in a proof assistant, enabling development and verification of solver-aided tools. Instead of exposing a low-level solver interface (e.g., SMTLib's [Barrett et al. 2016a] data types and commands), SpaceSearch provides a high-level interface inspired by Smten [Uhler and Dave 2014].

SpaceSearch's interface exposes solver functionality as operations to construct and to automatically solve *search problems*, leading to both compact high-level encodings and high-performance solver-aided tools [Uhler and Dave 2014]. A search problem is the problem of finding a *solution* in a given *search space* (think of this as a set of solutions). Search problems are generally useful — they may be used to find values with a certain property, e.g., solutions to a boolean formula; or to verify the absence of values with a certain property, e.g., counterexamples to the commutativity of integer addition.

This section presents an overview of SpaceSearch operations for constructing and solving search problems, a denotational semantics to formally reason about these operations, and an explanation of how the operations are implemented upon extraction. We show how to use SpaceSearch to build and verify a toy solver-aided tool for solving n-Queens problems.

2.1 SpaceSearch Interface

SpaceSearch exposes solver functionality as operations to construct and to automatically solve search problems. The basic SpaceSearch ADT interface (Figure 1) provides the search problem type, operations to construct search problems, and an operation to solve these problems. The SpaceSearch interface can be implemented either naively within the proof assistant (e.g., via the use of a finite set library), or efficiently by extraction to a solver-aided host language (e.g., Rosette).

Search Space Type. SpaceSearch uses the type $Space(A)$ to represent search problems (we use search problem and search space interchangeably) for solutions of some type A. SpaceSearch assigns

meaning to a search problem s by providing the function $[\![s]\!]$, which denotes s to a subset of the inhabitants of A (the powerset $\mathcal{P}(A)$). For example, the search problem of finding a "prime number greater than 1000" can be thought of as the problem of finding a value in the subset of numbers containing only "prime numbers greater than 1000".

Constructing Search Spaces. SpaceSearch provides four operations to construct search problems, the semantics of these operations is as follows. The $empty_A$ operation constructs a search problem with no solutions, $single_A(x)$ with exactly one solution x, and $union_A(s, t)$ with solutions of type A that are either in s or t. The $bind_{A,B}(s, f)$ operation creates a search problem by first applying f to every solution of type A in s, and then combining the solutions of type B from the resulting search problems into one. Note that the SpaceSearch Basic ADT does *not* provide space subtraction, which would require additional assumptions about the structure of the solution type, as discussed in Section 3.2. These operations are subscripted by the type of solutions in the search problem, but we omit subscripts that can be inferred from context.

Note that the operations for constructing search spaces do not require the corresponding SpaceSearch implementation to actually enumerate the entire space. Because ADTs hide their implementation details, SpaceSearch is free to choose whichever internal representation is most efficient.

Solving Search Spaces. SpaceSearch also provides the *search* operation, which takes a search space s, and either returns *None*, which means that the search space s is empty, or $Some(a)$, which means that a is a solution to s. In the case of multiple solutions to s, only one arbitrary solution is returned. To obtain additional solutions, a user can remove a from s and re-run *search*.

2.2 n-Queens Example

To illustrate SpaceSearch, we apply it to build and verify a simple solver-aided tool for solving n-queens problems. This case study, as well as the more complex case studies in Sections 5 and 6, uses the following methodology to build and verify a solver-aided tool. 1) We formally express the solver-aided tool's specification in a proof assistant. 2) We implement the optimized solver-aided tool in a proof assistant, using the operations of SpaceSearch. 3) We prove that the solver-aided tool meets its specification (and thus that its optimizations are correct), using the denotational semantics of SpaceSearch. For n-queens, the second step closely follows a Smten tutorial [Uhler 2014]; the other steps are novel.

1) n-Queens Specification. A solution to an n-queens problem places n queens on an $n \times n$ chessboard so that no two distinct queens attack each other, defined as two queens sharing the same column, row, or diagonal.

Formally, a set Q of n integer pairs is a solution to the n-queens problem iff all integers are in the range $[0..n)$ and for any distinct members $(x_1, y_1), (x_2, y_2) \in Q$, the following holds: $x_1 \neq x_2 \wedge y_1 \neq y_2 \wedge |x_1 - x_2| \neq |y_1 - y_2|$.

The following section describes a solver-aided tool that decides the n-queens problem for any given n. For any n, it either returns a solution to the n-queens problem, or it reports that no such solution exists.

2) n-Queens Solver-Aided Tool. Figure 2 shows a SpaceSearch implementation of an n-queens decision procedure. The implementation consists of three functions:

solveNQueens takes the problem size n and uses the *search* operation to find a solution in the space of non-attacking queens. This space is constructed by binding over a space of queen placements $placements(n, n)$, and only keeping those placements that are non-attacking.

```
— solveNQueens(n) decides the n-queens problem.
solveNQueens(n : Integer) : option(list(Integer × Integer))
  search(bind(placements(n, n), (λq.
    if noAttack(q) then single(q) else empty))).

— noAttack(q) checks whether a placement of queens q is non-attacking.
noAttack(q : list(Integer × Integer)) : bool :=
  distinct(map(fst, q)) ∧ distinct(map(snd, q)) ∧
  — optimization 2: use distinct to check for diagonal attack.
  distinct(map(plus, q)) ∧ distinct(map(minus, q))

— placements(n, m) is a space containing placements for m queens on an n × n chessboard.
placements(n, 0) := single([ ])
placements(n, i + 1) :=
  bind(range(0, n), (λy : Integer.
    bind(placements(n, i), (λq : list(Integer × Integer)
      — optimization 1: use i as the x coordinate.
      single((i, y) :: q)))))
```

Fig. 2. *n-Queens in SpaceSearch.* A solution to an n-queens problem places n queens on an $n \times n$ chessboard so that no two distinct queens attack each other horizontally, vertically, or diagonally. If the n-queens problem is solvable, *solveNQueens*(n) finds a solution using the *search* operation to provide a board configuration in the space of non-attacking queens. This space is constructed by binding over a space of queen placements (*placements*(n, n)), and only keeping those placements that are non-attacking (*noAttack*(q)).

noAttack(q) checks whether a placement of queens q is non-attacking. This is implemented by checking that the column (x-values, accessed using *fst*) of all queens is distinct, that the row (y-values, accessed using *snd*) of all queens is distinct, that the column plus row ($x + y$) of all queens is distinct (*plus* sums the components of a pair), and that the column minus row ($x - y$) of all queens is distinct (*minus* subtracts the components of a pair).

placements(n, m) is a space containing placements for m queens on an $n \times n$ chessboard. The space contains the placements that position the i^{th} queen (out of m) in the $i - 1^{\text{th}}$ column (x-value) and any row (y-value) contained in the space *range*$(0, n)$ of integers $[0..n)$.

3) n-Queens Correctness. It is not obvious that the Smten algorithm for deciding n-queens problems meets its specification. This is mainly due to two optimizations, which we will now prove correct using the SpaceSearch semantics and Coq.

The first optimization reduces the space of all queen placements to the space containing only the placements *placements*(n, n) that put each queen in a different column (x-value). We prove this optimization sound in Coq, using the intuition that *placements*(n, n) is a subspace of all possible queen placements. We also prove in Coq that this optimization is complete, namely that if a valid

arrangement of n queens exists, the search will return a solution. First, we note that any placement of two queens on the same column leads to an attack and is thus not a solution. Second, the order in which a list of queen positions is given does not affect whether it is a non-attacking arrangement; thus, if a non-attacking arrangement of n queens q exists, we may construct an equivalent counterpart q' that is in $placements(n, n)$ by sorting q by the x coordinate.

The second optimization improves the performance of the *noAttack* check. Instead of checking that no two queens share the same diagonal (distance between the two queens' x-values equals distance between the two queens' y-values), it checks that the sums and differences of all queen placements are distinct. We prove this optimization correct by formalizing the intuitive argument given in the Smten tutorial [Uhler 2014].

The tutorial uses the following tables to explain why the optimization is correct for a 4×4 chessboard:

y \ x	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

sum $(x + y)$

y \ x	0	1	2	3
0	0	1	2	3
1	−1	0	1	2
2	−2	−1	0	1
3	−3	−2	−1	0

difference $(x - y)$

The first table labels the cell at position x, y with the sum $x + y$, while the second table labels the cell x, y with the difference $x - y$. Observe that any two queens with a different sum of x, y are also on a different diagonal going from bottom-left to top-right. Similarly, any two queens with a different difference of x, y are also on a different diagonal going from top-left to bottom-right.

We can formally prove this optimization correct by showing that for all integers x_1, y_1, x_2, y_2, $|x_1 - x_2| = |y_1 - y_2|$ iff $x_1 + y_1 = x_2 + y_2$ or $x_1 - y_1 = x_2 - y_2$. The forward direction can be proven by casewise analysis on the absolute value expressions. The reverse direction can also be proven by casewise analysis on the disjunction. Thus for any two queens placed at (x_1, y_1) and (x_2, y_2), those queens are not on the same diagonal (that is, $|x_1 - x_2| \neq |y_1 - y_2|$) iff $x_1 + y_1 \neq x_2 + y_2$ and $x_1 - y_1 \neq x_2 + y_2$. This allows us to conclude that the optimized *noAttack* check indeed correctly enforces the rules for queens attacking.

Note that SpaceSearch enables us to verify the solver-aided tool that will eventually be executed, not just a model of an n-queens algorithm.

