TLP **18** (3–4): 470–483, 2018.
doi:10.1017/S147106841800025X

Shape Neutral Analysis of Graph-based Data-structures

GREGORY J. DUCK, JOXAN JAFFAR and ROLAND H. C. YAP
Department of Computer Science, National University of Singapore
(*e-mail:* {gregory,joxan,ryap}@comp.nus.edu.sg)

submitted 1 May 2018; accepted 11 May 2018

Abstract

Malformed data-structures can lead to runtime errors such as arbitrary memory access or corruption. Despite this, reasoning over data-structure properties for low-level heap manipulating programs remains challenging. In this paper we present a constraint-based program analysis that checks data-structure integrity, w.r.t. given target data-structure properties, as the heap is manipulated by the program. Our approach is to automatically generate a solver for properties using the type definitions from the target program. The generated solver is implemented using a Constraint Handling Rules (CHR) extension of built-in heap, integer and equality solvers. A key property of our program analysis is that the target data-structure properties are *shape neutral*, i.e., the analysis does not check for properties relating to a given data-structure graph *shape*, such as doubly-linked-lists versus trees. Nevertheless, the analysis can detect errors in a wide range of data-structure manipulating programs, including those that use lists, trees, DAGs, graphs, etc. We present an implementation that uses the Satisfiability Modulo Constraint Handling Rules (SMCHR) system. Experimental results show that our approach works well for real-world C programs.

KEYWORDS: Constraint Handling Rules, Satisfiability Modulo Constraint Handling Rules, Satisfiability Modulo Theories, Program Analysis, Data-structures, Memory Errors

1 Introduction

Low-level languages such as C and C++ are notorious for (subtle) bugs due to direct pointer manipulation. Program analysis may detect bugs, however, automating such analysis for data-structure manipulating programs in low-level languages is a challenging problem. Much of the existing work on data-structure analysis (Berdine *et al.* 2007; Berdine *et al.* 2005; Berdine *et al.* 2011; Dudka *et al.* 2011) focuses on (or depends on) *shape* properties, i.e., is the data-structure a tree or linked-lists, etc.? However, this complicates automated analysis:

1. Shape information is usually *implicit*.
2. Common data-structure shapes have *inductive* (a.k.a. *recursive*) definitions.
3. Data-structure integrity co-depends on *memory safety*.

For example, consider the following generic C `struct` declaration:

```
struct node { node *next1; node *next2; ...};
```

Automated shape discovery on type declarations alone is not feasible: this node could be for a tree, DAG, graph, doubly-linked-list, etc. Even if shape information were available

(or assumed), the next problem is that data-structure shapes have *inductive* (a.k.a. *recursive*) definitions, further complicating automated reasoning. For example, a list can be recursively defined as follows (in Separation Logic (Reynolds 2002)): $\mathsf{list}(l) \stackrel{\mathsf{def}}{=} \exists t : l = 0 \vee l \mapsto t * \mathsf{list}(t)$. Finally, data-structure reasoning for low-level programs is (co)dependent on *memory safety*, e.g., an object bounds error may clobber memory that invalidates the data-structure invariant. Conversely data-structure invariant violations may give rise to memory errors.

In this paper, we present a *shape neutral* data-structure analysis that aims to avoid the complications listed above. Instead of data-structure shape definitions, we analyze the program against a set of more general data-structure properties that hold for any canonical graph-based data-structures in standard idiomatic C. The properties include:

1. nodes are contiguous regions of memory.
2. nodes are the correct size.
3. nodes form a closed directed graph (no dangling links).
4. nodes do not overlap with each other.

Such general properties eliminate the need to infer (or assume) data-structure shapes, and can be derived solely from the types declared by the program.

Our automated shape neutral data-structure analysis for C programs is based on: (1) *symbolic execution* to generate path constraints for all possible paths through the program, and (2) a specialized *constraint solver* for shape neutral data-structure properties. For the latter, we first formalize the properties we aim to enforce, then use our formalization to derive a solver that can be implemented using *Constraint Handling Rules* (CHR) (Frühwirth 1998; Frühwirth 2009). Since the generated *verification conditions* (VCs) typically have a rich structure (quantifiers, conjunction, disjunction and negation), we implement the solver using the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) system (Duck 2012; Duck 2013). We demonstrate that our implementation is able to verify many data-structure manipulating C programs, including data-structures for lists, trees, DAGS, graphs, etc. We also compare against related tools for finding memory errors related to violations of the data-structure integrity constraints.

In summary, the main contributions of this paper are:

1. *Data-structure Integrity Constraints*: We propose and formalize a set of shape neutral data-structure integrity constraints based on the properties informally described above. The integrity constraints cover standard idiomatic C graph-based data-structures.
2. *Constraint Handling Rules Implementation of Shape Neutral Data-structure Properties*: We also present a constraint solver for the integrity constraints that can be implemented using a Constraint Handling Rules (CHR) extension of built-in heap, integer and equality solvers. The CHR solver is automatically generated from the target program's data-structure type declarations. Goals generated by program analysis can then be solved using the Satisfiability Modulo Constraint Handling Rules (SMCHR) system. SMCHR is suitable as it can efficiently handle goals with a complex Boolean structure, including negation. Furthermore, SMCHR allows different types of solvers (integer, heap, data-structure) to be seamlessly integrated.
3. *Evaluation*: Finally we present an experimental evaluation of our overall approach. We show that the proposed method is effective on "real world" data-structure manipulating code such as that from the `GNU GLib` library. We also compare our approach

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

$$sp((s_1;s_2),\phi) \stackrel{\text{def}}{=} sp(s_2, sp(s_1,\phi))$$
$$sp(x := E,\phi) \stackrel{\text{def}}{=} x = E[x'/x] \wedge \phi[x'/x]$$
$$sp(x := *p,\phi) \stackrel{\text{def}}{=} \mathsf{access}(\mathcal{H},p,x) \wedge \phi[x'/x]$$
$$sp(*p := E,\phi) \stackrel{\text{def}}{=} \mathsf{assign}(H',p,E,\mathcal{H}) \wedge \phi[H'/\mathcal{H}]$$
$$sp(p := \mathsf{malloc}(n),\phi) \stackrel{\text{def}}{=} \mathsf{alloc}(H',p,n,\mathcal{H}) \wedge \phi[H'/\mathcal{H},p'/p]$$
$$sp(\mathsf{abort}(),\phi) \stackrel{\text{def}}{=} \mathit{false}$$

(a) Summary of the symbolic execution rules.

```
(0) S1;
(1) while (b1) {
  (2) S2;
  (3) while (b2)
    (4) S3;
  (5) S4; }
(6) S5; (7)
```

(b) Program with loops.

Fig. 1. Strongest post-condition semantics and an example program.

against several existing state-of-the-art memory analysis/safety tools. We demonstrate that our tool can detect memory errors missed by other systems—especially regarding more complex data-structures involving multiple node types and sharing.

