

On Scalable Shape Analysis

Hongseok Yang, Oukseh Lee, Cristiano Calcagno,
Dino Distefano and Peter O'Hearn



RR-07-10

November 2007

Department of Computer Science



On Scalable Shape Analysis

Hongseok Yang

Queen Mary, University of London
hyang@dcs.qmul.ac.uk

Oukseh Lee

Hanyang University, Korea
oukseh@gmail.com

Cristiano Calcagno

Imperial College
ccris@doc.ic.ac.uk

Dino Distefano

Queen Mary, University of London
ddino@dcs.qmul.ac.uk

Peter O’Hearn

Queen Mary, University of London
ohearn@dcs.qmul.ac.uk

Abstract

Shape analysis is a precise form of pointer analysis, which can be used to verify deep properties of data structures such as whether or not they are cyclic, whether they are nested, etc. Shape analyses are also expensive, and the tremendous number of abstract states they generate is an impediment to their use in verification of sizeable programs. We start with an analysis that is able to analyze programs up to 1000 lines manipulating complex, nested structures, and progressively improve it until it is capable of analyzing programs of up to 10,000 lines. By experimental results we show that our analysis is precise. It identifies memory safety errors and memory leaks in several Windows and Linux device drivers and, after these bugs are fixed, it automatically proves integrity of pointer manipulation for these drivers.

This order of magnitude improvement in sizes of programs verified is obtained by combining several ideas. One is the local reasoning idea of separation logic, which reduces recomputation of analysis of procedure bodies, and which allows efficient transfer functions for primitive program statements. Another is an interprocedural analysis algorithm which aggressively discards intermediate states. The most important new technical contribution of the work is a new join operator, which greatly reduces the number of abstract states used by the analysis while not greatly reducing precision; the join is also integrated with procedure summaries in an interprocedural analysis.

1. Introduction

Automatic software verification has seen an upsurge of interest in recent years. By lowering aims from proving full functional correctness to weaker properties, such as absence of arithmetic errors or protocol properties of procedures, it has been possible to obtain practical and fully automatic proof methods for special classes of real-world software. This is exemplified by tools such as SLAM [2] and ASTRÉE [5], which have been used to verify properties of device drivers and avionics code.

Crucial in this reinvigoration of software verification has been the employment of methods from static program analysis. Analysis techniques lessen annotation burden, by inferring loop invariants and by automatically computing procedure summaries. This is particularly important for programs, such as device drivers, where invariants and procedure specifications may not be known beforehand and may otherwise be nontrivial to come by.

While these advances are impressive, a persistent trouble area stands in the way of verification-oriented program analysis for a wider range of properties: the heap. Shallow pointer analyses, which tabulate points-to information between dereferencing expressions of bounded length, often do not give enough information for verification purposes. For example, we might like to prove, automatically, that a device driver manipulating a collection of cyclic linked lists, themselves with sublists, does not dereference null or a dangling pointer. Even if the property (e.g., dereferencing null) is a weak-sounding one, proving it requires looking deeper into the heap than do points-to analyses, in fact unboundedly deep. One might say that the analysis needs to track indefinite dereferencing (by analogy with indefinite iteration), in the presence of dynamic allocation, deallocation, and destructive heap updates.

Shape analyses are precise, deep forms of pointer analysis, that infer the sort of information needed for verification [25, 26, 1, 14, 9, 6]. With a shape analysis it is possible to characterize, for example, whether a variable points to a cyclic or acyclic list, or a doubly-linked list, a list with back-pointers, etc. It is because of this precision that shape analyses are in principle useful for verification. However, again because of their precision, shape analyses oriented to verification are expensive; the analysis engine can encounter an enormous number of states. As a result, the in-principle promise of shape analysis has not yet translated into practice.

The purpose of this paper is to take steps towards addressing the central scalability problem for shape analysis. We start with an analysis that is able to analyze C programs of around 1000 LOC manipulating complex, nested structures, and progressively improve it until it can analyze programs of up to 10,000 LOC. On some programs, we observe savings in space and time reductions well beyond this order of magnitude improvement in LOC verified (of course, this observation is consistent with an analysis that is not linear-time).

The main thrust of this work is experimental in nature, and we liberally borrow from what has gone before: we want to see how far shape analysis can be pushed. We utilize the basic ideas on shape analysis for deep heap update, advanced by Sagiv et al first in [25]; namely, we use analogues of materialization of summary nodes (which we call rearrangement), followed by symbolic execution,

[Copyright notice will appear here once ‘preprint’ option is removed.]

followed by repackaging of the results (which we call abstraction). We utilize the idea of local reasoning, stemming from the frame rule in separation logic, which demonstrates that it is sound, when analyzing an instruction, to pass only a fragment of the input heap [20]. This can be used to speed up the analysis of both primitive instructions (as in [4, 9, 17]) and procedure calls [23, 10]. Finally, our treatment of procedures is based on the interprocedural analysis algorithm of Reps, Horowitz and Sagiv [22] (subsequently, RHS), which attempts to limit re-execution of procedure bodies using certain procedure summaries.

Although we re-use these ideas, we also provide some new perspectives on them. We reformulate RHS in a style that makes it easier to account for further optimizations used in our analyzer. One of the optimizations concerns the use of a join operator, whose incorporation into the original RHS presentation is non-trivial. Another is a conceptually simple but pragmatically useful optimization that aggressively discards unneeded states.

Concerning local reasoning, we take the ideas over wholesale. As far as new information is concerned, we provide more extensive experimental results than previously given. In particular, we identify limits to what gains localization can provide by providing negative experimental results, to complement the positive results reported in [23, 24, 10].

The most significant new technical material in the paper, and the most important optimization for scalability, is a new join operator. A join operator takes several abstract states, and finds a common generalization [8]. Using join can speed up an analysis, but care is needed not to lose too much information on generalization. Too much loss can lead to many false alarms, which are less tolerable in application of program analysis to verification than to other tasks such as compiler optimization. It is even possible to slow down an analysis, if the implementation of join is too costly. The truth is that the devil is in the detail: the problem of designing a good join operator is a balancing act between precision, expense (of the join), and gain (in state-space reduction).

Our new join operator leads to a dramatic reduction in the state space of the analysis. In Section 3 we quote an example of a program where our analysis without join produces a post-condition with over 3000 states, and with join the post-condition has only 1 state. This example program is *environment code*, which is used in the verification of a device driver. It nondeterministically generates an infinite number of linked structures of various sizes, which are passed to driver dispatch routines. The > 3000 states in the analysis without join do provide a finite overapproximation of this infinity, but to have to run the dispatch routines on > 3000 states (where each run does not use join) can be expensive, expensive enough that termination in reasonable time and space is not possible for our base analysis. The difference that a good join can make should be evident.

The way we arrived at our particular join operator was partly by design, and partly by experimentation. It has several distinguishing characteristics, the most important being the way that it treats a combination of predicates for possibly empty and necessarily nonempty lists. We will come back to this point several times in the paper.

We show by experimental results that our new join operator does not lead to an unacceptable loss of precision. Our experiments concern five device drivers ranging from 2500 to 10,000 LOC. The analysis identifies a number of genuine memory safety errors and memory leaks in these drivers, without any false alarms, and it proves integrity of pointer manipulation for the drivers after the bugs have been fixed.

Before continuing, we remark that, in passing from 1000 to 10,000 lines, we have held the data structure fixed – the numbers refer to code from the Windows IEEE 1394 (firewire) device driver

– so the comparison has some relevance. If we were to change this baseline to a program with simpler or more complex data structures than 1394 then the numbers could change. Indeed, two of the drivers that we analyze are smaller than 1394, but more difficult. If we were to strip out the environment code and instead manually supply preconditions then the programs we could analyze would again be larger, but it is more realistic to include environment code (furthermore, the preconditions are nontrivial to write). We offer these remarks just to warn the reader that the specific numbers must be interpreted with care (even if there is value in comparisons), and with that having been said now move on to the technical development.

2. Basic Setting

In this section we describe our initial analysis. We start by describing the inputs to our analysis: interprocedural control flow graphs with heap-manipulating commands.

