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

Konstantin Weitz
University of Washington, USA
weitzkon@cs.washington.edu

Steven Lyubomirksy
University of Washington, USA
sslyu@cs.washington.edu

Stefan Heule
Stanford University, USA
sheule@cs.stanford.edu

Emina Torlak
University of Washington, USA
emina@cs.washington.edu

Michael D. Ernst
University of Washington, USA
mernst@cs.washington.edu

Zachary Tatlock
University of Washington, USA
ztatlock@cs.washington.edu

Abstract

Many verification tools build on automated solvers. These tools reduce problems in an application domain (e.g., data-race detection) to queries that can be discharged with a highly optimized solver. However, tool builders rarely formally verify these reductions to ensure that the solver's output establishes the desired high-level property, decreasing confidence in the soundness of these tools.

This paper presents SpaceSearch, a library for developing solver-aided tools within a proof assistant. A user builds their solver-aided tool in Coq against the SpaceSearch interface, and then verifies that the results of the interface's operations can be lifted to establish the tool's desired high-level properties. Once verified, the tool can be extracted to an efficient implementation in a solver-aided language (e.g., Rosette) that instantiates 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 satisfies STOKE's x86 specification. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE. After these systems were patched by their developers, SaltShaker verified their agreement in under 2h. The second tool, BGProof$_{5S}$, is a verified version of an existing Border Gateway Protocol (BGP) router configuration checker. BGProof$_{5S}$ scales to checking industrial configurations spanning over 240 KLOC, identifying 19 configuration inconsistencies with no false positives.

1. Introduction

Solver-aided tools are used in a variety of domains including data-race detection [15, 23, 11], memory-model checking [32], and compiler optimization validation [17, 10]. 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, decreasing confidence in the soundness of the tool.

Past work has helped mitigate this problem by providing solver-aided host languages, such as Smten [34] and Rosette [30, 31]. These languages provide a higher-level interface to the underlying solver, which reduces the effort required to build solver-aided tools by orders of magnitude and, because the implementation is simpler, improves confidence in the tool's correctness.

However, solver-aided host languages are not designed to support formal reasoning about the meaning of solver calls in terms of the tool's application domain. Such reasoning is often necessary, since reductions to satisfiability typically depend on sophisticated domain knowledge, making them difficult to get right [35, 32, 10, 25]. For example, PEC is a solver-aided tool for verifying compiler loop optimizations [10]; it decomposes the proof of equivalence between the original and optimized code (its *application domain property*) by splitting the code at its branching points, and using an SMT solver to establish the equivalence of the resulting straight-line code fragments. Ensuring that the equivalence of these straight-line code fragments can be "stitched together" to prove the optimization correct requires reasoning in a higher-order logic [28].

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 [27, 26, 21], query incrementalization [22], and query parallelization [9]. 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, a library that provides a higher-level interface for building and formally verifying solver-aided tools within a proof assistant, i.e., SpaceSearch is a solver-aided host language for proof assistants. Using the expressive logic of the proof assistant, programmers can formally verify that the results of SpaceSearch's operations are sufficient to establish the desired application domain property. Once a solver-aided tool is implemented against this interface, it can be extracted (translated) to a solver-aided host language (e.g., Rosette) where the SpaceSearch interface is instantiated 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 modern SMT solvers. 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 evaluate SpaceSearch on two solver-aided tools. First, we built and verified SaltShaker, a solver-aided tool that checks, for all possible machine states, that an x86 instruction executed by RockSalt's Coq x86 semantics [19] behaves according to its instruction specifications extracted from STOKE [24]. SaltShaker verified the RockSalt semantics of over 15,000 instruction instantiations in under 2h, found 7 bugs in RockSalt, and found 1 bug in STOKE. We reported these bugs, and they were subsequently fixed by the respective developers.

Second, we modified Bagpipe [35], 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. To gain confidence that Bagpipe's reduction was correctly implemented, previous work[1] reimplemented Bagpipe as BGProof_V in Coq and verified its reduction. But that implementation times out on all industrial-scale configurations, so up to now, Bagpipe used the unverified implementation. This paper describes how we used SpaceSearch to extract BGProof_V (to Rosette) to obtain an efficient verified implementation dubbed $\text{BGProof}_{\mathsf{SS}}$. Instead of timing out, $\text{BGProof}_{\mathsf{SS}}$ with SpaceSearch runs on industrial configurations with over 240 KLOC, finds 19 inconsistencies, and due largely to parallelization, provides the same performance as the unverified Bagpipe prototype.