2 Preliminaries

Our analysis is based on defining a *data structure integrity constraint* (DSIC). The DSIC formula is derived from a schema which, when given type declarations, is instantiated into a first-order formula $\mathcal{D}$. Essentially, $\mathcal{D}$ states that data structures must have valid nodes, valid pointers, and nodes do not intersect. The formal presentation of $\mathcal{D}$ is given in Section 3.

The framework we use to analyze a program in pursuit of our integrity constraint $\mathcal{D}$ is a classic one: *Verification Condition Generation* (VCG) via *symbolic execution*, a method originating from Floyd (see e.g. (Matthews *et al.* 2006) for a succinct introduction). The overall algorithm is summarized as follows:

- The program, interpreted as a graph, is annotated with $\mathcal{D}$ at certain program points corresponding to a set of *cut-points* in the control flow graph.
- VCG is performed as follows. Suppose $\mathcal{D}$ holds when control reaches some cut-point p, then let q be the next subsequent cut-point encountered during program execution. We then show that q also satisfies $\mathcal{D}$. This is repeated for all cut-points.

This reduces shape neutral data-structure analysis (abbr. to $\mathcal{D}$-analysis) into proving that *Hoare triples* of the form $\{\mathcal{D}\}\ C\ \{\mathcal{D}\}$ are valid, where C is some code fragment (the analysis target) and $\mathcal{D}$ is the desired DSIC, defined later. Intuitively, a Hoare triple $\{A\}\ C\ \{B\}$ states that if A holds before execution of C, then B must hold after. Thus, $\{\mathcal{D}\}\ C\ \{\mathcal{D}\}$ is stating that the DSIC $\mathcal{D}$ is preserved by C. For branch-free C, our underlying methodology is *symbolic execution* as defined by the *strongest post condition* (SPC) *predicate transformer semantics* shown in Figure 1. All dashed variables (e.g. x') are implicitly existentially quantified, and the notation $\phi[x'/x]$ represents formula ϕ with variable x' substituted for variable x. We assume the standard definitions for sequences $(s_1;s_2)$, assignment $x := E$ and $\mathsf{abort}()$. Heap operations use special heap constraints defined below. A triple $\{\mathcal{D}\}\ C\ \{\mathcal{D}\}$ is established by symbolically executing $\mathcal{D}$ through C using the rules from Figure 1. This process generates a *path constraint* P. The triple holds iff *Verification Condition* (VC) ($P \models \mathcal{D}$) is proven *valid* with the help of a suitable constraint solver.

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

Cut-points are chosen to break loops into straight line program fragments amenable to symbolic execution. For example, consider the program with nested loops shown in Figure 1b. Also consider the formula $\mathcal{D}$ which we use to annotate points (0), (1), (3) and (7) (i.e., the chosen "cut-points"). The Hoare triples of interest are therefore:

$$\{\mathcal{D}\}\ \texttt{S1}\ \{\mathcal{D}\} \qquad \{\mathcal{D} \wedge \texttt{b1}\}\ \texttt{S2}\ \{\mathcal{D}\} \qquad \{\mathcal{D} \wedge \texttt{b2}\}\ \texttt{S3}\ \{\mathcal{D}\}$$
$$\{\mathcal{D} \wedge \neg\texttt{b2}\}\ \texttt{S4}\ \{\mathcal{D}\} \qquad \{\mathcal{D} \wedge \neg\texttt{b1}\}\ \texttt{S5}\ \{\mathcal{D}\}$$

It is important to note that we are not requiring integrity at *every* program point, rather only at the cut-points, which is a heuristic that works reasonably well in practice (see Section 5).

Heap Operations. To handle heap operations, we extend the $\mathcal{H}$-constraint language from (Duck *et al.* 2013). We assume, as given, a set of Values (typically $\mathsf{Values} \stackrel{\mathsf{def}}{=} \mathbb{Z}$) and define the set of Heaps to be all *finite partial maps* between values, i.e., $\mathsf{Heaps} \stackrel{\mathsf{def}}{=} (\mathsf{Values} \rightharpoonup_{\text{fin}} \mathsf{Values})$. Let $dom(H)$ be the *domain* of the heap H. We abuse notation and treat heaps H as sets of (pointer,value) pairs $\{(p, H(p)) \mid p \in dom(H)\}$. Conversely, a set of pairs S is a *heap* iff for all p, v, w we have that $(p, v), (p, w) \in S \rightarrow v = w$. A *heap partitioning constraint* is a formula of the form $H \doteq H_1 * H_2$, where H, H_1, H_2 are heap variables. Informally, the constraint $H \doteq H_1 * H_2$ states that heap H can be partitioned into two disjoint (separate) sub-heaps H_1 and H_2. The set-equivalent definition is as follows: $H = H_1 \cup H_2 \wedge dom(H_1) \cap dom(H_2) = \emptyset$.

We use the symbolic execution rules for heap operations from (Duck *et al.* 2013) summarized in Figure 1. By convention, the state of the program heap is represented by a *distinguished heap variable* $\mathcal{H}$ (of type Heaps). Each heap operation modifies $\mathcal{H}$ according to some heap constraint access, assign, and alloc defined as follows:

$$\mathsf{access}(\mathcal{H}, p, v) \stackrel{\mathsf{def}}{=} (p, v) \in \mathcal{H}$$
$$\mathsf{assign}(H, p, v, \mathcal{H}) \stackrel{\mathsf{def}}{=} \exists w : (p, w) \in H \wedge \mathcal{H} = (H - \{(p, w)\}) \cup \{(p, v)\}$$
$$\mathsf{alloc}(H, p, 1, \mathcal{H}) \stackrel{\mathsf{def}}{=} \exists w : \mathcal{H} = H \cup \{(p, w)\} \wedge p \notin dom(H)$$

We can extend the definition for arbitrary-sized alloc in the obvious way. Note that our definitions implicitly assume that accessing *unmapped memory* (i.e. any $p \notin dom(\mathcal{H})$) behaves the same way as abort() (see Figure 1).

3 Data-Structure Analysis

Data-structure analysis (or $\mathcal{D}$-analysis) aims to prove that a suitable *data-structure integrity constraint* (DSIC) is preserved by the program. Conversely, a program fails data-structure analysis if it is possible to generate a mal-formed data-structure that violates the DSIC. More formally, the analysis aims to prove Hoare triples of the form $\{\mathcal{D}(\mathcal{H}, p_1, .., p_n)\}\ C\ \{\mathcal{D}(\mathcal{H}, q_1, .., q_m)\}$ where C is some code fragment (e.g. a function definition), $\mathcal{H}$ is the global program heap, $\{p_1, .., p_n\}$ and $\{q_1, .., q_m\}$ are sets of live pointer variables, and $\mathcal{D}$ is a suitable DSIC defined below. For brevity we abbreviate the DSIC as $\mathcal{D}$ (without parameters). If the analysis is successful, then all execution paths through C preserve $\mathcal{D}$, and C is said to be $\mathcal{D}$-safe.

