Towards Memory Access Safety Analysis for Protected Environments

Gregory J. Duck, Joxan Jaffar, and Roland H. C. Yap

Department of Computer Science, National University of Singapore
{gregory,joxan,ryap}@comp.nus.edu.sg

Abstract. Preventing memory access errors is an important security consideration for programs implemented in low-level languages such as C. Some types of memory access errors can be protected against with the help of a suitable runtime environment. For example, a modern operating system protects against access to some invalid addresses, such as the null pointer. Furthermore, a garbage collector can prevent some kinds of memory access errors that are associated with manual memory management. Other kinds of memory access errors cannot reliably be protected against by a runtime environment. This paper presents a memory access safety analysis for detecting such errors. Our main contribution is an instantiation of the analysis that can determine the memory access safety of programs that manipulate recursively defined data-structures, such as linked-lists, trees, etc.

1 Introduction

Preventing memory errors (a.k.a. memory safety) is an important security consideration for programs implemented in low-level languages such as C. Here, memory errors refer to bugs related to the access or management of memory.

Memory errors are well known to be a key source of software vulnerabilities exploitable by attackers. A recent survey [16] presents a range of solutions which have been developed for the detection or prevention of memory errors. It is clear that memory errors continue to be a problem. For many years, it is listed as among the top software vulnerabilities, i.e. in the CWE/SANS top 25 errors list.

A key observation is that attackers tend to exploit specific kinds of memory errors; namely, those that access memory beyond the intention of the programmer. In this paper our focus is such memory “access” errors.

In general, memory errors are eliminated through two main approaches: (1) static solutions that aim to detect potential memory errors at compile time, i.e. through program analysis; and (2) dynamic solutions that aim to detect or eliminate memory errors at runtime. Tools such as Blast [3], Smallfoot [1], SLayer [2], etc., fall into the former category. Tools such as Cyclone [11], CCured [14], and SoftBound [13], ensure the safety of memory operations through runtime checking. Each approach has its advantages and disadvantages.
Other technologies offer partial protection. Garbage collection eliminates\(^1\) temporal memory errors associated with manual memory management, namely:

- **dangling-pointers** - using a pointer after freeing it: `free(p); *p=3;`
- **double-frees** - freeing a pointer twice: `free(p); free(p);`
- **invalid-frees** - freeing a non-malloc’ed pointer: `int p[3]; free(p);`

Other kinds of memory errors do not cause invalid memory access, including:

- **null-pointer-dereference** - dereference the null pointer: `p=0; *p=3;`
- **small-pointer-dereference** - dereference a low-valued pointer: `p=x; *p=3;`
  where \( x < B \) for some bound \( B \) (a generalization of null-pointer-dereference).

Such errors are not memory access errors assuming a modern operating system environment with protected memory, e.g. Windows or Linux. Instead, the operating system will raise an exception (which can be handled). Whilst this may deny service, it does not access nor corrupt memory. A protected environment (as per the title of this paper) is therefore defined as follows: ProtectedEnvironment = GarbageCollection + OperatingSystemProtection. Assuming a protected environment, some memory access errors may still exist:

- **wild-pointer-dereference** - dereference a low-valued pointer: `p=x; *p=3;` for integer \( x \geq B \)
- **uninitialized-ptr-deref** - dereference an uninitialized pointer: `int *p; *p=3;`
- **buffer-overflows** - accessing beyond the end (or start) of a buffer: `int p[3]; p[4]=3;`

In this paper, we propose a memory access safety analysis for detecting such errors. Our main contribution is an instantiation of the analysis for programs that manipulate recursively defined data-structures, such as lists, trees, etc. Whilst automated reasoning over such programs is generally considered difficult, we show the problem is tractable under our assumptions. In summary, the main contributions of this paper are:

- Section 3 introduces a simple static analysis for memory access safety using constraint-based symbolic execution over heaps;
- Section 4 instantiates the analysis to prove memory access safety for recursively defined data-structures;
- Section 5 presents a constraint solver for the desired properties necessary to implement the analysis; and
- Section 6 runs experiments to establish the validity of our approach.