2.1 Control Flow Graphs

Consider a language of atomic commands a defined by the grammar below:

$$\begin{aligned} B &::= E=E \mid E \neq E \\ E &::= x \mid \text{nil} \\ a &::= x := E \mid [E] := E \mid x := [E] \mid \text{free}(E) \mid x := \text{new}() \\ &\quad \mid \text{assume}(B) \mid f(). \end{aligned}$$

This language includes an assignment to a variable x ($x:=E$), an update of a heap cell E ($[E]:=E'$), the lookup of the content of cell E ($x:=[E]$), the disposal of cell E ($\text{free}(E)$), and the allocation of a new cell ($x:=\text{new}()$). The language also has an assume statement $\text{assume}(B)$ that filters out all states not satisfying B . Finally, it contains a call $f()$ to a parameterless procedure f .

Our language is chosen to be simple; we only consider unary cells and parameterless procedures and we assume that programs do not have local variables. This simplification is mainly for clarifying presentation, and it does not limit the techniques described in the paper. In fact, our implementation deals with C structs, which contain multiple fields, and C procedures, which can pass parameters, return a value and declare local variables.

An intraprocedural control flow graph is a finite graph $(N, E \subseteq N \times N)$ with two distinguished nodes s, e and a map L which labels nodes n in N with atomic commands a . The s node models the starting point of a procedure, and it is required to have no incoming edges. The e node models the exit of a procedure, and it is not allowed to have any outgoing edges.

A control flow graph (or an interprocedural control flow graph) is a finite collection of disjoint intraprocedural control flow graphs

$$G = \{(N_f, E_f, L_f, s_f, e_f)\}_{f \in F}.$$

Here F is the set of all the procedures defined by a program. Since we are considering a whole program analysis in the paper, we assume that F contains all the called procedures f (i.e., f 's such that its call $f()$ is in the range of L_g for some g) as well as the main procedure main . In the paper, we write N, E and L for $\cup_{f \in F} N_f, \cup_{f \in F} E_f$ and $\cup_{f \in F} L_f$, respectively.

2.2 Shape Domain

We give a description of our abstract domain parameterized by a collection \mathcal{P} of primitive predicates. By parameterizing the definition in this way, it is clear that our results apply to a number of specific domains. Our definitions are for an analysis where the abstract states are certain separation logic formulae called *symbolic heaps*, following the approach initiated in [4, 9].

The symbolic heaps q , are defined by the following grammar:

$$\begin{aligned} e &::= x \mid x' \mid \text{nil} \\ \Pi &::= \Pi \wedge \Pi \mid e=e \mid e \neq e \mid \text{true} \\ \Sigma &::= \Sigma * \Sigma \mid \text{emp} \mid \mathcal{P} \mid \text{true} \\ \mathcal{P} &::= \dots \\ q &::= \text{err} \mid \Pi \wedge \Sigma \end{aligned}$$

A symbolic heap q can be err , denoting the error state, or it has the form $\Pi \wedge \Sigma$, where Π and Σ describe properties of variables and the heap, respectively. We remind the reader of the meaning of Σ 's, since our analysis mostly works on the Σ part of symbolic heaps. The separating conjunction $\Sigma_0 * \Sigma_1$ of Σ_0 and Σ_1 holds for a heap if and only if the heap can be split into two disjoint parts, one making Σ_0 true and the other making Σ_1 true. emp means the empty heap, and true holds for all heaps. Throughout the paper, we use the fact that emp is the unit of $*$ so that $\Sigma * \text{emp} = \text{emp} * \Sigma = \Sigma$. Note that in each symbolic heap, we use both normal program variables and primed variables. These primed variables are used to denote cells that are not directly pointed to by program variables, and they are assumed to be (implicitly) existentially quantified outside of symbolic heaps.

Our first, simplest, instantiation of \mathcal{P} is as follows:

$$\begin{aligned} k &::= \text{PE} \mid \text{NE} \\ \mathcal{P} &::= (e \mapsto e) \mid \text{ls } k \ e \ e \end{aligned}$$

Here, $e \mapsto e'$ means a heap with only one cell e that stores e' . The list segment predicate $\text{ls } k \ e_0 \ e_1$ denotes heaps containing one list segment from e_0 to e_1 only. This list segment starts at cell e_0 and its last cell stores e_1 . The list is possibly empty if $k = \text{PE}$; otherwise (i.e., $k = \text{NE}$), the list is not empty.

The meanings of the segment predicates can be understood in terms of the definitions

$$\begin{aligned} \text{ls } \text{PE} \ e \ e' &\iff (e = e' \wedge \text{emp}) \vee (\text{ls } \text{NE} \ e \ e'), \\ \text{ls } \text{NE} \ e \ e' &\iff (e \mapsto e') \vee (\exists y'. e \mapsto y' * \text{ls } \text{NE} \ y' \ e'). \end{aligned}$$

These definitions are not *within* the shape domain (e.g., the domain does not have \vee), but are mathematical definitions in the metalanguage, used to verify soundness of operations on the predicates. Note that there is no problem with the recursion in $\text{ls } \text{NE}$: the recursive instance is in a positive position, and the definition satisfies monotonicity properties sufficient to ensure a least solution in a suitable sense.

The reader might have noticed that having $\text{ls } \text{PE}$ does not give us any extra expressive power: its meaning can be represented using two abstract states, one a emp and the other a $\text{ls } \text{NE}$. However, having $\text{ls } \text{PE}$ impacts on performance, as it represents disjunctive information, succinctly. We will return to this point several times throughout the paper.

A different instantiation of \mathcal{P} gives us the predicates for a variation on the domain of [3].¹

$$\begin{aligned} k &::= \text{PE} \mid \text{NE} \\ \mathcal{P} &::= (e \mapsto \vec{f} : \vec{e}) \mid \text{ls } k \ \phi \ e \ e \end{aligned}$$

Here, the points-to predicate $(e \mapsto \vec{f} : \vec{e})$ is for records with fields \vec{f} , and ϕ is a predicate that describes the shape of each node in a list. The definition of the nonempty list segment here is

$$\text{ls } \text{NE} \ \phi \ e \ e' \iff \phi(e, e') \vee (\exists y'. \phi(e, y') * \text{ls } \text{NE} \ y' \ e')$$

and the ϕ predicate gives us a way to describe composite structures. For example, if ϕ describes lists, then $\text{ls } \text{NE} \ \phi \ e \ e'$ describes lists of lists: see [3] for examples.

The experiments in this paper are done using this second instantiation of \mathcal{P} . It is similar to the domain from [3], but uses predicates

¹This instantiation assumes the change of the language where we have heap cells with multiple fields, instead of unary cells.

for both possibly empty and necessarily nonempty list segments. The experiments use this domain because of its ability to describe the composite structures found in device drivers. On the other hand the small examples in the text will be done using the simpler domain, without the ϕ 's.

We show some of the basic ideas with a simple example, a program that nondeterministically creates a linked list of arbitrary size (it should be clear how this is rendered as a flow graph).

```
// program "onelist.create"
x := nil;
while (NONDET) {d := new(); [d] := x; x := d;};
```

When we run our basic analysis algorithm, it returns three symbolic heaps at the end, which represents a disjunction:

$$\text{Inv} : \text{ls } \text{NE} \ x \ \text{nil} \quad \vee \quad x \mapsto \text{nil} \quad \vee \quad (x = \text{nil} \wedge \text{emp})$$

In fact, this disjunction is also (when we ignore variable d , which could be made local²) the invariant at the program point just inside the loop. We illustrate how this assertion is an invariant (fixed-point). Take the first disjunct, $\text{ls } \text{NE} \ x$. When we symbolically execute the body of the loop starting from this assertion we get a post-condition

$$\text{Post} : x \mapsto x' * \text{ls } \text{NE} \ x' \ \text{nil}$$

where x is the newly allocated location and x' is the old value of x . At this point we perform *abstraction*, which uses true implications in separation logic to simplify formulae. The particular abstraction rules from [9] swallow up the primed (existentially quantified) variables, and merge points-to facts into list segments, as in

$$x \mapsto x' * \text{ls } \text{NE} \ x' \ \text{nil} \implies \text{ls } \text{NE} \ x \ \text{nil}$$

When we simplify Post using this implication as a rewrite rule, it leads back into the invariant Inv that we started with. (In a similar way, starting from the other disjuncts leads back to the invariant.)

