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)): \[ \text{list}(l) \triangleq \exists t. \ (l = 0 \lor l \mapsto t \ast \text{list}(t)) \]. 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) (Fruhwirth 1998; Fruhwirth 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...
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 \( D \). Essentially, \( D \) states that data structures must have valid nodes, valid pointers, and nodes do not intersect. The formal presentation of \( D \) is given in Section 3.

The framework we use to analyze a program in pursuit of our integrity constraint \( D \) 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 \( D \) at certain program points corresponding to a set of cut-points in the control flow graph.
- VCG is performed as follows. Suppose \( D \) 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 \( D \). This is repeated for all cut-points.

This reduces shape neutral data-structure analysis (abbr. to \( D \)-analysis) into proving that Hoare triples of the form \( \{ D \} C \{ D \} \) are valid, where \( C \) is some code fragment (the analysis target) and \( D \) 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, \( \{ D \} C \{ D \} \) is stating that the DSIC \( D \) 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 \( \phi \) with variable \( x' \) substituted for variable \( x \). We assume the standard definitions for sequences \( (s_1; s_2) \), assignment \( x := E \) and \texttt{abort}(). Heap operations use special heap constraints defined below. A triple \( \{ D \} C \{ D \} \) is established by symbolically executing \( D \) through \( C \) using the rules from Figure 1. This process generates a path constraint \( P \). The triple holds iff Verification Condition (VC) \( (P \models D) \) is proven valid with the help of a suitable constraint solver.

\[
\begin{align*}
sp((s_1; s_2), \phi) & \overset{\text{def}}{=} sp(s_2, sp(s_1, \phi)) \\
sp(x := E, \phi) & \overset{\text{def}}{=} x = E[x'/x] \land \phi[x'/x] \\
sp(x := *p, \phi) & \overset{\text{def}}{=} \text{access}(H, p, x) \land \phi[x'/x] \\
sp(*p := E, \phi) & \overset{\text{def}}{=} \text{assign}(H', p, E, H) \land \phi[H'/H] \\
sp(p := \text{malloc}(n), \phi) & \overset{\text{def}}{=} \text{alloc}(H', p, n, H) \land \phi[H'/H, p'/p] \\
sp(\text{abort}(), \phi) & \overset{\text{def}}{=} \text{false}
\end{align*}
\]

(a) Summary of the symbolic execution rules. (b) Program with loops.

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

\[
\{ \mathcal{D} \} \text{S1} \{ \mathcal{D} \} \quad \{ \mathcal{D} \land b_1 \} \text{S2} \{ \mathcal{D} \} \quad \{ \mathcal{D} \land b_2 \} \text{S3} \{ \mathcal{D} \} \\
\{ \mathcal{D} \land \lnot b_2 \} \text{S4} \{ \mathcal{D} \} \quad \{ \mathcal{D} \land \lnot b_1 \} \text{S5} \{ \mathcal{D} \}
\]

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 $\text{Values} \equiv \mathbb{Z}$) and define the set of Heaps to be all finite partial maps between values, i.e., $\text{Heaps} \equiv (\text{Values} \rightarrow_{\text{fin}} \text{Values})$. Let $\text{dom}(H)$ be the domain of the heap $H$. We abuse notation and treat heaps $H$ as sets of (pointer,value) pairs $\{(p,H(p)) \mid p \in \text{dom}(H)\}$. 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 $H \equiv H_1 \uplus H_2$, where $H, H_1, H_2$ are heap variables. Informally, the constraint $H \equiv H_1 \uplus H_2$ 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: $H = H_1 \cup H_2 \land \text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset$.

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:

\[
\text{access}(\mathcal{H}, p, v) \equiv (p, v) \in \mathcal{H} \\
\text{assign}(H, p, v, \mathcal{H}) \equiv \exists w : (p, w) \in H \land \mathcal{H} = (H - \{(p, w)\}) \cup \{(p, v)\} \\
\text{alloc}(H, p, 1, \mathcal{H}) \equiv \exists w : \mathcal{H} = H \cup \{(p, w)\} \land p \notin \text{dom}(H)
\]

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 $p \notin \text{dom}(\mathcal{H})$) 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 $\{\mathcal{D}(\mathcal{H},p_1,\ldots,p_n)\} \ C \ \{\mathcal{D}(\mathcal{H},q_1,\ldots,q_m)\}$ where $C$ is some code fragment (e.g. a function definition), $\mathcal{H}$ is the global program heap, $\{p_1,\ldots,p_n\}$ and $\{q_1,\ldots,q_m\}$ 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
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 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 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:

```c
struct list_node {
    int val;
    list_node *elem;
    list_node *next;
};

struct tree_node {
    list_node *elem;
    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 D-analysis treats (a), (b), (c), (d), (e) as graphs of list_nodes. 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 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 D-analysis:

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

```c
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, \((\text{set}(xs,1,A); \text{set}(xs,1,B))\), that sets the second node’s value to integers \( A \) and \( B \) respectively. However, if \( xs \) was created with \text{make_bad}, the first call to \text{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 \text{set} executes \( A->val=B \) allowing for arbitrary memory to be overwritten.