## 2 Preliminaries

Our analysis uses the \( \mathcal{H} \) constraint language for reasoning over program heaps first described in [7]. Here we give a brief overview.

The \( \mathcal{H} \)-language is related to Separation Logic [15]. Here we assume as given a set of Values (typically Values \( \equiv \mathbb{Z} \)) and define the set of Heaps to be all finite partial maps between values, i.e. Heaps \( \equiv \) (Values \( \rightarrow_{\text{fin}} \) Values). Likewise we

\(^1\) Assuming we replace `free` with a NOP.
assume as given sets of variables \( \text{Vars}_\text{Values} \) and \( \text{Vars}_\text{Heaps} \) that are disjoint from each other. We define a restricted Separation Logic formula \( F \) as follows:

\[
H ::= \text{Vars}_\text{Heaps} \quad v ::= \text{Vars}_\text{Values} \quad F ::= \text{emp} \mid (v \mapsto v) \mid F * F
\]

A valuation \( \varphi \) maps \( \text{Vars}_\text{Heaps} \) to Heaps and \( \text{Vars}_\text{Values} \) to Values. Syntactically, a heap constraint \( C \) is a literal of the form \( (F \approx F) \). A valuation \( \varphi \) satisfies a heap constraint \( (F_1 \approx F_2) \) iff (1) \( \varphi(F_1) \) and \( \varphi(F_2) \) hold as ground Separation Logic formulae, and (2) \( \varphi(F_1) = \varphi(F_2) \) are the same heap.

Let \( \text{dom}(H) \) be the domain of the heap \( H \). We sometimes abuse notation and treat heaps \( H \) the set of pointer-value pairs \( \{(p,H(p)) \mid p \in \text{dom}(H)\} \). As shown in [7], heap constraints can be normalized into three basic forms:

\[
H \approx \text{emp} \quad (\text{Empty}) \quad H \approx (p \mapsto v) \quad (\text{Singleton}) \quad H \approx H_1 * H_2 \quad (\text{Separation})
\]

where \( H, H_1, H_2 \in \text{Vars}_\text{Heaps} \) and \( p, v \in \text{Vars}_\text{Values} \). Here (Empty) constrains \( H \) to be the empty heap (i.e. \( H = \emptyset \) as sets), (Singleton) constrains \( H \) to be the singleton heap mapping \( p \) to \( v \) (i.e. \( H = \{(p,v)\} \) as sets), and (Separation) constrains \( H \) to be heap that is partitioned into two disjoint sub-heaps \( H_1 \) and \( H_2 \) (i.e. \( H = H_1 \cup H_2 \) as sets and \( \text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset \)). A sound and complete constraint solver for normalized \( \mathcal{H} \)-constraints is presented in [7].

In addition to the basic heap constraints, we assume definitions for domain membership \( (p \in \text{dom}(H)) \), heap membership \( (\text{in}(H, p, v)) \), i.e. \( (p,v) \in H \) as sets), a sub-heap relation \( (H_1 \subseteq H_2) \); as defined in [7].

**Program Analysis via Symbolic Execution** Symbolic execution involves executing a program using symbolic values as inputs. The output of symbolic execution is a formula representing a given path through the program.

By convention we use a distinguished heap variable \( \mathcal{H} \) to represent the current program heap at any given program point. We can can therefore describe symbolic execution of programs as Hoare triples over the \( \mathcal{H} \)-language as follows:

\[
\{ \phi \} \; x := y[0] \; \{ \text{access}(\phi, y, x) \} \quad (\text{Heap access})
\]

\[
\{ \phi \} \; x[0] := y \; \{ \text{assign}(\phi, x, y) \} \quad (\text{Heap assignment})
\]

\[
\{ \phi \} \; x := \text{malloc}(1) \; \{ \text{alloc}(\phi, x) \} \quad (\text{Heap allocation})
\]