Returning to the formal development, we define SH to be the set of all symbolic heaps. The abstract domain of symbolic heaps comes with two operators:

$$\text{abs} : \text{SH} \rightarrow \text{SH} \quad \text{rearr}(e) : \text{SH} \rightarrow \mathcal{P}_{\text{fin}}(\text{SH}[e] \cup \{\text{err}\})$$

where $\text{SH}[e]$ denotes the set of heaps which contain $(e \mapsto e')$ for some e' . The first operator abs overapproximates a given symbolic heap, by throwing away the length information of linked lists in the heap. For instance, when abs is applied to $(x \mapsto x') * (x' \mapsto \text{nil})$ and $(x \mapsto x') * \text{ls } \text{NE} \ x' \ \text{nil}$, which describe linked lists of size two and at least two, it returns $\text{ls } \text{NE} \ x \ \text{nil}$, which means lists of size at least one. Note that while applying abs , we lost the length information in both symbolic heaps. This abs function is used in the separation-logic based analyses to enable fixed-point convergence (discovery of invariants). The next operator $\text{rearr}(e)$ does, if possible, case splitting of a given symbolic heap, so as to expose cell e explicitly with the points-to predicate. For instance, $\text{rearr}(x)(\text{ls } \text{NE} \ x \ \text{nil})$ returns $\{x \mapsto \text{nil}, (x \mapsto x') * \text{ls } \text{NE} \ x' \ \text{nil}\}$. Note that both elements in the result set contain the points-to fact about x explicitly, and they together overapproximate the given symbolic heap. Operator $\text{rearr}(e)$ can fail to expose $(e \mapsto -)$ from a given symbolic heap q (possibly because q does not ensure the allocatedness of e). The alternative output $\{\text{err}\}$ is used in such cases. We refer to [9, 3] for further details of abs and $\text{rearr}(e)$.

The abstract transfer function for atomic commands a (except the procedure call) is a function of type

$$[a] : \text{SH} \rightarrow \mathcal{P}_{\text{fin}}(\text{SH}).$$

²In fact, our implemented analysis uses a pre-pass which performs a live variable analysis, which has the effect of telling us that variables like d in this example can be existentially quantified at the end of the loop.

It is defined by the sequential composition of `rearr` (if a accesses a heap cell), a -specific transformation (coming from proof rules in separation logic) and `abs`. We give three example transfer functions for heap-cell update, disposal and allocation here; the abstract transfer functions for the other atomic commands can be found in [9, 3].

$$\begin{aligned} \llbracket [e] := e_0 \rrbracket (q) &= \\ & \{ \text{abs}(\Pi \wedge (e \mapsto e_0) * \Sigma) \mid \Pi \wedge (e \mapsto e') * \Sigma \in \text{rearr}(e)(q) \} \\ & \cup \{ \text{err} \mid \text{err} \in \text{rearr}(e)(q) \} \\ \llbracket \text{free}(e) \rrbracket (q) &= \\ & \{ \text{abs}(\Pi \wedge \Sigma) \mid \Pi \wedge (e \mapsto e') * \Sigma \in \text{rearr}(e)(q) \} \\ & \cup \{ \text{err} \mid \text{err} \in \text{rearr}(e)(q) \} \\ \llbracket [x := \text{new}()] \rrbracket (q) &= \\ & \{ \text{abs}(\Pi[x'/x] \wedge (x \mapsto y') * (\Sigma[x'/x])) \mid \Pi \wedge \Sigma \in q \} \\ & \quad (x', y' \text{ are fresh primed vars}) \end{aligned}$$

Our notion of symbolic heaps is slightly different from what have been studied in previous work on separation-logic-based program analyses. First, it includes two forms of list segment predicates $\text{lsPE } e \ e_0$ and $\text{lsNE } e \ e_0$, so that it expresses possibly empty linked lists directly and distinguishes such lists from non-empty lists. Although small, this change has a noticeable impact on the performance and precision of our analysis. Just keeping lsPE often leads to some false alarms, because it loses the information about the non-emptiness of the list.³ On the other hand, if we only had lsNE then the analysis would keep too many disjuncts, which affects performance. We will provide experimental results clarifying the impact of these decisions in Tables 1 and 3. Finally, technically, a symbolic heap contains non-formula `err`. The concretization of `err` is the singleton set of the error state, and it allows us to avoid unnecessary modification of the standard interprocedural analysis, in order to deal with `err`.

2.3 Interprocedural Analysis

Our shape analyzer is based on the RHS interprocedural analysis algorithm [22]. Our presentation of RHS is non-standard; it is a set-based presentation that makes it easier to explain further optimizations used in our analyzer, such as the use of the (partial) join operators. Throughout the section, we assume a fixed control flow graph $G = \{(N_f, E_f, L_f, s_f, e_f)\}_{f \in F}$.

The goal of RHS is to compute an overapproximation of reachable states at each node of a control flow graph, such that the overapproximation does not consider impossible execution paths (such as one call to f and two consecutive returns from f) and that the analysis avoids repeating the same computation. To achieve this goal, RHS uses the abstract domain \mathcal{A} defined by

$$\mathcal{A} \stackrel{\text{def}}{=} N \rightarrow \mathcal{P}_{\text{fin}}(\text{SH} \times \text{SH}), \text{ ordered pointwise.}$$

Note that for each node n we associate a set of symbolic heap pairs, instead of a set of symbolic heaps. The first component q of a pair (q, q_0) at n describes an input state of the procedure that contains n , and the second component q_0 denotes a resulting state at node n of the same procedure. Formally, this means that the concretization γ of A is given by

$$\gamma(A) \stackrel{\text{def}}{=} \lambda n \in N. \bigcup \{ \gamma(q) \times \gamma(q_0) \mid (q, q_0) \in A \}.$$

³ Some non-emptiness information can be encoded with disequalities, but it is difficult to deal with disequalities, since they might involve primed variables (i.e., existentially quantified variables): a naive approach can slow down the analysis or make the analysis diverge.

Here $\gamma(q)$ is the concretization of symbolic heap q defined by the semantics of separation logic formulae [9], and means a set of concrete states or the error state.

RHS is an iterative fixpoint algorithm with a function K of type $\mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A} \times \mathcal{A}$. The function K takes (A, B) where A describes the current analysis result at each node of the control flow graph G and B defines, at each node of G , the newly obtained symbolic heap pairs in the previous fixpoint iteration. K updates the analysis result A and the increment B by abstractly running all commands in the control flow graph once. Formally, we define K as follows:

$$\begin{aligned} K &: \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A} \times \mathcal{A} \\ K(A, B) &\stackrel{\text{def}}{=} (A', B') \\ A'(s_f) &\stackrel{\text{def}}{=} A(s_f) \cup \{ (q_0, q_0) \mid m \in \text{CallSite}(f) \wedge (q, q_0) \in B(m) \} \\ A'(n) &\stackrel{\text{def}}{=} A(n) \cup \bigcup_{(m,n) \in E} \llbracket L(m) \rrbracket_{(A,B)}(A(m), B(m)) \\ B'(n) &\stackrel{\text{def}}{=} A'(n) - A(n) \end{aligned}$$

Here $\text{CallSite}(f)$ is the set of nodes that call f , and $\llbracket a \rrbracket$ is the lifted abstract transfer function:

$$\llbracket a \rrbracket_{(A,B)}(X, Y) \stackrel{\text{def}}{=} \{ (q, q_1) \mid (q, q_0) \in Y \wedge q_1 \in \llbracket a \rrbracket(q_0) \} \quad (\text{when } a \text{ is not a procedure call})$$

$$\begin{aligned} \llbracket f() \rrbracket_{(A,B)}(X, Y) &\stackrel{\text{def}}{=} \{ (q, q_1) \mid (q, q_0) \in X \wedge (q_0, q_1) \in B(e_f) \} \\ & \cup \{ (q, q_1) \mid (q, q_0) \in Y \wedge (q_0, q_1) \in A(e_f) \}. \end{aligned}$$

Intuitively, function K updates the current analysis result A , by running commands in the control flow graph once with newly obtained analysis results. Note that function K treats the start nodes and non-start nodes differently. For the start nodes s_f of procedures, the analysis collects new inputs q_0 to the procedure f and adds pairs (q_0, q_0) to $A(s_f)$. For non-start nodes n , the analysis abstractly runs commands in n 's predecessors m with the previous analysis results $A(m)$ or increments $B(m)$ at those predecessors, and adds the resulting sets to $A(n)$.

