A Shape Neutral Analysis for Graph-based Data-structures

Gregory J. Duck  
Department of Computer Science  
National University of Singapore  
gregory@comp.nus.edu.sg

Joxan Jaffar  
Department of Computer Science  
National University of Singapore  
joxan@comp.nus.edu.sg

Roland H. C. Yap  
Department of Computer Science  
National University of Singapore  
ryap@comp.nus.edu.sg

ABSTRACT
Malformed data-structures can lead to runtime errors such as arbitrary memory access or corruption. Low level languages make it easy to have bugs which can lead to malformed data structures as they allow direct pointer manipulation – making reasoning over data-structure properties challenging. In this paper we present a constraint-based program analysis that checks data-structure integrity, w.r.t. given target data-structure properties, suitable for detecting data-structure bugs in low level code. A key property of our analysis is that it is 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. As a result, the analysis can be applied to a wide range of data-structure manipulating programs, including those that use lists, trees, DAGs, etc., all of which are specific kinds of graph data-structures. Furthermore, the analysis is powerful enough to detect certain classes of critical memory errors that can lead to data corruption or information leaks. Our analysis is modular and can be used to analyze code in libraries and components without requiring whole programs. Experimental results show that our approach works well in practice.

1. INTRODUCTION
Programs which deal with data-structures in low-level languages such as C and C++ can have (subtle) bugs due to direct pointer manipulation. Furthermore these bugs may lead to arbitrary memory modification, hence, critical code vulnerabilities. Low level languages neither have strong type systems (to help prevent errors) nor generic runtime protection mechanisms. Program analysis is therefore desirable to reduce or eliminate such bugs. However, automated analysis of such data-structure manipulating programs in low-level languages is a challenging problem. Much of the existing work on data-structure analysis [1, 3, 4, 8] focuses on (or depends on) shape properties, i.e. is the data-structure a tree or linked-lists, etc. There are several factors that complicate automatic reasoning over such properties, including:
1. Shape information is usually implicit, and thus must be inferred from the program.
2. Common data-structure shapes (lists, trees, etc.) have inductive (a.k.a. recursive) definitions.
3. Data-structure safety co-depends on memory safety.

The first problem is that shape information is usually implicit. For example, consider the following generic C struct declaration:

```
struct node { int val; node *next1; node *next2; }
```

This definition can be used to construct various types of data-structures, including (doubly) linked-lists, trees, DAGs or graphs. Shape analysis [1] aims to automatically infer the shape information based on program usage, but is incomplete in general.

Assuming shape information is available, another problem is that many common data-structures have inductive/recursive definitions – complicating automated reasoning. For example, a non-empty list can be recursively defined as a head node joined with a tail list, e.g. in Separation Logic [16]:

```
list(l) ≜ \exists t : l = 0 \lor l \rightarrow t*list(t)
```

Automated reasoning tools must rely on general (but incomplete) reasoning methods (e.g. unfolding + induction [12]), or specialized solvers that only work for a specific data-structure type (e.g. a list theory solver [2]).

The final problem is that data-structure reasoning for low-level programs is (co)dependent on memory safety, i.e. the absence of memory errors such as out-of-bounds (OOB) access that can potentially invalidate (clobber) any data-structure used by the program. Conversely invalid data-structures can cause memory errors.

In this paper we present a shape neutral data-structure analysis that aims to avoid or reduce the complications listed above. In place of precise data-structure definitions, we instead analyze the program against a set of more general data-structure properties that hold for most canonical graph-based data-structures in low-level languages such as C. Thus, we prescribe properties for well-behaved programs (i.e. good software engineering practices for data-structure manipulating code), namely:
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.

Example 1. (Data-structure Correctness) For example, consider the following code that uses struct node (defined above)
to construct a doubly linked-list:

```c
node *n = 0, p = 0;
for (i = 0; i < N; i++) {
    n = (node *)malloc(sizeof(node));
    n->val = i; n->next1 = p; n->next2 = 0;
    p = n;
} return p;
```

The data-structure constructed from this code fragment satisfies all of the above desired data-structure properties, namely: each node is a contiguous regions of memory of the correct size (guaranteed by malloc), the data-structure is closed (no dangling links are created), and no node will overlap with any other node (also guaranteed by malloc). None of these properties depend on the shape of the data-structure.

Importantly, all of the properties 1–4 are independent of the data-structure shape, i.e. the analysis is shape neutral.

The shape neutral analysis eliminates some of the problems listed above. Firstly, we eliminate the need to infer (or assume) a given shape used by the program. Rather the properties are purely dependent on the declared types. Secondly, we eliminate the need for reasoning over arbitrarily complex inductive definitions, which are replaced with more general non-inductive properties. In this paper we show how to automatically derive a constraint solver for the desired properties based on the type declarations contained within the program.

Like more traditional data-structure analysis, our shape neutral data-structure analysis is still (co)dependent on the memory safety of the underlying program. However, we observe that not all memory errors can corrupt memory assuming a modern operating system environment (e.g. Linux or Windows). For example, accessing unmapped memory will lead to a program abort (a.k.a. crash) that cannot corrupt data-structure integrity. We extend the shape neutral data-structure analysis to automatically detect the specific class of memory errors that can corrupt (mapped) memory. We show that, despite analyzing for more general properties, the shape neutral data-structure analysis is nevertheless powerful enough to detect such memory errors.

The main advantages of our shape neutral data-structure analysis are the following:

- **Scope**: The analysis is applicable to a wide variety of data-structures, including lists, trees, DAGs, etc.

- **Modularity**: The analysis can be used to analyze each function/module/library individually. In comparison, many other analysis tools require a complete program making them unsuitable for analyzing libraries or modules.