where auxiliary macros access, assign, alloc, and free expand as follows:

\[
\text{access}(\phi, y, x) \triangleq \exists H, x_0 : \mathcal{H} = (y \mapsto x) * H \land \phi[x_0/x]
\]

\[
\text{assign}(\phi, x, y) \triangleq \exists H_0, H, v : H_0 = (x \mapsto v) * H \land \mathcal{H} = (x \mapsto y) * H \land \phi[H_0/\mathcal{H}]
\]

\[
\text{alloc}(\phi, x) \triangleq \exists H_0, v, x_0 : \mathcal{H} = (x \mapsto v) * H_0 \land \phi[H_0/\mathcal{H}, x_0/x]
\]

Here the notation \( \varphi[x/y] \) means formula \( \varphi \) with variable \( x \) substituted for \( y \). We generalize the rules for malloc \( (n) \), constant \( n > 1 \) and for \( x[i] \) in the obvious way. The above triples correspond to the Strongest Post-Condition Encoding from [7]. Since we assume garbage collection, we also assume that calls to free have been replaced by a NOP, i.e. \( \{ \phi \} \text{free}(x) \; \{ \phi \} \).
3 Memory Access Safety Analysis

Memory access safety analysis protects against only certain kinds of memory errors. The key idea is to partition a given program heap $H$ into two parts: a footprint heap $H$ and a frame heap $F$, i.e. $H \approx H*F$. Intuitively, the footprint represents the portion of the heap that some given code is allowed to access (i.e. read from or write to), and the frame represents everything else. A program commits a memory access error if it accesses the frame $F$:

**Definition 1 (Memory Access Error).** Given a frame heap $F$, then a read or write operation to pointer $p$ is a memory access error if $p \in \text{dom}(F)$. Conversely, a read/write operation is memory access safe if $p \not\in \text{dom}(F)$.

Note that our definition is somewhat unconventional: memory access safety merely requires $p$ not to be within the frame $F$, i.e. $p \in \text{dom}(H)$. To illustrate the difference, consider the code snippet $(p = 0; \ast p = 3)$. The write operation is a memory error (null-pointer-dereference) since $p \not\in H$. However, null-pointer-dereference is not a memory access error since $p \not\in F$. For readability, we shall also define:

\[
\text{safe}(x, F) \overset{\text{def}}{=} (x \not\in \text{dom}(F))
\]

A code fragment $C$ is memory access safe w.r.t. a precondition $P$ and a frame $F$ iff executing any state $\sigma$ satisfying $(P \land \bar{H} \approx H*F)$ will not cause a memory access error in $C$. Memory access safety ensures that the frame remains unchanged:

**Theorem 1.** If $C$ is memory access safe w.r.t. precondition $P$ and frame $F$, then $\{P \land \bar{H} \approx H*F\} C \{\exists H': \bar{H} \approx H'*F\}$.

**Proof.** (Sketch) By induction over the statements in $C$.

Theorem 1 analogous to the Frame Rule from Separation Logic.

**Analysis Definition** Our memory access analysis uses constraint-based symbolic execution. The basic idea is to test for memory access safety as we build path constraints of the code. The analysis is parameterized as follows:

1. a code fragment $C$ (i.e. the code we wish to analyze); and
2. an initial assumption $\phi_{\text{init}}(H, F)$, an invariant $\phi_{\text{inv}}(H, F)$ and an exit condition $\phi_{\text{exit}}(H, F)$.