The most interesting part of RHS is the abstract run of a procedure call $\llbracket f() \rrbracket$. The issue here is that the fixpoint iteration gives not just new inputs to f but also new input-output pairs of f . To incorporate these two kinds of new information correctly, $\llbracket f() \rrbracket$ takes the union of two sets; the first is the result of applying the newly obtained input-output pairs $B(e_f)$ to the old inputs X , and the second comes from applying the old input-output pairs $A(e_f)$ to the new inputs Y .

Given a finite set Q_0 of input symbolic heaps to `main`,⁴ the RHS algorithm iteratively generates the below sequence $\{(A_k, B_k)\}_k$

$$\begin{aligned} A_0(n) &\stackrel{\text{def}}{=} \text{if } (n = \text{main}) \text{ then } \{ (q, q) \mid q \in Q_0 \} \text{ else } \{ \} \\ B_0(n) &\stackrel{\text{def}}{=} A_0(n) \quad (A_{k+1}, B_{k+1}) \stackrel{\text{def}}{=} K(A_k, B_k), \end{aligned}$$

until it finds an index k with $k > 0$ and $B_k = \emptyset$; A_k with the first such index k is the result of RHS.

The correctness of RHS comes from the fact that it computes a post-fix point of the following function K^s for a standard but inefficient interprocedural analysis:

$$\begin{aligned} K^s &: \mathcal{A} \rightarrow \mathcal{A} \\ K^s(A)(s_f) &\stackrel{\text{def}}{=} A(s_f) \cup \{ (q_0, q_0) \mid m \in \text{CallSite}(f) \wedge (q, q_0) \in A(m) \} \\ K^s(A)(n) &\stackrel{\text{def}}{=} A(n) \cup \bigcup_{(m,n) \in E} \llbracket L(m) \rrbracket_A^s(A(m)) \end{aligned}$$

where $\llbracket a \rrbracket^s$ is defined below:

$$\begin{aligned} \llbracket a \rrbracket_A^s X &\stackrel{\text{def}}{=} \{ (q, q_1) \mid (q, q_0) \in X \wedge q_1 \in \llbracket a \rrbracket(q_0) \} \quad (\text{when } a \neq f()) \\ \llbracket f() \rrbracket_A^s X &\stackrel{\text{def}}{=} \{ (q, q_1) \mid (q, q_0) \in X \wedge (q_0, q_1) \in A(e_f) \}. \end{aligned}$$

⁴ In our experiments, we used the singleton set $\{\text{emp}\}$ for Q_0 .

Program	NO JOIN		JOIN	
	NE	PE	NE	PE
<code>onelist_create.c</code>	3	3	2	1
<code>twolist_create.c</code>	9	9	4	1
<code>firewire_create.c</code>	3969	3087	37	1

Table 1. Creation routines. Reports the number of states in the postcondition with join turned on and turned off, and the base list predicates chosen to be either nonempty ls only (NE), or both nonempty and possibly empty ls (PE).

THEOREM 1. *The result A of RHS satisfies: $\gamma(K^s(A)) \subseteq \gamma(A)$ and $\gamma(A_0) \subseteq \gamma(A)$.*

3. Join

We now consider the first of our optimizations, the join operator. We begin this section with an example that illustrates and motivates our operator, then move on to the formal definition of the operator itself, and finally show how it can be integrated with the RHS algorithm, thus updating the analysis described in the previous section.

3.1 Illustration and Intuition

A join operator takes two symbolic states in a program analysis and attempts to find a common generalization.

Recall the program `onelist_create` from Section 2.2. It nondeterministically creates an infinite collection of linked lists, which the analysis algorithm overapproximates with the three assertions

$$\text{ls NE } x \text{ nil} \quad \vee \quad x \mapsto \text{nil} \quad \vee \quad (x = \text{nil} \wedge \text{emp})$$

Now, if you look at the first two disjuncts there is evident redundancy: If you know that either x points to nil or a nonempty linked list, then that is the same as knowing you have a nonempty linked list. So, our join replaces the first two disjuncts with just the list segment formula, giving us

$$\text{ls NE } x \text{ nil} \quad \vee \quad (x = \text{nil} \wedge \text{emp})$$

It is possible to take yet a further step, using the notion of a possibly empty list segment. If you know that either you have a nonempty list, or that $x = \text{nil} \wedge \text{emp}$, then that is the same as having

$$\text{ls PE } x \text{ nil}$$

and our join operator produces this formula from the previous two.

Thus, using join we have gone from a position where we have three disjuncts in our postcondition, to where we have only one. The saving that this possibly gives us is substantial, especially for more complicated programs or more complicated data structures. Table 1 gives an indication. `onelist_create.c` in the table is the C program that corresponds to the already referred `onelist_create`, and `twolist_create.c` is a similar C program that creates two disjoint linked lists. `firewire_create.c` is the environment code we use in the analysis of the 1394 firewire driver: it creates five cyclic linked lists, which share a common header node, with head pointers in some of the lists, and nested sublists.

There are two points to note about the table. The first is just the great saving, in number of states, given by join (e.g., from 3087 down to 1). This is particularly important with environment code, like `firewire_create.c`, which is run as a harness to generate heaps on which driver routines will subsequently be run. The second is the distinction between NE and PE. In the table we keep track of two versions of our analysis, one where ls NE is the only list predicate used by the analysis, and another where we use both ls NE and

ls PE. Having both PE and NE gives significant further reduction; we will return to this point in Section 5.

This illustration shows some of the aspects of our join operator, but not all. In the illustration join worked perfectly, never losing any information, but this is not always the case. Part of the intuition is that you generalize points-to facts by list segments when you can. So, considering

$$y \mapsto \text{nil} * (\text{ls NE } x \text{ nil}) \quad \vee \quad (\text{ls NE } y \text{ nil}) * x \mapsto \text{nil}$$

our join will produce

$$(\text{ls NE } y \text{ nil}) * (\text{ls NE } x \text{ nil}).$$

This formula is less precise than the disjunction, in that it loses the information that one or the other of the lists pointed to by x and y has length precisely 1. (Fortunately, it is unusual for programs to rely on this sort of disjunctive information.)

We have tried to keep the intuitive description simple, but the truth is that the join must deal with disequalities, equalities, and generalization of “nothing” by ls PE in ways that are nontrivial. It also must deal with the existential (primed) variables specially. In the end, for instance, when our join is given

$$\begin{aligned} q_0 &\stackrel{\text{def}}{=} x \neq y \wedge x' \neq y \wedge (\text{ls NE } x x' * y \mapsto x') \quad \text{and} \\ q_1 &\stackrel{\text{def}}{=} x \neq y \wedge z' \neq y \wedge (x \mapsto z' * \text{ls NE } y z' * \text{ls NE } z' y'), \end{aligned}$$

it will produce

$$x \neq y \wedge \text{ls NE } x v' * \text{ls NE } y v' * \text{ls PE } v' w'.$$

Now we turn to the formal definition.