This paper's contributions include:

- 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 3).

[1] Submitted as supplemental material

$$
\begin{aligned}
&\mathit{Space} : \mathit{Type} \to \mathit{Type}\\
&[\![\ \_\ ]\!] : \mathit{Space}(A) \to \mathcal{P}(A)\\
&\mathit{empty}_A : \mathit{Space}(A)\\
&\mathit{single}_A : A \to \mathit{Space}(A)\\
&\mathit{union}_A : \mathit{Space}(A) \to \mathit{Space}(A) \to \mathit{Space}(A)\\
&\mathit{bind}_{A,B} : \mathit{Space}(A) \to (A \to \mathit{Space}(B)) \to \mathit{Space}(B)\\
&[\![\mathit{empty}]\!] = \emptyset\\
&[\![\mathit{single}(x)]\!] = \{x\}\\
&[\![\mathit{union}(s,t)]\!] = [\![s]\!] \cup [\![t]\!]\\
&[\![\mathit{bind}(s,f)]\!] = \bigcup_{a \in [\![s]\!]} [\![f(a)]\!]\\
&\mathit{search}_A : \mathit{Space}(A) \to \mathit{option}(A)\\
&\mathit{search}(s) = \mathit{None} \implies [\![s]\!] = \emptyset\\
&\mathit{search}(s) = \mathit{Some}(a) \implies a \in [\![s]\!]
\end{aligned}
$$

Figure 1. SpaceSearch Basic ADT.

- Various backends for SpaceSearch's high-level interface that enable solving search problems using brute force, parallel, incremental, and SAT/SMT search (Section 4).
- An evaluation of SpaceSearch via the construction of two solver aided tools: SaltShaker checks the correctness of x86 instruction semantics in RockSalt (Section 5), and $\text{BGProof}_{\mathsf{SS}}$ checks the correctness of Border Gateway Protocol (BGP) configurations (Section 6).

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 [3] data types and commands), SpaceSearch provides a high-level interface inspired by Smten [34]. This 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 [34].

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

2.1 SpaceSearch Interface

The SpaceSearch 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, which we call *search spaces*, 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 spaces. 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. These operations are subscripted by the type of solutions in the search problem, but we omit subscripts that can be easily inferred from the 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 which ever internal representation is most efficient for conducting searches over a particular ADT's space.

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, the interface only specifies that one arbitrary solution is returned.

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. A solution to an n-queens problem places n queens on an $n \times n$ chessboard so that no two queens attack each other, defined as two queens sharing the same column, row, or diagonal. We first describe a solver-aided algorithm for finding a solution to an n-queens problem, and then use the semantics of SpaceSearch to show the algorithm's correctness.

n-Queens Algorithm. Figure 2 shows a SpaceSearch implementation of the n-queens solver taken from a Smten tutorial [33]. 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.

$$
\begin{array}{l}
solveNQueens(n : Integer) : option(list(Integer \times Integer)) \\
\quad search(bind(placements(n,n), (\lambda q. \\
\quad\quad \texttt{if } noAttack(q) \texttt{ then } single(q) \texttt{ else } empty))).
\end{array}
$$

$$
\begin{array}{l}
placements(n, 0) := single([\,]) \\
placements(n, S(x)) := \\
\quad bind(range(0,n), (\lambda y : Integer. \\
\quad bind(placements(n,x), (\lambda q : list(Integer \times Integer) \\
\quad\quad single((x,y) :: q)))))
\end{array}
$$

$$
\begin{array}{l}
noAttack(q : list(Integer \times Integer)) : bool := \\
\quad distinct(map(fst, q)) \wedge distinct(map(snd, q)) \wedge \\
\quad distinct(map(plus, q)) \wedge distinct(map(minus, q))
\end{array}
$$

Figure 2. N-Queens in SpaceSearch.

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

$noAttack(q)$ 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).

n-Queens Algorithm Correctness. The Smten algorithm employs two optimizations, which we 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 can prove this optimization correct in Coq, as any placement of two queens on the same column leads to an attack, and is thus not a solution.

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 [33].

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

Figure 3. Ensembles.