All of $\phi_{\text{init}}$, $\phi_{\text{inv}}$ and $\phi_{\text{exit}}$ are formulae over a footprint $H$ and frame $F$. We define the following:

\[
\text{precond}(\phi, H, F) \overset{\text{def}}{=} (\bar{H} \approx H*F \land \phi(H, F))
\]

\[
\text{invariant}(\phi, F) \overset{\text{def}}{=} (\exists H : \bar{H} \approx H*F \land \phi(H, F))
\]

\[
\text{postcond}(\phi, F) \overset{\text{def}}{=} \text{invariant}(\phi, F)
\]

The terminology footprint and frame is also shared with Separation Logic.
analyze([return x], σ) \equiv (σ \models \text{postcond}(φ_{exit}, F)) \quad (\text{VC-exit})

analyze([x := y[0]; S], σ) \equiv
\begin{cases}
\sigma \models \text{safe}(y, F) \\
\text{analyze}([S], \text{access}(σ, y, x))
\end{cases} \quad (\text{VC-Access})

analyze([x[0] := y; S], σ) \equiv
\begin{cases}
\sigma \models \text{safe}(x, F) \\
\text{analyze}([S], \text{assign}(σ, x, y))
\end{cases} \quad (\text{VC-Assign})

analyze([x := malloc(1); S], σ) \equiv \text{analyze}([S], \text{alloc}(σ, x))

analyze([if G then T else E endif; S], σ) \equiv
\begin{cases}
\text{analyze}([T; S], G \land σ) \land \text{analyze}([E; S], \neg G \land σ)
\end{cases}

analyze([while G do B done; S], σ) \equiv
\begin{cases}
\sigma \models \text{invariant}(φ_{inv}, F) \\
\sigma \models \text{invariant}(φ_{inv}, F) \\
\text{analyze}([B; while G do B done; S], G \land \text{precond}(φ_{inv}, H, F)) \\
\text{analyze}([S], \neg G \land \text{precond}(φ_{inv}, H, F))
\end{cases} \quad (\text{VC-Inv})

analyze([x := f(y_1, ..., y_n); S], σ) \equiv
\begin{cases}
\sigma \models P \\
\text{analyze}([S], Q)
\end{cases} \quad (\text{VC-Func})

The initial symbolic state \(σ_0\) is therefore \(σ_0 = \text{precond}(φ_{init}, H, F)\). The purpose of the \(φ_{inv}\) is to handle loops, where \(\text{invariant}(φ_{inv}, F)\) will be the loop invariant.\(^3\) Note that we defer loop invariant discovery (if required) to the analysis instantiation. Any property we wish to hold after \(C\) exits is captured by \(φ_{exit}\).

The analysis schema is shown in Figure 1. The analysis is defined in terms of a Strongest Post Condition (SPC) Predicate Transformer Semantics (PTS), and is shown in Figure 1. Here the function \text{analyze}([C], σ)\) takes as input some code fragment \(C\) and a symbolic state \(σ\), and returns a Boolean value: where \(true\) indicates the analysis “passed”, i.e. \(C\) is memory access safe and the exit conditions were satisfied, or \(false\) indicates otherwise. The brackets \([..]\) around code fragments are added for readability. We also assume that the code fragment is terminated with a return statement.

The general form for an analysis step over a statement \(s\) is the rule:
\[
\text{analyze}([s; S], σ) \equiv \text{analyze}([S], σ')
\]

where \(\{σ\}\ \{σ'\}\) is the corresponding Hoare Logic axiom (including those from Section 2). If \(s\) reads/writes to memory via pointer \(p\), then we additionally

\(^3\) We may also allow a different \(φ_{inv}\) per loop.
generate a Verification Condition (VC) that checks if the operation is memory access safe, i.e. if \((\sigma \models \text{safe}(p, F))\) holds as per (VC-Access) and (VC-AssIGN). If-then-else is handled by branching the analysis to cover the then \(T\) and else \(E\) cases. Loops are handled by proving that invariant\((\phi_{inv}, F)\) is a loop invariant. This includes checking the base case (VC-Inv), and the inductive case \(G \wedge \text{precond}(\phi_{inv}, H, F)\), for fresh \(H\), over the loop body \(B\). If the loop invariant holds, then the analysis continues with \(\neg G \wedge \text{precond}(\phi_{inv}, H, F)\), for fresh \(H\). For function calls, we assume a (separately proven) triple \(\{P\} x := f(y_1, \ldots, y_n) \{Q\}\).