3 THE SPACESEARCH INTERFACE

The SpaceSearch interface exposes operations to construct and solve search problems in proof assistants. SpaceSearch bundles its operations by functionality into Abstract Data Types (ADTs) [Liskov and Zilles 1974]. In general, ADTs provide (1) operations for introducing values of some abstract type and (2) operations for eliminating values of that type. This section presents the *specifications* of SpaceSearch ADTs for constructing and solving search problems, including a machine-checkable denotational semantics (Section 3.1), extensions to support optimizations including parallelization and incrementalization (Section 3.2), and a discussion of design tradeoffs (Section 3.3). The next section details *implementations* of these ADTs.

3.1 Constructing Basic Search Problems

Basic ADT. SpaceSearch denotes (Figure 1) a search problem for solutions of type A to a subset of A. In the theorem prover, this subset is represented by an *ensemble*—a function that maps every value of type A to a proposition *Prop* (i.e. a logical claim). An ensemble s contains the value a if and

$$
\begin{array}{lcl}
\mathcal{P}(A) & := & A \to \mathit{Prop} \\
\emptyset & := & \lambda a.\, \mathit{False} \\
\{x\} & := & \lambda a.\, a = x \\
s \cup t & := & \lambda a.\, s(a) \vee t(a) \\
\bigcup_{a \in s} f(a) & := & \lambda b.\, \exists a.\, s(a) \wedge (f(a))(b)
\end{array}
$$

Fig. 3. Ensembles. An ensemble of A represents a subset of A as a function that maps every value of type A to a proposition *Prop*. An ensemble s contains the value a if and only if the proposition $s(a)$ is true.

$$
\begin{array}{l}
\mathit{bv} : \mathbb{N} \to \mathit{Type} \\
[\![\_]\!] : \mathit{bv}(n) \to \{m : \mathbb{N} \mid m < 2^n\} \\
\mathit{bvZero}_n : \mathit{bv}(n) \\
[\![\mathit{bvZero}_n]\!] = 0 \\
\ldots
\end{array}
\qquad
\begin{array}{l}
\mathit{Integer} : \mathit{Type} \\
[\![\_]\!] : \mathit{Integer} \to \mathbb{Z} \\
\mathit{intPlus} : \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
[\![\mathit{intPlus}(n, m)]\!] = [\![n]\!] + [\![m]\!] \\
\mathit{intFull} : \mathit{Space}(\mathit{Integer}) \\
[\![\mathit{intFull}]\!] = \lambda n.\, \mathit{True} \\
\ldots
\end{array}
$$

Fig. 4. SpaceSearch BitVector and Integer ADTs. The BitVector ADT provides the $\mathit{bv}(n)$ type for bit vectors of size n, as well as constants and operations on bit vectors. These operations are more efficient than the native bit vector operations provided by Coq. The Integer ADT provides the type of integers *Integer*. The *intFull* space contains infinitely many integers, and can therefore not be constructed using only the operations provided by the Basic ADT.

only if the proposition $s(a)$ is true, i.e., if $s(a)$ is a provable logical claim. This is summarized in Fig. 3.

Using ensembles, the empty set $\emptyset$ is the function that maps every element a in A to the false proposition *False*, the singleton $\{x\}$ is the function that maps every element a to the proposition that is only true if a is equal to x, the binary union $s \cup t$ is the function that maps every element a to the proposition that is only true if a is either contained in s or in t, and the infinitary union $\bigcup_{a \in s} f(a)$ is the function that maps every element b to the proposition that is only true if there exists a value a in s, such that b is in the ensemble returned by $f(a)$.

3.2 Constructing Advanced Search Problems

The Basic ADT is the least common denominator of all possible backend implementations of search problems. However, some extensions are required to increase the efficiency and expressiveness for specialized backends.

Specialized Search Problems. While the Basic ADT can be used to construct search problems for any of Coq's native types, such problems cannot always be solved efficiently. For example, we initially tried to build SaltShaker (Section 5) using Coq's native implementation of bit vectors. But we found that even simple space constructions, like the space of all 32-bit vectors equal to 5, cannot be searched efficiently directly within Coq (e.g., within a day). SpaceSearch therefore also exposes ADTs to construct *specialized* search spaces that certain solvers can search more efficiently.

Figure 4 describes SpaceSearch's *BitVector ADT*, which provides the $\mathit{bv}(n)$ type for bit vectors of size n, as well as constants and operations on bit vectors (not shown). Elements of type $\mathit{bv}(n)$ are

denoted to the natural numbers up to 2^n. Using the Rosette Backend, these bit vectors are extracted to the bit vector theory provided by the underlying SMT solver. The result is that the aforementioned space construction (of all 32-bit vectors equal to 5) can be searched in fractions of a second.

Infinite Search Problems. The Basic ADT is sufficient to construct full spaces of finite types, e.g., the full space of the *bool* type is: $union(single(true), single(false))$. But these basic operations cannot be used to construct infinite spaces, like the space of all integers. This is because *single* and *empty* are finite spaces, and *union* and *bind* map finite spaces to finite spaces. SpaceSearch thus provides additional ADTs to construct *infinite* search spaces.

Figure 4 describes SpaceSearch's *Integer ADT*, which provides the *Integer* type. Elements n of type *Integer* are denoted to the mathematical integers $\mathbb{Z}$ with $[\![n]\!]$ (this function has the same syntax as, but is different from, the function provided by the Basic ADT). The ADT also provides constants and operations on integers, such as *intPlus*, which are denoted to the corresponding constants and operations on the mathematical integers.

The most interesting value provided by the Integer ADT is the *intFull* space. This space contains every one of the infinitely many integers, and is thus denoted as the ensemble that returns the true proposition *True* for every integer. As we will see in Section 3.3, providing this space has implications on the solvability of search problems.

Callable Search Problems. During the design of the SpaceSearch library, we were confronted with the choice of either building SpaceSearch as a Domain Specific Language (DSL) or as a set of ADTs (as described above).

A library that provides its interface in the form of a DSL defines a language (usually as an Abstract Syntax Tree (AST)), and provides an interpreter for this language. For SpaceSearch, the DSL would be a language to construct search problems, whose AST $SpaceAST(A)$ would have the constructors:

$$
\begin{aligned}
&emptyAST_A : SpaceAST(A)\\
&singleAST_A : A \to SpaceAST(A)\\
&unionAST_A : SpaceAST(A) \to SpaceAST(A) \to SpaceAST(A)\\
&\ldots
\end{aligned}
$$

What differentiates the ADT approach from the DSL approach is that a user of the DSL interface knows exactly which operations can be used to construct search problems (because the user can pattern match on an AST), whereas a user of the ADT only knows that there are some operations to construct search problems (but there may be more).

To support extensibility with various optimizations like SMT theories, parallelization, and incrementalization, we opted to provide the SpaceSearch interface in the form of ADTs, where adding new operations is guaranteed not to break user code (e.g., extending an ADT cannot cause pattern matches to become incomplete).

A weakness of the SpaceSearch ADT interface is that it cannot be efficiently implemented directly in Coq. To see why, consider the expression $bind(s, f)$. No matter how the space s is implemented, an implementation of *search* in Coq will have to invoke f for every element in s (which can be very slow or impossible). This is because functions in Coq programs can only be applied to arguments, not inspected or optimized as would be possible with a DSL-based interface.

One way SpaceSearch mitigates this issue is with the use of extraction (i.e., pretty-printing) from Coq to Racket. Once extracted, the SpaceSearch interface can be efficiently implemented because the target language's interpreter can inspect a function's source code, and can thus provide an

$$
\begin{aligned}
&\mathit{Callable} : \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\
&\mathit{call}_{A,B} : \mathit{Callable}(A, B) \to (A \to B) \\
&\mathit{callableBind}_{A,B} : \mathit{Space}(A) \to \mathit{Callable}(A, \mathit{Space}(B)) \to \mathit{Space}(B) \\
&\mathit{callableBind}(s, r) = \mathit{bind}(s, \mathit{call}(r))
\end{aligned}
$$

Fig. 5. SpaceSearch Callable ADT. *callableBind* is a version of *bind* whose second argument is a value that can be called with *call*, but depending on the *Callable* type, may also support other operations, such as looking at the value's abstract syntax tree. These additional operations may be used to improve the performance of *callableBind* over normal *bind*.

$$
\begin{aligned}
&\mathit{minus}_A : \mathit{Space}(A) \to \mathit{Space}(A) \to \mathit{Space}(A) \\
&[\![\mathit{minus}(s, t)]\!] = [\![s]\!] \setminus [\![t]\!] \\
&\mathit{incSearch}(s, t, f) := \mathit{search}(\mathit{bind}(\mathit{minus}(s, t), f))
\end{aligned}
$$

Fig. 6. SpaceSearch Minus ADT and Incremental Search. $\mathit{minus}(s, t)$ subtracts solutions of t from s. This operation is non-monotonic, and can therefore not be implemented using the Basic ADT's operations. $\mathit{minus}(s, t)$ can be used to implement the incremental search function *incSearch*, which efficiently searches $\mathit{bind}(s, f)$ assuming that $[\![\mathit{bind}(t, f)]\!] = \emptyset$.

efficient implementation of *bind*. For example, if the interpreter recognizes that the function f's source code is equivalent to *single*, it can just replace $\mathit{bind}(s, f)$ with s.