3.2 Formal Definition

We now define the (partial) binary operator `pjoin` on symbolic heaps. To specify it we use a predicate `pjoin_sg`($\Sigma_0, \Sigma_1, \Sigma$), which signifies that Σ_0 and Σ_1 can be joined to give Σ . Note that it is concerned with the spatial parts of symbolic heaps only. The definition of `pjoin_sg` is parameterized by another ternary predicate ϵ . Intuitively, $\epsilon(e_0, e_1, e)$ holds when e_0, e_1 are expressions from the first and second arguments to the join and e is an expression in the result of the join, and it means that e_0, e_1, e are considered the same during the application of the join operator. For instance, given two symbolic heaps

$$(y \mapsto z' * \text{ls NE } z' y') \quad \text{and} \quad (\text{ls NE } y x')$$

the `pjoin` operator constructs (and uses)

$$\epsilon = \{(y, y, y), (z', x', v'), (y', x', w')\},$$

which describes that the primed variables z', y' in the first symbolic heap correspond to x' in the second heap and they are renamed to v', w' in the result of the join. The ϵ relation also says that y relates to itself and is not renamed in the result of the join.

Predicate `pjoin_sg` is defined by the following rules.

$$PE \sqcup NE = NE \sqcup PE = PE \sqcup PE = PE \quad NE \sqcup NE = NE$$

$$\frac{}{\text{pjoin_sg}_\epsilon(\text{emp}, \text{emp}, \text{emp})}$$

$$\frac{\text{pjoin_sg}_\epsilon(\Sigma_0, \Sigma_1, \Sigma)}{\text{pjoin_sg}_\epsilon(\text{true} * \Sigma_0, \text{true} * \Sigma_1, \text{true} * \Sigma)}$$

$$\frac{\epsilon(e_0, e_1, e) \quad \epsilon(e'_0, e'_1, e') \quad k = k_0 \sqcup k_1 \quad \text{pjoin_sg}_\epsilon(\Sigma_0, \Sigma_1, \Sigma)}{\text{pjoin_sg}_\epsilon(\text{ls } k_0 \ e_0 \ e'_0 * \Sigma_0, \text{ls } k_1 \ e_1 \ e'_1 * \Sigma_1, \text{ls } k \ e \ e' * \Sigma)}$$

$$\frac{\epsilon(e_0, e_1, e) \quad \epsilon(e'_0, e'_1, e') \quad \text{pjoin_sg}_\epsilon(\Sigma_0, \Sigma_1, \Sigma)}{\text{pjoin_sg}_\epsilon(\text{ls } k_0 \ e_0 \ e'_0 * \Sigma_0, (e_1 \mapsto e'_1) * \Sigma_1, \text{ls } k_0 \ e \ e' * \Sigma)}$$

$$\frac{\exists e_1. \epsilon(e_0, e_1, e) \quad \epsilon(e'_0, e_1, e') \quad e_1 \notin \text{LHS}(\Sigma_1) \quad \text{pjoin_sg}_\epsilon(\Sigma_0, \Sigma_1, \Sigma)}{\text{pjoin_sg}_\epsilon(\text{ls } k_0 \ e_0 \ e'_0 * \Sigma_0, \Sigma_1, \text{ls } PE \ e \ e' * \Sigma)}$$

Here $\text{LHS}(\Sigma)$ is a set of expressions that appear in the left hand side of the `pointsto` predicate or as a first expression argument of `ls`. In the above, we omitted the symmetric versions of the last two rules. The third rule has to do with generalizing two lists, the fourth abstracts a `pointsto` by a list, and the last is about generalizing (or synthesizing) possibly empty lists.

The (partial) join operator `pjoin` ($\Pi_0 \wedge \Sigma_0, \Pi_1 \wedge \Sigma_1$) first searches for witnesses Σ, ϵ of

$$\text{pjoin}_\epsilon(\Sigma_0, \Sigma_1, \Sigma)$$

where ϵ relates non-identical expressions (e_0, e_1, e) only when e is a primed variable and at least one of e_0 and e_1 is a primed variable.⁵ If this search does not succeed, `pjoin` is undefined. Otherwise, it returns $\Pi \wedge \Sigma$ where Π is

$$\bigwedge \left(\begin{array}{l} \{e=e' \mid e=e' \text{ has no primed vars, it occurs in } \Pi_0 \text{ and } \Pi_1\} \\ \cup \{e \neq e' \mid e \neq e' \text{ has no primed vars, it occurs in } \Pi_0 \text{ and } \Pi_1\} \end{array} \right).$$

3.3 Extending RHS with Join

The extension of the RHS with `pjoin` changes the definition of K in the previous section by replacing a union by join. Let N_c be the set of control-flow-graph nodes with incoming backedges,⁶ which correspond to loops of a program. If n is an exit node, or a node in N_c , or a call node (i.e., $L(n) = f()$),

$$A'(n) \stackrel{\text{def}}{=} \text{pjoin}^\ddagger(A(n), \bigcup_{(m,n) \in E} \llbracket L(m) \rrbracket_{(A,B)}(A(m), B(m))).$$

Intuitively, the effect of this is as follows. Given a new addition to a procedure summary, join is applied to simplify the disjunctions in the addition, and then it is applied to combine the simplified new and the old elements in the post-part of the summary. Additionally, join is applied before a procedure is called (before the summary concept is considered) to reduce the number of inputs to the procedure, and it is also applied at the end of while loops.

The operator `pjoin`[‡] in the redefinition of $A'(n)$ is an operator on sets of symbolic heap pairs (in keeping with the types of the operators used in RHS), defined in terms of `pjoin` and an auxiliary

⁵ In order to prevent the loss of precision, the implementation of `pjoin` ensures that when ϵ is restricted to primed variables only, $\{(e_0, e_1) \mid \exists e. \epsilon(e_0, e_1, e)\}$ defines a partial isomorphism except for the following case: if `ls k x' y'` appears in Σ_0 or Σ_1 , x' and y' can relate to the same primed variable.

⁶ When control flow graphs are traversed from starting nodes of procedures by depth first traversal, we get a collection of traversing trees, together with additional edges. Backedges are the ones that go from nodes to their ancestors in these trees.

```

R1 := ∅;
while (Q1 ≠ ∅) do
  choose and remove q1 from Q1;
  q := q1; Q'1 := Q1;
  while (Q'1 ≠ ∅) do
    choose and remove q'1 from Q'1;
    if (pjoin(q, q'1) is defined and equals q')
      then (Q1 := Q1 - {q1}; q := q')
  od;
  R1 := R1 ∪ {q}
od;
R := ∅;
while (R1 ≠ ∅) do
  choose and remove q1 from R1;
  q := q1; Q'0 := Q0;
  while (Q'0 ≠ ∅) do
    choose and remove q'0 from Q'0;
    if (pjoin(q, q'0) is defined and equals q')
      then (Q0 := Q0 - {q'0}; q := q')
  od;
  R := R ∪ {q}
od;
return (Q0 ∪ R)

```

Figure 1. Definition of `pjoin`[†](Q_0, Q_1)

operator `pjoin`[†] defined on sets of symbolic heaps. `pjoin`[†] takes two finite sets Q_0, Q_1 of symbolic heaps, and abstracts $Q_0 \cup Q_1$ by repeatedly applying `pjoin` first to two elements of Q_1 and then to two elements chosen from Q_0 and Q_1 . The precise definition of `pjoin`[†] is given in Fig. 1. The operator `pjoin`[‡] is defined using `pjoin`[†]:

$$\text{pjoin}^\ddagger : \mathcal{P}_{\text{fin}}(\text{SH} \times \text{SH}) \times \mathcal{P}_{\text{fin}}(\text{SH} \times \text{SH}) \rightarrow \mathcal{P}_{\text{fin}}(\text{SH} \times \text{SH})$$

$$\text{pjoin}^\ddagger(A, B) \stackrel{\text{def}}{=} \bigcup_{q \in \text{dom}(A) \cup \text{dom}(B)} (\{q\} \times \text{pjoin}^\dagger(\text{rng}(A, q), \text{rng}(B, q)))$$

where $\text{dom}(A)$ is $\{q \mid \exists q_0. (q, q_0) \in A\}$ and $\text{rng}(A, q)$ is $\{q_0 \mid (q, q_0) \in A\}$.