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

sum $(x + y)$

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

difference $(x - y)$

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

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. The optimized $noAttack$ check therefore enforces the rules of the puzzle.

While we omit both proofs for brevity, we 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) [16]. 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 SpaceSearch ADTs for constructing and solving search problems.

3.1 Constructing 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 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 $\bot$, 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

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

Figure 4. SpaceSearch Integer and BitVector ADTs.

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 $f(a)$.

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: $\mathit{union}(\mathit{single}(\mathit{true}), \mathit{single}(\mathit{false}))$. But these basic operations cannot be used to construct infinite spaces, like the space of all integers. SpaceSearch thus provides additional ADTs to construct *infinite* search spaces.

Figure 4 describes SpaceSearch's *Integer ADT*, which provides the $\mathit{Integer}$ type. Elements n of type $\mathit{Integer}$ 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 $\mathit{intPlus}$, which are denoted to the corresponding constants and operations on the mathematical integers.

The most interesting value provided by the Integer ADT is the $\mathit{intFull}$ space. This space contains every one of the infinitely many integers, and is thus denoted as the ensemble that returns the true proposition $\top$ for every integer. As we will see in Section 3.2, providing this space has potential implications on the solvability of search problems.

Specialized Search Problems. While the Basic ADT can be used to construct search problems for any of Coq's native types, these problems cannot always be solved efficiently. For example, we initially tried to build SaltShaker 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 (i.e., within a day). SpaceSearch therefore also exposes ADTs to construct *specialized* search spaces that certain solvers can search more efficiently.

$$Callable : Type \to Type \to Type$$
$$call_{A,B} : Callable(A, B) \to A \to B$$
$$callableBind_{A,B} : Space(A) \to Callable(A, Space(B)) \to Space(B)$$
$$callableBind(s, r) = bind(s, call(r))$$

Figure 5. SpaceSearch Callable ADT.

$$minus_A : Space(A) \to Space(A) \to Space(A)$$
$$[\![minus(s, t)]\!] = [\![s]\!] \setminus [\![t]\!]$$
$$incSearch(s, t, f) := search(bind(minus(s, t), f))$$

Figure 6. SpaceSearch Minus ADT and Incremental Search.

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

Callable Search Problems A weakness of the SpaceSearch 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 is very slow or impossible), because in Coq, the only way to learn anything about a function is via function invocation.

SpaceSearch gets around this problem with the use of extraction. 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 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 $bind(s, f)$ with s.

To also allow the efficient implementation of $bind$ in Coq, SpaceSearch provides $callableBind$ (Fig. 5), a version of $bind$ whose second argument is a value that can be called, but depending on the $Callable$ type, may also support other operations, such as looking 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 $Callable(A, B)$ can be called using the $call$ operation, which converts r to a function from A to B. The $callableBind(s, r)$ operation thus has the semantics of $bind(s, call(r))$.

$$preciseSearch_A : Space(A) \to option(A)$$
$$preciseSearch(s) = None \implies [\![s]\!] = \emptyset$$
$$preciseSearch(s) = Some(a) \implies a \in [\![s]\!]$$

$$heuristicSearch_A : Space(A) \to option(option(A))$$
$$heuristicSearch(s) = None \implies \top$$
$$heuristicSearch(s) = Some(None) \implies [\![s]\!] = \emptyset$$
$$heuristicSearch(s) = Some(Some(a)) \implies a \in [\![s]\!]$$

Figure 7. Search ADTs.

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.

Figure 6 describes the Minus ADT and the incremental search function $incSearch(s, t, f)$. This function returns all the solutions of $bind(s, f)$ assuming that $bind(t, f)$ has already been searched and has no solutions. As a result, $incSearch(s, t, f)$ avoids having to perform the bind on a portion of the space that is already known not to return a solution.

The $incSearch(s, t, f)$ 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, SaltShaker applies a computationally expensive verification function to every element in a frequently changing set of instructions.

3.2 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 takes as input a search space s and returns either $None$, 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, 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.