Finally (VC-Exit) is generated for the return statement to confirm the triple \(\{\text{precond}(\phi_{init}, H, F)\} C \{\text{postcond}(\phi_{exit}, F)\}\). This triple may, in turn, be used for function composition as input to (VC-FUNC).

**Example 1 (Basic Analysis).** Consider the following code fragment \(C\):

\[
i = 0; \ p[i] = 0; \ i++; \ p[i] = 0; \ \text{return 0;}
\]

Furthermore we define:

\[
\phi_{init}(H, F) \overset{\text{def}}{=} (H \models (p \mapsto v) \cdot (p+1 \mapsto w)) \quad \text{and} \quad \phi_{inv} = \phi_{exit} = \text{true}
\]

Here, \(\phi_{init}\) constrains \(H\) to contain two cells pointed to by \(p\) and \(p+1\). The initial symbolic state is \(\sigma_0 = \text{precond}(\phi_{init}, H, F)\), and subsequent symbolic states are:

\[
\begin{align*}
\sigma_1 &= \sigma_0 \land i = 0 \\
\sigma_2 &= \sigma_1[H_0/\bar{H}]\land H_0 = (p+i \mapsto v) \cdot H' \land \bar{H} \models (p+i \mapsto 0) \cdot H' \\
\sigma_3 &= \sigma_2[i_0/i] \land i = i_0 + 1 \\
\sigma_4 &= \sigma_3[H_1/\bar{H}] \land H_1 = (p+i \mapsto w) \cdot H'' \land \bar{H} \models (p+i \mapsto 0) \cdot H''
\end{align*}
\]

For brevity we keep the existential quantifiers implicit. The analysis generates the VCs (\(\sigma_1 \models \text{safe}(p+i, F)\)) and (\(\sigma_3 \models \text{safe}(p+i, F)\)). The former expands to:

\[
(\bar{H} = H \circ F \land H = (p \mapsto v) \cdot (p+1 \mapsto w) \land i = 0) \models p + i \notin \text{dom}(F)
\]

Intuitively, the VC holds since if \(p+i \in \text{dom}(H)\), then \(p+i \notin \text{dom}(F)\), since \(H\) and \(F\) must be separate as per the \(\bar{H} = H \circ F\) constraint. More rigorously, the VC can be solved using the \(\mathcal{H}\)-solver from [7]. The same is true for the second VC. The analysis therefore passes, and thus the fragment is memory access safe. □

Example 1 also works for weaker initial assumptions, e.g. if \(\phi_{init}(H, F)\) were \((p \notin \text{dom}(F) \land p+1 \notin \text{dom}(F))\), allowing for \(p\) to be null. On the other hand, if we further weaken the initial assumption, e.g. if \(\phi_{init}(H, F)\) were \((p \notin \text{dom}(F))\), then the analysis fails, since the address \(p+1\) may point into frame \(F\).

Memory access analysis for simple programs with basic pointers and arrays, such as Example 1, reduces to solving over \(\mathcal{H}\)-formulae. For the rest of this paper, we consider an instantiation of the analysis for programs that manipulate recursively defined data-structures.

---

\(^4\) The analysis, as specified in Figure 1, naïvely executes both branches. This can be optimized, e.g. by using interpolation [10]. We leave this as future work.
4 Memory Access Safety for Data-Structures

Our aim is to instantiate the memory access analysis for programs that manipulate recursively-defined data-structures, such as linked-lists, trees, graphs, etc.

The key property of interest is whether or not a heap is closed. A heap $H$ is closed if all (non-null) pointers in $H$ point back into $H$. We shall first define closed heaps for skeleton data-structures corresponding to the definition: $(\text{struct node} \{ \text{struct node} *\text{next} \})$ Later we shall generalize the definitions to account for non-skeleton data-structures with multiple fields. The basic idea behind the skeleton and generalized definitions is the same, save for obfuscation.

