Automatic Inference of Frame Axioms Using Static Analysis*

Zvonimir Rakamarić, Alan J. Hu
Department of Computer Science, University of British Columbia, {zrakamar,ajh}@cs.ubc.ca

Abstract

Many approaches to software verification are currently semi-automatic: a human must provide key logical insights — e.g., loop invariants, class invariants, and frame axioms that limit the scope of changes that must be analyzed.

This paper describes a technique for automatically inferring frame axioms of procedures and loops using static analysis. The technique builds on a pointer analysis that generates limited information about all data structures in the heap. Our technique uses that information to over-approximate a potentially unbounded set of memory locations modified by each procedure/loop; this over-approximation is a candidate frame axiom.

We have tested this approach on the buffer-overflow benchmarks from ASE 2007. With manually provided specifications and invariants/axioms, our tool could verify/falsify 226 of the 289 benchmarks. With our automatically inferred frame axioms, the tool could verify/falsify 203 of the 289, demonstrating the effectiveness of our approach.

1. Introduction

Automatic formal verification has long been a dream in software engineering. Unfortunately, undecidability and tractability problems have prevented full realization of this dream. Fully automatic verification tools typically sacrifice soundness — i.e., they can fail to detect bugs — (e.g., [15, 8, 21, 32, 1, 23, 2]) or scalability (e.g., [10]), or both. Our long-term aim is to produce sound, highly scalable, automatic program verification tools that can handle the complexity of real code without producing too many false error reports.

Many scalable and sound software verification tools are currently semi-automatic: they rely on user-provided annotations about the modularity of the software to achieve better scalability and precision (e.g., [19, 4, 17, 7, 9, 30]).[1] Typical required annotations include procedure pre- and postconditions, loop invariants, and frame axioms.

This paper focuses on frame axioms. Formal analysis of software always confronts some version of the frame problem [29]: knowing what is *not* changed by a piece of code is necessary for correct and efficient verification. For straight-line code and scalar variables, computing what changes and what does not is straightforward. In the presence of pointers, unbounded arrays, and heap-allocated data structures, however, with the corresponding looping/recursive code to manipulate them, computing precisely what changes is exceedingly difficult (undecidable in general). Frame axioms allow the user to aid this computation by suggesting candidate logical formulas that delimit what memory locations can be modified by a loop or procedure body; if the verification tool can prove these formulas to be inductive invariants, then it can use them as assumptions during verification of other assertions. Because frame axioms are so helpful to the verification process, many tools and specification styles support them, e.g.: *modifies* clauses in Spec# [7] and HAVOC [9], *assignable* clauses in JML [27], and *assigns* clauses in Caduceus [17]. Unfortunately, these frame axioms are often very complex and difficult to write, as they must carefully balance between looseness and tightness in order to be inductive, as well as being strong enough to prove desired properties of the program. We have found writing frame axioms to be the most tedious part of annotating a program's procedures and loops.

In this paper, we present a novel, automatic method to infer candidate frame axioms. Our two main goals are scalability to non-trivial code bases and sufficient precision to replace most or all manual annotation of frame axioms. Our method starts from a recent shallow shape analysis approach that offers scalability to hundreds of thousands of lines of code, with better precision than previous, highly scalable pointer analyses [26]. The analysis summarizes the points-to relation as a graph; our algorithm performs a graph traversal to create a logical formula characterizing what could be modified via *any* sequence of pointer-chasing. This formula is the candidate frame axiom. The

*This work was supported by the Natural Sciences and Engineering Research Council of Canada and a University of British Columbia Graduate Fellowship.

[1] User-provided annotations are also useful for verifying incomplete programs, but our focus is on scalability and precision when verifying a complete code base — very little can be inferred about missing code.

worst-case complexity of the algorithm is exponential in the size of a graph, so we must evaluate our approach empirically. We have implemented our algorithm in a modular extended static checker for C programs. To evaluate scalability, we ran our tool on several medium-sized open-source C programs and had no difficulty scaling to several tens of thousands of lines of code. To evaluate the precision of our analysis, we tested our tool on a benchmark suite of challenging buffer-overflow examples proposed at ASE 2007 [24]. With manually provided specifications and invariants/axioms, our tool could verify/falsify 226 of the 289 benchmarks. Using our new automatic inference approach, we were able to infer, completely automatically, frame axioms precise enough to verify/falsify 203 of the 226 benchmarks, demonstrating the effectiveness of our inference approach.

1.1. Related Work

Our inference approach uses a simple shape analysis as a starting point. There is a vast literature on shape analysis (and related analyses of side-effects, pointers, etc.). We are using one particular, recently published result, which offers high scalability and context-sensitive heap information [26]. We believe our approach could be adapted to other pointer analyses that produce similar summary graphs of the data structures in the program.

There has been some work on automatic inference of procedure pre- and postconditions, and loop invariants (e.g., [11, 18, 3, 20, 22, 28, 16]). Our work shares the same motivation with these works: making semi-automatic program verification more automatic. We address the completely different problem of inferring frame axioms, however, so this body of work is complementary to ours.