**Formalization.** We shall now formalize the integrity constraint \( D \). We assume, as given, a set of node types \( \text{Types} = \{ \text{type}_0, \ldots, \text{type}_n \} \) that are used by the program, e.g. list_node and \text{tree_node} defined above. We treat each \( \text{type} \in \text{Types} \) as a set of fields, e.g. \( \text{tree_node} = \{ \text{elem}, \text{left}, \text{right} \} \). W.l.o.g., we shall assume all fields are renamed apart. Given \( \text{Types} \), we define set \( \text{Fields} \) as all fields, and \( \text{PtrFields} \subseteq \text{Fields} \) as all fields with a pointer-to-node type. We also treat \( \text{Fields} \) and \( \text{PtrFields} \) as sequences by choosing an arbitrary field ordering. The sets \( \text{Types} \), \( \text{Fields} \) and \( \text{PtrFields} \) are derived from all \text{struct} declarations in scope.

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

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

\[
p \leftrightarrow 1 * (p+1) \leftrightarrow q * q \leftrightarrow 2 * (q+1) \leftrightarrow r * r \leftrightarrow 3 * (r+1) \leftrightarrow 0
\]

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

**Definition 1 (Node Pointers)**

Let \( \text{type} \in \text{Types} \) be a node type, then value \( p \in \text{Values} \) is a node-pointer if

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

\[1\]

As a simplification, we assume that the \( i \)th field is stored in address \( p + i \), and that \( \text{sizeof}(\text{int}) = \text{sizeof}(\text{void}*). \)
Essentially, a non-null value $p$ is a valid node-pointer for $type \in \text{Types}$ if the contiguous addresses $p, p + 1, \ldots, p + |type| - 1$ are allocated in the corresponding field heaps. For example, $q$ from Figure 3b is valid since $q \in \text{dom}(F_{\text{val}})$ and $q + 1 \in \text{dom}(F_{\text{next}})$.

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

**Definition 2 (Closed)**

Field heaps $F_{\text{field}_1}, \ldots, F_{\text{field}_{|\text{Fields}|}}$ are closed if for all field $\in \text{PtrFields}$ and for all $p, v$ such that $(p, v) \in F_{\text{field}}$, then $v$ is a valid $T$-node-pointer (Definition 1) where $\text{typeof}(\text{field}) = (T \ast)$.

We define:

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

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

\[ H \approx F_1 \ast \cdots \ast F_m \land \text{closed}(F_1, \ldots, F_m) \]  \hspace{1cm} (CLOSED)

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

\[ \text{node}_{T_1}(p_1, F_1, \ldots, F_m) \land \cdots \land \text{node}_{T_n}(p_n, F_1, \ldots, F_m) \]  \hspace{1cm} (PTRS)

We can now define the integrity constraint $D$:

**Definition 3 (Data-structure Integrity Constraint)**

The data-structure integrity constraint $D$ is defined by combining the above components (via textual substitution) as follows:

\[ D(H, p_1, \ldots, p_n) \overset{\text{def}}{=} \exists F_1, \cdot \cdot \cdot, F_m : (\text{CLOSED}) \land (\text{PTRS}) \]  \hspace{1cm} \Box

**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
global heap $\mathcal{H}$ into a footprint heap $Fp$ and a context heap $Cxt$ as follows: $\mathcal{H} \approx Fp \ast Cxt$. 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:

$$\mathcal{D}_M(\mathcal{H}, Cxt, p_1, \ldots, p_n) \overset{\text{def}}{=} \exists Fp: \mathcal{H} \approx Fp \ast Cxt \wedge \mathcal{D}(Fp, p_1, \ldots, p_n)$$

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

$$\{\mathcal{D}_M(\mathcal{H}, Cxt)\} P; (x := *p) \{p \not\in \text{dom}(Cxt)\} \quad \text{(Spatial Memory Safety)}$$

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:

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

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

$$W \overset{\text{def}}{=} (Fp' = \mathcal{H} - Cxt) \wedge X_f \in \text{Fields} \wedge Y_p \in \text{Allocs} \quad Y_p \overset{\text{def}}{=} Z_p = Z_{f \in \text{type}(\ast p)}$$

$$X_f \overset{\text{def}}{=} F'_f \subseteq Fp' \wedge \text{dom}(F_f) \subseteq \text{dom}(F'_f) \quad Z_p \overset{\text{def}}{=} p + \text{offsetof}(\text{type}(\ast p), f) \in \text{dom}(F'_f)$$