Let $H$ be a data-structure comprised of skeleton nodes. Then $p$ is a skeleton node pointer w.r.t. $H$ if $p$ is null or $p$ points to a skeleton node in $H$.

**Definition 2 (Skeleton Node Pointer).** A value $p \in \text{Values}$ is a skeleton node pointer w.r.t. heap $H$, written as $\text{node}_{\text{skel}}(p, H)$, if $p = 0$ or $p \in \text{dom}(H)$.

$H$ is a skeleton closed heap if all the next fields in $H$ are skeleton node pointers.

**Definition 3 (Skeleton Closed Heap).** A heap $H$ is skeleton closed, written as $\text{closed}_{\text{skel}}(H)$, if for all $p, v$ such that $H(p) = v$, then $\text{node}_{\text{skel}}(v, H)$.

Essentially all pointers within a skeleton closed heap $H$ are either null or point back into $H$.

**Example 2 (Skeleton Lists).** Skeleton heaps can be used to implement various simple linked-list data-structures. Assuming $l$ is the list’s head pointer:

$$L \equiv (l \mapsto p) \ast (p \mapsto q) \ast (q \mapsto 0) \quad \text{(straight linked-list)}$$
$$L \equiv (l \mapsto p) \ast (p \mapsto q) \ast (q \mapsto l) \quad \text{(circular linked-list)}$$
$$L \equiv (l \mapsto p) \ast (p \mapsto q) \ast (q \mapsto p) \quad \text{(lasso-list)}$$
$$L \equiv (l \mapsto p) \ast (p \mapsto 0) \ast (q \mapsto 0) \quad \text{(disjoint-list)}$$

for some $p$ and $q$. All such $L$ are skeleton closed since $p, q, l \in \text{dom}(L)$. In contrast, the heap $L \equiv (l \mapsto p) \ast (p \mapsto q)$ is not closed if $q \neq 0$ and $q \not\in \{p, l\}$.

The $\text{closed}_{\text{skel}}(H)$ property is shape impartial, i.e. it generalizes all kinds of list data-structures, including straight, circular, lasso, and even disjoint lists.

We can use the $\text{closed}_{\text{skel}}$ and $\text{node}_{\text{skel}}$ properties to prove memory access safety for programs that manipulate skeleton data-structures.

**Example 3 (In-place List Reverse).** Consider the code for in-place skeleton list reversal $\text{rev}(xs)$ program shown below:

```c
struct node *prev = 0;
while (xs != 0)
    { struct node *next = xs->next; xs->next = prev;
      prev = xs; xs = next; }
```
Our aim is to show that the program is memory access safe w.r.t. the following analysis parameters:

\[
\phi_{\text{init}}(H, F) \overset{\text{def}}{=} \text{closed}_{\text{skel}}(H) \land \text{node}_{\text{skel}}(H, xs)
\]

\[
\phi_{\text{inv}}(H, F) \overset{\text{def}}{=} \phi_{\text{init}}(H, F) \land \text{node}_{\text{skel}}(H, \text{prev})
\]

Symbolic execution of the loop body proceeds as follows:

\[
\sigma_0 = \bar{H} \approx H \land \text{closed}_{\text{skel}}(H) \land \text{node}_{\text{skel}}(xs, H) \land \text{node}_{\text{skel}}(\text{prev}, H) \land xs \neq 0 \tag{1}
\]

\[
\sigma_1 = \sigma_0 \land \bar{H} \approx (xs \mapsto \text{next}) * H' \tag{2}
\]

\[
\sigma_2 = \sigma_1[H_0/\bar{H}] \land H_0 \approx (xs \mapsto v) * H'' \land \bar{H} \approx (xs \mapsto \text{prev}) * H'' \tag{3}
\]

\[
\sigma_3 = \sigma_2[\text{prev}_0/\text{prev}] \land \text{prev} = xs \tag{4}
\]

\[
\sigma_4 = \sigma_3[\text{xs}_0/xs] \land xs = \text{next} \tag{5}
\]