We are not aware of any work on automatic inference of frame axioms. The closest related work to ours is [31], which attacks the more difficult problem of inferring procedure summaries that are sufficiently precise to prove verification conditions. Frame axioms form part of these procedure summaries. Because they are attempting a more ambitious objective, they created their own static analysis, which is more precise (flow-sensitive as well as context-sensitive), and is therefore by design more expensive and less scalable. We cannot compare results directly, because their tool is for Java, whereas ours is for C, but we report results on more and larger examples. Our tool also can analyze the much more complicated pointer manipulation that occurs in C programs, which theirs does not. On the other hand, for small Java procedures, their tool infers usable, complete procedure summaries, whereas our goal is only to infer frame axioms.

```
1 typedef struct {int f1;
2                 int f2;} Elem;
3
4 Elem* alloc(int size) {
5   return (Elem*)malloc(size * sizeof(Elem));
6 }
7
8 void init(int size) {
9   Elem *a1 = alloc(size), *a2 = alloc(size);
10
11  // set fields f1 of a1 to 1
12  for (int i = 0; i < size; i++) {
13    a1[i].f1 = 1;
14  }
15
16  // set fields f1 of a2 and
17  // fields f2 of a1 to 0
18  for (int i = 0; i < size; i++) {
19    a2[i].f1 = 0;
20    a1[i].f2 = 0;
21  }
22
23  // check if fields f1 of a1 are set to 1
24  for (int i = 0; i < size; i++) {
25    assert(a1[i].f1 == 1);
26  }
27 }
```

Figure 1. Our illustrative example.

2. Illustrative Example

Throughout the paper, we will use a simple running example to illustrate the basic concepts as well as our new automatic inference approach. The code of the example is presented in Figure 1. First, we define type `Elem` that is a structure consisting of two integer fields `f1` and `f2`. The example has two procedures called `alloc` and `init`. The procedure `alloc` allocates an array of `size` elements of type `Elem`. The procedure `init` starts by allocating arrays `a1` and `a2` by calling procedure `alloc` on line 9. Both arrays have an unspecified size `size`. Then, two loops are executed:

1. The loop on line 12 sets field `f1` of all elements in `a1` to 1.
2. The loop on line 18 sets field `f1` of all elements in `a2` to 0, and also field `f2` of all elements in `a1` to 0.

In the end, we check whether field `f1` of all elements in `a1` is set to 1 using the assertion on line 25. Obviously, the assertion is not going to fail: First of all, it is clear that fields `f1` of `a1` are set to 1 in the first loop on line 13. Second, the loop on line 18 does not change those fields: it updates fields `f1` of different array `a2` and also different fields `f2`

of array a1. Precisely such important facts about preservation of values of memory locations are necessary for verification of this example. We capture them using modifies clauses (i.e. frame axioms). As we'll see in the next section, it is often tedious to specify the modifies clauses manually. Therefore, the goal of this paper is to infer as much as possible completely automatically.

3. Background

3.1. Modeling the Semantics of Memory

Because of the vast size of available memory in today's computer systems, faithfully representing each memory allocation and access in a static verifier is not going to scale. Therefore, verification tools rely on memory models that trade precision for scalability, and in turn, they define its programming language operational semantics with respect to the chosen memory model.

To make this paper self-contained, we briefly introduce here our memory model and the respective operational semantics of C. For details we refer the reader to the related work, since our operational semantics is mostly based on the one used in HAVOC [9], and also similar to Caduceus [17] and *VerifiedC* [30].

The main idea behind our memory model is to divide the memory into disjoint objects (or regions). Each object is identified by its reference, and has a fixed size determined when the object is being allocated. A pointer in our memory model is therefore a pair consisting of a reference and an offset; the reference uniquely defines the object into which the pointer points to; the byte offset in the object defines the byte being pointed to.

Our semantics for C programs depends on three fundamental types, the uninterpreted type `ref` of object references, the type `int` of integers, and the type `ptr` = `ref` × `int` of pointers. Each variable in a C program, regardless of its static type, contains a pointer value. A *pointer* is a pair containing an object reference and an integer offset. An integer value is encoded as a pointer value whose first component is the special constant `null` of type `ref`. Note that because of the integer offset component, our memory model can precisely capture byte offsets and low-level pointer arithmetic inside an object. On the other hand, since object references are uninterpreted, the objects are essentially "infinitely apart" and we cannot model pointer arithmetic between objects. However, this is not a serious drawback since such pointer manipulations are very rare in practice.

The heap of a C program is modeled using two map variables, `Mem` and `Alloc`, and a map constant `Size`. The variable `Mem` maps pointers to pointers and intuitively represents the contents of the memory at a pointer location. The variable `Alloc` maps object references to the set {`UNALLOCATED, ALLOCATED`} and is used to model memory allocation. The constant `Size` maps object references to positive integers and represents the size of the object. For instance, the procedure call `malloc(n)` for allocating a memory buffer of size `n` returns a pointer $\texttt{Ptr}(o,0)$ where o is an object reference such that $\texttt{Alloc}[o] = \texttt{UNALLOCATED}$ and $\texttt{Size}[o] \geq n$ before the call, and $\texttt{Alloc}[o] = \texttt{ALLOCATED}$ after the call.[2]

3.2. Specification Language

The modular style of verification we are employing requires a specification language for program annotations, in the form of invariants and procedure pre- and post-conditions. The specification language of our modular verifier is the same as the one used by HAVOC [9]. It allows succinct expression of many interesting properties of low-level programs that manipulate unbounded data structures.