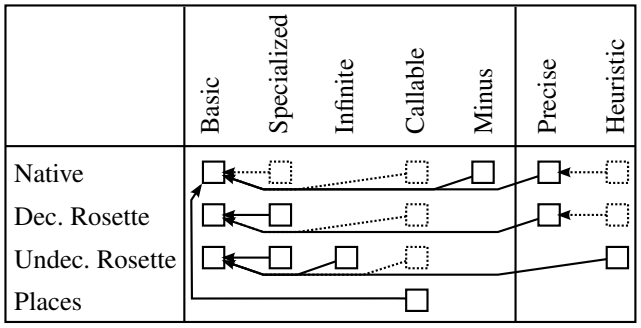

Figure 8. SpaceSearch Backend ADT Instances. A box means that the row's backend provides an instance of the column's ADT. An arrow $i \leftarrow j$ means that j depends on i. A dotted instance is automatically derived from the instance it depends on (indicated by a dotted arrow).

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.

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 this library 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 [5] style testing. Such an implementation of $heuristicSearch_A(s)$ 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 $None$.

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; 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 [29] 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.

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

Figure 9. SpaceSearch Native Backend.

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 $in(a, s)$ predicate is true iff a is an element of the list s. $empty$ is then just the empty list, $single(x)$ is the singleton list of x, $union(s, t)$ concatenates the two lists s and t, and $bind(s, f)$ 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 $listMinus(l, l')$ 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. In the Callable ADT, $Callable(A, B)$ 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

$empty$	$\triangleq$	`(lambda () (assert false))`
$single(x)$	$\triangleq$	`(lambda () x)`
$union(s,t)$	$\triangleq$	`(lambda ()` `  (if (symbolic-bool) (s) (t)))`
$bind(s,f)$	$\triangleq$	`(lambda () ((f (s))))`
$preciseSearch(s)$	$\triangleq$	`(solve (s))`
$heuristicSearch(s)$	$\triangleq$	`(solve (s))`

Figure 10. SpaceSearch Decidable and Undecidable Rosette Backend.

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 [30, 31], 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 [30, 31] 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 $\forall\, x\, y.\, x \wedge y \iff \neg(\neg x \vee \neg y)$ 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 [4] formula and solving it with an off-the-shelf SMT solver.

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 (i.e., 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 determinstic 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.

Extraction to Rosette in this simplified fashion 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 overcome this problem 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 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

$Callable(A, B)$:= $Worker(A, B)$
$call(r)$:= $runWorker(r)$
$callableBind(l, w)$ $\triangleq$ `(bind (map (spawn w) l) get)`

```
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))))
```

Figure 11. SpaceSearch Distributed Places Backend.

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 judgemental 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, as each has its strengths and weaknesses.

4.3 Distributed Places Backend

The Places Backend provides a parallel, distributed implement of SpaceSearch's Callable ADT using the Distributed Places [29] 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. Every $Space(A)$ is thus implemented as a $list(A)$.

The $callableBind(l, w)$ implementation first `spawn`s a new thread with the worker w for every element (solution) a of the list (space) l, then `get`s 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 `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 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

$succ : Worker(natural, list(natural)) \triangleq$ `(quote succ)`
$sqrt : Worker(integer, list(double)) \triangleq$ `(quote sqrt)`
$runWorker \triangleq$

```
(lambda (w a) (cond
  ((eq? w (quote succ)) (list (+ 1 a)))
  ((eq? w (quote sqrt)) (list (sqrt a) (- (sqrt a))))))
```

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

$$
\begin{array}{l}
rocksalt : instr \to rocksaltOracleType \to state \to state \\
rocksaltOracleType := \mathbb{N} \to bv(1) \\
spec : (i : instr) \to specOracleType(i) \to state \to state \\
specOracleType(i) := bv(specOracleBits(i)) \\
specOracleBits : instr \to \mathbb{N} \\
\\
saltShaker(i : instr) : option(state \times specOracleBits(i)) := \\
\quad preciseSearch(\\
\quad\quad bind(stateFull, (\lambda s_0 : state. \\
\quad\quad bind(bvFull(specOracleBits(i)), (\lambda o : specOracleType(i). \\
\quad\quad\quad \texttt{if } exists(\lambda w.\ spec(o, s_0) =? \\
\quad\quad\quad\quad\quad\quad rocksalt(flagOracle(w), s_0)) \\
\quad\quad\quad \texttt{then } empty \texttt{ else } single(s_0, o)))))) \\
\\
flagOracle(w : bv(5)) : rocksaltOracleType := \lambda\, i\, n. \\
\quad \texttt{if } i < 5 \texttt{ then } w[i] \texttt{ else } bvZero \\
\\
state := \{ \\
\quad eax : bv(32);\ ecx : bv(32);\ edx : bv(32);\ ebx : bv(32); \\
\quad esp : bv(32);\ ebp : bv(32);\ esi : bv(32);\ edi : bv(32); \\
\quad cf : bv(1);\ pf : bv(1);\ zf : bv(1);\ sf : bv(1);\ of : bv(1) \\
\} \\
stateFull : Space(state) := \\
\quad bind(bvFull(32), (\lambda eax'.\ bind(bvFull(32), (\lambda ebx'. \ldots . \\
\quad\quad single(\{eax := eax'; ebx := ebx'; ...\})))))
\end{array}
$$