Another way SpaceSearch enables the efficient implementation of *bind* is with *callableBind* (Fig. 5), a version of *bind* whose second argument is a value that can be called (by denoting it to a function). Depending on the *Callable* type, such operations can also support other operations, such as inspecting and optimizing the value's abstract syntax tree. The *callableBind* operation takes as input a search space s and a callable r, calls r on every solution in s, and returns the search problem containing all the solutions produced by calling r. A callable r of type $\mathit{Callable}(A, B)$ can be called using the *call* operation, which converts r to a function from A to B. The $\mathit{callableBind}(s, r)$ operation thus has the semantics of $\mathit{bind}(s, \mathit{call}(r))$.

Space Minus and Incremental Search. All of the operations in the aforementioned ADTs are monotonic: whenever the input spaces to these operations grow in size, the output size grows or stays the same. However, in some applications, it is useful to be able to reduce the size of a space. In particular, having a notion of subtracting spaces allows for performance gains by incrementalizing searches: if a previous call to *search* has established that a large space s is empty, and another space s' is constructed which overlaps substantially with s, then $\mathit{search}(\mathit{minus}(s', s))$ can be significantly faster than searching s' "from scratch" with $\mathit{search}(s')$.

Figure 6 describes the Minus ADT and the incremental search function $\mathit{incSearch}(s, t, f)$. This function returns all the solutions of $\mathit{bind}(s, f)$ assuming that $\mathit{bind}(t, f)$ has already been searched and has no solutions. As a result, $\mathit{incSearch}(s, t, f)$ avoids having to perform the bind on a portion of the space that is already known not to return a solution. *minus* can also be used for building search spaces, iteratively enumerating solutions, or subtracting out "uninteresting" parts of a larger search space whose counterexamples prove to be unhelpful.

The $\mathit{incSearch}(s, t, f)$ function is useful for applications that apply computationally expensive functions f to spaces that change often, but only in relatively few, easily isolated ways. For example,

$$
\begin{aligned}
&\mathit{preciseSearch}_A : \mathit{Space}(A) \to \mathit{option}(A)\\
&\mathit{preciseSearch}(s) = \mathit{None} \implies [\![s]\!] = \emptyset\\
&\mathit{preciseSearch}(s) = \mathit{Some}(a) \implies a \in [\![s]\!]\\
&\mathit{heuristicSearch}_A : \mathit{Space}(A) \to \mathit{option}(\mathit{option}(A))\\
&\mathit{heuristicSearch}(s) = \mathit{Some}(\mathit{None}) \implies [\![s]\!] = \emptyset\\
&\mathit{heuristicSearch}(s) = \mathit{Some}(\mathit{Some}(a)) \implies a \in [\![s]\!]\\
&\mathit{heuristicSearch}(s) = \mathit{None} \implies \mathit{True}
\end{aligned}
$$

Fig. 7. Search ADTs. *preciseSearch*(*s*) either returns *Some* solution in *s*, or *None* if no solution exists. *heuristicSearch*(*s*) may additionally simply fail (the *None* case), and won't decide whether *s* contains a solution or not. This is useful for undecidable search problems and for applying heuristic solvers.

SaltShaker (Section 5) applies a computationally expensive verification function to every element in a frequently changing set of instructions, where each change is relatively small.

3.3 Solving Search Problems

This section explains how to find solutions to search problems constructed using SpaceSearch ADTs. SpaceSearch's interface comes with ADTs to perform both precise and heuristic-based search. These ADTs are formalized in Fig. 7.

Precise Search. The *preciseSearch* operation is equivalent to the search described in Section 2.1, taking as input a search space *s* and returning either *None*, meaning *s* is empty, or *Some*(*a*), meaning that *a* is a solution to *s*. In the case of multiple solutions to *s*, the interface only specifies that one of them has to be returned, without specifying which one.

Unlike Smten, *preciseSearch* does not wrap search results in the IO monad. This enables the use of SpaceSearch in pure functional code, and simplifies reasoning in Coq. It is also safe because all SpaceSearch implementations (including Rosette) are pure (always returning the same solution for the same search problem), and support nested search queries. To also support impure and non-nesting implementations, SpaceSearch's interface could easily be extended with another ADT that wraps search results in the IO monad.

Heuristic Search. The *heuristicSearch* operation takes as input a search space *s* and returns either *Some*(*None*), which means that the search space *s* is empty; *Some*(*Some*(*a*)), which means that *a* is a solution to *s*; or *None*, which means that the *heuristicSearch* operation could not determine whether *s* contains a solution or not, e.g., due to timeout.

The *heuristicSearch* operation is important because some search problems are undecidable and thus lack a *preciseSearch* operation that always returns a result. For example, the *intFull* operation from the Integer ADT can be used to construct undecidable search problems (search problems constructed using only Basic ADT are, however, always decidable). One of the challenges of SpaceSearch is to statically prevent the use of *preciseSearch* on undecidable search problems. Section 4 shows how the separation of the SpaceSearch interface into multiple ADTs achieves this goal.

The *heuristicSearch* operation is not only useful for undecidable problems, but it can also be used to perform QuickCheck [Claessen and Hughes 2000] style testing. Such an implementation of $\mathit{heuristicSearch}_A(s)$ generates multiple elements of type *A*, and tests whether one of these elements is a solution to the search space *s*. If *a* is a solution, the operation returns *Some*(*Some*(*a*)). If none of the elements is a solution, it returns *None*.

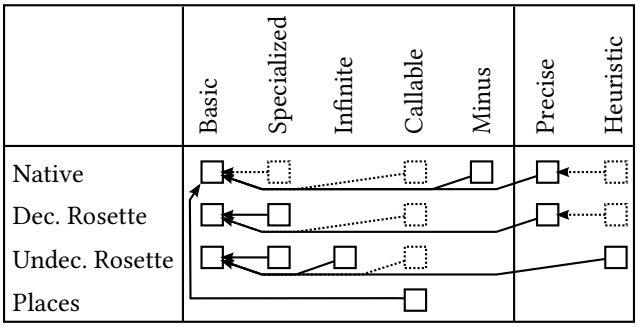

Fig. 8. SpaceSearch Backend ADT Instances. A box means that the row's backend provides an instance of the column's ADT. An arrow $i \leftarrow j$ means that j depends on i. A dotted instance is automatically derived from the instance it depends on (indicated by a dotted arrow).

$$
\begin{array}{lcl}
\mathit{Space}(A) & := & \mathit{list}(A) \\
[\![s]\!](a) & := & \mathit{in}(a, s) \\
\mathit{empty} & := & [\,] \\
\mathit{single}(x) & := & [\,x\,] \\
\mathit{union}(s, t) & := & \mathit{append}(s, t) \\
\mathit{bind}(s, f) & := & \mathit{flatten}(\mathit{map}(f, s)) \\
\\
\mathit{minus}(s, t) & := & \mathit{listMinus}(s, t)
\end{array}
\qquad
\begin{array}{lcl}
\mathit{preciseSearch}([\,]) & := & \mathit{None} \\
\mathit{preciseSearch}(a :: \_) & := & \mathit{Some}(a) \\
\mathit{heuristicSearch}(s) & := & \mathit{Some}(\mathit{preciseSearch}(s)) \\
\\
\\
\mathit{Callable}(A, B) & := & A \to B \\
\mathit{call}(r) & := & r \\
\mathit{callableBind}(s, r) & := & \mathit{bind}(s, r)
\end{array}
$$

Fig. 9. SpaceSearch Native Backend. Instantiates the *Space* type with the type of lists. *preciseSearch*(l) simply returns the first element in l. *heuristicSearch* just performs a *preciseSearch*, and will thus never return an imprecise result. The native *Callable*(A, B) is just a function. The Native Backend is useful for efficiently searching small problems, testing directly within Coq, and verifying that SpaceSearch's interface is not vacuous.

4 THE SPACESEARCH BACKENDS

SpaceSearch provides three backends that implement the various ADTs contained in the SpaceSearch interface. The Native Backend (Section 4.1) provides an inefficient, but provably correct, implementation of SpaceSearch directly in Coq (thus ensuring the consistency of the library interface); the Rosette Backend (Section 4.2) implements SpaceSearch interfaces using Rosette primitives on extraction to Racket; and the Places Backend (Section 4.3) implements SpaceSearch's Callable ADT using Distributed Places [Tew et al. 2014] on extraction to Racket. Figure 8 provides an overview of the ADT instances provided by each backend, as well as the dependencies between these ADTs instances.

This section describes contributions related to SpaceSearch's backends, including a translation from Smten's interface to Rosette's symbolic execution API (Section 4.2), an application of proof assistant extraction mechanisms to support solver-aided tool development (Section 4.2) without modifying the host language interpreter (as done in Haskell for Smten, and Racket for Rosette), and extensions to the Smten interface to support new optimizations including parallelization and incrementalization (Section 4.3).

$$
\begin{array}{lcl}
\mathit{empty} & \triangleq & \texttt{(lambda () (assert false))} \\
\mathit{single}(x) & \triangleq & \texttt{(lambda () x)} \\
\mathit{union}(s,t) & \triangleq & \texttt{(lambda () (if (symbolic-bool) (s) (t)))} \\
\mathit{bind}(s,f) & \triangleq & \texttt{(lambda () ((f (s))))}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{preciseSearch}(s) & \triangleq & \texttt{(solve (s))} \\
\mathit{heuristicSearch}(s) & \triangleq & \texttt{(solve (s))}
\end{array}
$$

Fig. 10. SpaceSearch Decidable and Undecidable Rosette Backends. Rosette extends Racket with symbolic values, assertions, and search for valid instantiations (instantiations that do not violate any assertions) of a symbolic value. Both Rosette Backends extract a search space s to a symbolic value thunk, whose only valid instantiations are solutions in s. The Decidable Rosette Backend implements those SpaceSearch ADTs that only allow the construction of decidable search problems, the Undecidable Rosette Backend instantiates all other ADTs as well.