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

The next step is to prove that the quantifier-free formula $(W \wedge \text{path} \wedge \neg \text{post})$ 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 (Fruhwirth 2009) using the following basic solver schema customized for the types declared by the program:

$$\text{closed}(F_0, \ldots, F_m) \wedge (p, v) \in F_f \iff \text{node}_f(v, F_0, \ldots, F_m)$$

where $f \in \text{PtrFields}$ and $f \in \text{type}$

$$\text{node}_f(p, F_0, \ldots, F_m) \iff p = 0 \lor \left( \bigwedge_{f \in \text{type}} \text{p + offsetof(type, f) \in dom(F_f)} \right)$$

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
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 \( \text{Types} = \{\text{list\_node}\} \), the corresponding \( \text{D}\)-solver is:

\[
\begin{align*}
\text{closed}(F_{\text{val}}, F_{\text{next}}) \land (p, v) \in F_{\text{next}} & \implies \text{node}(v, F_{\text{val}}, F_{\text{next}}) \\
\text{node}(p, F_{\text{val}}, F_{\text{next}}) & \implies p = 0 \lor (p \in \text{dom}(F_{\text{val}}) \land p+1 \in \text{dom}(F_{\text{next}}))
\end{align*}
\]

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

\[
\begin{align*}
\mathcal{H} & \equiv Fp*Cxt, Fp \simeq F_{\text{val}}*F_{\text{next}}, \text{node}(\text{xs}, F_{\text{val}}, F_{\text{next}}), \text{xs}+1 \in \text{dom}(Cxt) \\
(H) & \\
\mathcal{D} & \equiv Fp*Cxt, Fp \simeq F_{\text{val}}*F_{\text{next}}, \text{node}(\text{xs}, F_{\text{val}}, F_{\text{next}}), \text{xs}+1 \in \text{dom}(Cxt), \text{xs} > 0 \\
(D) & \\
\mathcal{I} & \equiv Fp*Cxt, Fp \simeq F_{\text{val}}*F_{\text{next}}, \text{node}(\text{xs}, F_{\text{val}}, F_{\text{next}}), \text{xs}+1 \in \text{dom}(Cxt), \text{xs} > 0, \text{xs} = 0 \\
(I) & \\
\mathcal{H} & \equiv Fp*Cxt, Fp \simeq F_{\text{val}}*F_{\text{next}}, \text{node}(\text{xs}, F_{\text{val}}, F_{\text{next}}), \text{xs}+1 \in \text{dom}(Cxt), \text{xs} > 0, \text{xs}+1 \in \text{dom}(F_{\text{next}}) \\
(H) &
\end{align*}
\]

This VC is valid iff the constraints in Figure 4 1) are unsatisfiable. The solver steps are shown in Figure 4. Here \( (H) \), \( (D) \), 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 \text{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 \((p \in \text{dom}(H))\) (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 \text{node} rule), or may match element constraints propagated by the \( \mathcal{H} \)-solver (e.g., the \text{closed} rule).

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

\[
\begin{align*}
\text{closed}(F_{\text{val}}, F_{\text{next}}, F_{\text{elem}}, F_{\text{left}}, F_{\text{right}}) \land (p, v) \in F_{\text{next}} & \implies \text{node}\text{-list\_node}(v, F_{\text{val}}, F_{\text{next}}) \\
\text{closed}(F_{\text{val}}, F_{\text{next}}, F_{\text{elem}}, F_{\text{left}}, F_{\text{right}}) \land (p, v) \in F_{\text{left}} & \implies \text{node}\text{-tree\_node}(v, F_{\text{elem}}, F_{\text{left}}, F_{\text{right}}) \\
\text{closed}(F_{\text{val}}, F_{\text{next}}, F_{\text{elem}}, F_{\text{left}}, F_{\text{right}}) \land (p, v) \in F_{\text{right}} & \implies \text{node}\text{-tree\_node}(v, F_{\text{elem}}, F_{\text{left}}, F_{\text{right}}) \\
\text{node}\text{-list\_node}(p, F_{\text{val}}, F_{\text{next}}) & \implies p = 0 \lor (p \in \text{dom}(F_{\text{val}}) \land p+1 \in \text{dom}(F_{\text{next}})) \\
\text{node}\text{-tree\_node}(p, F_{\text{elem}}, F_{\text{left}}, F_{\text{right}}) & \implies p = 0 \lor (p \in \text{dom}(F_{\text{elem}}) \land p+1 \in \text{dom}(F_{\text{left}}) \land p+2 \in \text{dom}(F_{\text{right}}))
\end{align*}
\]