Figure 12. SaltShaker.

of solutions that are easily enumerable, and the callable r performs an expensive computation.

5. SaltShaker: Verifying x86 Semantics

5.1 SaltShaker Overview

RockSalt [19] is a formal checker for the safety of Native Client [36] code, a sandbox developed by Google. Part of this checker is a specification of a subset of the x86 instruction set that powers most desktops and servers today. 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 instruction is sound with respect to another x86 specification.

The official Intel x86 specification [8] (a 3,800 page document using English and informal pseudo-code) details that certain instructions can have *undefined* output; that is, for particular locations, the CPU is free to store arbitrary bits. RockSalt (and other formal x86 semantics) model this explicitly by being parameterized over an *oracle* that provides a stream of bits that can be used to make non-deterministic choices to fill the undefined output locations. More formally, RockSalt provides

$$rocksalt : instr \to rocksaltOracleType \to state \to state$$

which, for a given instruction, oracle and input machine state, returns a new state that captures the result of execution the instruction. $rocksaltOracleType$ is modeled as $\mathbb{N} \to bv(1)$ and can be thought of as providing a stream of bits to make any non-deterministic choice that may be required. In addition to inherent undefined outputs in x86, RockSalt can also use the oracle to over-approximate some outputs of an instruction if providing a precise semantics is difficult or not needed.

To prove the soundness of RockSalt for a given subset of instructions, we compare its semantics to another specification. This specification similarly provides a function $spec$ that produces an output state given an instruction, oracle and input state. For practical purposes, we assume that the specification also provides a function $specOracleBits : instr \to \mathbb{N}$, which returns the number of non-deterministic choice bits that are required for a given instruction.[2]

We instantiate $spec$ with the semantics found in STOKE [24], 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 [7]. The instantiation is implemented by pretty-printing STOKE's semantics of instructions to Rosette functions (using a small custom extension of STOKE), and then extracting $spec$ and $specOracleBits$ from that.

RockSalt and STOKE support slightly differently 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.

With these definitions, 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 implementation cpu running i. Such a specification compliant cpu makes a concrete choice for all unspecified outputs and is a function of type $instr \to state \to state$ that no longer requires an oracle.

[2] In practice, this is easy to ensure, as the official Intel specification says exactly how many output bits are undefined.

Formally, we require

$$\forall s_0\ P\ cpu.\ (\forall o.\ P(rocksalt(i, o, s_0))) \rightarrow P(cpu(i, s_0))$$

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_0 and specification oracle o_s such that there is no RockSalt oracle o_r for which the specification $spec(o_s, s_0)$ and RockSalt $rocksalt(o_r, s_0)$ are equal.

To make this check practical, we make the following (sound) approximation. Instead of existentially quantifying over the infinite space of all RockSalt oracles, SaltShaker searches over oracles that only provide at most five non-deterministic bits (and returns 0 otherwise). 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.

This approach 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. Figure 12 provides all definitions and shows the implementation of SaltShaker.

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?

We ran SpaceSearch on all 40 opcodes that are supported by both RockSalt and STOKE, using a single computer with an Intel i7-4790K CPU and 32 GiB of memory. 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`). We tested 15,255 different instruction instantiations (or just instructions, for short), 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.).

Q1: Verifying these 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 underlying issues in RockSalt and 1 issue in STOKE[3]. All of these have been fixed since.

Specifically, RockSalt (1) computed wrong result 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 parity flag (of 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 [19] is likely correct, because the bugs that we found were mostly in the computation of flags or were introduced in a refactoring after the release of the verified NaCL implementation (issue 1 above).

STOKE computed the 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 thought the official Intel semantics does specify its behavior).

Q2: 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,

[3] We group failures by the conceptually underlying issue 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 underlying issue.