For this paper, we will informally introduce only the part of the specification language necessary for annotating our running example. The example with manually provided annotations required for the verification to go through is given in Figure 2.[3] As usual, we denote preconditions with `requires`, postconditions with `ensures`, loop invariants with `invariant`, and modifies clauses with `modifies`.

The procedure `alloc` has one precondition, `size>0`, requiring that its integer parameter `size` be greater than 0 at every call. Furthermore, it ensures that the heap object pointed to by the return pointer (denoted with `$return`) is allocated, its size is equal to `size*sizeof(Elem)`, and also that the offset component of `$return` is 0.

In procedure `init`, all three loops had to be annotated with loop invariants and modifies clauses to be able to prove the assertion on line 44. Each loop has a necessary invariant `0<=i<=size` that bounds the counter `i`. Apart from the usual basic expressions, such as `0<=i<=size`, the specification language also supports annotations, again borrowed from HAVOC, convenient for constructing potentially unbounded sets of pointers (such as `Array`) and for manipulating those sets (such as `Incr` and `Union`).

The expression $\texttt{Array}(p, \mathit{size}, \mathit{idx})$, where p is a pointer and *size* and *idx* are integers, refers to the unbounded set of pointers

$$\{p, p+\mathit{size}, p+2*\mathit{size}, \ldots, p+(\mathit{idx}-1)*\mathit{size}\}.$$

We use it to specify a set of memory locations up to index *idx* belonging to an array whose element size is *size*. For instance, in the invariant on line 17, the expression

[2] We currently do not model failure of memory allocation, but it would be easy to do so.

[3] For better readability, we omit the syntactic clutter that pushes the annotations through the C frontend of our prototype verifier.

```
1 typedef struct {int f1;
2                 int f2;} Elem;
3
4 requires size > 0;
5 ensures Allocates($return);
6 ensures Size($return) == size*sizeof(Elem);
7 ensures OffsetOf($return) == 0;
8 Elem* alloc(int size) {
9   return (Elem*)malloc(size * sizeof(Elem));
10 }
11
12 requires size > 0;
13 void init(int size) {
14   Elem *a1 = alloc(size), *a2 = alloc(size);
15
16   invariant 0 <= i <= size;
17   invariant Forall(x,
18                    Array(a1, sizeof(Elem), i),
19                    x→f1 == 1);
20   modifies Incr(Array(a1,
21                       sizeof(Elem), New(i)),
22                 OFFSET(Elem, f1));
23   // set fields f1 of a1 to 1
24   for (int i = 0; i < size; i++) {
25     a1[i].f1 = 1;
26   }
27
28   invariant 0 <= i <= size;
29   modifies Union(
30     Incr(Array(a2, sizeof(Elem), New(i)),
31          OFFSET(Elem, f1)),
32     Incr(Array(a1, sizeof(Elem), New(i)),
33          OFFSET(Elem, f2)));
34   // set fields f1 of a2 and
35   // fields f2 of a1 to 0
36   for (int i = 0; i < size; i++) {
37     a2[i].f1 = 0;
38     a1[i].f2 = 0;
39   }
40
41   invariant 0 <= i <= size;
42   // check if fields f1 of a1 are set to 1
43   for (int i = 0; i < size; i++) {
44     assert(a1[i].f1 == 1);
45   }
46 }
```

Figure 2. Our illustrative example annotated with necessary preconditions, postcondtions, loop invariants, and modifies clauses.

`Array`$(\texttt{a1},\texttt{sizeof(Elem)},\texttt{i})$ captures elements of the array `a1` up to index `i`.

The set expression `Incr`(C,n) increments each element of the set of pointers C by the offset n. On line 20, it is used to increment all pointers in the set defined with `Array` by the offset of field `f1` in the structure type `Elem`. Similarly, the set expression `Decr`(C,n) decrements each element of the set of pointers C by the offset n.

To be able to reason about sets of pointers, we use the expression `Forall`$(\texttt{x},S,\phi)$, which says that for all elements `x` of some set of pointers S, formula ϕ has to hold. For example, on line 17, we use `Forall` to say that fields `f1` of all elements in `a1` up to index `i` are set to 1.

Each modifies clause `modifies` C refers to a set of pointers C in the pre-state of the respective procedure or loop. It specifies which memory locations get modified by the procedure/loop. The set C has to be carefully specified. If the set is a subset of the memory locations that actually get modified, the frame axiom generated from the modifies clause will fail when the verifier checks it. If the set is too coarse of an over-approximation, the verifier will not be able to prove many interesting properties later on. Modifies clauses are therefore often complex, as can be seen from the one on line 29, which says that the loop modifies only fields `f1` of the array `a2` (first `Incr` expression of the `Union`) and fields `f2` or the array `a1` (second `Incr` expression). Note that in the loop modifies clauses, `New(i)` indicates that we are not referring to the value of `i` in the pre-state, but to the value of `i` being changed by the loop (i.e. in the post-state).