In this section, we formalize the DSIC necessary to implement our analysis. Later, we use the formalism as the basis for the implementation using the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) system.

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

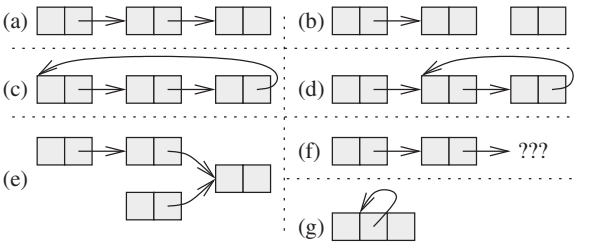


Fig. 2. Various list data-structure shapes. Here (???) indicates a dangling pointer.

Graph-based Data-structures. For our purposes, a *data-structure* is a *directed graph* of *nodes*. Each node has an associated *type* that corresponds to a C struct declaration. A data-structure is considered valid if the following conditions hold, including:

1. *Valid nodes*: Each node is a contiguous region of memory whose size is large enough to fit the corresponding node type. Partially allocated nodes (e.g., size too small) are disallowed.
2. *Valid pointers*: All non-null pointers stored within the data-structure must point to another valid node. Invalid, interior or dangling-pointers are disallowed. The null pointer is treated as a special case that indicates the non-existence of a link.
3. *Separated nodes*: Nodes must not overlap in memory.

These conditions are desirable for most standard graph-based data-structures implemented in idiomatic C, including linked-lists, trees, DAGS, graphs, etc., or any other data-structure type that can be described as a graph of nodes and uses standard pointers. Our $\mathcal{D}$-analysis is specific to the above properties, and does not include any other data-structure property. In particular, the analysis is *shape neutral*, and does not aim to analyze for, nor enforce, a given *shape* of the graph. As such, our $\mathcal{D}$-analysis is applicable to any graph-based structure, including *cyclic* data-structures such as circular linked-lists.

Data-structures in C are declared using some combination of `struct` declarations with pointer and data fields. It is not necessarily apparent what the intended shape of the data-structure is based on the type declarations alone. For example, consider the following `struct` definitions:

```
struct list_node {     struct tree_node {
    int val;               list_node *elem;
    list_node *next; }     tree_node *left; tree_node *right; };
```

For example, the `list_node` definition can be used to construct: (a) a linked-list; (b) a disjoint list; (c) a circular linked-list; (d) a lasso-list; (e) a list with sharing, or any combination of the above, as illustrated in Figure 2. Our $\mathcal{D}$-analysis treats (a), (b), (c), (d), (e) as graphs of `list_node`s. List (f) is *invalid* since the last pointer is dangling (violates *valid pointers*). Likewise list (g) is invalid since it contains overlapping nodes (violates *separated nodes*).

The $\mathcal{D}$-analysis aims to detect code that violates the DSIC. For example, consider the following "malicious" function `make_bad`, which deliberately constructs a mal-formed linked-list (with overlapping nodes as per list (g)), and thus should fail the $\mathcal{D}$-analysis:

```
struct list_node *make_bad(void) {
    struct list_node *xs = malloc(3*sizeof(void *));
```

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

```
    xs->val = 0; xs->next = (struct list_node *)&xs->next;
    xs->next->next = NULL; }
```

Such data-structure violations can lead to counter-intuitive behavior. For example, consider the following "benign" `set` function that sets the n^{th} member of a linked list:

```
void set(list_node *xs, int n, int v) {
    while (xs && (n--) > 0) xs = xs->next;
    if (xs) xs->val = v; }
```

Next consider the seemingly benign code fragment, (`set(xs,1,A); set(xs,1,B);`), that sets the second node's value to integers `A` and `B` respectively. However, if `xs` was created with `make_bad`, the first call to `set` clobbers the `next` field of the first node with value `A`. The second node now appears to be at address `A`. The second call to `set` executes `A->val=B` allowing for arbitrary memory to be overwritten.

Formalization. We shall now formalize the integrity constraint $\mathcal{D}$. We assume, as given, a set of *node types* $\mathsf{Types} = \{type_0, ..., type_n\}$ that are used by the program, e.g. `list_node` and `tree_node` defined above. We treat each $type \in \mathsf{Types}$ as a set of *fields*, e.g. $\texttt{tree\_node} = \{\texttt{elem}, \texttt{left}, \texttt{right}\}$. W.l.o.g., we shall assume all fields are renamed apart. Given Types, we define set Fields as all fields, and $\mathsf{PtrFields} \subseteq \mathsf{Fields}$ as all fields with a pointer-to-node type. We also treat Fields and PtrFields as sequences by choosing an arbitrary field ordering. The sets Types, Fields and PtrFields are derived from all `struct` declarations in scope.

Suppose heap $\mathcal{H}$ is a valid data-structure, then $\mathcal{H}$ is composed of a set of disjoint *node* heaps. Given a *node pointer* p of type $(T *)$, then a heap $N_p \in \mathsf{Heaps}$ is a *node heap* for pointer p if it spans the contiguous range of addresses $p, p+1, .., p+|T|-1$.[1]

An alternative (and unconventional) way to decompose a data-structure is based on fields. Given a valid data-structure $\mathcal{H}$ and a field $field \in \mathsf{Fields}$, then we define the *field heap* F_{field} to be the sub-heap of $\mathcal{H}$ containing all address-value pairs associated with the given *field*. For example, suppose $\mathcal{H}$ is a 3-node linked-list of type `list_node` (defined above), and encodes the sequence $1, 2, 3$. We assume the nodes have addresses p, q, and r respectively. Heap $\mathcal{H}$ is therefore representable in Separation Logic (Reynolds 2002) notation as follows:

$$p \mapsto 1 * (p{+}1) \mapsto q * q \mapsto 2 * (q{+}1) \mapsto r * r \mapsto 3 * (r{+}1) \mapsto 0$$

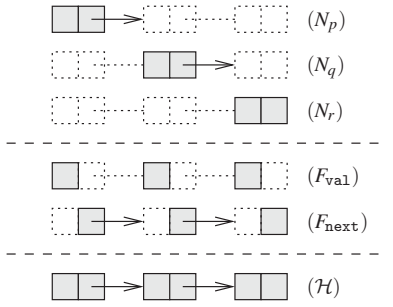
Heap $\mathcal{H}$ contains three node sub-heaps $N_p, N_q, N_r \subset \mathcal{H}$ and two field sub-heaps $F_{\texttt{val}}$, $F_{\texttt{next}} \subset \mathcal{H}$ defined in Figure 3b and illustrated in Figure 3a. The heap $\mathcal{H}$ is essentially the disjoint-union of all the field heaps, i.e., $\mathcal{H} \doteq F_{\texttt{val}} * F_{\texttt{next}}$. Given a set of field heaps, then we can define a valid *node-pointer* p as follows:

Definition 1 (*Node Pointers*)
Let $type \in \mathsf{Types}$ be a node type, then value $p \in \mathsf{Values}$ is a *type-node-pointer* if

- $p = 0$ (null pointer) ; or
- $p + i \in dom(F_{field})$ for each $field \in type$, where *field* is the i^{th} field of *type*. □

[1] As a simplification, we assume that the i^{th} field is stored in address $p + i$, and that $sizeof(\texttt{int}) = sizeof(\texttt{void} *)$.