Heap operations (2) and (3) are memory access safe, i.e. \( \sigma_{i \in \{1, 2\}} \models \text{safe}(xs, F) \). Furthermore the loop invariant holds, i.e. \( \sigma_4 \models \text{invariant}(\phi_{\text{inv}}, F) \). All the generated VCs can be solved using a suitable theorem prover (see Section 5). We can extrapolate these results to show that the \( \text{rev}(xs) \) function is memory access safe. □

Since \( \text{closed}_{\text{skel}}(H) \) is shape impartial, the \( \text{rev}(xs) \) function is memory access safe for any kind of list (straight, circular, lasso, etc.).

**Generalized Access Safety** We can generalize Definitions 2 and 3 to cover more complex data-structures; such as lists with data and trees respectively:

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

```c
struct tree_node {
    int val;
    struct tree_node *left;
    struct tree_node *right;
};
```

Our generalization must consider three main issues:

1. **Node sizes**: allow nodes with a size greater than one cell;
2. **Field types**: allow the mixture of pointer and non-pointer data; and
3. **Overlapping fields**: disallow arbitrarily overlapping nodes/fields.

For simplicity we will also assume that all data-structures use one kind of node type \( \text{struct node} \) (or simply \( \text{node} \) for short). This can be further generalized.

Given a struct definition for type \( \text{node} \) with fields \( \{\text{field}_0, ..., \text{field}_n\} \), we define \( \text{Ptrs}(\text{node}) \) to be the set of all \( \text{field indices} \) (from 0) that have type \( (\text{node} *) \), i.e.:

\[
\text{Ptrs}(\text{node}) = \{ i \mid \text{typeof}(\text{field}_i) = (\text{node} *) \}
\]

Thus \( \text{Ptrs}([\text{list_node}]) = \{1\} \), \( \text{Ptrs}([\text{tree_node}]) = \{1, 2\} \), etc. Given a pointer \( p \) for type \( \text{node} \), we assume \( \text{field}_i \) resides at address \( p + i \).

To exclude arbitrarily overlapping nodes/fields, we assume the given heap \( H \) is partitioned into \( n \) separate sub-heaps, i.e. \( H \approx H_0 *...* H_n \), one for each field. This assumption is reasonable for most standard data-structures, i.e. linked-lists, trees, etc., in idiomatic C. We can generalize a \( \text{node pointer} \) as follows:
Definition 4 (Node Pointer). A value $p \in Values$ is a node pointer w.r.t. the heap and partitioning $H \equiv H_0 \ast \ldots \ast H_n$, written as $\text{node}_{node}(p, H_0, \ldots, H_n)$, if $p = 0$ or $p + i \in \text{dom}(H_i)$ for all $i \in 0..n$. □

The generalized definition no longer restricts nodes to a single heap cell. Similarly we can generalize the notion of a closed heap as follows:

Definition 5 (Closed Heap). A heap and partitioning $H \equiv H_0 \ast \ldots \ast H_n$ is closed, written as $\text{closed}_{node}(H_0, \ldots, H_n)$, if for all $p, v$ such that $H_i(p) = v$ for $i \in \text{Ptrs}(node)$, then $\text{node}_{node}(v, H_0, \ldots, H_n)$. □

For the skeleton definition there is only one field, so $\text{Ptrs}(node) = \{0\}$, $H_0 = H$, and the above definitions reduce to Definition 2 and 3. We shall write $\text{node}_{node} = \text{node}$ and $\text{closed}_{node} = \text{closed}$ when the type $\text{node}$ is clear from context.