Most existing automatic data-structure analysis tools, such as [4, 8, 3], are restricted to reasoning over one particular class of data-structures, namely linked-lists. The trade-off is that, for some programs/functions, safety is sometimes inescapably dependant on shape. As such, our analysis is most suited to programs that use automatic memory management, such as C++’s std::shared_ptr (reference counting) [10] or the Boehm garbage collector [5] (for C programs). We highlight that automatic memory management via shared_ptr is standard (considered good) programming practice in modern dialects of C++.

In summary, the main contributions of this paper are:

1. We propose and formalize a shape neutral data-structure analysis for heap manipulating programs implemented in low-level languages such as C and C++. We extend the analysis to detect exploitable memory errors that can corrupt data-structures.

2. We present an implementation of the analysis based on automatic solver generation. The analysis itself utilizes constraint-based symbolic execution and can analyze libraries without whole programs.

3. We present experimental results to evaluate our overall approach. We show that the proposed method is effective on “real world” data-structure manipulating code such as that from the GLib library. We compare favorably against several other memory safety tools (both analysis and runtime-based tools). We show the analysis and tool is more general and can detect errors that other tools cannot handle.

This paper presents a “complete” approach, including analysis formalization, solver algorithms for the desired properties, and an implementation with experimental evaluation.

The paper is organized as follows. Section 2 introduces constraint-based symbolic execution methods for heap manipulating programs. Section 3 formalizes the desired data-structure properties and analysis. Section 4 extends the analysis to detect specific classes of memory errors that can corrupt data-structures. Section 5 presents a solver implementation for the desired data-structure properties. Section 6 presents an experimental evaluation of the proposed solution. Finally, we conclude in Section 7.

2. PRELIMINARIES

The high-level aim of the shape neutral data-structure analysis (abbr. to D-analysis) is to prove Hoare triples of the form \( \{D\} C \{D\} \), where \( C \) is some code fragment (the analysis target) and \( D \) is a desired data-structure invariant (to be defined later).

Symbolic Execution

Our analysis method is based on symbolic execution. As symbolic execution is a standard method for program analysis, we shall only give a brief overview.

We define symbolic execution in terms of a strongest post condition (SPC) predicate transformer semantics. Each symbolic execution rule \( s \) of the form \( sp(s, \phi) = \psi \), where \( s \) is a statement, \( \phi \) a pre-condition and \( \psi \) is the strongest post-condition resulting from symbolically executing \( \phi \) through \( s \). A summary of the symbolic execution rules is shown in Figure 1. Here all dashed variables, e.g. \( x' \), are implicitly existentially quantified. We assume standard definitions for all integer (a.k.a. non-heap) operations, such as integer assignment \( x := E \).

Loops (while \( c \) do \( B \)) are handled by computing a loop invariant \( I \) satisfying \( \phi \models I \) and \( \{I\} B \{I\} \) hold. Function calls \( f() \) are handled by separately proving a triple \( \{P\} f() \{Q\} \), e.g. by analyzing the body of \( f \). Then provided \( \phi \models P \), symbolic execution over function call \( f \) yields \( Q \).

To prove a Hoare triple of the form \( \{\phi\} C \{\psi\} \), we symbolically execute \( \phi \) through all paths in \( C \). Branching (e.g. if-then-else) is handled by computing paths for both branches. Given the resulting set \( P \) of all path constraints, we generate a Verification Condition (VC) of the form \( p \models \phi \) for all \( p \in P \). The Hoare triple is valid if all generated VCs are valid.
Heap Operations
To handle heap operations we employ the $\mathcal{H}$-constraint language from [7]. 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. $\text{Heaps} \equiv (\text{Values} \to \text{Values})$. Separation Logic [16] also works with a similar heap representation. 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 $(p, v), (p, w) \in S \implies v = w$ for all $p, v, w$. A heap partitioning constraint is a formula of the form $H \models H_1 \cup H_2$, where $H, H_1, H_2$ are heap variables. Informally, the constraint $H \models H_1 \cup H_2$ 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:

$$H = H_1 \cup H_2 \land \text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset$$

The symbolic execution rules for heap constraints is also shown 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, namely access, assign, and alloc, defined as follows (in set notation):

$$\text{access}(H, p, v) \overset{\text{def}}{=} (p, v) \in H$$
$$\text{assign}(H, p, v, \mathcal{H}) \overset{\text{def}}{=} \exists w : (p, w) \in H \land \mathcal{H} = (H - \{(p, w)\}) \cup \{(p, v)\}$$
$$\text{alloc}(H, p, n, \mathcal{H}) \overset{\text{def}}{=} \exists w : \mathcal{H} = H \cup \{(p, w)\} \land p \notin \text{dom}(H)$$

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 $p \notin \text{dom}(H)$) behaves the same way as $\text{abort}()$ (see Figure 1). For example, the triple $\{\mathcal{H} = \emptyset\}; p[0] = 3$ holds since pointer $p$ is unmapped.

The handling of memory deallotation is left open. A simple model of deallotation can be defined as follows:

$$\text{dealloc}(H, p, \mathcal{H}) \overset{\text{def}}{=} \exists w : (p, w) \in H \land \mathcal{H} = H - \{(p, w)\}$$

The shape neutral data-structure analysis is most useful for programs that employ automatic memory management (AMM), such as that provided by C++’s shared_ptr or conservative garbage collection such as the Boehm GC [5]. For such programs we can model free as a NOP $sp(\text{free}(p), \phi) = \phi$.

3. DATA-STRUCTURE ANALYSIS
Data-structure analysis (or $D$-analysis) aims to prove that a suitable data-structure invariant is preserved by the program. Conversely, a program fails data-structure analysis if it generates a mal-formed data-structure that violates the invariant. More formally, data-structure analysis aims to prove Hoare triples of the form $\{D(H)\} \ C \ \{D(H)\}$ where $C$ is some code fragment (e.g. a function definition), $D$ is the global program heap and $D$ is a suitable data-structure invariant that is defined below.1 If the analysis is successful, then all execution paths through $C$ preserve the data-structure invariant $D$.

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 definition. A data-structure is considered valid if several 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 to indicate the non-existence of a link, e.g. with a null-terminated linked list.
3. Separated nodes: Two nodes may not overlap and share memory.