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press



(a) Sub-heap illustrations

$$p \mapsto 1 * (p{+}1) \mapsto q \quad (N_p)$$
$$q \mapsto 2 * (q{+}1) \mapsto r \quad (N_q)$$
$$r \mapsto 3 * (r{+}1) \mapsto 0 \quad (N_r)$$
$$p \mapsto 1 * q \mapsto 2 * r \mapsto 3 \quad (F_{\texttt{val}})$$
$$(p{+}1) \mapsto q * (q{+}1) \mapsto r * (r{+}1) \mapsto 0 \quad (F_{\texttt{next}})$$

(b) Sub-heap expressions

Fig. 3. A list example.

Essentially, a non-null value p is a valid node-pointer for $type \in \mathsf{Types}$ if the contiguous addresses $p, p+1, .., p+|type|-1$ are allocated in the corresponding field heaps. For example, q from Figure 3b is valid since $q \in dom(F_{\texttt{val}})$ and $q+1 \in dom(F_{\texttt{next}})$.

In order for a data-structure $\mathcal{H}$ to be valid, all non-null values p stored in any $field \in \mathsf{PtrField}$ must be valid node-pointers of the corresponding $type$. Thus, the graph structure represented by $\mathcal{H}$ is *closed*, i.e., no invalid (uninitialized, wild, or dangling) links.

Definition 2 (*Closed*)
Field heaps $F_{field_1}, .., F_{field_{|\mathsf{Fields}|}}$ are *closed* if for all $field \in \mathsf{PtrFields}$ and for all p, v such that $(p, v) \in F_{field}$, then v is a valid T-node-pointer (Definition 1) where $typeof(field) = (T\ *)$. □

We define:

- $\mathsf{node}_{type}(p, F_1, .., F_m)$ to be the relation satisfying Definition 1 for field heaps $\{F_1, .., F_m\}$;
- $\mathsf{closed}(F_1, .., F_m)$ to be the relation satisfying Definition 2.

Our DSIC $\mathcal{D}(\mathcal{H})$ is defined as follows: Given the set Types, we derive the sets Fields and $\mathsf{PtrFields}$. The heap $\mathcal{H}$ must be partitionable into field heaps, and the field heaps must be closed under Definition 2:

$$\mathcal{H} \doteq F_1 * \cdots * F_m \wedge \mathsf{closed}(F_1, \cdots, F_m) \qquad (\textsc{Closed})$$

At a given program point, there may be zero or more variables $p_1, .., p_n$ pointing to nodes of types $T_1, .., T_n \in \mathsf{Types}$. These pointers must be valid under Definition 1, i.e.:

$$\mathsf{node}_{T_1}(p_1, F_1, .., F_m) \wedge \cdots \wedge \mathsf{node}_{T_n}(p_n, F_1, .., F_m) \qquad (\textsc{Ptrs})$$

We can now define the integrity constraint $\mathcal{D}$:

Definition 3 (*Data-structure Integrity Constraint*)
The *data-structure integrity constraint* $\mathcal{D}$ is defined by combining the above components (via textual substitution) as follows:

$$\mathcal{D}(\mathcal{H}, p_1, .., p_n) \stackrel{\mathsf{def}}{=} \exists F_1, \cdots, F_m : (\textsc{Closed}) \wedge (\textsc{Ptrs}) \quad \square$$

Spatial Memory Safety. The basic analysis of Section 3 assumes that all memory outside the data-structure is *unmapped*, which is unrealistic in practice. We extend our memory model to account for some arbitrary context of mapped memory by splitting the

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

global heap $\mathcal{H}$ into a *footprint heap* Fp and a *context heap* Cxt as follows: $\mathcal{H} \doteq Fp * Cxt$. The data-structure resides in the footprint heap Fp, and the context heap Cxt represents any other mapped memory, such as the stack, globals, free-lists, etc. Buggy code may access Cxt via a *spatial memory error*, such as an object bounds overflow, thus violating memory safety. To detect such errors, we extend the DSIC as follows:

Definition 4 (*Data-structure Integrity Constraint II*)
Let $\mathcal{D}$ be the basic data-structure integrity constraint from Definition 3, then:

$$\mathcal{D}_M(\mathcal{H}, Cxt, p_1, .., p_n) \stackrel{\text{def}}{=} \exists Fp : \mathcal{H} \doteq Fp{*}Cxt \wedge \mathcal{D}(Fp, p_1, .., p_n) \quad \square$$

Finally, to prove spatial memory safety, it must be shown that

$$\{\mathcal{D}_M(\mathcal{H}, Cxt)\}\ P; (x := *p)\ \{p \notin dom(Cxt)\} \qquad \text{(Spatial Memory Safety)}$$

for all reads of pointer p. Likewise, we similarly must verify all writes $(*p := x)$.

4 Solving for Data-Structures

The $\mathcal{D}_M$-analysis depends on determining the validity of the *Verification Conditions* (VCs) generated by symbolic execution, which are of the form:

$$path(\mathcal{H}, Cxt, F_1, .., F_m) \models \exists Fp', F'_1, .., F'_m : post(\mathcal{H}, Cxt, F'_1, .., F'_m) \qquad \text{(VC)}$$

where *path* is the path constraint, *post* is the post-condition, $\mathcal{H}$ is the global heap, Cxt is the context heap, $F_1, .., F_m$ are the initial field heaps, Fp' is the modified footprint and $F'_1, .., F'_m$ are the modified field heaps. Validity can be established by a two-step process: (1) generating witnesses W for $Fp', F'_1, .., F'_m$; and (2) proving that $(W \wedge path \wedge \neg post)$ is *unsatisfiable* using a solver. Witnesses W are built using the following schema:

$$W \stackrel{\text{def}}{=} (Fp' = \mathcal{H} - Cxt) \wedge X_{f \in \mathsf{Fields}} \wedge Y_{p \in \mathsf{Allocs}} \quad Y_p \stackrel{\text{def}}{=} Z^p_{f \in \mathsf{typeof}(*p)}$$
$$X_f \stackrel{\text{def}}{=} F'_f \subseteq Fp' \wedge dom(F_f) \subseteq dom(F'_f) \qquad Z^p_f \stackrel{\text{def}}{=} p + \mathsf{offsetof}(\mathsf{typeof}(*p), f) \in dom(F'_f)$$

Here $T_{x \in \{a,..,z\}}$ is shorthand for $(T_a \wedge .. \wedge T_z)$, and Allocs is defined to be all allocated pointers (i.e. $p = \mathsf{malloc}(..)$)) in *path*. Intuitively, Fp' is the heap difference $\mathcal{H} - Cxt$, and F'_{field} is the heap that is (1) a sub-heap of Fp', and (2) has the same domain as F_{field} save for any new addresses created by allocations.