4.1 Native Backend

The Native Backend (Fig. 9) instantiates the Basic ADT's *Space* type with the type of lists, and denotes a list to an ensemble using the *in* predicate. The $\mathit{in}(a, s)$ predicate is true iff a is an element of the list s. *empty* is then just the empty list, $\mathit{single}(x)$ is the singleton list of x, $\mathit{union}(s, t)$ concatenates the two lists s and t, and $\mathit{bind}(s, f)$ first maps f over every element in s, and then flattens the resulting list. Infinite ADTs are not supported; for example, the Native Backend does not provide an instance for the Integer ADT, because *intFull*—the list of all integers—does not exist.

The Native Backend instantiates the Precise ADT's *preciseSearch* operation by simply returning the first element in the list, if there is one. This implementation therefore depends on details of the Native Backend's Basic ADT implementation, namely that Spaces are lists. Space subtraction also depends on the fact that Spaces are lists in the Native Backend. The Native Backend's implementation of *minus* calls the $\mathit{listMinus}(l, l')$ function, which simply removes all elements in the list l' from the list l.

The Heuristic and Callable ADTs are implemented using existing operators from the Precise ADT and Basic ADT respectively. In the Heuristic ADT, *heuristicSearch* just performs a *preciseSearch*, and will thus never return an imprecise result (since all Basic ADT spaces are finite, the search is decidable). In the Callable ADT, $\mathit{Callable}(A, B)$ is implemented as an ordinary function $A \to B$, and *call* is thus just the identity function; using this definition of callable, *callableBind* is implemented as a direct invocation of *bind*. The Heuristic and Callable ADT instances do not depend on any implementation details of the Native Backend, and can thus be automatically derived from any Precise ADT and Basic ADT instance respectively.

We found the Native Backend useful for efficient search of small search problems, for testing a solver-aided tool directly within Coq, and for verifying that SpaceSearch's interface is not vacuous, i.e., it can be implemented and proven correct.

4.2 Rosette Backend

The Rosette Backends provide an efficient implementation of SpaceSearch's ADTs using the Rosette language [Torlak and Bodik 2013, 2014], which extends Racket with primitives for constructing solver-aided tools. A program using the Rosette Backends can be proven correct against the SpaceSearch interface in Coq, but can only be executed after extraction to Racket.

Rosette Background. Rosette [Torlak and Bodik 2013, 2014] extends Racket with the following solver-aided primitives: *symbolic values*, which are created using functions such as (`symbolic-bool`) and (`symbolic-integer`); *assertions*, which are created using the `assert` construct; and *solver queries*, which are made via the `solve` construct. The `solve`(e) query takes an expression e and tries to find a concrete assignment to any symbolic values in e such that no assertion in e is violated. If `solve` finds a valid concrete assignment c, it returns the expression e, where symbolic values have been replaced with concrete values from the assignment c.

For example, the following Rosette program checks De Morgan's law $\forall xy.x \wedge y \iff \neg(\neg x \vee \neg y)$ by checking that its negation is unsatisfiable:

```
(solve
  (let ((x (symbolic-bool)) (y (symbolic-bool)))
  (if (eq? (and x y) (not (or (not x) (not y))))
    (assert false)
    (cons x y))))
```

The `solve` query tries to find an assignment to x and y such that the if-condition evaluates to false to avoid the assertion failure. If `solve` found such an assignment (which it does not), it would return the tuple (`cons x y`) concretized with the values of the assignment (i.e., a counterexample). With this encoding, the verified property holds iff every execution of the program fails.

The `solve` query works by translating the input expression into an SMT-LIB [Barrett et al. 2016b] formula and solving it with an off-the-shelf SMT solver. For those familiar with SMT-LIB, the above program gets translated to:

```
(declare-const x Bool)
(declare-const y Bool)
(define-const a Bool (and x y))
(define-const b Bool (not (or (not x) (not y))))
(assert (not (and (=> a b) (=> b a))))
(check-sat)
```

Extraction to Rosette. SpaceSearch provides two Rosette Backends. The Decidable Rosette Backend implements those SpaceSearch ADTs that only allow the construction of decidable search problems (e.g., it does not implement the Integer ADT), and can therefore implement the Precise ADT. The Undecidable Rosette Backend also implements SpaceSearch ADTs that allow the construction of undecidable search problems, but can therefore only implement the Heuristic ADT.

The Rosette Backends implement SpaceSearch ADTs using *parameters*, which are extracted to the Racket terms described in Fig. 10. Coq's built-in extraction mechanism compiles Coq expressions to a target language, in our case Racket. Coq's extraction mechanism also supports the definition of extraction parameters—uninterpreted expressions that, at extraction time, are instantiated with an expression in the target language. In Fig. 10., $p \triangleq e$ indicates that the parameter p is extracted to the target language expression e.

To a first approximation, both Rosette Backends extract a search space to a symbolic value such that the valid instantiations of the symbolic value are equal to the solutions of the search space. In particular, *single*(x) evaluates to the concrete value x, i.e., a symbolic value with exactly one instantiation; *union*(s, t) evaluates to an symbolic value that, depending on the value of a symbolic boolean, is either the symbolic value s or the symbolic value t; *empty* evaluates to (`assert false`),

i.e., a symbolic value with no instantiations; and *bind*(s, f) evaluates to a call of the function f with the symbolic value s.

The *preciseSearch*(s) and *heuristicSearch*(s) operations both call the `solve` query, which returns an instantiation of the symbolic value s (if there is one), and therefore a solution to the search problem that s represents. Rosette's `solve` query is deterministic (in that it always returns the same result when invoked with the same arguments) when used with a deterministic SAT/SMT solver like Z3. The only difference between *preciseSearch*(s) and *heuristicSearch*(s) is that for *preciseSearch*(s), we know that it can never be called with an undecidable search problem, and thus that it will always either return a solution or indicate that s is empty.

Naive extraction to Rosette leads to problems with the evaluation order of symbolic values. For example, (`if (symbolic-bool) 42 (assert false)`) has the solution 42, while the following term has no solution, because the assertion is executed before the if statement:

```
(let ((x (assert false))) (if (symbolic-bool) 42 x))
```

The Rosette Backends addresses this issue by wrapping all symbolic values in functions that take no arguments (thunks); and evaluating these thunks in the appropriate order.

Specialized ADT Extraction. The implementation of the BitVector and Integer ADTs in Rosette is straightforward: the ADTs' constants and operations are parameterized and extracted to the appropriate Rosette constants and operations.

Using SpaceSearch's specialized ADTs is one way to build efficient solver-aided tools. Another way is to develop solver-aided tools using native Coq data types, and on extraction use another feature of Coq's extraction mechanism: the feature of literally replacing Coq definitions (not parameters) with arbitrary target language expressions. While this approach is arguably less principled, it also has two advantages. First, it enables the efficient use of existing frameworks with SpaceSearch (we used this feature in SaltShaker). Second, it simplifies reasoning because native Coq types can provide judgmental equalities, whereas ADTs can only provide propositional equalities. For example, the native integer $0 + n$ is judgementally equal to n, whereas the ADT Integer *intPlus*(*intZero*, n) is only propositionally equal to n. SpaceSearch supports both approaches, allowing developers to choose the approach best suited to their goals.

4.3 Distributed Places Backend

The Places Backend provides a parallel, distributed implementation of SpaceSearch's Callable ADT using the Distributed Places [Tew et al. 2014] Racket library.

Distributed Places Background. The Distributed Places library provides the `dynamic-place` operation that takes the name of a function, and runs the function identified by that name on some thread of some node of a cluster. The `dynamic-place` operation also allows communication between the thread and its caller, by creating a channel that is both passed to the function running on the cluster and returned to the caller of the operation. This communication channel can be used to send and receive messages using the `put` and `get` operations respectively.

Extraction to Places. The Places Backend implements the Callable ADT as described in Fig. 11. The Callable ADT does not implement the Basic ADT, but instead reuses the existing Native Basic ADT implementation via Coq's typeclasses. Every *Space*(A) is thus implemented as a *list*(A).

The *callableBind*(l, w) implementation first `spawns` a new thread with the worker w for every element (solution) a of the list (space) l, then `gets` the list of results generated by each thread, and finally flattens these result lists into one final result list. The `spawn`(w, a) operation runs the worker w on task a and returns the result. This is implemented by first spawning a thread with

$$
\begin{array}{lcl}
\mathit{Callable}(A, B) & := & \mathit{Worker}(A, B) \\
\mathit{call}(r) & := & \mathit{runWorker}(r) \\
\mathit{callableBind}(l, w) & \triangleq & \texttt{(bind (map (spawn w) l) get)}
\end{array}
$$

```
spawn := (lambda (w a)
  (let ((ch (dynamic-place (quote worker-place))))
    (put ch w) (put ch a) ch)))

worker-place := (lambda (ch)
  (put ch (runWorker (get ch) (get ch))))
```

Fig. 11. SpaceSearch Distributed Places Backend. The Distributed Places library function (`dynamic-place f`) runs the function, identified by the name f, on some thread of some node of a cluster. The Distributed Places Backend implements $\mathit{callableBind}(l, w)$ using `dynamic-place` to run the callable w in parallel and distributed, for each solution in l.