Handling Negation. Some VCs may contain negated \( \mathcal{D} \)-constraints \text{node} and \text{closed}. We can eliminate all negated \( \mathcal{D} \)-constraints by applying the following rewrite rules:
Shape neutral data-structure analysis

Fig. 5. \( D_M \)-analysis benchmarks for safe library code.

\[
\forall \text{field} \in \text{PtrFields} \quad \neg \text{closed}(F_1, ..., F_m) \quad \text{if} \quad (s, t) \in F_{\text{field}} \land \neg \text{node type}(t, F_1, ..., F_m) \quad \text{then} \quad p \neq 0 \land \left( \forall \text{field} \in \text{type} \quad p + i \notin \text{dom}(F_{\text{field}}) \right)
\]

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

\[
(s, t) \in F_{\text{next}} \land t \neq 0 \land \left( t \notin \text{dom}(F_{\text{val}}) \lor t+1 \notin \text{dom}(F_{\text{next}}) \right)
\]

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

5 Experiments

We have implemented a prototype \( D_M \)-analysis tool (called \( D \)-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 \texttt{clang} front-end. The plug-in implements the \( D_M \)-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.

\( D \)-tool Verification on Safe Modules. As the \( D \)-tool can analyze partial programs—one important use case is to analyze libraries (or modules). Figure 5 tests the \( D \)-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
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 \(\text{nodes} + \text{fields}\) where \(\text{nodes}\) is the number of node types and \(\text{fields}\) is the number of fields used by the data-structure, \((\text{Sh?})\) 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 \(D/M\) is the number of functions proven \(D\)-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 \(D_M\)-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 \(D_M\)-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 \(D\) VCs failing because of partially initialized data-structures. As future work, the \(D\)-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 \(D_M\)-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.
(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 $D$-tool targets specific memory errors (spatial), but is not limited to specific types of data-structures. The $D$-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, $D$-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:

\[
\text{list_node} \; *xs = \text{make_list}(n); \; \text{set}(xs, m, v);
\]

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. $xs[1].next$;

![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.](https://doi.org/10.1017/S147106841800025X)

<table>
<thead>
<tr>
<th>Type</th>
<th>List</th>
<th>DAG</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Analysis</td>
<td>BMC</td>
<td>BC</td>
</tr>
<tr>
<td>overlap_node</td>
<td>U c U n n n</td>
<td>U c t.o. B B</td>
<td>U c U B B B</td>
</tr>
<tr>
<td>wrong_size</td>
<td>U s U n B B</td>
<td>U s U B B</td>
<td>U s U B B</td>
</tr>
<tr>
<td>safe</td>
<td>Y S S S N N N N S S t.o. br. b.r. S t.o. t.o. br. b.r.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Score</td>
<td>9 4 9 6 8 5 9 4 1 7 7 9 0 3 8 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- \texttt{cast\_int}: manufacture an invalid pointer from an integer, e.g. (\texttt{list\_node *})\texttt{i} for integer \texttt{i};
- \texttt{uninit\_ptr}: neglecting to initialize a pointer, e.g. Figure 2(f);
- \texttt{uninit\_ptr\_stk}: neglecting to initialize a pointer on the stack; and
- \texttt{arith\_ptr}: arbitrary pointer arithmetic, e.g. (\texttt{xs-3})\rightarrow\texttt{next}.

Each unsafe variant is \textit{exploitable} in that it demonstrably (by compiling and running the program) overwrites memory outside of the footprint. In addition to bounded linked-lists (\textit{List}), we also test a variant that uses parameterized \textit{DAG} and a parameterized graph \textit{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 \{\texttt{S, U}\}; BMC-tools should report \{\texttt{B, N}\}; and dynamic bounds checking tools should report \{\texttt{B, C, N}\}. All tools were fast (<10s) provided no timeout/bounds-reached condition occurred. Our experimental comparison excludes \texttt{SMALL\_FOOT} (no C support), \texttt{SLAYER} (crashed with error), \texttt{Verifast} (requires manual proofs), and \texttt{LowFat} for \textit{DAG} and \textit{Graph} (bug not reachable). The \texttt{D\_tool} performs as expected (total score 27/27) for all data-structure shapes. The results for \texttt{SPACE\_INVADER} were mixed, even for lists. In contrast, Predator performed flawlessly for list data-structures but less well in the \textit{DAG} and \textit{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 \textit{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 \texttt{D\_tool} focuses on general data structures, modularity and (limited) memory safety.

\section{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 \texttt{D\_solver}) in combination with built-in heap, integer and equality solvers. The \texttt{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 \texttt{D\_tool} employs the \texttt{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 \texttt{D\_solver}), and allows for the integration of different kinds of solvers (i.e., integer, heap and the \texttt{D\_solver}). Our experimental results are promising, with the \texttt{D\_tool} able to detect memory errors that are missed by other tools.
Acknowledgements

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

References


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