The next step is to prove that the quantifier-free formula $(W \wedge path \wedge \neg post)$ is unsatisfiable. For this we use a combination of an integer solver, an extension of the heap solver from (Duck *et al.* 2013) (a.k.a., the $\mathcal{H}$-solver) and a specialized solver for data-structure constraints defined below (a.k.a., the $\mathcal{D}$-solver). The $\mathcal{D}$-solver is implemented using the *Constraint Handling Rules* (CHR) solver language (Frühwirth 2009) using the following basic solver schema customized for the types declared by the program:

$$\mathsf{closed}(F_0, .., F_m) \wedge (p, v) \in F_f \Longrightarrow \mathsf{node}_{type}(v, F_0, .., F_m) \text{ where } f \in \mathsf{PtrFields} \text{ and } f \in type$$
$$\mathsf{node}_{type}(p, F_0, .., F_m) \Longrightarrow p = 0 \vee \Big(\bigwedge_{f \in type} p + \mathsf{offsetof}(type, f) \in dom(F_f) \Big)$$

The rules encode the greatest relations satisfying Definitions 2 and 1 respectively.

Given a set of types, the schema is automatically instantiated to generate a specialized CHR solver (the $\mathcal{D}$-solver) for the corresponding $\mathcal{D}$-constraints. The $\mathcal{D}$-solver can then be

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

$$
\begin{array}{lll}
1) & \{\mathcal{H} \doteq Fp{*}Cxt, Fp \doteq F_{\mathtt{val}}{*}F_{\mathtt{next}}, \mathsf{node}(xs, F_{\mathtt{val}}, F_{\mathtt{next}}), \underline{xs{+}1 \in dom(Cxt)}\} & (\mathcal{H}) \\
2) & \{\mathcal{H} \doteq Fp{*}Cxt, Fp \doteq F_{\mathtt{val}}{*}F_{\mathtt{next}}, \underline{\mathsf{node}(xs, F_{\mathtt{val}}, F_{\mathtt{next}})}, \underline{xs{+}1 \in dom(Cxt)}, xs > 0\} & (\mathcal{D}) \\
3a) & \{\mathcal{H} \doteq Fp{*}Cxt, Fp \doteq F_{\mathtt{val}}{*}F_{\mathtt{next}}, \mathsf{node}(xs, F_{\mathtt{val}}, F_{\mathtt{next}})), xs{+}1 \in dom(Cxt), \underline{xs > 0}, \underline{xs = 0}\} & (I) \\
4a) & \mathit{false} & \\
3b) & \{\underline{\mathcal{H} \doteq Fp{*}Cxt, Fp \doteq F_{\mathtt{val}}{*}F_{\mathtt{next}}}, \mathsf{node}(xs, F_{\mathtt{val}}, F_{\mathtt{next}}), \underline{xs{+}1 \in dom(Cxt)}, xs > 0, & (\mathcal{H}) \\
 & \underline{xs{+}1 \in dom(F_{\mathtt{next}})}\} & \\
4b) & \mathit{false} &
\end{array}
$$

Fig. 4. Solver steps for an example memory safety VC.

used to solve VCs using the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) system (Duck 2012; Duck 2013) in combination with existing heap, integer, and equality built-in solvers. For example, assuming $\mathsf{Types} = \{\mathtt{list\_node}\}$, the corresponding $\mathcal{D}$-solver is:

$$\mathsf{closed}(F_{\mathtt{val}}, F_{\mathtt{next}}) \wedge (p, v) \in F_{\mathtt{next}} \implies \mathsf{node}(v, F_{\mathtt{val}}, F_{\mathtt{next}})$$
$$\mathsf{node}(p, F_{\mathtt{val}}, F_{\mathtt{next}}) \implies p = 0 \vee \big(p \in dom(F_{\mathtt{val}}) \wedge p{+}1 \in dom(F_{\mathtt{next}})\big)$$

Consider the statement $S = (\mathtt{xs{=}xs{->}next})$. Assuming that the $\mathcal{D}_M$ property (Definition 4) holds before S, we can prove S to be memory safe using the VC:

$$\mathcal{H} \doteq Fp{*}Cxt \wedge Fp \doteq F_{\mathtt{val}}{*}F_{\mathtt{next}} \wedge \mathsf{node}(xs, F_{\mathtt{val}}, F_{\mathtt{next}}) \models xs{+}1 \notin dom(Cxt)$$

This VC is valid iff the constraints in Figure 4 1) are unsatisfiable. The solver steps are shown in Figure 4. Here $(\mathcal{H})$, $(\mathcal{D})$, and (I) represent inferences made by the $\mathcal{H}$-solver, $\mathcal{D}$-solver, and integer solver respectively. The constraints used by each inference step are underlined. Step 2) introduces a disjunction which leads to two branches 3a) and 3b). Since all branches lead to *false* the original goal is unsatisfiable, hence proving the VC is valid.

One of the main features of the SMCHR system is the ability to extend existing built-in solvers with new constraints implemented using CHR. For example, the built-in $\mathcal{H}$-solver works by propagating *heap element* constraints of the form $((p, v) \in H)$ or $(p \in dom(H))$ (Duck *et al.* 2013). The $\mathcal{D}$-solver extends the $\mathcal{H}$-solver with new rules that interact with these constraints. Solver communication is two-way, e.g., the $\mathcal{D}$-solver may propagate new element constraints (e.g., the `node` rule), or may match element constraints propagated by the $\mathcal{H}$-solver (e.g., the `closed` rule).