Our tool needs two important facts to be able to discharge the assertion on line 44. The facts are captured by the annotations on line 17 and 29 we just explained. First, the invariant on line 17 ensures that after the loop, the field `f1` of all elements in `a1` is set to 1. In addition, the modifies annotation on line 29 ensures that the second loop does not modify the `f1` fields of `a1` that the first loop just set. The modifies clause says that the loop modifies `f1` fields of elements of array `a2` and `f2` fields of elements of `a1`, leaving therefore `f1` fields of `a1` unchanged. To be able to generate these modifies sets automatically, we have to be able to distinguish `a1` from `a2` although they are allocated using the same `malloc` instruction on line 9, and also to conclude which fields (offsets) of array elements are being modified.

3.3. Data Structure Analysis (DSA)

Data Structure Analysis (DSA) [26] is a highly scalable and fast, context-sensitive (with full *heap cloning*), field-sensitive, conservative pointer analysis. The term "heap cloning" refers to a property important for achieving true context-sensitivity — heap objects are not distinguished just by allocation site, but also by (acyclic) call paths leading to their allocation, i.e. the calling context in which they were

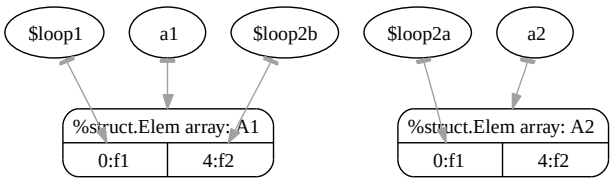

Figure 3. Simplified Data Structure Graph for procedure `init`. The nodes $\$loop1$, $\$loop2a$, and $\$loop2b$ are temporary helper pointer variable nodes not visible in the source code; $\%struct.Elem$ denotes the type of a node; flag *array* marks array nodes; $f1$ and $f2$ are fields at offsets 0 and 4, respectively. The fields $f1$ and $f2$ are integer and not pointer fields, and therefore have no outgoing edges.

created. In our illustrative example from Section 2, both arrays get created with the same call to `malloc` on line 9, but from different contexts (line 14). Such patterns are common because support for data structure operations is often going to be encapsulated in a library, and therefore it is important to be able to handle such cases precisely.

DSA constructs a representation of the heap in the form of Data Structure Graphs (DS graphs); it creates one DS graph per procedure plus an additional one for global storage. The separate globals graph is a key optimization allowing procedure graphs to contain only the parts of global storage reachable from that procedure. A DS graph consists of a set of nodes (DS nodes) and a set of edges. The DS graph for procedure `init` of our motivating example is shown in Figure 3. We distinguish two types of DS nodes: heap nodes with a number of fields at different offsets (e.g. nodes $A1$ and $A2$ in the example graph), and pointer variable nodes that point into heap nodes (e.g. nodes $a1$ and $\$loop1$). A pointer variable node is named after the pointer variable it represents and has one edge. A heap node has one outgoing edge per pointer field. Each heap node represents a potentially unbounded number of memory locations. A DS graph edge is defined by its source node and offset (i.e. offset of the respective pointer field in the source node), and its end node and offset. For instance, the edge coming out of $\$loop2b$ is defined by $\langle \$loop2b, 0\rangle \rightarrow \langle A1, 4\rangle$.

Instead of just providing the usual pairs of references that may alias (points-to/alias information), the explicit heap representation DSA constructs includes objects that are maybe not directly necessary for identifying aliases. That additional limited information about linked data structures present in the heap and explicit tracking of reachability relations between heap objects makes DSA a simple form of shape analysis. It can identify different instances of data structures and provide structural and type information for each identified instance (e.g. all data structures on which array indexing is performed, such as $A1$ and $A2$, are marked with the *array* flag). Having explicit representation of heap objects and their connectives (and therefore also reachability information) is important for our approach since it involves describing sets of objects that DS nodes represent by traversing paths through which they are reachable from global variables, procedure parameters, etc.

Another important feature of the algorithm is conservative field-sensitivity in a type-unsafe language such as C. DSA tracks fields precisely in the type-safe parts of the heap/program, while in the presence of type-unsafe operations it conservatively collapses all fields of an object. The field-sensitivity of DSA enabled us to take advantage of our precise memory model described in Section 3.1, and therefore to generate modifies sets at the level of granularity of byte offsets instead of whole heap objects. We can see where this is important on our illustrative example from Figure 2: to prove the final assertion, our automatic inference algorithm has to be able to conclude that the loop on line 36 modifies only field `f2` of array `a1`.

4. Automatic Frame Axiom Generation

Given the DS graphs generated by DSA (Section 3.3), our tool chain creates candidate frame axioms for the verifier via a three-step process. First, we process the DS graph to compute an over-approximation of the set of memory locations that can be modified by each function or loop body, which we call the *modifies set*. Next, we encode this set into a typical program specification logic, for use as a modifies clause annotation. The final step is the standard conversion of the modifies clauses into frame axioms used internally by the verification tool.