`dynamic-place`, where we use `quote` to pass the name of the `worker-place` function, and then sending the worker w and task a to that thread. Upon receipt, the `worker-place` function calls the `runWorker` function to run the worker w on the task a, and then sends the result of this call back to via the communication channel.

The Places Backend must also provide instances for the *Callable* type and *call* operation of the Callable ADT. Ideally, the Places Backend would have the same instantiation as the Native Backend: a *Callable* is a function. This would give users of the Backend maximal flexibility by running arbitrary functions with arbitrary inputs and outputs. However, such an implementation is not possible because each callable, its input, and its output are sent over a network, which means that these values have to be *serializable* (i.e., it must be possible to convert them to a list of bits), and functions are *not* serializable, both in Coq and in Racket.

It is, however, possible to serialize *statically defined functions*. These are functions that have a globally visible name at compile time; e.g., `worker-place` is a statically defined function, but `(lambda (x) x)` is not. The `dynamic-place` operation takes the name of a statically defined function, sends that name over the network, and runs the function with that name on the thread that it spawns. This means that users of `dynamic-place` can send arbitrary functions, as long as they are defined statically.

The Places Backend exposes this capability of the Places library (sending statically defined functions) in Coq as follows. The Places Backend instantiates *Callable* and *call* with two parameters, *Worker* and *runWorker* (which is a statically defined function) respectively, but the Places Backend does not specify how to extract them. The extraction is specified by the user of the backend. Specifically, a user can define and extract parameters representing elements of the *Worker* type, and then extract *runWorker* to an arbitrary function. For example, a user can specify

$$
\begin{array}{l}
\mathit{succ} : \mathit{Worker}(\mathit{natural}, \mathit{list}(\mathit{natural})) \triangleq \texttt{(quote succ)} \\
\mathit{sqrt} : \mathit{Worker}(\mathit{integer}, \mathit{list}(\mathit{double})) \triangleq \texttt{(quote sqrt)} \\
\mathit{runWorker} \triangleq \texttt{(lambda (w a) (cond} \\
\quad \texttt{((eq? w (quote succ)) (list (+ 1 a)))} \\
\quad \texttt{((eq? w (quote sqrt)) (list (sqrt a) (- (sqrt a))))))},
\end{array}
$$

```
rocksalt : (i : instr) → bv(rocksaltOracleBits(i)) → state → state
rocksaltOracleBits : instr → ℕ
spec : (i : instr) → bv(specOracleBits(i)) → state → state
specOracleBits : instr → ℕ

saltShaker(i : instr) : option(state × specOracleBits(i)) :=
  preciseSearch(
    bind(stateFull, (λs : state.
      bind(bvFull(specOracleBits(i)), (λo_s : bv(specOracleBits(i)).
        if exists(bvFull(5), λo_r : bv(5).
              spec(i, o_s, s) == rocksalt(i, extend(o_r), s))
        then empty else single(s, o_s))))))

state := {
  eax : bv(32); ecx : bv(32); edx : bv(32); ebx : bv(32);
  esp : bv(32); ebp : bv(32); esi : bv(32); edi : bv(32);
  cf : bv(1); pf : bv(1); zf : bv(1); sf : bv(1); of : bv(1)
}

stateFull : Space(state) :=
  bind(bvFull(32), (λeax′. bind(bvFull(32), (λebx′. ...
    single({eax := eax′; ebx := ebx′; ...})))))
```

Fig. 12. SaltShaker. Given instruction i, *saltShaker*(i) searches for an inconsistency, that is a machine state s and specification oracle o_s (with *specOracleBits*(i) bits) such that there is no RockSalt oracle o_r (with 5 bits, as a sound efficiency optimization) for which the specification *spec*(i, o, s_s) and RockSalt *rocksalt*(i, o, s_r) are equal.

where *callableBind*$([1, 4, 9], sqrt)$ evaluates to the list $[-1, 1, -2, 2, -3, 3]$ of the square roots of $[1, 4, 9]$, and *callableBind*$(\ [1, 2, 3], succ)$ evaluates to the list $[2, 3, 4]$ of the successors of $[1, 2, 3]$.

A user of the Places Backend must ensure serializability. The *Worker* type in the Callable ADT makes this easy—serializability is ensured when for all A and B: all values of *Worker*(A, B) are serializable, and *Worker*(A, B) is only inhabited if all values of A and B are serializable.

The *callableBind*(s, r) operation can be more efficient than a normal *bind* whenever s has a medium-sized number of solutions that are easily enumerable, and the callable r performs an expensive computation.

5 SALTSHAKER: VERIFYING X86 SEMANTICS

5.1 SaltShaker Overview

RockSalt [Morrisett et al. 2012] is a formal checker for the safety of Native Client [Yee et al. 2009] code, a sandbox developed by Google. Part of this checker is a specification of a subset of the x86 instruction set. Since Rocksalt is developed in Coq, it has a relatively small trusted code base, but

(among other things) it relies on the correctness of its x86 semantics. We developed SaltShaker, which checks that the RockSalt semantics, for a given set of instructions, is sound with respect to another x86 specification. This case study follows the same methodology as the overview: 1) We formally express the tool's specification, using a proof assistant. 2) We implement an optimized solver-aided tool, using the operations of SpaceSearch. 3) We prove that the solver-aided tool meets its specification (thus also ensuring its optimizations are correct), using the denotational semantics of SpaceSearch.

1) SaltShaker Specification. SaltShaker checks that the RockSalt semantics of a given instruction i is sound, where soundness is defined as follows: any property P provable in RockSalt about the output of i also holds for the output of any x86-specification-compliant CPU running i. The RockSalt semantics is modeled as $rocksalt(i, o, s)$ (see Fig. 12) which, for a given instruction i, oracle o, and input machine state s, returns a new state that captures the result of executing the instruction i. $bvFull(n)$ is the space of all bit vectors of size n.

The oracle o provides $rocksaltOracleBits(i)$ many bits that determine i's *non-deterministic behavior*. For example, i's semantics can use the oracle to store arbitrary bits for particular machine-state locations, or to non-deterministically follow both sides of a branch. This non-deterministic behavior is used by the RockSalt semantics 1) to model *undefined* behavior; and 2) to *over-approximate* behavior of an instruction if providing a precise semantics is difficult or not needed.

To check that the output of i also holds for the output of any x86-specification-compliant CPU running i, SaltShaker requires a formalization of the official Intel x86 specification [Intel 2015], which is a 3,800-page document using English and informal pseudo-code. SaltShaker assumes this formalization is provided as a function *spec*, and a function *specOracleBits* with similar meaning as the RockSalt equivalent. While the *spec* models undefined behavior, it must *not* over-approximate instructions.

Formally, given an instruction i, if SaltShaker returns *None*, that implies that there is no counterexample, and that RockSalt is sound with respect to another x86 specification. Namely, for any $cpu : instr \to state \to state$, that is specification-compliant ($\exists o, \forall s, cpu(i, s) = spec(i, o, s)$), the following proposition holds:

$$\forall s\ P.\ (\forall o.\ P(rocksalt(i, o, s))) \to P(cpu(i, s))$$

2) SaltShaker Solver-Aided Tool. Since the soundness property quantifies over the proposition P, it is higher-order and thus cannot be directly encoded in the logic of a first-order solver like Z3. Instead, SaltShaker uses SpaceSearch to search for a start state s and specification oracle o_s such that there is no RockSalt oracle o_r for which the specification $spec(i, o, s_s)$ and RockSalt $rocksalt(i, o, s_r)$ are equal. Figure 12 provides all definitions and shows the implementation of SaltShaker.

To make this check practical, SaltShaker makes the following (sound) approximation. Instead of existentially quantifying over the space of all RockSalt oracles, SaltShaker searches over oracles that only provide at most five non-deterministic bits, and extends that oracle to the full size using 0s. This choice is useful because most often only the flag registers are undefined (and there are only five flags in our machine state). Furthermore, the space of all oracles of this kind is finite and small (2^5 to be precise), and therefore the existential quantifier can be replaced by a disjunction.

The RockSalt semantics $rocksalt(i, o, s_r)$ is a program written in Coq. The extraction, used by SpaceSearch, desugars all Coq language constructs to a small core language (whose computational aspects are only: functions, function applications, variables, definitions, and pattern matching). It requires little engineering effort to translate the core language to Rosette (this functionality is privided by SpaceSearch). For example, Rocksalt's entry point definition in Coq

```
Definition rocksalt (i:instr) (o:oracle) (s:state) : state :=
  RTL_step_list (instr_to_rtl i) (rtl_state o s)
```

is translated to the following Rosette definition:

```
(define rocksalt (lambda (i o s)
  (RTL_step_list (instr_to_rtl i) (rtl_state o s)))
```

SaltShaker's *spec* and *specOracleBits* are parameterized, and on extraction to Rosette, instantiated with the x86 semantics defined by STOKE [Schkufza et al. 2013], a stochastic super-optimizer that uses SMT solvers to prove the equivalence of optimizations on x86 programs. The specification used in STOKE was largely automatically learned [Heule et al. 2016].