These conditions are desirable for most standard graph-based data-structure types in idiomatic C, including any 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.2 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, e.g. binary trees versus doubly linked-lists. 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.

Example 2. (Lists and trees) Consider the following C struct definitions:

1In general the data-structure invariant $D$ may depend on more parameters beyond the global heap $H$, such as the current set of live program variables. However, for brevity, we use the notation $D(H)$ and keep any additional parameters implicit.
2Examples of “non-standard” pointers include XOR-linked-lists and tagged-pointers. Such structures are not covered by our analysis.
list_node *make_bad(void) {
    list_node *xs;
    xs->val = 0;
    xs = malloc(sizeof(list_node)+sizeof(void *));
    xs->next = (list_node *)&xs->next;
    xs->next->next = 0;
    if (xs) xs->val = v; }
    return xs; }

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

Figure 2: List functions for Example 2.

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

Our $D$-analysis is shape neutral by design. For example, the list_node definition can be used to construct: (a) a linked-list; (b) a circular linked-list; (c) a lasso-list; or even (d) a disjoint list, as illustrated below:

(a) | (b) | (c) | (d) | (e) | (f)

Our $D$-analysis simply treats (a), (b), (c), (d) as graphs of list_nodes. List (e) is invalid since the last pointer is dangling (violates valid pointers). Likewise list (f) is invalid since it contains overlapping nodes (violates separated nodes).

Consider the list functions make_bad and set defined in Figure 2. The first function, make_bad, is “malicious” in the sense that it deliberately constructs a malformed linked-list (with overlapping nodes as per list (f)). As a data-structure property has been violated, the make_bad function should therefore fail the $D$-analysis. The second function, set, sets the $i^{th}$ value of a linked-list. This function is “benign” and, therefore should pass the $D$-analysis.

The $D$-analysis is not restricted to lists. For the sake of later examples, we include a possible definition of tree_node, which can be used to construct binary trees, DAGs, graphs, or other exotic data-structures.

Data-structure violations can lead to counter-intuitive behavior. For example, consider the following code fragment:

node *xs = make_bad(); set(xs,1,A); set(xs,1,B); 

This code fragment constructs a list $xs$, and then successively sets the second node’s value to integers $A$ and $B$ respectively. If $xs$ were a “normal” linked-list then the above code fragment is benign, albeit suboptimal. However, since the nodes in $xs$ overlap, 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 therefore sets $A\rightarrow \text{val}=B$. In other words, for a “bad” list $xs$, the code fragment

is equivalent to $\ast A=B$. By carefully choosing integer values $A$ and $B$, a hypothetical attacker could exploit this to arbitrarily overwrite memory (which is usually considered a critical code vulnerability).

Formalization

We shall now formalize the data-structure invariant $D$. 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 tree_node from Example 2. We treat each $\text{type} \in \text{Types}$ as a set of fields, e.g. $\text{tree_node} = \{\text{elem, left, right}\}$. W.l.o.g. we shall assume all fields are renamed apart. Given Types, we define Fields as the set of all fields, and $\text{PtrFields}$ as the set of all fields with a pointer-to-node type:

$$
\text{Fields} \overset{\text{def}}{=} \{\text{field} \mid \text{type} \in \text{Types} \land \text{field} \in \text{type}\}
$$

$$
\text{PtrFields} \overset{\text{def}}{=} \{\text{field} \mid \text{field} \in \text{Fields} \land \text{type} \in \text{Types} \land 
\text{typeof(field)} = (\text{type} \ast)\}
$$

Here typeof returns the declared C type of a field. We also treat Fields and $\text{PtrFields}$ as sequences by choosing an arbitrary field ordering. The sets $\text{Types}$, Fields and $\text{PtrFields}$ can be straightforwardly derived from all 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$^3$ $\text{type}$, then a heap $N_p \in \text{Heaps}$ is a node heap for pointer $p$ if it exclusively spans the contiguous range of addresses $p, p+1, \ldots p+|\text{type}| - 1$.

An alternative way to decompose data-structure is based on fields. Given a valid data-structure $H$ and a field $\text{field} \in \text{Fields}$, then we define the $\text{field heap} F_{\text{field}}$ to be the sub-heap of $H$ containing all address-value pairs associated to the given field.

Example 3. (Field Heaps) For example, suppose $H$ is a 3-node linked-list of type list_node defined above, and encodes the sequence 1,2,3. We assume the nodes have ad-

For our formalism we shall use native pointers. The approach can be easily generalized to pointer wrappers, such as C++’s shared_ptr.

We assume that the $i^{th}$ field is stored in address $p+i$, and that $\text{sizeof(int)} = \text{sizeof(void *)}$. Generalizing such assumptions is left for future work.
addresses \( p, q, \) and \( r \) respectively. Heap \( \mathcal{H} \) is therefore representable (in Separation Logic notation) as follows:

\[
p \mapsto 1 \ast (p+1) \mapsto q \ast q \mapsto 2 \ast (q+1) \mapsto r \ast r \mapsto 3 \ast (r+1) \mapsto 0
\]

Heap \( \mathcal{H} \) contains three node sub-heaps \( N_p, N_q, N_r \in \mathcal{H} \) and two field sub-heaps \( F_{val}, F_{next} \subset \mathcal{H} \) defined as follows:

\[
\begin{align*}
p \mapsto 1 \ast (p+1) & \mapsto q & \text{ (node heap } N_p) \\
q \mapsto 2 \ast (q+1) & \mapsto r & \text{ (node heap } N_q) \\
r \mapsto 3 \ast (r+1) & \mapsto 0 & \text{ (node heap } N_r) \\
p \mapsto 1 \ast q & \mapsto 2 \ast r & \mapsto 3 & \text{ (field heap } F_{val}) \\
(p+1) \mapsto q \ast (q+1) & \mapsto r \ast (r+1) & \mapsto 0 & \text{ (field heap } F_{next})
\end{align*}
\]

The heaps \( \mathcal{H}, N_p, N_q, N_r, F_{val} \) and \( F_{next} \) are illustrated in Figure 3.