4.1. From DS Graphs to Modifies Sets

The goal of the first step is to compute an over-approximation of the set of memory locations that can be modified by parts of the program code. Because we intend to generate annotations for loops and procedures (for use inductively as frame axioms), our analysis centers on characterizing what memory locations can be modified by a given procedure or loop body.

The algorithm is a straightforward traversal and marking of the DS graphs. The analysis is ordered by a bottom-up traversal of the program's call graph. (Cycles in the call graph are broken arbitrarily.) For each procedure or loop body, we can identify all store operations and mark the target address's corresponding location, which is defined by its node and offset, in the procedure's or globals DS graph as (potentially) modified. In addition, for any procedures

called from this procedure or loop body, we copy the markings from the callee's DS graphs to the corresponding node in the current DS graph, if any. In other words, we mark any changes made by the current procedure/loop body, as well as copying over any changes made by any callee that is visible to the current procedure/loop body. Note that all modified locations marked in the globals graph do not have to be copied over since the globals graph is shared by all procedures. Therefore, the globals graph is always searched first when marking modified locations.

The result is that we compute the following sets of $\langle DSnode, offset \rangle$ pairs:

- For each procedure, we find a set of memory locations (i.e. $\langle DSnode, offset \rangle$ pairs) in the respective DS graph that are being modified by that procedure or its callees, and that are visible to its callers, i.e. nodes reachable from globals or procedure parameters.

- For each loop, we find a set of $\langle DSnode, offset \rangle$ pairs modified by that loop or by procedures called from the loop, and that are visible outside the loop, i.e. nodes reachable from globals or loop variables that are live at the loop header.

Note that each $\langle DSnode, offset \rangle$ pair can represent an unbounded set of memory locations.

The modifies set we compute might not be guaranteed to be an over-approximation, because when we break cycles in the call graph, we may lose behaviors of the original program. Fortunately, this localized unsoundness in our analysis does not compromise the overall soundness of the verification, because the candidate frame axioms (like any other annotation) are checked when they are used during the verification process.

4.2. Modifies Sets to Modifies Clauses

The modifies sets are then passed to the second stage of our algorithm, which tries to characterize these sets of memory locations as formulas in a logic for program specification and verification, as in Section 3.2. Details of this specification language are largely unimportant, as most verification tools use languages with very similar expressiveness. One significant issue is that our specification language, like most others, does not currently have constructs for describing unbounded recursive data structures. (We plan to address this in future work.) Accordingly, the second stage starts by breaking any cycles in the DS graphs, which can represent such data structures, yielding directed acyclic graphs (DAGs). As before, this localized potential unsoundness does not compromise the overall soundness of the verification process.

For each node in the DS graph, we generate a logic formula that tries to over-approximate the set of memory locations that the pair $\langle DSnode, 0 \rangle$ represents. The formulas are generated by walking over the topologically sorted (each node before all nodes to which it has outbound edges) nodes of a DS graph (DAG) starting from variables that can appear in the respective modifies clause. We call such variables *root variables*. The root variables for modifies clauses for procedures are the globals and the procedure parameters; the root variables for loops are globals and variables live at the loop header. A path in a DS graph to a node represented as a formula will be a sequence of pointer arithmetic operations, memory dereferences, and `Array` set constructors. The pseudocode of the algorithm is given in Figure 4.

The input of the algorithm is a set of root nodes R and a DS graph. For each node in the graph reachable from the root nodes, the algorithm generates a list of expressions, each expression representing one path to the $\langle DSnode, 0 \rangle$ pair from a root node. We call such expressions *path expressions*. If a node n is a variable node, the path expression to the beginning of the object its edge e points to is simply

$$\texttt{Decr}(n.varName, e.endOffset)$$

and is generated on line 6. Note that while $n.varName$ and $e.endOffset$ are actually evaluated by our algorithm, `Decr` becomes a part of the path expression we are recursively constructing and is not evaluated. Before we add the path expression to the $e.endNode$, we always have to case-split on whether the $e.endNode$ represents an array or not. If a node n is a heap node, the algorithm iterates through all of its outgoing edges on line 12. For each edge e, it loops on line 13 through all path expression to the current node n. Then, for each path p, the path expression

$$\texttt{Decr}(\texttt{Deref}(\texttt{Incr}(p, e.startOffset)), e.endOffset)$$

to the beginning of the object the edge points to is generated on line 14. The path expression captures the fact that each outgoing edge from a heap node represents a memory dereference, which is represented by the `Deref` expression. Again, before adding the newly generated path expression to the end node, we have to case-split on whether the $e.endNode$ represents an array or not on line 15.