STOKE's instruction semantics are expressed in a DSL, which required little engineering effort to translate to Rosette (we implemented a small custom extension to STOKE that pretty prints instruction semantics to Rosette). For example, STOKE's `notl %edx` instruction semantics

```
rdx := 0x032 ∘ !rdx[31:0]
```

is translated to the following Rosette definition:

```
(define (notl s)
  (define new-rdx (concat (bv 0 32) (bvnot (extract 31 0 (state-rdx s)))))
  (update-state-rdx s new-rax))
```

3) SaltShaker Correctness. SaltShaker is incomplete but sound: whenever SaltShaker returns *None*, then for any start state and specification oracle, there is a RockSalt oracle such that the specification and RockSalt are equal, which implies the soundness property. We have proven this formally in the Coq proof assistant. Whenever SaltShaker returns $Some((s, o))$, the tuple (s, o) may be a counterexample to RockSalt's soundness.

5.2 SaltShaker Evaluation

Our evaluation of SpaceSearch seeks to answer the following research questions:

- **Q1**: Can SpaceSearch be used to build and verify solver-aided tools that are both efficient and effective?
- **Q2**: Can SpaceSearch be used with unchanged, existing Coq developments?
- **Q3**: Are specialized SMT data-structures more efficient than native Coq data structures?
- **Q4**: Does incrementalizing the search improve performance over repeated searches?

SaltShaker's *spec* and *specOracleBits* is parameterized, and on extraction to Rosette, instantiated with the x86 semantics defined by STOKE [Schkufza et al. 2013], a stochastic super-optimizer that uses SMT solvers to prove the equivalence of optimizations on x86 programs. The specification used in STOKE was largely automatically learned [Heule et al. 2016]. To translate STOKE's semantics to Rosette, we implemented a small custom extension to STOKE that pretty prints instruction semantics to Rosette.

RockSalt and STOKE support slightly different subsets of the machine state, and thus in SaltShaker, *state* consists of the common subset. In particular, this includes the eight 32-bit general purpose registers (*eax*, *ecx*, *edx*, . . .), as well as five 1-bit flags: *cf* (carry), *pf* (parity), *zf* (zero), *sf* (sign), and *of* (overflow). STOKE provides a semantics for x86-64, the 64-bit extension of x86, whereas Rocksalt only supports 32 bits. Because x86-64 is largely backwards compatible, it is sufficient to map the parts of the machine state common to both architectures. However, a mapping is more difficult for memory, as addresses are 32 and 64 bits respectively. At the moment, memory is not part of our machine state.

In x86, every opcode (e.g., `add`) gives rise to many different instruction variants, for different operand sizes (8, 16, or 32 bits) and operand types (constant or register), and every instruction variant can be instantiated with different concrete operands (e.g., `add eax, 8` or `add al, bl`). These are called instruction instantiations, or just instructions, for short.

SaltShaker proves the soundness of Rocksalt's semantics for a given x86 instruction instantiation (e.g. `add eax, 8`), by showing the equivalence between the instruction's Rocksalt semantics and STOKE semantics for all possible machine states. However, because there are a large number of instruction instantiations (e.g. there are 2^{64} variants of `add`'s 64-bit immediate alone), it is infeasible to use SaltShaker to prove the soundness of Rocksalt's semantics for all instruction instantiations.

Instead, we ran SaltShaker on all 40 opcodes that are supported by both RockSalt and STOKE. We tested 15,255 different instruction instantiations, using both random operands (random registers or random constants) as well as constants from a fixed set of "interesting" bit patterns (e.g., 0, -1, 2^n for various n, etc.). Because opcodes behave similarly for all their different instantiation, this provides high confidence that the RockSalt semantics is in fact sound for all instruction instantiations. If SaltShaker is run with the aim of gaining a full soundness guarantee for a particular program, i.e., any property proven about the output of the program using the RockSalt semantics will hold for any execution of the program on any x86-specification-compliant CPU, it suffices to prove the soundness of Rocksalt's semantics only for those instructions actually used in the program to be verified (e.g. the Chrome browser consists of about 2 million distinct instructions).

Q1, Efficiency & Effectiveness: Using a single computer with an Intel i7-4790K CPU and 32 GiB of memory, verifying our 15,255 instructions took 1.8h (0.43s per instruction), and initially 72.7% percent of instructions showed at least 1 bit where the semantics of RockSalt and STOKE differ. We investigated the differences, consulted the manual to determine the correct behavior, and reported all issues to the developers of RockSalt and STOKE. After working with the developers, we were able to trace all of the inconsistencies to 7 root-causes in RockSalt and 1 root-cause in STOKE[2]. All of these have been fixed since we reported them.

Specifically, RockSalt (1) computed wrong results and flags due to using a location that had already been overwritten (several instructions affected); (2) incorrectly computed on 32-bit values for 16-bit versions of `bsf` and `bsr`; (3) used the wrong bits to compute the parity flag (for all instructions with a parity flag); (3) computed wrong flags for addition/subtraction with carry/borrow; (4) computed wrong flags for comparison, addition, and subtraction; (5) computed wrong flags for multiplication; and (7) computed wrong flags for `shld` and `shrd`.

Despite these errors, the implementation of NaCL that was verified using the RockSalt semantics [Morrisett et al. 2012] is likely correct, because the bugs that we found were mostly in the computation of flags, or in the case of defect 1 above, were introduced in a refactoring after the release of the verified NaCL implementation.

STOKE computed an incorrect result for the `rcr` instruction due to a bug in STOKE's pretty-printer, which is used by SaltShaker. The bug cannot be triggered when using STOKE directly to reason about programs.

SaltShaker reported a false positive on 57 instruction instantiations that use the RockSalt oracle to non-deterministically set the instruction's 32-bit results (whereas the flag oracle we use only allows for at most 5 non-deterministic choices). SaltShaker also found 113 instruction instantiations (1 opcode) where the STOKE semantics is over-approximating (i.e., leaves an output unspecified even though the official Intel semantics does specify its behavior).

[2]We group failures by the root cause — the conceptually distinct underlying defect in the source code of the semantics. If a single function computes the parity flag for all instructions with a parity flag, then we consider this a single root cause defect.

SaltShaker depends on RockSalt, STOKE, and about 500 LOC to bridge gaps between the two. Above that, SaltShaker's implementation is about 70 LOC, and the specification is about 30 LOC. The proof of correctness is about 50 LOC. Note that the RockSalt semantics "cancel out" (a la Appel's VST [Appel 2011]) when verifying an actual program.

Q2, Legacy Coq Code: In building SaltShaker, we made only slight modifications to RockSalt (20 LOC). Specifically, we replaced the bit vector extraction to OCaml with an extraction to Rosette (5 LOC), extracted a frequently-used combination of bit vector operations to a more efficient implementation (6 LOC), and rewrote a function that was inefficient due to Racket's call by value semantics (9 LOC).

This compatibility with existing Coq frameworks is one of the strengths of SpaceSearch over low-level solver interfaces. The *bind* operator enables this compatibility. Once we constructed the space of all machine states s, we were able to call *bind* on s, and then just write a function for checking that RockSalt's x86 interpreter is equivalent to the specification for a *concrete* machine state.

In fact, SaltShaker can even bind over some parts of an instruction (e.g., all possible values of an immediate operand) and can thus verify billions of instruction instantiations in seconds. We did not use this feature in our evaluation, however, because STOKE, a tool that was developed with an SMT solver in mind, only provides semantics for *concrete* instruction instantiations over *symbolic* machine state.

Q3, Specialized SMT Datastructures: We initially tried to verify SaltShaker using Coq's native implementation of bit vectors. While the space of all bit vectors is efficiently searchable, even simple space constructions, like the space of all 32-bit vectors that are equal to 5, cannot be searched efficiently. SpaceSearch's support of SMT data-structures is thus crucial for the construction of efficient solver-aided tools.

Q4, Incrementalization: SaltShaker cannot feasibly check all x86 instruction instantiations (the space of 32-bit immediates alone is already too large). Instead, we recommend a user of SaltShaker only check those instruction instantiations that are actually used in the program that is being verified against Rocksalt, rerunning SaltShaker whenever the set of used instructions changes.

To improve the performance of this workflow, we provide a version of SaltShaker that takes as input a list (space) of instructions t that have already been checked and a list (space) of instructions s that need to be checked. SaltShaker then incrementally checks the instructions in s with:

$$incSearch(s, t, \lambda i.optionToSpace(saltShaker(i)))$$

Fig. 13 shows the results of the following experiment: Given a test set of 15,255 instructions, we split it into 15 partitions of 1,017 instructions each. We compare the time to run a monolithic search over partitions 1 through n, to run an incremental search over partitions 1 through n assuming that 1 through $n-1$ have already been searched, and to run *saltShaker* on each individual instruction in partition n only.

The results of this experiment show that incremental search is much faster than monolithic search. However, incremental search incurs a slight overhead compared to running the verification function on only the new instructions. This overhead is caused by the (currently) quadratic algorithm used to subtract the spaces.

6 BAGPIPE: VERIFYING BGP CONFIGURATIONS

6.1 Bagpipe Overview

To watch a video, send an email, or check the news on the Internet, you have to communicate with a server that is potentially on the other side of the world. This is hard, because the Internet is not a single coherent network, but rather made up of many smaller, interconnected but autonomous

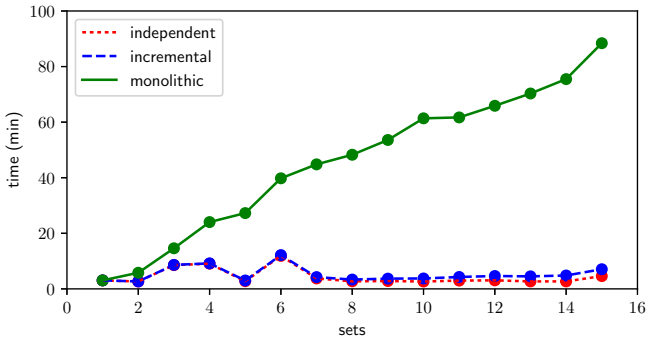

Fig. 13. Performance gains of incrementalizing SaltShaker. Given 15 disjoint sets of instructions and an x-coordinate, we compare the time to run a monolithic search over sets 1 through x, to run an incremental search over sets 1 through x assuming that 1 through $x-1$ have already been searched, and to run an independent search on set x only.