The heap \( \mathcal{H} \) is the disjoint-union of all the field heaps, e.g. \( \mathcal{H} \approx F_{val} \ast F_{next} \). 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 type-node-pointer if

\[
\begin{align*}
- & p = 0 \text{ (null pointer)} \text{ or } \\
- & p + i \in \text{dom}(F_{field}) \text{ for each field } \text{type}, \text{ where } \text{field} \text{ is the } i^{th} \text{ field of type}.
\end{align*}
\]

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

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

**Definition 2. (Closed)** Field heaps \( F_{field_1}, \ldots, F_{field_{|\text{Fields}|}} \) are closed if for all field \( \in \text{PtrField} \) where \( \text{typeof}(\text{field}) = (\text{type} \ast) \); and for all \( p, v \) such that \( (p, v) \in F_{field} \), then \( v \) is a valid type-node-pointer as per Definition 1.

For notational convenience, we rename

\[
\{F_{field_1}, \ldots, F_{field_{|\text{Fields}|}}\} = \{F_1, \ldots, F_m\}
\]

by associating each field with an index \( 1..m \) where \( m = |\text{Fields}| + 1 \). Furthermore, we define:

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

Our data-structure invariant \( \mathcal{D}(\mathcal{H}) \) is defined as follows. Given the set \( \text{Types} \), we derive the sets \( \text{Fields}, \text{PtrFields} \), and the set of field heaps \( F_1, \ldots, F_m \) as defined above. The heap \( \mathcal{H} \) must be partitionable into field heaps as follows:

\[
\mathcal{H} \approx F_1 \ast \cdots \ast F_m \quad \text{(PARTITION)}
\]

Furthermore, the field heaps must be closed as per Definition 2:

\[
\text{closed}(F_1, \cdots, F_m) \quad \text{(CLOSED)}
\]

Finally, at any given program point there may be zero or more live variables \( p_1, \ldots, p_n \) containing pointers to nodes (with the corresponding types \( \text{type}_1, \ldots, \text{type}_n \)). All such pointers must be valid as per Definition 1, i.e.

\[
\begin{align*}
& \text{node}_{\text{type}_1}(p_1, F_1, \ldots, F_m) \land \\
& \cdots \land \\
& \text{node}_{\text{type}_n}(p_n, F_1, \ldots, F_m) \quad \text{(POINTERS)}
\end{align*}
\]

We define the data-structure invariant \( \mathcal{D} \) as follows:

**Definition 3. (Data-structure Invariant)** The data-structure invariant \( \mathcal{D} \) is defined by combining the above components (via textual substitution) as follows:

\[
\mathcal{D}(\mathcal{H}, p_1, \ldots, p_n) \overset{\text{def}}{=} \exists F_1, \cdots, F_m : \\
\quad \text{closed}(F_{val}, F_{next}) \land \\
\quad \text{node}_{\text{type}}(p_1, F_1, \ldots, F_m) \land \cdots \land \text{node}_{\text{type}}(p_n, F_1, \ldots, F_m) \quad \text{(POINTERS)}
\]

We see that this definition satisfies the informal conditions introduced above. Namely: valid nodes is enforced by a combination of (CLOSED) and (POINTERS); valid pointers is enforced by (CLOSED) and separated nodes is enforced by (PARTITION).

The \( \mathcal{D} \)-analysis can prove the \( \mathcal{D} \)-unsafety of programs.

**Example 4. (Overlapping Nodes Analysis)** Consider the make_bad function from Example 2. Symbolic execution over the function body yields the path constraint:

\[
\begin{align*}
\text{path}(\mathcal{H}, x_0) &= \left( \mathcal{D}(H_0) \land \text{alloc}(H_0, x_0, 3, H_1) \land \text{assign}(H_1, x_0, 0, H_2) \land \text{assign}(H_2, x_0, 1, H_3) \land \text{assign}(H_3, x_0, 2, 0, H_3) \right) \\
\text{path}(\mathcal{H}, x_0) &= \exists F_{val}, F_{next} : \\
\quad \text{closed}(F_{val}, F_{next}) \land \\
\quad \text{node}_{\text{list}}(x_0, F_{val}, F_{next})
\end{align*}
\]

This VC is not valid. To see why, consider the address \( x_0+1 \in \text{dom}(H) \) and the following proof sketch:

1. \( x_0+1 \in \text{dom}(F_{next}) \) by \( \text{node}_{\text{list}}(x_0, F_{val}, F_{next}) \).
2. \( \text{node}_{\text{list}}(x_0, F_{val}, F_{next}) \) by (1), \( \text{closed}(F_{val}, F_{next}) \) and \( \text{path}(H, x_0) \).
3. \( x_0+1 \in \text{dom}(F_{val}) \) by (2).
4. Contradiction between (1), (3), and \( \mathcal{D} \approx F_{val} \ast F_{next} \).

Informally, the VC does not hold since pointer \( x_0+1 \) (the overlap location) cannot simultaneously be in the \( F_{val} \) and \( F_{next} \) heaps, which are disjoint. The make_bad function therefore \( \mathcal{D} \)-unsafe and fails the analysis.

Unsurprisingly, the make_bad function fails the analysis, indicating that the function is doing something suspicious. In contrast, the benign set function passes the analysis, indicating that the data-structure invariant \( \mathcal{D} \) will be preserved by the function. The \( \mathcal{D} \)-analysis is therefore useful in determining whether programs adhere to the desired data-structure properties.

### 3.1 Additional Considerations

We briefly cover some additional considerations for implementing the \( \mathcal{D} \)-analysis.
Functions and Loops
The analysis assumes that all function calls are \( D \)-safe. Supposing function \( f \) calls function \( g \), then we assume
\[
\{D(H)\} g() \{D(H)\}
\]
holds. This triple can be later established by running the analysis on \( g \), the functions called by \( g \), and so on. The analysis is therefore modular, and can be used to analyze each function individually. Overall program correctness depends on all functions passing the analysis.