An array heap node represents an array of unbounded number of elements that the algorithm captures using the `Array` expression introduced in Section 3.2. The algorithm generates array expressions

$$\texttt{Array}(path, \texttt{sizeof}(e.endNode), \texttt{Size}(path))$$

on lines 8 and 16. Note that while the size of each array element $\texttt{sizeof}(e.endNode)$ can be known at compile time, the total size of the array is usually not since arrays tend to be dynamically allocated. However, because our memory

```
1:  Q ← nodes with no predecessor
2:  while Q is non-empty do
3:    pop node n from Q
4:    if n is a variable node and n in R then
5:      e ← n.edge
6:      path ← Decr(n.varName, e.endOffset)
7:      if e.endNode is array node then
8:        path ←
            Array(path, sizeof(e.endNode),
                  Size(path))
9:      end if
10:     e.endNode.addPath(path)
11:   else if n is a heap node then
12:     for all edges e of n do
13:       for all paths p of n do
14:         path ← Decr(Deref(
              Incr(p, e.startOffset)), e.endOffset)
15:         if e.endNode is array node then
16:           path ←
                Array(path, sizeof(e.endNode),
                      Size(path))
17:         end if
18:         e.endNode.addPath(path)
19:       end for
20:     end for
21:   end if
22:   for all edges e of n do
23:     remove edge e from graph
24:     if e.endNode has no other incoming edges then
25:       push e.endNode into Q
26:     end if
27:   end for
28: end while
```

Figure 4. Algorithm for generating formulas that describe DS graph nodes reachable from a set of root nodes R.

model has the map `Size` where size of each object is stored during allocation, with the expression $\texttt{Size}(path)$ we are referring to this map when looking for a dynamic size of the array object *path* points to. The ability of the DSA to recognize array heap nodes, and the ability of our algorithm to precisely express the potentially unbounded set of memory locations the array nodes represent, is crucial for the precision of the generated modifies sets.

In the end, because the generated path expressions point to the beginning of an object, we have to offset them to point exactly to the modified memory location inside the object. The modifies set for the respective procedure or loop is then the union of such path expressions.

4.3. Modifies Clauses to Frame Axioms

Finally, frame axioms are constructed from the modifies clauses in the standard manner of the many tools that support modifies clauses. Formally, the modifies clause `modifies` C, where C is a set of pointers, is translated into the following frame axiom:[4]

$$\forall \texttt{x} : \texttt{ptr} \left(\begin{array}{l} \texttt{old}(\texttt{Alloc})[\texttt{Obj}(\texttt{x})] == \texttt{UNALLOCATED} \\ \texttt{||}\ \ (\texttt{x} \in \texttt{old}(C)\ \texttt{\&\&}\ \texttt{Obj}(\texttt{x})\texttt{!=null}) \\ \texttt{||}\ \ \texttt{old}(\texttt{Mem})[\texttt{x}] == \texttt{Mem}[\texttt{x}] \end{array} \right).$$

Informally, the axiom states that the contents of `Mem` remains unchanged at each pointer that is allocated and both not a member of C and not `null` in the pre-state of the procedure/loop. Because of the flow-insensitivity of DSA and also of our algorithm (i.e. flow-insensitive marking of modified locations even if they had not been allocated), a loop frame axiom might contain memory locations that are allocated only later on. Such locations are uninitialized and can point to essentially anything. Therefore, leaving them in the frame axioms would mean that anything could be modified, which is highly imprecise and would prevent proving many interesting properties. We prevent this by adding path-sensitivity by restricting the set of modified locations just to the ones that have been allocated (i.e. not equal to `null`) at the point where frame axiom had been asserted (procedure or loop entry).

The automatically generated frame axioms are our best effort to be as precise as possible, and in general do not have to be sound. However, as mentioned already, this does not affect the soundness of the verification, since all of the generated frame axioms are checked during verification.

4.4. Example Run

We now illustrate how the presented algorithm generates path formulas on the DS graph of our running example given in Figure 3. The root nodes for the second loop in the example are $a1$ and $a2$. The algorithm starts by putting all variable nodes (i.e. nodes with no predecessor) into Q. Nodes $\$loop1$, $\$loop2a$, and $\$loop2b$ are just going to be popped on line 3 and their edges removed in the loop on line 22 since these variable nodes are not in root nodes. Node $a1$ is a root node. It has one edge whose end node is the array heap node $A1$. Therefore, the path

$$\texttt{Array}(\texttt{Decr}(a1, 0), \texttt{sizeof}(A1), \texttt{Size}(\texttt{Decr}(a1, 0)))$$

will be added to the paths of node $A1$. Also, in the loop on line 22, the node $A1$ will be pushed onto Q since it has no more incoming edges. The same thing will happen with $a2$

[4] The expression $\texttt{old}(\phi)$ denotes the value of ϕ in the pre-state of the procedure/loop.

in the next iteration of the while loop. Then, $A1$ and $A2$ will be removed from Q since they have no outgoing edges, Q is empty, and we are done.