THEOREM 2. *When A is the result of RHS extended with `pjoin`, we have that $\gamma(K^s(A)) \subseteq \gamma(A)$ and $\gamma(A_0) \subseteq \gamma(A)$, where K^s is the functional from the standard interprocedural analysis in Sec. 2.3.*

4. Localization

The next optimization is localization. The basic idea is to pass only the relevant part of each input symbolic heap q , when q is given as an input to a procedure. For instance, when a procedure f that disposes a linked list x is called with the symbolic heap

$$\text{ls } NE \ x \ \text{nil} * \text{ls } NE \ y \ \text{nil},$$

the localization optimization passes only `ls NE x nil` to f and later combines the result `emp` of the procedure with the remaining part `ls NE y nil`. In this way, the localization reduces the amount of time analyzing f by passing smaller symbolic heaps, and also helps f to compute more general summaries in $A(e_f)$; the summary $(\text{ls } NE \ x \ \text{nil}, \text{emp})$ can be used also for an input abstract state `ls NE x nil * z ↦ nil`.

Formally, we assume two operators that do the splitting and combining of symbolic heaps:

$$\text{comb} : F \rightarrow \text{SH} \times \text{SH} \rightarrow \text{SH} \quad \text{split} : F \rightarrow \text{SH} \rightarrow \text{SH} \times \text{SH}.$$

Both operators are parameterized by callee f . Given callee f , `split` splits a symbolic heap q into (q_f, q_0) where q_f is the part of q

relevant to callee f and q_0 is the remainder of q . Operator `comb` does the opposite, and combines two symbolic heaps q'_f and q'_0 into one symbolic heap. Intuitively, `split` computes the relevant portion of q by carving out all the unreachable parts of q from variables used in f or procedures called by f . Operator `comb` first renames primed variables in the two symbolic heaps, and then puts the resulting symbolic heaps together by conjoining their Σ -parts with $*$ and their Π -parts with \wedge .

In order to incorporate localization to our analysis, we change two places in the analysis that involve the call and return of procedures. First, we change the definition of A' for all start nodes of procedures:

$$A'(\mathfrak{s}_f) \stackrel{\text{def}}{=} A(\mathfrak{s}_f) \cup \{(q'_0, q'_0) \mid m \in \text{CallSite}(f) \wedge (q, q_0) \in B(m) \wedge \text{split}_f(q_0) = (q'_0, q_1)\}.$$

Note that the new definition passes only the relevant part q'_0 of the symbolic heap q_0 at a call site, which is often much smaller than q_0 itself. Next, we change the definition of $\llbracket f() \rrbracket$ in order to put back the split-out portions of procedure inputs:

$$\llbracket f() \rrbracket_{(A,B)}(X, Y) \stackrel{\text{def}}{=} \{(q, \text{comb}_f(q_1, q''_0)) \mid ((q, q_0) \in X \wedge \text{split}_f(q_0) = (q'_0, q''_0) \wedge (q'_0, q_1) \in B(e_f)) \vee ((q, q_0) \in Y \wedge \text{split}_f(q_0) = (q'_0, q'_0) \wedge (q'_0, q_1) \in A(e_f))\}$$

The correctness of the new analysis comes from the fact that it computes a post-fix point of the following function K^l :

$$K^l(A)(\mathfrak{s}_f) \stackrel{\text{def}}{=} A(\mathfrak{s}_f) \cup \{(q'_0, q'_0) \mid m \in \text{CallSite}(f) \wedge (q, q_0) \in A(m) \wedge \text{split}_f(q_0) = (q'_0, q_1)\}$$

$$K^l(A)(n) \stackrel{\text{def}}{=} A(n) \cup \bigcup_{(m,n) \in E} \llbracket L(m) \rrbracket_A^l(A(m))$$

where $\llbracket a \rrbracket^l$ is defined below:

$$\begin{aligned} \llbracket a \rrbracket_A^l X &\stackrel{\text{def}}{=} \{(q, q_1) \mid (q, q_0) \in X \wedge q_1 \in \llbracket a \rrbracket(q_0)\} \text{ (when } a \neq f()) \\ \llbracket f() \rrbracket_A^l X &\stackrel{\text{def}}{=} \{(q, \text{comb}_f(q_1, q''_0)) \mid (q, q_0) \in X \wedge \text{split}_f(q_0) = (q'_0, q'_0) \wedge (q'_0, q_1) \in A(e_f)\}. \end{aligned}$$

THEOREM 3. *If A is the result of RHS extended with `pjoin` and localization, then $\gamma(K^l(A)) \subseteq \gamma(A)$ and $\gamma(A_0) \subseteq \gamma(A)$.*

We make two further remarks. First, the correctness of K^l itself can be given by slightly modifying the correctness argument given in [10]. Next, the localization involves the so-called issue of cutpoints [23]. When q is split into q_f and q_0 , a primed variable x' appearing both in q_f and q_0 loses the connection. To see this, note that x' is implicitly existentially quantified. Initially, we have $\exists x'.q$ and later we have $\exists x'.q_f$ and $\exists x'.q_0$. Our implementation considers this issue, and uses the approach proposed in [10], which is to introduce auxiliary non-primed variables x_0 during splitting and conjoin equalities $x_0 = x'$ to the split heaps q_f and q_0 .

5. Experiments Involving Localization and Join

We have implemented our analysis in OCaml, using the CIL compiler infrastructure [19].

Our target in this paper was a collection of five device drivers. One, `t1394Diag.c`, is from the IEEE 1394 (firewire) driver for Windows, and the others were for Embedded Linux. We present our experimental results in a graded way, in an effort to show what some of the optimizations cannot do, as well as what they can. Our best results will be presented later, in Tables 4 and 5.

In our experiments timeout was set to 90 minutes. In practice, when we observed the memory consumption exceeding 2GB we would inexorably proceed to timeout: the experiments were run on an ordinary laptop with 2GB RAM, and exceeding this caused communication with the disk.

To begin, we sought to measure the effects of the join and localization optimizations, compared to our baseline analysis; see Table 2. In order to get measurements for our base analysis, we could not use an entire one of our drivers. So, we stripped out part of `t1394Diag.c`. Program `t1394Diag.PnpRemoveDevice.c` in Table 2 is a specific dispatch routine of the 1394 driver, and the similarly-named `t1394Diag.PnpRemoveDevice_cut.c` is a version of this program with enough code removed to get it past our base analysis. The LOC reported in both cases includes data structure definitions and environment code which nondeterministically creates input heaps to run the dispatch routine (or its fragment) on. In these cases the environment code comprised 92 LOC, and the rest was the dispatch routine or its fragment.

In the first line of the table we can see the significant time and space savings afforded by join, and the not insignificant time savings of locality. Line two shows how join is enough to verify the entire `t1394Diag.PnpRemoveDevice.c`, while locality (on its own) is not. However, for a larger program, `cdrom.c`, we find that join on its own is not sufficient, that *both* locality and join are needed.

The measurements in Table 2 were done with the option PE for possibly empty as well as nonempty list segments, rather than NE for nonempty only, because PE is so clearly superior. An indication that this is the case was provided earlier, in Table 1, and we further confirm this contention with experiments in Table 3. The difference is so great that it is a determining factor in being able to, or not being able to, verify one of the programs at all. An explanation for this difference is that the join with possibly empty list segments reduces the search space to a significantly larger degree than does the join for nonempty lists only.

The test programs in Table 3 are the device drivers that are the target of our main experiments. `t1394Diag.c` is from Windows and the others from Embedded Linux. `t1394Diag.c` is the entire driver, containing as a part the similarly-named programs in Table 2. The environment code used for the entire drivers is more complex than when analyzing one routine. A driver typically contains a collection of dispatch routines. The environment code works by nondeterministically generating candidate heaps for input, and nondeterministically calling the dispatch routines over and over.

Although the combination of join and locality allows us to verify our largest example, we are still in a situation where our analyzer fails to converge on two of our test programs, and this brings us to our next optimization.

6. Discarding Intermediate Results

The final optimization is to remove unnecessary intermediate results. The idea is to detect when RHS completes the computation of a procedure f and to remove all the intermediate results used to compute the outputs of f from its given inputs.

Let `proc` be a function that, given a node n in the control flow graph, returns the name of the procedure containing n . Say that a node n is reachable from procedure f when n can be reached from \mathfrak{s}_f by following edges in G or procedure calls. Define a predicate `done` that holds for a procedure f and an increment B of RHS if and only if $B(n')$ is empty for all nodes n' reachable from f .