Loops are handled by establishing that \( D(H) \) is a loop invariant in the standard way.

Multiple Data-structures
At first glance, the above formalism appears to assume a single data-structure represented by \( D(H) \). It is not uncommon for programs to use multiple disjoint data-structures at once. However, since \( D \) is shape neutral, and allows disjoint graphs, e.g., we can treat multiple data-structures as a single combined data-structure, as illustrated by Example 2.

Interpretation
As \( D \) is an abstraction, failure of the \( D \)-analysis should be interpreted as “possibly” \( D \)-unsafe. In contrast, a function passing the analysis is “definitely” \( D \)-safe. The usefulness of the analysis is to be evaluated via experimentation in Section 6.

4. MEMORY CONTEXT SAFETY
The \( D \)-analysis can be used to verify that a program does not construct malformed or corrupt graph-based data-structures, according to the properties defined in Section 3. Violating such properties can lead to memory errors and arbitrary memory access, as shown in Example 2. In this section we extend the \( D \)-analysis to explicitly check for such memory errors, namely, any memory errors that can access (read or write) to mapped memory outside the footprint of the data-structure. Such errors are critical from a security point of view since arbitrary memory access can lead to memory corruption or information leaks.

The basic analysis of Section 3 assumes that all memory outside the data-structure is unmapped. This assumption is unrealistic in practice (e.g., real programs also have stack, globals, free-lists, etc.). We extend our memory model to account for some arbitrary context of mapped memory by splitting the global heap \( H \) into a footprint heap \( Fp \) and a context heap \( Cxt \) as follows:
\[
H = Fp + Cxt \quad Cxt = H - Fp \quad Fp = H - Cxt
\]

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, and/or caller context memory, etc. We extend the data-structure invariant to account for the footprint and context heaps as follows:

Definition 4. (Data-structure Invariant II) Let \( D \) be the basic data-structure invariant from Definition 3, then:
\[
D_M(H, Cxt, p_1, \ldots, p_n) \overset{\text{def}}{=} \exists Fp : H = Fp + Cxt \land D(Fp, p_1, \ldots, p_n)
\]

Note that \( Cxt \) is not existentially quantified. The extended \( D_M \)-analysis (\( D + \) memory safety analysis) therefore aims to prove triples of the form
\[
\{D_M(H, Cxt)\} C \{D_M(H, Cxt)\} \quad \text{(Target)}
\]
for a given context \( Cxt \). The context \( Cxt \) must be “preserved” by \( C \), i.e., \( C \) does not permanently modify any memory outside the footprint of the data-structure \( Fp \).

Note that the triple (Target) alone does not guarantee that \( C \) does not access context \( Cxt \). Indeed, \( C \) may read from \( Cxt \), or even write to \( Cxt \) provided any modification is undone before \( Cxt \) completes. To guarantee that \( C \) does not access \( Cxt \), we also need to strengthen our analysis method.

First we shall define a context error to be a specific kind of memory error that accesses (reads or writes) to the context heap \( Cxt \), as follows:

Definition 5. (Context Error) Given a context heap \( Cxt \), then a pointer \( p \) dereference is a context error w.r.t \( Cxt \) if \( p \in \text{dom}(Cxt) \).

A context error is a type of memory error in the classical sense, i.e. a read or write to a pointer \( p \) outside of the footprint. The difference is that a context error specifically requires that \( p \) be mapped memory, and thus accessing \( p \) will not cause the program to abort.

We extend the \( D \)-analysis symbolic execution method to additionally detect context errors. Given a code fragment \( C \) containing a statement \( s \) of the form \( (x := e) \) or \( (s := e) \), accessing pointer \( p \), then for all symbolic paths \( path(H, Cxt) \) through \( C \) to \( s \) we prove the VC
\[
\text{path}(H, Cxt) \models p \notin \text{dom}(Cxt) \quad \text{(Context-Safety)}
\]

A context error is detected if any path violates this condition. A code fragment is context safe if no path leads to a context error.

Example 5. (Context Safety) Consider the following unsafe code fragment:
\[
xs = \text{make_list}(); \quad \text{val} = xs[10];
\]
The code constructs a linked-list \( xs \), and then erroneously attempts to retrieve the 11th element using an array subscript. This is a potential memory context error, since the memory corresponding to \( xs[10] \) may be mapped and is not in the footprint for \( xs \).

The lack of context safety can be proven as follows. Assuming that \( \text{make_list} \) is \( D_M \)-safe, then the assignment is context safe iff the VC
\[
H = Fp + Cxt \land F_{val} + F_{next} \land \text{closed}(F_{val}, F_{next}) \land \text{node_list}(xs, F_{val}, F_{next}) \implies xs + 20 \notin \text{dom}(Cxt)
\]
holds. The VC can be shown to be invalid using the following counter example: let \( F_{val} \) be \((xs \mapsto 0)\), \( F_{next} \) be \((xs + 1 \mapsto 0)\), and \( Cxt \) be \((xs + 20 \mapsto 0)\), as illustrated below:

Then the above heaps satisfy the path constraint, but \( xs + 20 \in \text{dom}(Cxt) \) and therefore the VC is invalid. The code fragment is therefore not context safe.

In contrast the following modified code fragment
xs = make_list(); val = xs[0];

is context safe since the path constraint implies \( xs+0 \notin \text{dom}(Cxt) \).