The paths generated by the algorithm are

$$\texttt{Array}(\texttt{Decr}(a1,0),\texttt{sizeof}(A1),\texttt{Size}(\texttt{Decr}(a1,0)))$$

to the memory location $\langle A1,0\rangle$, and

$$\texttt{Array}(\texttt{Decr}(a2,0),\texttt{sizeof}(A2),\texttt{Size}(\texttt{Decr}(a2,0)))$$

to the memory location $\langle A2,0\rangle$. The loop modifies memory locations pointed by $\$loop2a$ and $\$loop2b$, which correspond to pairs $\langle A2,0\rangle$ and $\langle A1,4\rangle$. Therefore, the expression

$$\texttt{Incr}(\texttt{Array}(\texttt{Decr}(a2,0),\ \texttt{sizeof}(A2),\texttt{Size}(\texttt{Decr}(a2,0))),0)$$

represents the first set of modified memory locations, while the expression that offsets all pointers by 4

$$\texttt{Incr}(\texttt{Array}(\texttt{Decr}(a1,0),\ \texttt{sizeof}(A1),\texttt{Size}(\texttt{Decr}(a1,0))),4)$$

represents the second set. The final modifies set for the second loop is the union of these two sets, which corresponds to the modifies set we provided manually on line 29 in Figure 2.

Program	#TCs	#Annot	#Mod	#Infer
apache	24	24/24	90	9/24
bind	20	4/20	8	4/4
edbrowse	6	6/6	14	6/6
gxine	2	2/2	0	2/2
libgd	8	4/8	4	4/4
MADWiFi	6	6/6	8	2/6
NetBSD-libc	24	24/24	72	20/24
OpenSER	102	102/102	204	102/102
samba	4	4/4	2	4/4
sendmail	67	46/67	58	46/46
SpamAssassin	2	2/2	4	2/2
wu-ftpd	24	2/24	2	2/2
Total	289	226/289	466	203/226

Table 1. Results showing the quality of the automatically generated modifies clauses. "#TCs" is the number of testcases; "#Annot" is the number of testcases our tool discharged with the manually provided annotations; "#Mod" is the number of required modifies clauses; "#Infer" is the number of testcases with the automatically generated modifies clauses our tool successfully discharged.

5. Experiments

We have implemented the inference algorithm in our tool SMACK (Static Modular Assertion ChecKer), a modular, annotation-based, extended static checker of C programs. In the spirit of modular verification, SMACK verifies programs annotated with procedure specifications and loop invariants. It uses the LLVM compiler framework [25] to parse input programs and annotations. The LLVM output is translated by SMACK into a BoogiePL [13] program based on the operational semantics of C memory accesses introduced in Section 3.1. BoogiePL is the input language of the BOOGIE verifier [5], which, in turn, generates a verification condition (VC) from the input program whose validity implies partial correctness of the input. The VC generation in BOOGIE is performed using a variation [6] of the standard *weakest precondition* transformer [14]. We check the generated VC using the accompanying Z3 theorem prover [12].

We assessed the usability of our technique and the precision of the generated modifies clauses on the buffer-overflow benchmark suite proposed at ASE 2007, containing testcases derived from a number of buffer-overflow vulnerabilities in open-source programs [24]. The suite has 22 vulnerabilities from 12 programs, totaling 289 testcases (faulty and patched versions) with different difficulty levels and around 18000 LOC. First, we manually annotated most of the benchmarks with pre- and postconditions, loop invariants, and modifies clauses necessary for the verification/falsification to go through. We checked NULL pointer dereference, buffer-overflow, and buffer-underflow properties for each pointer dereference. Then, we removed all of the manually provided modifies clauses and, instead, used the ones generated by our automatic approach. We again ran all of the experiments to measure the quality of the automatically generated modifies clauses. The results for this set of experiments are presented in Tables 1 and 2.

Table 1 assesses the quality (i.e., precision) of the automatically generated modifies clauses. We managed to manually annotate and check with SMACK 226 out of the 289 testcases. We had to skip 63 testcases because they either require bit-precise reasoning, which our tool currently does not support, or they were too complex to be completely annotated manually with the limited time we had. The annotation of these 226 testcases required, among other annotations, 466 modifies clauses, ranging in complexity from simple lists of variables that get modified to complex expressions involving unions of unbounded sets of pointers (`Array` expressions) and pointer arithmetic. After removing all of the manually provided modifies clauses and replacing them with the automatically generated ones, SMACK discharged successfully 203/226 testcases (or 90%). This clearly shows the effectiveness of our technique: in most

Program	MTime(s)	ITime(s)
apache	42.5	44.1
bind	11.5	11.5
edbrowse	14.5	13.6
gxine	3.8	3.8
libgd	13.3	13.3
MADWiFi	3.9	3.9
NetBSD-libc	61.6	94.5
OpenSER	275.3	276.4
samba	7.7	7.8
sendmail	120.4	120.9
SpamAssassin	4.3	4.3
wu-ftpd	3.9	3.9
Total	559.1	585.1

Table 2. Verification time for testcases using manually provided (MTime) vs. automatically inferred (ITime) modifies clauses. Verification was run on an Intel Pentium D at 2.8Ghz.

cases, the automatically generated modifies clauses are precise enough for the verification to succeed, or for finding a bug without introducing false errors.