Now, we incorporate the optimization by changing some of the defining clauses for A of K in our analysis. We insert the discarding clause like

$$A'(n) \stackrel{\text{def}}{=} \text{if } (\text{done}(\text{proc}(n), B)) \text{ then } \{\} \text{ else } \dots$$

when n is not a start or exit node of a procedure. We do not discard the analysis result at the exit node e_f , because it gives the current summary of function f , and we also keep the result at the start node \mathfrak{s}_f , because it prevents the analysis from re-analyzing the same

Program	LOC	No Join & No Locality (Sec)	No Join & No Locality (Mb)	Join & No Locality (Sec)	Join & No Locality (Mb)	No Join & Locality (Sec)	No Join & Locality (Mb)	Join & Locality (Sec)	Join & Locality (Mb)
t1394Diag_PnpRemoveDevice_cut.c	973	523.17	1133.69	0.64	3.69	184.87	575.82	0.36	2.70
t1394Diag_PnpRemoveDevice.c	1825	X	X	2.70	9.58	X	X	1.21	5.16
pci-driver.c	2532	X	X	1.75	9.09	X	X	0.55	4.42
cdrom.c	6218	X	X	X	X	X	X	107.91	357.58

Table 2. Experimental results on localization and join. Timeout (X) set at 90mins. Experiments run on Intel Core Duo 2Ghz with 2GB RAM. PE predicate base set. Results do not use further optimizations on intermediate states.

Program	LOC	Non Empty		Possibly Empty	
		(Sec)	(Mb)	(Sec)	(Mb)
pci-driver.c	2532	1.37	7.13	0.55	4.42
cdrom.c	6218	203.81	650.77	107.91	357.58
t1394Diag.c	10240	X	X	119.40	425.16
md.c	6635	X	X	X	X
ll_rw_blk.c	5469	X	X	X	X

Table 3. Experimental results on nonempty versus possibly empty list segments. Base list predicates chosen to either be nonempty ls only (NE), or both nonempty and possibly empty ls (PE). Timeout (X) set at 90mins. Experiments run on Intel Core Duo 2Ghz with 2GB RAM. Results use join but do not use further optimizations on intermediate states.

inputs. Formally, we change K as follows:

$$\begin{aligned}
A'(s_f) &\stackrel{def}{=} A(s_f) \cup \{(q_0, q_0) \mid m \in \text{CallSite}(f) \wedge (q, q_0) \in B(m)\} \\
A'(e_f) &\stackrel{def}{=} \text{pjoin}^\dagger(A(e_f), \bigsqcup_{(m, e_f) \in E} \langle L(m) \rangle_{A, B}(A(m), B(m))) \\
A'(n) &\stackrel{def}{=} \text{if done}(\text{proc}(n), B) \text{ then } \{ \\
&\quad \text{else } \text{pjoin}^\dagger(A(n), \bigsqcup_{(m, n) \in E} \langle L(m) \rangle_{A, B}(A(m), B(m))) \\
&\quad \text{(when } n \text{ is a node in } N_c \text{ or a call node)} \\
A'(n) &\stackrel{def}{=} \text{if done}(\text{proc}(n), B) \text{ then } \{ \\
&\quad \text{else } A(n) \sqcup \bigsqcup_{(m, n) \in E} \langle L(m) \rangle_{A, B}(A(m), B(m)) \\
&\quad \text{(otherwise)}
\end{aligned}$$

7. Experiments on Keeping and Discarding

We then re-ran our analysis on the drivers, with the setting of PE for possibly empty and nonempty lists and with the join and locality optimizations turned on; see Table 4. With the further optimization of discarding intermediate states, we were finally able to verify all of our drivers.

These verifications establish that programs do not dereference null or a dangling pointer, and that there are no memory leaks. In order to verify these properties, we had to fix bugs in the drivers, discovered in the course of analyzing them. We approached this in a stop-first way. If our analyzer encountered a memory safety or leak error, it would stop immediately. A potential memory leak is indicated if some points-to or ls predicates cannot be reached from program variables.⁷ A potential safety violation is indicated when a statement that dereferences or disposes e is run in a state where rearrangement is not able to reveal a points-to involving e . We would then fix the (potential) bug and run the analyzer again. This stop-first approach would not be desirable in a bug-catching application of program analysis, where one would like to return a list of the errors in a piece of code. For our aim of verification, it is more appropriate, and in any case sufficed for our experiments.

Our claim that the optimizations in our analysis do not lead to an unacceptable loss of precision is supported by the successful verifications of the fixed drivers. Also, the bugs that we fixed along the way were all genuine ones for the programs that were input to

⁷To do this reachability check, the analyzer views $\text{ls } kee'$ and $e \mapsto e'$ as edges from e to e' .

our analysis; in that sense, and in the sense of verification of the final programs, we found that our analyzer did not lead to false alarms.

Of course, these remarks should be understood in context. The bugs that we found were for the code with our environment model (a nondeterministic creation routine). That does not imply that the bugs would show up in the normal execution setting of the drivers, i.e., the Windows or Embedded Linux kernels. As with tools such as SLAM, determining whether or to what extent an environment model is suitably faithful to an OS kernel's view of the data structures is highly nontrivial.

We also remark that we have not solved the problem of analyzing arrays (indices as well as their contents) and pointer arithmetic in this paper. The analyzer assumes that all array accesses and heap accesses via pointer arithmetic are safe. Thus, the “verification” done by our analysis is relative, to the validity of this assumption in a given program. (That is why we have spoken, e.g., of integrity of pointer manipulation, rather than the more blanket “memory safety”.) We decide to make this assumption in our analysis because designing a precise, scalable analyzer for arrays and pointer arithmetic is an open question, and it is more sensible not to try to solve several challenging problems at the same time.

We performed one final experiment. Given the timeout observed for `md.c` and `ll_rw_blk.c` when states were not discarded, we wondered whether the analysis would terminate on those programs if we gave it more RAM. So, we re-ran those experiments on a (different) machine with more RAM; see Table 5. We were indeed able to observe convergence without discarding intermediate states, but the RAM required went beyond 2GB, which would have caused swapping between RAM and disk on our other experiments. In any case, the programs in Table 5 are at the edge of what our analysis can handle, when running on an ordinary laptop.

That our analyzer has more trouble with the smaller programs in Table 5 is not inconsistent. Like the firewall driver, those programs use several linked lists, together with nested lists. A difference is that the nesting is sometimes of a cyclic list. More importantly, though, `md.c` and `ll_rw_blk.c` use additional fields, that are not themselves used to tie up linked lists, in a way that makes it more difficult for our join operator to generalize.

A typical case is a field that always points either to `nil` or to a pointer that itself points immediately to `nil`: that is, a list

segment of length zero or one, and no longer. We could, soundly, generalize this with the `lsPE` predicate, but that would lead to a loss of precision that leads to false alarms when analyzing these drivers. What our analysis does instead is maintain this zero-or-one information precisely, and this introduces case distinctions which increase the number of abstract states generated. It might be possible to formulate a more powerful, but not too imprecise, join operator by considering a single predicate for list segments of length zero or one; that could possibly lead to a speedup in the analysis of these programs.

More importantly, though, these remarks just serve to underline our earlier remark that designing a good join is a balancing act, where experimentation is crucial. A possible direction for future work is to have methods for refining or adapting a join operator, by analogy with counterexample-guided abstraction refinement, and adaptive methods for shape analysis [16, 3, 27, 12].

8. Related Work

The field of shape analysis, and more generally of heap verification, has seen rapid growth in recent years. In this section we confine our comments to related analyses, excluding verification methods that require user-supplied loop invariants or procedure specifications.

There are two outstanding problems in the area of shape analysis. One concerns how one might best describe the states of an analysis. Many techniques are being proposed relevant to this question (e.g., [26, 1, 14, 21, 9, 15, 6, 28, 11]). As well as precision and expressiveness, this question concerns the effort a programmer or analysis designer would need to put in to press a shape analysis into service, and there have been works that seek to reduce this effort by adapting the analysis to the structures found in a program [16, 3, 27, 12].