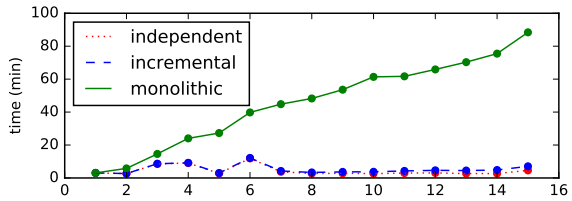

Figure 13. Performance gains of incrementalizing SaltShaker.

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: 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: 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. BGProof$_{SS}$: Verifying BGP Configurations

6.1 BGProof$_{SS}$ Overview

Whenever someone wants to watch a video, send an email, or check the news on the Internet, they have to communicate with a server that is potentially on the other side of the world. The Internet itself is a network made up of smaller, interconnected but autonomous networks run by Internet Service

$$\begin{array}{l}
verifyISP(c : Config, s : Property) : option(Path \times Anno) := \\
\quad preciseSearch(callableBind(\\
\quad\quad bind(fullPath(c), \lambda p.single((c, s, p))), verifyPathWorker)) \\
\\
verifyPathWorker : Worker(Config \times Property \times Path, \\
\quad Space(Path \times Anno)) \triangleq (\texttt{quote verifyPathWorker}) \\
runWorker \triangleq (\texttt{lambda (_ csp) (verifyPath csp)})) \\
\\
optionToSpace(Some(a)) := single(a) \\
optionToSpace(None) := empty \\
\\
verifyPath(c, s, p) : Space(Path \times Anno) := \\
\quad optionToSpace(preciseSearch(\\
\quad\quad bind(fullAnnouncement, (\lambda a : Anno. \\
\quad\quad\quad \texttt{if } check(c, s, p, a) \texttt{ then } empty \texttt{ else } single(t, a))))
\end{array}$$

Figure 14. BGProof$_{SS}$.

Providers (ISPs) like Comcast, MIT, and IBM. For the end-to-end communication over the Internet to work, each ISP must notify all other ISPs of the destinations (like the video/e-mail/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 all ISPs have been notified of all destinations, anyone can communicate with anyone else on the Internet.

To ensure reliable and secure communication, ISPs must configure their BGP routers to restrict how route announcements can be used and exchanged. For example, ISPs configure their routers with policies to never use route announcement to bogus destinations like *localhost*. Because BGP gives ISPs freedom 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.

Bagpipe is a solver-aided tool, written in Rosette, that enables ISPs to express desirable properties and automatically check them for a given set of router configurations. To gain confidence that Bagpipe's reduction was correctly implemented, previous work reimplemented Bagpipe as BGProof$_V$ in Coq and verified its reduction. But that implementation times out on all industrial-scale configurations, so up to now, Bagpipe used the unverified implementation. In this section, we use SpaceSearch to extract BGProof$_V$ to Rosette, dubbed BGProof$_{SS}$, and thus actually run the verified algorithm.

Figure 14 describes BGProof$_{SS}$'s checking algorithm. $verifyISP$.[4] Given a desirable property $s : Property$ and a set of router configurations $c : Config$, the $verifyISP$ function

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

checks that any announcement $a : Anno$ forwarded along any path $p : Path$ through the network satisfies the desirable property s (where the space of paths $fullPath$ is derived from the router configurations c).

BGProof$_{SS}$'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.

BGProof$_{SS}$'s algorithm is subtle, as it only checks that a single announcement forwarded along a single path satisfies the desired property, but it does not check that multiple announcements forwarded along multiple paths concurrently also satisfy the desired property. However, we have proved BGProof$_{SS}$ is sound, based on the verification of BGProof$_V$.

6.2 BGProof$_{SS}$ 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?

Just like in the original Bagpipe paper [35], 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. BGProof$_{SS}$ ran this experiment on Amazon EC2 with 2 instances of type `c3.8xlarge`, each with 32 virtual-cores and 60 GB of memory.

Q1: BGProof$_{SS}$ ran the experiment in a total of 82h, the cost for which is about $30 using EC2 spot instances. During the verification, BGProof$_{SS}$ 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.

Q2: BGProof$_{SS}$ can check the desired property for each path independently, which means that, apart from a short startup period, BGProof$_{SS}$'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.