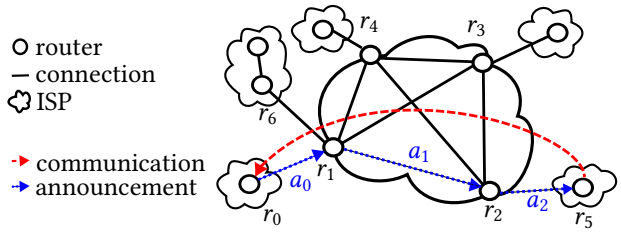

Fig. 14. BGP Background.

networks operated by Internet Service Providers (ISPs) like Comcast, MIT, and IBM. Internet communication requires that each ISP notify all other ISPs of the destinations (like the video/email/news server) that it can communicate with either directly, or indirectly through another ISP. ISPs do so by *sending* route announcements via the Border Gateway Protocol (BGP). Once an ISP has *received* an announcement with route r to a destination d, it can *use* r to forward traffic to d.

For example, Fig. 14 shows the relevant announcements for the router r_5 of one ISP to establish communication with r_0 of another ISP.

(1) r_0 *sends* an announcement a_0 to notify its neighbors that it can reach router r_0 directly.
(2) r_1 *receives* a_0 and *sends* a_1 to notify its neighbors that it can reach r_0 via r_0.
(3) r_2 *receives* a_1 and *sends* a_2 to notify its neighbors that it can reach r_0 via r_1.
(4) r_5 *receives* a_2 and can *use* it to reach r_0 via r_2.

To ensure reliable, secure, and profitable communication, ISPs must configure their BGP routers to restrict how route announcements can be *used*, *received*, and *sent*. For example, the router r_1 may be configured to never forward announcements to r_6, because the resulting routes wouldn't be profitable for r_1's ISP, or because the resulting routes to r_4 may be insecure. In practice, network operators write these configurations in imperative, loop-free, vendor-specific configuration languages which provide rules for how a router should process incoming announcements. An example snippet from a Cisco router configuration which tests and sets various announcement flags (here med and communities) is shown on the left of Figure 15.

```
route-policy set-community-attrs
  set med +100
  if (destination in westbound-out) then
    set community (low-priority, no-forward) additive
  elseif (community matches-any (cdn-link, dod)) then
    set community (hi-priority) additive
  else
    drop
  endif
end-policy
```

```
(implies
    (external? receiver)
   (not (has-community? 'confidential sent)))
```

Fig. 15. *Example BGP Configuration and Policy.* The example Cisco configuration (left, adapted and anonymized from a regional ISP) tests various flags set on an incoming BGP announcement and either sets additional flags or drops the announcement. The example ISP-wide Bagpipe policy (right) specifies that announcements received by the ISP with the BLOCK flag set should never be forwarded outside the ISP (i.e. *sent*), however the ISP is free to *use* such announcements for routing traffic itself.

Because ISPs have broad flexibility to configure their routers, BGP provides very few general guarantees—essentially all desirable properties have to be proven for a particular set of router configurations. This is challenging because all routers in an ISP interact concurrently by forwarding messages to one another according to their low-level configuration rules, making it difficult to predict ISP-wide behavior.

Bagpipe is a solver-aided tool, written in Rosette, that enables ISPs to specify desirable properties of the ISP's BGP network, and automatically checks them for a given set of router configurations written in Cisco or Juniper configuration languages. This section describes our SpaceSearch-based, verified revision to the Bagpipe [Weitz et al. 2016] prototype. This case study follows the same methodology as the overview: 1) We formally express the tool's specification, using a proof assistant. 2) We implement a solver-aided tool, using the operations of SpaceSearch. 3) We prove that the solver-aided tool meets its specification, using the denotational semantics of SpaceSearch.

1) Bagpipe Specification. Bagpipe provides a specification language to express desirable safety properties of a *single* ISP's BGP network. These specifications are written as assertion on the announcements *used* by an ISP for routing, as well as assertions on announcements *sent* and (internally) *received* by the ISP. A specification may for example state that "the ISP will never *send* announcements tagged as confidential", "the ISP will never *use* an announcement for invalid destinations like *localhost*", or "the ISP will never *use* an announcement, if a more profitable announcement is also available". Shown on the right of Figure 15 is the first policy, expressed in Bagpipe's specification language. Liveness properties, such as "the ISP will eventually send an *announcement*", cannot be checked by Bagpipe. Bagpipe's specification language is detailed in the original Bagpipe paper [Weitz et al. 2016].

Bagpipe assumes that the ISP's neighbors may send any announcements, and checks that an ISP's router configurations c correctly implement a desirable property s written in Bagpipe specification language. This means that s must hold for all possible announcements that could ever be used, sent, or received, by the ISP; even if these announcements are concurrently forwarded along multiple paths through the ISP's network.

2) Bagpipe Solver-Aided Tool. Figure 16 describes Bagpipe's checking algorithm, *verifyISP*.[3] Given a desirable property s and a set of router configurations c, the *verifyISP* function checks, using the *check* function, that any announcement a forwarded along any path p through the ISP's

[3]The presentation of this algorithm is simplified. Check the original paper for more details.

```
verifyISP(c : Config, s : Property) : option(Path × Announcement) :=
  preciseSearch(callableBind(
    bind(fullPath(c), λp.single((c, s, p))),
    verifyPathWorker))

verifyPathWorker : Worker(Config × Property × Path, Space(Path × Announcement)) ≜
  (quote verifyPathWorker)

runWorker ≜ (lambda (_ csp) (verifyPath csp)))

optionToSpace(Some(a)) := single(a)
optionToSpace(None) := empty

verifyPath((c, s, p)) : Space(Path × Announcement) :=
  optionToSpace(preciseSearch(
    bind(fullAnnouncement, (λa : Announcement.
      if check(c, s, p, a) then empty else single((t, a))))))
```

Fig. 16. Bagpipe. Given a desirable property s and a set of router configurations c, the *verifyISP* function checks that any announcement a forwarded along any path p through the network satisfies the desirable property s (where the space of paths *fullPath* is derived from the router configurations c).

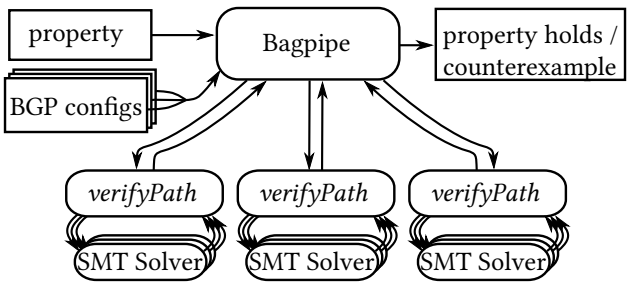

Fig. 17. Bagpipe Workflow.

network satisfies s.[4] The space of paths *fullPath* is derived from the router configurations c. Each path is and starts at an external router, then traverses internal routers, and finally exits via an external router. For example, in Fig. 14, the loop-free paths through the center ISP's network are $[r_0, r_1, r_2, r_5], [r_0, r_1, r_6], \ldots, [r_5, r_2, r_1, r_6], \ldots$.

Bagpipe's algorithm uses the Distributed Places Backend to check the desirable property in parallel, for the set of all paths. To do so, *verifyISP* enumerates all paths, and binds each path p (along with the configuration c and desirable property s) to the *verifyPath* function (which is called indirectly via the *Worker*/*runWorker* mechanism). The *verifyPath* function in turn uses the Rosette Backend to check the desired property symbolically for all announcements a. This workflow is illustrated in Fig. 17.

[4]The BGP specification (RFC 4271) ensures such paths are loop-free.

3) Bagpipe Correctness. Bagpipe's algorithm is subtle, as it relies on the *initial network reduction* to only check that a single announcement forwarded along a single path satisfies the desired property, instead of checking that multiple announcements forwarded along multiple paths concurrently also satisfy the desired property.

The pen-and-paper initial network reduction described in [Weitz et al. 2016] argues that Bagpipe's behavior is sound. Using the Coq proof assistant, we have formally verified Bagpipe's soundness, using a formalization of the initial network reduction.

6.2 Bagpipe Evaluation

In this evaluation of SpaceSearch, we wanted to answer the following research questions:

- **Q1**: Can SpaceSearch be used to build and verify solver-aided tools that are both efficient and effective?
- **Q2**: Can SpaceSearch's parallelization API improve solver-aided tool performance?
- **Q3**: Can verification with SpaceSearch find bugs in existing solver-aided tools?

Just like in the original Bagpipe paper [Weitz et al. 2016], we ran an experiment which checked desirable properties for three ISPs: the nation-wide ISP Internet2, the regional ISP BelWü, and the local ISP Selfnet. Their configurations total over 240,000 lines of Cisco and Juniper code. Bagpipe ran this experiment on Amazon EC2 with 2 instances of type `c3.8xlarge`, each with 32 virtual-cores and 60 GB of memory.