5. SOLVING FOR DATA-STRUCTURES

The \( D_M \)-analysis depends on determining the validity of the Verification Conditions (VCs) generated by symbolic execution. The generated VCs are of the form:

\[
p_{\text{path}}(Cxt) \models p_{\text{post}}(Cxt) \quad (\text{VC})
\]

where \( p_{\text{path}} \) is some path constraint, and \( p_{\text{post}} \) is some desired post-condition (i.e. \( D \)-safety condition, memory context safety condition, loop-invariant or function-call precondition). The (VC) can be expanded into the following form:

\[
\exists x : P \models \exists y : Q
\]

where \( P \) and \( Q \) are conjunctions of heap, integer and \( D \)-constraints (such as node and closed), as demonstrated in Example 5. Proving validity is a two-step process:

1. Generating witnesses \( W \) for \( y \) (if necessary); and
2. Proving that \( P \land W \land \neg Q \) is unsatisfiable using a constraint solver.

Building Witnesses

Some of the generated VCs contain existential variables \( y \) that must be eliminated. This occurs when the analysis generates VCs for \( D \)-safety of the form:

\[
p_{\text{path}}(H, Cxt, Fp, F_1, \ldots, F_m) \models D_M(H, Cxt) \quad (\text{VC-PATH})
\]

where \( p_{\text{path}} \) is some path constraint, \( Fp \) is the initial footprint, and \( F_1, \ldots, F_m \) are the initial field heaps. VC-PATH expands to:

\[
\cdots \models \exists Fp', F_1', \ldots, F_m' : H = Fp' \ast Cxt \land Fp' = F_1' \ast \ldots \ast F_m' \land \cdots
\]

where \( Fp' \) is the modified footprint, and \( F_1', \ldots, F_m' \) are the modified field heaps. The first step is to eliminate the existential quantifiers by building witnesses for \( Fp', F_1', \ldots, F_m' \). We achieve this by generating constraints \( W \) as follows:

- the witness for \( Fp' \) is the unique heap satisfying:

\[
Fp' = H - Cxt \quad (\text{as sets})
\]

- and the witness for \( F_1' \) is the unique heap satisfying:

\[
F_1' \subseteq \text{dom}(F_1) \quad (\text{as sets})
\]

\[
\text{dom}(F_1') \subseteq \text{dom}(F_1) \land p + 1 \in \text{dom}(F_1')
\]

for all corresponding calls \( p = \text{mallo}c(\text{sizeof}(\text{node})) \) along the path, and where field is the \( i \)-th field of node (indexed from 0).

Essentially, the domains of \( F_1' \) and \( F_m' \) are equal save for any new nodes allocated along the path via malloc.

Note that this witness construction method is a heuristic. If the memory allocations are not in a canonical form, i.e. of the form \( \text{malloc}(\text{sizeof}(\text{node})) \), then the VC may be valid but the witness construction method is not applicable. Nevertheless, this method works well in practice for standard idiomatic C programs.

Constraint Solving

The resulting VCs contain a mixture of integer, heap and \( D \)-constraints (node and closed). There are existing solvers for integer and heap [7] constraints. In this section we build a solver for data-structure constraints using the Constraint Handling Rules (CHR) solver scripting language. We choose CHR for two main reasons: (1) it is expressive enough to encode the desired properties, and (2) our underlying solver infrastructure already supports solvers implemented in CHR. The underlying ideas presented here can likely be implemented in other ways.

Each CHR is a rule of the form \( (\text{Head} \implies \text{Body}) \) where Head is a conjunction of constraints, Body is a conjunction or disjunction of constraints, and \( \text{vars}(Head) \subseteq \text{vars}(Head) \). Each rule has both a logical and operational interpretation. The logical interpretation is the implication \( \forall : \text{Head} \implies \text{Body} \). Operationally, each rule is interpreted as a multi-set rewrite rule that rewrites a constraint store \( S \land \text{Head} \) to \( S \land \theta \). Body provides there exists a matching substitution \( \theta \) such that \( \theta \cdot \text{Head} = \text{Head}' \). The rewrite rules are exhaustively applied until a fixed-point is reached, or failure occurs (rewrite to false). Disjunction is handled via backtracking search. For more details about CHR see [9].

The basic idea is to generate a data-structure property solver using CHR and the following schema. The solver depends on the set Types, so a different and customized solver is generated based on the problem being analyzed. Firstly, for each \( F_i \in \text{PtrFields} \) we generate the rule:

\[
\text{closed}(F_0, \ldots, F_m) \land (p, v) \in F_i \implies \text{(CLOSED)}
\]

\[
\text{node}(v, F_0, \ldots, F_m) \implies \text{(NODE)}
\]

where \( \text{typeof}(F_i) = \{\text{types}\} \). The (CLOSED) rule extends the heap element constraint propagation algorithm for the \( \mathcal{H} \)-solver from [7]. Next we generate a rule for each type \( \text{in Types} \).

\[
\text{node}(p, F_0, \ldots, F_m) \implies \text{(NODE)}
\]

\[
p = 0 \lor (p \in \text{dom}(F_i) \land p + n \in \text{dom}(F_i))
\]

where \( \text{type} = \{F_0, \ldots, F_m\} \subseteq \{F_0, \ldots, F_m\} = \text{Fields} \). Rules (CLOSED) and (NODE) encode the greatest relations satisfying Definitions 2 and 1 respectively.

Example 6. (\( D \)-Solver) Assuming that Types = \{list_node\} as defined in Section 3, then the CHR encoding of the closed heap solver is:

\[
\text{closed}(F_{\text{val}}, F_{\text{next}}) \land (p, v) \in F_{\text{next}} \implies \text{node}(v, F_{\text{val}}, F_{\text{next}})
\]

\[
\text{node}(p, F_{\text{val}}, F_{\text{next}}) \implies \text{node}(v, F_{\text{val}}, F_{\text{next}})
\]

Consider the statement \( S = (xs = xs \rightarrow \text{next}) \). Assuming \( D_M \) holds before \( S \), we can prove \( S \) context safe via the VC:

\[
H = Fp \ast Cxt \land Fp = F_{\text{val}} \ast F_{\text{next}} \land \text{node}(xs, F_{\text{val}}, F_{\text{next}}) = \text{xs+1} \notin \text{dom}(Cxt)
\]

This VC is valid if the constraints in Figure 4 1) are unsatisfiable. The solver steps are shown in Figure 4. Here \( \text{O} \), \( \text{D} \), and \( \text{I} \) represent inferences made by the \( \mathcal{H} \)-solver, \( D \)-solver, and an integer solver respectively. The constraints used in the inference 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 that the VC is valid.