It is also possible to instantiate the $\mathcal{D}$-solver schema with multiple types. For example, assuming $\mathsf{Types} = \{\mathtt{list\_node}, \mathtt{tree\_node}\}$, the following $\mathcal{D}$-solver rules will be generated:

$$\mathsf{closed}(F_{\mathtt{val}}, F_{\mathtt{next}}, F_{\mathtt{elem}}, F_{\mathtt{left}}, F_{\mathtt{right}}) \wedge (p, v) \in F_{\mathtt{next}} \implies \mathsf{node}_{\mathtt{list\_node}}(v, F_{\mathtt{val}}, F_{\mathtt{next}})$$
$$\mathsf{closed}(F_{\mathtt{val}}, F_{\mathtt{next}}, F_{\mathtt{elem}}, F_{\mathtt{left}}, F_{\mathtt{right}}) \wedge (p, v) \in F_{\mathtt{left}} \implies \mathsf{node}_{\mathtt{tree\_node}}(v, F_{\mathtt{elem}}, F_{\mathtt{left}}, F_{\mathtt{right}})$$
$$\mathsf{closed}(F_{\mathtt{val}}, F_{\mathtt{next}}, F_{\mathtt{elem}}, F_{\mathtt{left}}, F_{\mathtt{right}}) \wedge (p, v) \in F_{\mathtt{right}} \implies \mathsf{node}_{\mathtt{tree\_node}}(v, F_{\mathtt{elem}}, F_{\mathtt{left}}, F_{\mathtt{right}})$$
$$\mathsf{node}_{\mathtt{list\_node}}(p, F_{\mathtt{val}}, F_{\mathtt{next}}) \implies p = 0 \vee \big(p \in dom(F_{\mathtt{val}}) \wedge p{+}1 \in dom(F_{\mathtt{next}})\big)$$
$$\mathsf{node}_{\mathtt{tree\_node}}(p, F_{\mathtt{elem}}, F_{\mathtt{left}}, F_{\mathtt{right}}) \implies$$
$$p = 0 \vee \big(p \in dom(F_{\mathtt{elem}}) \wedge p{+}1 \in dom(F_{\mathtt{left}}) \wedge p{+}2 \in dom(F_{\mathtt{right}})\big)$$

Handling Negation. Some VCs may contain negated $\mathcal{D}$-constraints `node` and `closed`. We can eliminate all negated $\mathcal{D}$-constraints by applying the following rewrite rules:

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

						malloc			zmalloc		
	Struct.	#Nd	Sh?	*Func.*	LOC	Time	$\mathcal{D}$	M	Time	$\mathcal{D}$	M
GLib	*doubly linked-list*	1+3	✗	foreach append last prepend insert nth insert_before concat remove _remove_link remove_all remove_link delete_link copy copy_deep reverse nth_prev nth_data find find_custom position index first length insert_sorted insert_sorted_real insert_sorted_with_data sort sort_real sort_with_data sort_merge	545	0.52	28	**33**	0.48	**33**	**33**
	singly linked-list	1+2	✗	foreach append last prepend insert insert_before concat remove remove_all remove_link _remove_link delete_link copy copy_deep reverse nth nth_data find find_custom position index length insert_sorted insert_sorted_real insert_sorted_with_data sort sort_real sort_with_data sort_merge	507	0.25	27	**31**	0.26	**31**	**31**
	red-black tree	2+14	✗	remove_all insert replace remove steal lookup find_node foreach first_node node_next traverse node_pre_order node_in_order node_post_order search node_search height nnodes	858	4.05	**27**	**27**	4.02	**27**	**27**
VF	*binary tree*	1+3	✗	init_tree free_tree contains add maximum remove main	157	0.20	6	**7**	0.20	**7**	**7**
	graph	1+4	✓	schorr_waite	38	0.06	**1**	**1**	0.06	**1**	**1**
libf	*234 tree*	1+8	✓	_tree_singleton tree_is_empty tree_is_singleton _tree_search tree_search_any tree_search_min tree_search_max tree_search_lt _tree_size tree_depth _tree_foldl _tree_map	452	5.94	**12**	**12**	5.98	**12**	**12**
	23 finger tree	3+17	✓	_seq_is_empty _seq_length dig_length tree_length _seq_lookup tree_lookup dig_lookup seq_push_front _seq_replace_front _seq_peek_front _seq_foldl tree_foldl dig_foldl _seq_map tree_map dig_map	830	62.93	**21**	**21**	64.09	**21**	**21**

Fig. 5. $\mathcal{D}_M$-analysis benchmarks for safe library code.

$$\neg\mathsf{closed}(F_1, .., F_m) \longrightarrow \bigvee_{field \in \mathsf{PtrFields}} \left(\begin{array}{l} (s,t) \in F_{field} \wedge \\ \neg\mathsf{node}_{type}(t, F_1, .., F_m) \end{array} \right) \qquad \neg\mathsf{node}_{type}(p, F_1, .., F_m) \longrightarrow p \neq 0 \wedge \left(\bigvee_{field \in type} p + i \notin dom(F_{field}) \right)$$

where $typeof(field) = (type\ *)$, index $i = offsetof(field, type)$, and variables s, t are assumed fresh. These rules implement the negations of Definitions 1 and 2 respectively. For example, $\neg\mathsf{closed}(F_{\mathtt{val}}, F_{\mathtt{next}})$ can be rewritten to

$$(s,t) \in F_{\mathtt{next}} \wedge t \neq 0 \wedge \big(t \notin dom(F_{\mathtt{val}}) \vee t{+}1 \notin dom(F_{\mathtt{next}})\big)$$

That is, in order for $\mathsf{closed}(F_{\mathtt{val}}, F_{\mathtt{next}})$ to be violated, there must exist a heap cell $(s,t) \in F_{\mathtt{next}}$ such that (1) t is non-null, and (2) t does not point to a valid node, i.e. $(t \notin dom(F_{\mathtt{val}}))$ or $(t{+}1 \notin dom(F_{\mathtt{next}}))$. As with the CHR rules, the rewrite rules are generated automatically.

5 Experiments

We have implemented a prototype $\mathcal{D}_M$-analysis tool (called $\mathcal{D}$-tool) as a (LLVM 2018) plug-in. The tool takes as input a C program that is first converted into the LLVM *Intermediate Representation* (IR) using the `clang` front-end. The plug-in implements the $\mathcal{D}_M$-analysis as described in Section 3, and automatically generates a specialized solver as described in Section 4. The VCs are solved using a constraint solver back-end, namely the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) (Duck 2012) system, using the generated solver in combination with existing built-in heap, integer and equality solvers. The SMCHR system also supports goal transformation using rewrite rules, which is used to implement negation. The result is either SAFE if all generated VCs are proved valid, or (possibly) UNSAFE otherwise. The entire process (i.e. compilation, VC generation, solver generation, and solving) is automatic. All experiments were run on an Intel i7-4770 CPU clocked at 3.4GHz.