The second problem is scalability. Shape analyses are expensive, and this is the principal impediment to them being used “for real”, for verification of substantial software. There has been comparatively little work on this second problem. The analysis in [13] has been applied to non-trivial code, but the abstract domain there is rather imprecise and not well suited to verification; it could not be used to verify memory safety of the device drivers that we consider. [15] considers the question of efficiency of transfer functions in 3-valued analyses. The abstract domain we consider here already has efficient transfer functions because it is based on [9], which leverages the local reasoning idea of [20]. The question we address concerns the efficiency of an overall analysis, rather than individual transfer functions, and limiting the number of abstract states tracked by the analysis is crucial for this.

On this note, [17] presents a novel abstract domain which cuts down on the number of abstract states, in a different way from that here. The largest example reported in [17] comprises 278 LOC, (also from the IEEE 1394 driver and thus not yet approaching the experiments achieved here). However, the savings compared to their base analysis are significant, and the abstract domain intriguing; it will be interesting to see if the ideas can help with more substantial programs.

Another way to increase performance is to use a slicer, that removes shape-irrelevant statements from a program [12]. In case there are many such irrelevant statements, this can make a significant difference. In this paper, we are concerned with the problem of how to limit the number of states that need to be considered, for the relevant statements. In our most advanced examples, it seems difficult to soundly slice away much of the code. But, it would certainly make sense to use slicing and state-space reduction methods, together.

In our experiments we fixed our abstract domain, in order to separate out scalability from the first of the open problems mentioned above. The domain is a variant of the one in [3]. It deals

with composite, possibly nested, variations on liked lists, but does not deal with trees and other non-linear structures. We expect that the improvements in this paper do not depend wholly on the particular abstract domain, that analogous results could be obtained for other domains. We warn, though, about the non-orthogonal nature of the question of what is a good join and the question of what base predicates (or summary nodes) an analysis uses, as we illustrated in Tables 1 and 3.

The general lines of our attack exploit two existing general ideas, join and local reasoning. In the case of shape analyses, several join operators have been proposed in the literature [18, 14, 7]. Our join is related to these, but its detailed formulation was also driven by the problem of verifying device driver programs, and this problem caused many variations to be tried. A difference with these other works is the way that we distinguished possibly empty and nonempty lists; this choice was determined by pragmatic considerations, and turned out to be performance-critical. We refer again to the gains illustrated in numbers of states (Table 1) and the time and space speedups in our other experiments. The previous join operators in shape analysis have not reported gains comparable to these here.

This being said, neither we nor the referenced works can claim to have found the ultimate join for shape domains. Further work is needed to better understand the tradeoffs involved.

The local reasoning idea stems from work surrounding the frame rule in separation logic [20]. We are not the first to exploit this idea in automatic verification. [4] uses localization to obtain efficient transfer functions for primitive operations in an abstract domain based on separation logic, and [17] uses the idea in a graph-based analysis. [23] and [10] use the idea in interprocedural analysis, with [23] contributing a key heuristic based on reachability. In common with these works our results suggest that the local reasoning idea can help with scalability. But, as we mentioned in the introduction, an additional contribution of this paper is the identification of limits to its positive effect, as is evident from Table 2.

The RHS algorithm was previously used to very good effect in SLAM. We are not aware of published modifications to RHS for join and for discarding intermediate states. In implementation, our modifications required a significantly different treatment of the working set, for which care is required, and which motivated our alternate formulation.

9. Conclusions

Scalability is a central problem in shape analysis, critical to its potential wider use in verification of real-world code. The main contribution of this paper is the demonstration that it is possible to scale a shape analysis in a way that leads to an order of magnitude increase in the size of program it can handle (holding the complexity of the data structures constant). This involved a mixture of reuse of existing ideas (materialization and abstraction, RHS, local reasoning, a shape domain for composite structures), several optimizations to RHS (supported by an alternate formulation), and, most importantly, a new join operator that greatly reduces the state space of the analysis. The crucial part of our contribution is our experimental validation involving verification of five device drivers.

No previous shape analysis has reported experimental results approaching our verification of the safety of pointer manipulation in several entire device drivers in the thousands LOC, including one of 10,000 LOC. The closest related results are those of [7, 3]. [7] analyzed the Linux scull driver, which has 879 LOC, and the analysis was done with manually given preconditions rather than nondeterministic environment code as here. [3] analyzed several 1394 driver routines of around 700 LOC, with environment code included. It timed out on a routine of 1800 LOC when run with

Program	LOC	Keep		Discard		Bugs Found	
		(Sec)	(Mb)	(Sec)	(Mb)	Leaks	Safety
pci-driver.c	2532	0.55	4.42	0.75	3.19	0	0
cdrom.c	6218	107.91	357.58	91.45	84.79	0	2
t1394Diag.c	10240	119.40	425.16	137.78	73.24	33	10
md.c	6635	X	X	1819.53	1010.81	6	5
ll_rw_blk.c	5469	X	X	947.20	511.43	3	1

Table 4. Keep vs Discard run on Intel Core Duo 2Ghz with 2GB.

Program	LOC	Keep		Discard	
		(Sec)	(Mb)	(Sec)	(Mb)
md.c	6635	2491.90	3146.47	2022.83	1027.52
ll_rw_blk.c	5469	1594.86	2257.80	1007.38	500.12

Table 5. Keep vs Discard on a Dual-Core Opteron 2.2Ghz with 4GB.

environment code, and terminated when the routine was supplied with a particular precondition.

In addition to being able to verify larger pieces of code than before, we observed significant time and space reductions on code that could be analyzed before. On our starting program (Table 2) we saw a reduction in running time from 523sec to 0.36sec and space reduction from >1GB to 5MB. (Even with these reductions, our most difficult example, md.c, still requires >1GB in space.)

As we mentioned above, many interesting techniques are being advanced in the field of shape analysis, and more generally in heap verification, and we hope that other ideas can lead to further improvements in scalability. In particular, a better understanding of the tradeoffs surrounding join operators for shape domains might lead to better control over the space requirements of these analyses.

References

- [1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, volume 3385 of *LNCS*, pages 164–180. Springer, 2005.
- [2] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.
- [3] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O’Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *CAV 2007*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.
- [4] J. Berdine, C. Calcagno, and P. O’Hearn. Symbolic execution with separation logic. In *3rd APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
- [5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.
- [6] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *SAS 2006.*, volume 4134 of *LNCS*, pages 52–70. Springer, 2006.
- [7] B. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *14th SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.
- [8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [9] D. Distefano, P. W. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
- [10] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *13th SAS*, volume 4134 of *LNCS*, pages 240–260, 2006.
- [11] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, volume 4590 of *LNCS*, pages 379–392. Springer, 2007.
- [12] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. *PLDI'07*. pp. 256-265.
- [13] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323. ACM, 2005.
- [14] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, volume 3444 of *LNCS*, pages 124–140. Springer, 2005.
- [15] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, volume 4144 of *LNCS*, pages 547–561. Springer, 2006.
- [16] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *CAV 2005*, volume 3576 of *LNCS*, pages 519–533. Springer, 2005.
- [17] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *13th TACAS*, volume 4424 of *LNCS*, pages 3–18. Springer, 2007.
- [18] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *11th SAS*, volume 3148 of *LNCS*, pages 265–279. Springer, 2004.
- [19] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL:intermediate language and tools for analysis and transformation of C programs. In *11th CC*, 2002. pp213-228.
- [20] P. O’Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL 2001*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
- [21] A. Podelski and T. Wies. Boolean heaps. In *SAS*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.
- [22] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *232nd POPL*, pages 49–61, 1995.
- [23] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *32nd POPL*, pp296–309, 2005.
- [24] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th SAS*, volume 3672 of *LNCS*, pages 284–302. Springer, 2005.
- [25] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.
- [26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
- [27] M. Češka, P. Erlebach, and T. Vojnar. Generalised multi-pattern-based

verification of programs with linear linked structures. *Formal Aspects Comput.*, 2007.

- [28] V.Kuncak and M.C.Rinard. An overview of the Jahob analysis system: project goals and current status. In *IPDPS*, 2006.