7. Related Work

Solver-aided tools. Advances in solver technology, including SMT, SAT, and model-finding, have made solver-aided tools a compelling option in many domains; here, we briefly mention a handful of representative examples. Boogie [13] and related tools [12, 11] enable general-purpose verification by compiling verification conditions to SMT queries. Alive [17] and PEC [10] verify compiler optimizations using a solver back end. Batfish [6] verifies data plane properties using a Datalog solver; Vericon [2] 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 [34] and Rosette [30, 31] 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 [34].

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, by providing operations for parallelization and incrementalization, SpaceSearch pushes the boundary on automatically orchestrating solver queries even further. 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 solver have been built and verified in proof assistants like Coq and Isabelle [14, 20, 18], 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 do not provide the additional features of SMT solvers.

Witness checkers [1] 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. SpaceSearch provides a Coq interface against which users build their solver-aided tool and verify that the results of the interface's operations 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 BGProof$_{SS}$, 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. BGProof$_{SS}$ 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. We found that SpaceSearch can be used with almost unchanged 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.

References

[1] M. Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *CPP*. 2011.

[2] T. Ball et al. "VeriCon: Towards Verifying Controller Programs in Software-defined Networks". In: *PLDI*. 2014.

[3] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[4] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.smt-lib.org`. 2016.

[5] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ICFP*. 2000.

[6] A. Fogel et al. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015.

[7] S. Heule et al. "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set". In: *PLDI*. 2016.

[8] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals, Revision 325462-057US*. 2015.

[9] J. Jeon et al. "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*. 2015.

[10] S. Kundu, Z. Tatlock, and S. Lerner. "Proving Optimizations Correct Using Parameterized Program Equivalence". In: *2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009.

[11] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. "Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers". In: *CAV*. 2009.

[12] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR*. 2010.

[13] K. R. M. Leino. *This is Boogie 2*. Tech. rep. 2008.

[14] S. Lescuyer and S. Conchon. "Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme". In: *FroCoS 2009*. 2009.

[15] G. Li and G. Gopalakrishnan. "Scalable SMT-based Verification of GPU Kernel Functions". In: *FSE*. 2010.

[16] B. Liskov and S. Zilles. "Programming with Abstract Data Types". In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. 1974.

[17] N. P. Lopes et al. "Provably Correct Peephole Optimizations with Alive". In: *PLDI*. 2015.

[18] F. Mari. "Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL". In: *TCS* 50 (2010).

[19] G. Morrisett et al. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *PLDI*. 2012.

[20] D. Oe et al. "versat: A Verified Modern SAT Solver". In: *VMCAI*. 2012.

[21] P. Panchekha and E. Torlak. "Automated Reasoning for Web Page Layout". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016.

[22] P. M. Phothilimthana et al. "Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.

[23] M. Said et al. "Generating Data Race Witnesses by an SMT-based Analysis". In: *NFM*. 2011.

[24] E. Schkufza, R. Sharma, and A. Aiken. "Stochastic Superoptimization". In: *ASPLOS*. 2013.

[25] H. Sigurbjarnarson et al. "Push-Button Verification of File Systems via Crash Refinement". In: *OSDI'16*. 2016.

[26] H. Sigurbjarnarson et al. "Push-Button Verification of File Systems via Crash Refinement". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.

[27] R. Singh et al. "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*. 2014.

[28] Z. Tatlock and S. Lerner. "Bringing Extensibility to Verified Compilers". In: *PLDI*. 2010.

[29] K. Tew et al. "Distributed Places". In: *TFP*. 2014.

[30] E. Torlak and R. Bodik. “A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages”. In: *PLDI*. 2014.

[31] E. Torlak and R. Bodik. “Growing Solver-aided Languages with Rosette”. In: *Onward!* 2013.

[32] E. Torlak, M. Vaziri, and J. Dolby. “MemSAT: Checking Axiomatic Specifications of Memory Models”. In: *PLDI*. 2010.

[33] R. Uhler. *Tutorial 2 - Symbolic Computation*. `https://github.com/ruhler/smten/blob/master/tutorials/T2-SymbolicComputation.txt`. 2014.

[34] R. Uhler and N. Dave. “Smten with Satisfiability-based Search”. In: *OOSPLA*. 2014.

[35] K. Weitz et al. “Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver”. In: *OOPSLA*. 2016.

[36] B. Yee et al. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *S&P*. 2009.