Handling Negation

Some VCs may generate negated data-structure property constraints node and closed, e.g. \( P \land W \land \neg Q \) where \( Q = \text{closed}(F_1, ..., F_n) \). We can eliminate all negated \( D \)-constraints by applying the following rewrite rules:

\[
\neg \text{closed}(F_1, ..., F_m) \\
\forall \text{field} \in \text{PtrFields} \left( (s, t) \in F_{\text{field}} \land \neg \text{node}_{\text{type}}(t, F_1, ..., F_m) \right) \quad \text{(NOT-CLOSED)}
\]

\[
\neg \text{node}_{\text{type}}(p, F_1, ..., F_m) \\
p \neq 0 \land \left( \bigvee_{\text{field} \in \text{PtrFields}} p + i \notin \text{dom}(F_{\text{field}}) \right) \quad \text{(NOT-NODE)}
\]

where type\((\text{field}) = (\text{type} \ast)\) in (NOT-CLOSED), index \( i = \text{offsetof} (\text{field}, \text{type}) \) in (NOT-NODE), and variables \( s, t \) are assumed fresh. Rules (NOT-CLOSED) and (NOT-NODE) implement the negations of Definitions 1 and 2 respectively. For example, \( \neg \text{closed}(F_{v1}, F_{next}) \) can be rewritten to

\[
(s, t) \in F_{next} \land t \neq 0 \land (t \notin \text{dom}(F_{va1}) \lor t+1 \notin \text{dom}(F_{next}))
\]

6. EXPERIMENTS

We have implemented a prototype \( D_M \)-analysis tool (called \( D \)-tool) as a LLVM [14] plug-in. The plug-in takes as input the LLVM Intermediate Representation (IR), thus source code is not needed, and generates VCs according to the \( D_M \)-analysis symbolic execution method described in Sections 2 to 4. The tool checks both \( D \)-safety and memory context safety as described in Sections 3 and 4. The VCs are solved using an implementation of the \( D \)-solver as described in Section 5, in combination with standard heap and integer solvers. The \( D \)-solver is scripted using CHR and implemented using the Satisfiability Modulo Constraint Handling Rules (SMCHR) [6] system. Our tool is available for download. All experiments were run on a Intel i5-2500K CPU clocked at 3.3GHz.

Verifying Safety

Figure 5 tests the \( D \)-tool against several memory safe functions that manipulate data-structures from libraries, including Singularly Linked Lists (SLL), Doubly Linked Lists (DLL), Binary Trees (BT), Balanced Binary Trees (BBT) (multi-node type GTree and GTreeNode from GLib), and Binary

\[\text{http://www.comp.nus.edu.sg/~gregory/dtool/}\]

![Figure 4: Solver steps for Example 6.](image)

![Figure 5: \( D_M \)-analysis benchmarks for safe programs.](image)

Graphs (BGs). Our benchmarks are sourced from the GNU GLib library (version 2.38.0) and the Verifast distribution. The following functions were tested:

- GLib GList, GSList, GTree: All functions in the corresponding modules except those that use arrays.\(^6\)
- Verifast tree: functions add, contains, init_tree, maximum, and remove.
- Verifast node: schorr_waite (the Schorr Waite algorithm) adapted for binary graphs.

In Figure 5, LOC is the total source-lines-of-code, Time is the total time (in seconds), \#VC is the number of generated verification conditions, %Pass is the percentage of the VCs that passed, and %Safe is the fraction of functions that were proven to be safe.

Overall, both \( D_M \) analysis performed very well, with the majority (78/84) of functions verified to be safe. The tool fails to prove \( D \)-safety for some functions. For example, consider the following code fragment from insert (for GSList):\(^7\)

\[
\text{new_list} = \text{malloc}(\text{sizeof}(\text{node})); \ldots
\]

\[
\text{while } ((\text{position}-- > 0) \&\& \text{tmp_list}) \{ \ldots
\]

\[
\text{new_list->next} = \text{prev_list->next};
\]

Our simple strategy of requiring \( D_M \) to be a loop invariant is insufficient for this example, since the initialization of new_list is postponed until after the loop. The \( D_M \)-analysis will pass this example if either (after modification) initialization occurs before the loop, or under the assumption that malloc returns zeroed memory.\(^7\) If we assume the latter, then all (84/84) functions pass the analysis and are therefore \( D_M \)-safe.

Tool Comparison

Several tools exist that aim to detect memory errors either through program analysis or runtime checking. In this section, Support for array manipulating programs is future work. Some implementations of malloc, such as that provided by the Boehm GC [5], satisfy this assumption.
We deliberate keep the test suite as simple as possible to give each tool the best chance of success. Each unsafe variant introduces a context error. The variants are:

- overlap_node: exploitable from Example 2;
- 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 (see Example 5);
- 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. by removing ys->next = xs from make_list;
- uninit_ptr_stk: neglecting to initialize a pointer on the stack; and
- arith_ptr: unsafe 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, which could be exploited as the basis of an attack. In addition to bounded linked-lists, we also test a variant that uses parameterized binary trees instead of lists. For this version the tree size n is an input parameter rather than a fixed constant.

The results of the experiments are shown in Figure 6. Here (✓) indicates the following:

- For Safe? = N, the tool correctly detects an error.
- For Safe? = Y, the tool proves safety (for D, Pr) or did not detect an error (for CB, LL, AS)

A contrary result is indicated by (✗), e.g. no error was detected for unsafe programs. Essentially (✓) indicates a correct result, and (✗) an incorrect result. Otherwise, (t.o.) indicates a timeout (of one minute) and (b.r.) indicates the bound was reached for bounded model checking-based tools (CB, LL). For these experiments we assume a bound B = 100. However, we note that all errors should be detectable at much lower depths, i.e. B = 100 is more than sufficient. All tools were fast (< 1s) provided no timeout/bounds-reached condition occurs.