We analyzed the 23 testcases for which the automatically generated modifies clauses are not good enough. In all cases, the problems are loop modifies clauses, in particular, certain idioms of loops iterating over arrays. The root cause is the loss of precision because DSA conservatively over-approximates an unbounded array by a single element. The resulting overly conservative modifies clauses can cause either verification complexity to blow-up or some annotation to fail erroneously. In these 23 testcases, however, SMACK never erroneously reported a violation of the correctness properties; the failure was manifestly a failure of the analysis, not a false bug report.

Table 2 gives cumulative execution times of SMACK for the 203 testcases that SMACK could discharge with automatically generated modifies clauses. The results again support our automatic inference technique — the performance penalty we paid for using automatically generated modifies clauses is negligible compared to the effort needed for manually specifying them when the technique was not available.

Because the size of the testcases in the buffer-overflow benchmark suite is relatively small, the running times of DSA and our automatic inference algorithm are just a few milliseconds. Therefore, although annotating and checking these programs using SMACK is beyond the scope of this paper, we assessed the scalability of the inference algorithm on a number of open-source applications: the bftpd FTP server, the muh irc-bouncer, the gzip compression utility, the Pure-FTPd FTP server, the CUDD decision diagram package (we actually run the analysis on nanotrav — a simple reachability analysis program included with the CUDD package), and the Spin explicit-state model-checker.

Benchmark	LOC	Time(s)
bftpd 2.0	3843	2.5
gzip 1.2.4	5809	2.9
muh 2.2a	6294	2.7
Pure-FTPd 1.0.21	26320	3.8
Spin 5.1.4	29672	122.5
CUDD/nanotrav 2.4.1	67578	61.0
Total	139516	195.4

Table 3. Total running times of DSA and our automatic inference algorithm on a number of open-source benchmarks. These were on an AMD Opteron 254 at 2.8Ghz.

The running times are in Table 3 and clearly show the scalability of the prototype implementation of our approach. Our biggest example, CUDD, took only 61s. We believe the running time for Spin is the longest because it has an unusually big DS graph of around 2500 nodes for global storage. Since the complexity of our expression generator algorithm is worst-case exponential in the size of a DS graph, it is understandable that slowdown of the analysis is possible on big graphs, which is confirmed by the Spin example. However, as can be seen from the published DSA results [26], the usual maximal graph size is only a couple hundred of nodes and such big graphs do not occur often in practice.

6. Conclusion and Future Work

We have described a technique for automatically inferring frame axioms of procedures and loops using static analysis. We evaluated the inference technique on a benchmark suite proposed in ASE 2007. Our inference technique automatically generated frame axioms of sufficient quality to discharge approximately 90% of the benchmark examples that we could solve with manually provided frame axioms. In no cases did the automatic frame axioms produce false error reports or fail to falsify the buggy examples in the benchmark suite. The inference algorithm is also very fast. We have demonstrated scalability to several tens of thousands of lines of code, and we expect scalability to hundreds of thousands of lines of code in the near future.

We have immediate plans for future work along a number of directions. Adding support for recursive data structures based on the version of the reachability predicate described in [9] is at the top of our list. Another promising direction is to use disjointness information about sets of heap objects, which is also captured by DSA, to improve performance of the extended static checker by safely separat-

ing memory into disjoint regions. Using DSA, we can also find objects that are potentially accessed in a type-unsafe manner. We could employ this information to use a more refined, bit-precise memory model that would capture low-level, type-unsafe information for such accesses on demand, instead of using it for each memory access.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Intl. SPIN Workshop on Model Checking of Software (SPIN)*, pages 163–181, 2006.

[2] D. Babić and A. J. Hu. Calysto: Scalable and precise extended static checking. In *Intl. Conf. on Software Engineering (ICSE)*, pages 211–220, 2008.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Symp. on Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2005.

[6] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 82–87, 2005.

[7] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, pages 49–69, 2005.

[8] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software — Practice and Experience*, 30(7):775–802, 2000.

[9] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.

[10] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symp. on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[13] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[14] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[15] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symp. on Operating System Design and Implementation (OSDI)*, pages 1–16, 2000.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[17] J. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Intl. Conf. on Formal Engineering Methods (ICFEM)*, pages 15–29, 2004.

[18] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, pages 500–517, 2001.

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[20] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Principles of Programming Languages (POPL)*, pages 191–202, 2002.

[21] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Symp. on Principles of Programming Languages (POPL)*, pages 232–244, 2004.

[23] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14, 2007.

[24] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *Conf. on Automated Software Engineering (ASE)*, pages 389–392, 2007.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Intl. Symp. on Code Generation and Optimization (CGO)*, pages 75–88, 2004.

[26] C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.

[27] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *Software Engineering Notes*, 31(3):1–38, 2006.

[28] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 119–134, 2005.

[29] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh Univ. Press, 1969.

[30] W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop*, 2007.

[31] M. Taghdiri, R. Seater, and D. Jackson. Lightweight extraction of syntactic specifications. In *Intl. Symp. on Foundations of Software Engineering (FSE)*, pages 276–286, 2006.

[32] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Symp. on Principles of Programming Languages (POPL)*, pages 351–363, 2005.