Q1: Bagpipe ran the experiment in a total of 82h, the cost for which is about $30 using EC2 spot instances. During the verification, Bagpipe found 19 cases where the configurations did not implement a desirable property, verified 4 desirable properties, and issued no false positives. This is the same as in the original paper.

Bagpipe's proofs and implementation are about 3,000 LOC, of which about 500 LOC are specification (this includes the full semantics of BGP). We are unable to cleanly split the lines between proof and implementation, as Bagpipe is developed using intrinsic verification which blurs the distinction.

Q2: Bagpipe can check the desired property for each path independently, which means that, apart from a short startup period, Bagpipe's algorithm is embarrassingly parallel. With over 1,000,000 paths to check, parallelization over paths thus improved performance roughly by the number of CPU cores used during the evaluation.

Q3: The unverified Bagpipe prototype did *not* correctly reduce domain-level properties (i.e., ISP BGP policies) to solver queries. Our verification of the Bagpipe prototype identified two bugs: (1) Bagpipe did not correctly verify policy specifications in the presence of withdrawn route announcements. (2) Bagpipe also created incorrect paths in which a router r appeared twice if an update messaged entered and exited the ISP at r without being forwarded within the ISP. These bugs did not seem to lead to user-visible failures, such as missed alarms or false alarms when running Bagpipe over real BGP configurations. Using a fixed version, we completed a formal verification of Bagpipe's soundness, thus establishing the correctness of the initial network reduction Bagpipe depends on.

7 RELATED WORK

Solver-aided tools. Advances in solver technology, including SAT, SMT, and model-finding, have made solver-aided tools a compelling option in many domains; here, we briefly mention a handful of representative examples. Boogie [Leino 2008] and related tools [Lahiri et al. 2009; Leino 2010] enable general-purpose verification by compiling verification conditions to SMT queries. Alive [Lopes et al. 2015] and PEC [Kundu et al. 2009] verify compiler optimizations using a solver back end.

Batfish [Fogel et al. 2015] verifies data plane properties using a Datalog solver; Vericon [Ball et al. 2014] verifies policies for software-defined networking controllers using an SMT solver. All of these tools reduce queries in some application domain to queries answerable by an automated solver. But none of these tools come with mechanically-checked proofs that this reduction is sound.

Solver-Aided Domain Specific Languages. Solver-aided host languages like Smten [Uhler and Dave 2014] and Rosette [Torlak and Bodik 2013, 2014] provide a higher-level interface to underlying solvers, and automatically optimize the orchestration and construction of solver queries. The higher-level interface often leads to less developer effort and order-of-magnitude improvements in code size, while maintaining the performance of hand-crafted solver-aided tools [Uhler and Dave 2014].

SpaceSearch itself can be viewed as a solver-aided language, whose interface is inspired by Smten, and which executes solver-aided tools efficiently by extracting them to Rosette. But SpaceSearch extends the state of the art in three ways. First, by exposing its interface in a proof assistant, along with a formal semantics, solver-aided tool developers can verify their optimizations and domain reductions. Second, SpaceSearch extends automatic solver orchestration with parallelization and incrementalization. Finally, by splitting its interfaces across various ADTs, SpaceSearch is easily extensible with advanced solver features and provides static guarantees about a solver's timeout behavior.

Integration of Proof Assistants and Solvers. Various SAT solvers have been built and verified in proof assistants like Coq and Isabelle [Lescuyer and Conchon 2009; Marić 2010; Oe et al. 2012], and they provide an interface against which solver-aided tools can be verified. But verified solvers are currently too slow to be used in most solver-aided tools, and they do not provide the additional features of SMT solvers. Witness checkers [Armand et al. 2011] alleviate these performance problems by checking the correctness of each individual SMT solver result, instead of verifying the entire solver. However, even a witness-checked solver still provides a low-level solver interface, and is thus harder to use than the interface provided by solver-aided languages. Both verified and witness-checked solvers can be used as drop-in replacements for the solver invoked by SpaceSearch.

8 CONCLUSION

This paper presented SpaceSearch, a library that provides a high-level interface for building and formally verifying solver-aided tools within a proof assistant. In essence, SpaceSearch is a solver-aided host language for proof assistants. A user builds their solver-aided tool in Coq against the SpaceSearch interface, and the user then verifies that the results provided by the interface are sufficient to establish the tool's desired high-level properties. Once verified, the tool can be extracted to several backends, including a backend in the Rosette solver-aided language that instantiates the SpaceSearch interface with calls to the Z3 SMT solver. Through its backends, SpaceSearch combines the strong correctness guarantees of a proof assistant with the high performance of modern SMT solvers.

Our evaluation on two solver-aided tools, SaltShaker and Bagpipe, showed that SpaceSearch can be used to build and verify solver-aided tools that are both efficient and effective. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE in under 2h. Bagpipe scales as well as its unverified hand-crafted predecessor, checking industrial configurations spanning over 240 KLOC and identifying 19 configuration inconsistencies with no false positives. Verifying Bagpipe also uncovered two previously unknown bugs in an earlier prototype version of Bagpipe. We found that SpaceSearch can be used with existing Coq developments; that SpaceSearch's SMT data-structures are more efficient than native Coq data-structures; and that SpaceSearch's incrementalization and

parallelization improve performance. These results show that SpaceSearch is a practical approach to developing efficient, verified solver-aided tools.

ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107, by the United States Air Force under Contract No. FA8750-15-C-0010, and by Air Force Research Laboratory and DARPA under agreement number FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was additionally supported by a generous gift from Google.

REFERENCES

Andrew W Appel. 2011. Verified software toolchain. In *ESOP*. Springer, 1–17.

Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses.* Springer Berlin Heidelberg, Berlin, Heidelberg, 135–150. https://doi.org/10.1007/978-3-642-25379-9_12

Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *PLDI*. ACM, 282–293. https://doi.org/10.1145/2594291.2594317

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016a. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`. (2016).

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016b. The Satisfiability Modulo Theories Library (SMT-LIB). www.smt-lib.org. (2016).

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. Montreal, Canada, 268–279. https://dl.acm.org/citation.cfm?id=1988046

Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *NSDI*. USENIX Association, Oakland, CA. https://dl.acm.org/citation.cfm?id=2789803

Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *PLDI*. ACM. https://doi.org/10.1145/2908080.2908121

Intel. 2015. Intel 64 and IA-32 Architectures Software Developer Manuals, Revision 325462-057US. (Dec. 2015). http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 377–394. https://doi.org/10.1007/978-3-319-21668-3_22

Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM.

Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. 2009. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In *CAV*. Springer Verlag. http://research.microsoft.com/apps/pubs/default.aspx?id=80360

K. Rustan M. Leino. 2008. *This is Boogie 2.* Technical Report. Microsoft Research. http://research-srv.microsoft.com/en-us/um/people/leino/papers/krml178.pdf

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LPAR'10)*. Springer-Verlag, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

Stéphane Lescuyer and Sylvain Conchon. 2009. *Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme.* Springer Berlin Heidelberg, Berlin, Heidelberg, 287–303. https://doi.org/10.1007/978-3-642-04222-5_18

Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based Verification of GPU Kernel Functions. In *FSE (FSE'10)*. ACM, 187–196. https://doi.org/10.1145/1882291.1882320

Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. ACM.

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI (PLDI 2015)*. ACM, 22–32. https://doi.org/10.1145/2737924.2737965

Filip Marić. 2010. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *TCS* 411, 50 (2010), 4333 – 4356. https://doi.org/10.1016/j.tcs.2010.09.014

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *PLDI (PLDI '12)*. ACM, 395–404. https://doi.org/10.1145/2254064.2254111

Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. 2012. *versat: A Verified Modern SAT Solver*. Springer Berlin Heidelberg, Berlin, Heidelberg, 363–378. https://doi.org/10.1007/978-3-642-27940-9_24

Pavel Panchekha and Emina Torlak. 2016. Automated Reasoning for Web Page Layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 181–194. https://doi.org/10.1145/2983990.2984010

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 396–407. https://doi.org/10.1145/2594291.2594339

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *NFM (NFM'11)*. Springer-Verlag, 313–327. http://dl.acm.org/citation.cfm?id=1986308.1986334

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *ASPLOS (ASPLOS '13)*. ACM, 305–316. https://doi.org/10.1145/2451116.2451150

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016a. Push-Button Verification of File Systems via Crash Refinement. In *OSDI'16*. USENIX Association, GA, 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016b. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular Synthesis of Sketches Using Models. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. 395–414. https://doi.org/10.1007/978-3-642-54013-4_22

Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. In *PLDI (PLDI '10)*. ACM, 111–121. https://doi.org/10.1145/1806596.1806611

Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda. 2014. *Distributed Places*. https://doi.org/10.1007/978-3-642-45340-3_3

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Onward! (Onward! 2013)*. ACM, 135–152. https://doi.org/10.1145/2509578.2509586

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *PLDI (PLDI '14)*. ACM, 530–541. https://doi.org/10.1145/2594291.2594340

Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking Axiomatic Specifications of Memory Models. In *PLDI (PLDI'10)*. ACM, 341–350. https://doi.org/10.1145/1806596.1806635

Richard Uhler. 2014. Tutorial 2 - Symbolic Computation. https://github.com/ruhler/smten/blob/master/tutorials/T2-SymbolicComputation.txt. (2014).

Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-based Search. In *OOSPLA (OOPSLA)*. ACM, 157–176. https://doi.org/10.1145/2660193.2660208

Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *OOPSLA*.

B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P*. 79–93. https://doi.org/10.1109/SP.2009.25