$\mathcal{D}$-tool Verification on Safe Modules. As the $\mathcal{D}$-tool can analyze partial programs—one important use case is to analyze libraries (or modules). Figure 5 tests the $\mathcal{D}$-tool against several memory safe functions that manipulate data-structures sourced from the following C libraries: the GNU GLib library (version 2.38.0) representative of real library

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

Type	*Tool*	Lang.	Static?	Auto?	Modular?	Mem. Safety?	Data. Structs?
Sep. Log.	SmallFoot	custom	✓	✗	✓	✓	*lists, trees*
	SpaceInvader	C	✓	✓	✗	✓	*lists*
	SLAyer	C	✓	✓	✗	✓	*lists*
	Predator	C	✓	✓	✗	✓	*lists*
	VeriFast	C	✓	✗	✓	✓	*any* (limited)
BMC	LLBMC	C	✓	✓	✗	*limited*	*any* (bounded)
	CBMC	C	✓	✓	✗	*limited*	*any* (bounded)
BC	LowFat	C/C++	✗	✓	✗	*limited*	*any*
	$\mathcal{D}$-tool	C	✓	✓	✓	*limited*	*any*

Fig. 6. Summary of related tools and trade-offs.

code used by a large number of programs, Verifast (abbr. VF) (Jacobs *et al.* 2011) distribution (manually verified safe modules), and the `libf` library[2]. These benchmarks test a wide variety of data-structure types and shapes, including: singly-linked-lists, doubly-linked-lists, red-black-trees, binary-trees, binary-graphs, 234-trees, and 23-finger-trees (Hinze and Paterson 2006). In Figure 5, #Nd is the pair *nodes*+*fields* where *nodes* is the number of node types and *fields* is the number of fields used by the data-structure, (Sh?) indicates whether the data-structure is designed for *sharing* (i.e., each node may have multiple parent nodes), LOC is the total source-lines-of-code, Time is the total time (in seconds), and $\mathcal{D}$/M is the number of functions proven $\mathcal{D}$-safe/memory-safe (see (Spatial Memory Safety)) respectively. Ideal results are highlighted in **bold**. We test two versions of the analysis: *uninitializing* `malloc` and *zero-initializing* `zmalloc`, such as that used by (Boehm and Weiser 1988).

The GLib benchmarks represent standard C data-structures, namely linked-lists (singly or doubly) and trees (red-black balanced binary trees). The typical usage of GLib assumes no data-structure sharing, so each node has at most one parent node. The Verifast benchmarks contain an alternative tree implementation, and binary graphs. For the graph benchmark, we verify the $\mathcal{D}_M$-safety of the *Schorr Waite algorithm*.[3] The `libf` library implements 234-trees (for immutable maps and sets) and 23-finger-trees (Hinze and Paterson 2006) (for immutable sequences). Finger trees are a relatively complex data-structure, with 3 node types and an intricate shape. Furthermore, the `libf` library employs automatic memory management via garbage collection, and is specifically designed to allow data-structure sharing.

Figure 5 shows that the $\mathcal{D}_M$-analysis performs well on library data structures, with all functions automatically verified to be safe under zmalloc. This result is sufficient for programs using allocators such as (Boehm and Weiser 1988). For malloc the results were less precise, with some $\mathcal{D}$ VCs failing because of partially initialized data-structures. As future work, the $\mathcal{D}$-analysis could be improved by considering weaker forms of the DSIC to account for uninitialized fields.

Comparing Memory Safety Tools. We also compare against several existing memory safety analysis tools, summarized in Figure 6, which are classified into four main types: (1) our $\mathcal{D}_M$-analysis tool; (2) *Separation Logic*-based analysis tools such as SmallFoot (Berdine *et al.* 2005), SpaceInvader (Distefano *et al.* 2006), SLAyer (Berdine *et al.* 2011), Predator (Dudka *et al.* 2011; Dudka *et al.* 2013) and Verifast (Jacobs *et al.* 2011); (3) *Bounded Model Checking* (BMC) based analysis tools such as LLBMC (Merz *et al.* 2012) and CBMC (Kroening and Tautschnig 2014); and (4) *Bounds Checking*

[2] `https://github.com/GJDuck/libf`
[3] Verifast verifies the Schorr Waite algorithm for trees and not general graphs.

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

		List						DAG					Graph				
Type		Analysis			BMC		BC	Analysis			BMC		Analysis			BMC	
Prog.	Safe?	$\mathcal{D}$-tool	Invader	Predator	CBMC	LLBMC	LowFat	$\mathcal{D}$-tool	Invader	Predator	CBMC	LLBMC	$\mathcal{D}$-tool	Invader	Predator	CBMC	LLBMC
`overlap_node`	N	**U**	c	**U**	n	n	n	**U**	c	t.o.	b	**B**	**U**	c	**U**	**B**	**B**
`wrong_node`		**U**	**U**	**U**	n	**B**	**C**	**U**	s	t.o.	**B**	b.r.	**U**	t.o.	t.o.	**B**	**B**
`wrong_size`		**U**	s	**U**	n	**B**	**B**	**U**	s	**U**	**B**	**B**	**U**	t.o.	**U**	**B**	**B**
`not_array`		**U**	s	**U**	**B**	**B**	**B**	**U**	s	t.o.	**B**	**B**	**U**	t.o.	t.o.	**B**	**B**
`cast_int`		**U**	**U**	**U**	**B**	**B**	n	**U**	**U**	t.o.	**B**	**B**	**U**	t.o.	t.o.	**B**	b.r.
`uninit_ptr`		**U**	c	**U**	**B**	**B**	n	**U**	**U**	t.o.	**B**	**B**	**U**	t.o.	**U**	**B**	**B**
`uninit_ptr_stk`		**U**	**U**	**U**	**B**	**B**	n	**U**	**U**	t.o.	**B**	**B**	**U**	t.o.	t.o.	**B**	**B**
`arith_ptr`		**U**	s	**U**	**B**	**B**	**B**	**U**	s	t.o.	**B**	**B**	**U**	t.o.	t.o.	**B**	**B**
`safe`	Y	**S**	**S**	**S**	**N**	**N**	**N**	**S**	**S**	t.o.	b.r.	b.r.	**S**	t.o.	t.o.	b.r.	b.r.
Score		**9**	4	**9**	6	8	5	**9**	4	1	7	7	**9**	0	3	8	7

Fig. 7. Comparison versus various tools. Key: S/s=safe, U/u=unsafe, B/b=bug-detected, N/n=no-bug-detected, C/c=crash, t.o.=time-out, b.r.=bound-reached, uppercase/bold=positive-result, lowercase=negative-result.

(BC) instrumentation tools such as LowFat (Duck and Yap 2016; Duck *et al.* 2017). Different approaches have different trade-offs: (Static?) whether the tool is based on static program analysis; (Auto?) whether the tool is fully automatic, or requires user intervention (e.g., annotations); (Modular?) whether the tool can be used to analyze individual functions (i.e., suitable for libraries), or requires a complete program including an entry point (e.g., the `main` function); (Mem. Safety?) whether the tool checks for all types of classical memory errors (including null-pointers), otherwise the tool is *limited* to some specific subset; (Data. Structs?) lists the types of graph-based data-structure that are compatible with the tool. Clearly there are different trade-offs between the different classes of tools. Separation Logic-based tools can be used to prove "full" memory safety, including null-pointer and temporal memory errors (use-after-free), but are either (1) limited to narrow classes of data-structures, such as lists, or (2) are not automatic and require user annotations. In contrast, the $\mathcal{D}$-tool targets specific memory errors (spatial), but is not limited to specific types of data-structures. The $\mathcal{D}$-tool does not target use-after-free errors which typically depend on the data-structure shape—i.e., that the freed node is not *shared* thereby creating a dangling pointer. Only 3 tools (SmallFoot, Verifast, $\mathcal{D}$-tool) are modular. In contrast, the other tools require the whole program for analysis. BMC-based tools are automatic but check for weaker notions of memory safety. Bounded model checking may also fail to detect errors that are beyond the search horizon of the tool. Dynamic bounds checking differs from static analysis-based methods in that it cannot be used to prove that the whole program is error free. At best, dynamic analysis tools can only prove that specific paths are error free.