For AS, a result marked with † indicates that the program crashed with a segmentation fault. This should not be interpreted as a bug in AS. Rather, each unsafe variant contains a memory error, and the nature of the error may be inadvertently changed (i.e. from mapped to unmapped access) by simply altering the runtime environment. This result is nevertheless considered correct because an error (of sorts) was detected.

The D_M-analysis tool is designed to detect memory context errors, and performs as expected. Pr performs well in detecting context errors provided the data-structure is a list. For trees, Pr works less well, and seems to resort to infinite unfolding leading to timeouts. The LL and CB tools have mixed results, and fail to detect some kinds of context errors. AS (a runtime tool) also fails to detect some types of memory context errors. This is because dynamic memory safety tools, such as AS, use a different notion of “memory error” that is purely based on object bounds. Namely, dereferencing a pointer p to object O is not considered a memory error provided p is within the bounds of O. In contrast, our tool (and related tools such as Pr) will consider such a dereference an error if O is outside of the footprint heap. Since AS has no notion of “footprint”, it does not detect all errors under our definitions. We note that this comparison is not intended to say one tool is better than another (which de-
pends on many factors). Rather this is simply an evaluation based on memory context errors.

Finally, we note that all of the tools {CB, LL, Pr, AS} do “whole” program analysis, meaning that a complete program (including an entry main function) must be provided. This can be a restrictive if one for code which is primarily written as libraries or software components. In contrast, the D-tool can be used to analyze individual functions, and whole program analysis is possible by verifying all functions are safe. Our tool is therefore suitable for analysis of code with no specific entry routine, such as the library code from Figure 5.

7. RELATED WORK AND 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, and therefore are a reasonable analysis target. It can be demonstrated, via example, that violating such properties can lead to surprising and unexpected results, such as arbitrary memory access. Since the analysis is shape neutral, it can be applied to a wide variety of programs manipulating different (and possibly multi-node) data-structures, including linked-lists, trees, DAGS, etc., which encompasses a large class of data-structures. Despite analyzing for more general properties, the analysis is nevertheless powerful enough to detect memory errors that can lead to memory corruption and information leaks.

Experimental results are promising, and show that a prototype implementation of the analysis scales well, and can detect errors that other tools miss.

Related Work

The formal basis of the D-analysis is most closely related to Separation Logic (SL) [16] – a popular extension of Hoare Logic for heap manipulating programs. The D-analysis is based on an alternative extension, namely the strongest post condition SPC encoding as introduced in [7], also for heap manipulating programs. The SPC-encoding encodes heap operations as heap constraints, as summarized in Section 2, that can be solved using a suitable constraint solver. Furthermore, the SPC encoding does not enforce any strict inbuilt notion of memory safety. This allows for a more fine-grained treatment of memory errors. For example, our D-analysis treats unmapped memory access, such as null pointer access, the same as an abort(), which reflects real-world behavior. In contrast SL enforces a stronger form memory safety which generally depends on data-structure shape, however, this makes automated reasoning difficult.

Our D-tool is related to other SL-based memory safety and program verification tools such as Predator [8], SLAyer [4], Smallfoot [3] and Verifast [11] (Verifast is for manual verification). Since these tools are based on SL, they attempt to detect others kinds of memory errors beyond context errors for the D-tool. However, this comes at a price: stronger memory safety is more difficult to prove, and therefore these SL-based tools have significant practical limitations. Many automated SL-based tools such as SLAyer and Smallfoot are limited to (some variant of) list-based data-structure reasoning. In contrast, the D-tool can reason over arbitrary data-structures including trees, DAGS, graphs, etc. Other tools, such as Verifast, are more general but are not automatic (require user annotations) and function as proof assistants. The D-tool is fully automatic.

Another key difference between the tools is the underlying specification of the desired data-structure properties. The D-analysis decomposes the data-structure based on field heaps, where all cells belonging to a given field are grouped into a single heap, as demonstrated in Example 3. A more common approach is to decompose based on node heaps. For example, consider a classical definition of list segment in SL:

\[
\text{ls}_{\text{seg}}(h, t) \triangleq \exists v, n : (h = t) \lor \text{node}(h, v, n) \land \text{ls}_{\text{seg}}(n, t)
\]

\[
\text{node}(h, v, n) \triangleq h \rightarrow \forall \pi(h+1) \rightarrow n
\]

Such definitions are recursive, allowing the possibility of infinite unfolding assuming naive reasoning. Alternatively tools may rely on specialized solvers, such as that described in [2] for lists. However, this means the tool is specialized to the given data-structure type. In contrast, the definitions for node and closed are non-recursive. The solver template from Section 5 is therefore terminating, and can be adapted to any arbitrary graph-based data-structure.

Other memory safety tools that are not based on SL include LLMB [15] and CBMC [13] (bounded model checkers) and runtime memory safety checking such as AddressSanitizer [17]. Unlike the D-tool and SL-based alternatives, these tools do not do data-structure-based reasoning, and primarily aim to detect out-of-bounds memory access such as buffer overflows. Dynamic tools such as AddressSanitizer have no notion of memory footprint and therefore cannot detect some kinds of memory errors altogether.

Future Work

For future work, the D-analysis could be extended to include checking for other types of memory errors, such as null-pointer access. However, such analysis for non-list data-structures is challenging since null pointer information is often left implicit by the programmer, e.g. in function parameters.

Another extension is to introduce a byte-precise model of data-structure layouts. Furthermore, this work can be extended to support programs that manipulate arrays in addition to data-structures. The main challenge is to extend the set of underlying solvers to support array reasoning in a compatible way.

8. REFERENCES