For our experiments, we compare variants of a simple *safe program* consisting of the following basic template:

```
list_node *xs = make_list(n); set(xs, m, v);
```

Here the `set` function is defined in Section 3, and `make_list` constructs a linked-list of length n. We also evaluate several unsafe variants of the basic template, including:

- `overlap_node`: list with overlapping nodes, e.g., Figure 2(g);
- `wrong_node`: using the wrong node type, e.g. `list_node` for a `tree_node` function;
- `wrong_size`: passing the wrong size to `malloc`;
- `not_array`: attempt to access a list element via an array subscript, e.g. `xs[1].next`;

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

- `cast_int`: manufacture an invalid pointer from an integer, e.g. (`list_node` $*$)i for integer i;
- `uninit_ptr`: neglecting to initialize a pointer, e.g. Figure 2(f);
- `uninit_ptr_stk`: neglecting to initialize a pointer on the stack; and
- `arith_ptr`: arbitrary pointer arithmetic, e.g. `(xs-3)->next`.

Each unsafe variant is *exploitable* in that it demonstrably (by compiling and running the program) overwrites memory outside of the footprint. In addition to bounded linked-lists (*List*), we also test a variant that uses parameterized *DAG* and a parameterized graph *Graph* in place of lists.

The experimental results are shown in Figure 7 with the result key summarized by the caption. In the ideal case, we expect the following: static analysis tools should report {S, U}; BMC-tools should report {B, N}; and dynamic bounds checking tools should report {B, C, N}. All tools were fast ($<10s$) provided no timeout/bounds-reached condition occurred. Our experimental comparison excludes SmallFoot (no C support), SLAyer (crashed with error), Verifast (requires manual proofs), and LowFat for *DAG* and *Graph* (bug not reachable). The $\mathcal{D}$-tool performs as expected (total score 27/27) for all data-structure shapes. The results for SpaceInvader were mixed, even for lists. In contrast, Predator performed flawlessly for list data-structures but less well in the *DAG* and *Graph* tests. For the latter, Predator appears to resort to (infinite) unfolding leading to timeouts. The results for the BMC-based tools are also generally positive. CBMC and LLBMC detect most memory errors for the unsafe test cases, demonstrating that the BMC approach is effective at detecting bugs. The total scores are LLBMC (22/27) and CMBC (21/27). There are also some anomalous results, e.g., SpaceInvader reports unsafe programs as safe. CBMC reports a false null-pointer error for the *DAG*/`overlap_node` test case. LowFat, as a bounds checker, primarily detects errors relating to bad pointer arithmetic, and may not detect memory errors relating to bad casts (type confusion) or uninitialized pointers. We highlight that, while different tools embody different tradeoffs (Figure 6), the $\mathcal{D}$-tool focuses on general data structures, modularity and (limited) memory safety.

6 Conclusion

This paper presented a shape neutral data-structure analysis for low-level heap manipulating programs. The analysis validates several key properties of graph-based data-structures including the validity of nodes, pointers, and the separation between nodes. Such properties are standard for graph-based data-structures implemented in idiomatic C. Our approach therefore caters for a broad range of heap-manipulating code.

Our analysis methodology is based on using symbolic execution to generate Verification Conditions (VCs) which are then solved using a specialized data-structure property solver (a.k.a., the $\mathcal{D}$-solver) in combination with built-in heap, integer and equality solvers. The $\mathcal{D}$-solver itself is implemented using Constraint Handling Rules (CHR), and is automatically generated from the type declarations contained within the target program. For solving VCs, our $\mathcal{D}$-tool employs the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) system. The SMCHR system is well suited for this task, as it can handle VCs with a rich Boolean structure, rewrite rules (for negation), CHR solvers (for the $\mathcal{D}$-solver), and allows for the integration of different kinds of solvers (i.e., integer, heap and the $\mathcal{D}$-solver). Our experimental results are promising, with the $\mathcal{D}$-tool able to detect memory errors that are missed by other tools.

https://doi.org/10.1017/S147106841800025X Published online by Cambridge University Press

Acknowledgements

This research was partially supported by MOE2015-T2-1-117 and R-252-000-598-592.

References

Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., and Yang, H. 2007. Shape Analysis for Composite Data Structures. In *Computer Aided Verification*. Springer.

Berdine, J., Calcagno, C., and O'Hearn, P. 2005. SmallFoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*. Springer.

Berdine, J., Cook, B., and Ishtiaq, S. 2011. SLAyer: Memory Safety for Systems-level Code. In *Computer Aided Verification*. Springer.

Boehm, H. and Weiser, M. 1988. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience 18,* 9.

Distefano, D., O'Hearn, P., and Yang, H. 2006. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

Duck, G. 2012. SMCHR: Satisfiability Modulo Constraint Handling Rules. *Theory and Practice of Logic Programming 12,* 4-5, 601–618.

Duck, G. 2013. Satisfiability Modulo Constraint Handling Rules (Extended Abstract). In *International Joint Conference on Artificial Intelligence*. AAAI.

Duck, G., Jaffar, J., and Koh, N. 2013. Constraint-based Program Reasoning with Heaps and Separation. In *Constraint Programming*. Springer.

Duck, G. and Yap, R. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.

Duck, G., Yap, R., and Cavallaro, L. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium*. The Internet Society.

Dudka, K., Peringer, P., and Vojnar, T. 2011. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Computer Aided Verification*. Springer.

Dudka, K., Peringer, P., and Vojnar, T. 2013. Byte-Precise Verification of Low-Level List Manipulation. In *Static Analysis*. Springer.

Frühwirth, T. 1998. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming 37*.

Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press.

Hinze, R. and Paterson, R. 2006. Finger Trees: A Simple General-purpose Data Structure. *Journal of Functional Programming 16,* 2.

Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. 2011. VeriFast: a Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal methods*. Springer.

Kroening, D. and Tautschnig, M. 2014. CBMC: C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

LLVM 2018. `http://llvm.org`.

Matthews, J., Moore, J., Ray, S., and Vroon, D. 2006. Verification Condition Generation Via Theorem Proving. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer.

Merz, F., Falke, S., and Sinz, C. 2012. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *Verified Software: Theories, Tools, Experiments*. Springer.

Reynolds, J. 2002. Separation Logic: A Logic for Shared Mutable Data Objects. In *Logic in Computer Science*. IEEE.