Example 4 (In-place List Reverse Version II). Consider the in-place list reverse function from Example 3 once more. This time we assume nodes of type $\text{list}_{node}$ defined above. We use the following definitions:

\[
\begin{align*}
\phi_{\text{init}}(H, F) & \overset{\text{def}}{=} H \equiv H_0 \ast H_1 \land \text{closed}(H_0, H_1) \land \text{node}(xs, H_0, H_1) \\
\phi_{\text{inv}}(H, F) & \overset{\text{def}}{=} \exists H'_0, H'_1 : \land \left\{ H \equiv H'_0 \ast H'_1 \land \text{closed}(H'_0, H'_1) \land \text{node}(xs, H'_0, H'_1) \land \text{node}(\text{prev}, H'_0, H'_1) \right\}
\end{align*}
\]

The analysis proceeds in the same way as Example 3. For example, the VCs for the heap access/assignment become:

\[
\mathcal{H} \equiv H \ast F \land H \equiv H_0 \ast H_1 \land \text{node}(xs, H_0, H_1) \models \text{safe}(xs, F)
\]

This trivially holds since $xs \in \text{dom}(H_1)$, and thus $xs \notin \text{dom}(F)$. Therefore in-place list reverse is memory access safe w.r.t. to type $\text{list}_{node}$. □

Analysis Instantiation Schema We generalize Example 4 into an analysis instantiation schema for closed heaps as follows:

\[
\begin{align*}
\phi_{\text{init}}(H, F) & \overset{\text{def}}{=} \land \left\{ H \equiv H_0 \ast \ldots \ast H_n \land \text{closed}(H_0, \ldots, H_n) \land \text{node}(p, H_0, \ldots, H_n) \land \text{node}(x_1, H_0, \ldots, H_n) \land \ldots \land \text{node}(x_i, H_0, \ldots, H_n) \right\} \\
\phi_{\text{inv}}(H, F) & \overset{\text{def}}{=} \exists H'_0, \ldots, H'_n : \land \left\{ H \equiv H'_0 \ast \ldots \ast H'_n \land \text{closed}(H'_0, \ldots, H'_n) \land \text{node}(p, H'_0, \ldots, H'_n) \land \text{node}(y_1, H'_0, \ldots, H'_n) \land \ldots \land \text{node}(y_j, H'_0, \ldots, H'_n) \right\} \\
\phi_{\text{exit}}(H, F) & \overset{\text{def}}{=} \exists H'_0, \ldots, H'_n : \land \left\{ H \equiv H'_0 \ast \ldots \ast H'_n \land \text{closed}(H'_0, \ldots, H'_n) \land \text{node}(p, H'_0, \ldots, H'_n) \land \text{node}(r, H'_0, \ldots, H'_n) \right\}
\end{align*}
\]

where $x_1, \ldots, x_i$ are all input parameters, $y_1, \ldots, y_j$ are all live program variables, $p$ is a ghost variable, and $r$ is a possible return value, all of type $\text{node} \ast$. If no such $r$ exists then we can treat $r = 0$. □
The purpose of the ghost variable \( p \) is for function composition. In addition to memory access safety, the analysis also proves the triple \( \{P\} C \{Q\} \) where:

\[
P \equiv \bar{H} \simeq H^*F \land H \simeq H_0^*..H_n^* \land \text{closed}(H_0, ..., H_n) \land \text{node}(p, H_0, ..., H_n)
\]

\[
Q \equiv \exists H', H'_0, ..., H'_n : \bar{H} \simeq H'^*F \land H' \simeq H'_0^*..H'_n^* \land \text{closed}(H'_0, ..., H'_n) \land \text{node}(p, H'_0, ..., H'_n) \land \text{node}(r, H'_0, ..., H'_n)
\]

This means that: (1) the output heap \( H' \) remains closed, and (2) all valid node pointers \( p \) before \( C \) and the return value \( r \) are valid node pointers after \( C \). This triple is used to instantiate \((\text{VC-Func})\) from Figure 1 for function composition.

The reader may notice that these definitions assume a single data-structure \( H \). In practice, a program may maintain several disjoint data-structures of type \( \text{node} \). However, since the closed property is shape impartial, it allows for disjoint graphs. Thus, w.l.o.g., we can treat a collection of disjoint data-structures as one single combined structure, thereby simplifying the analysis.

5 Solving for Closed Heaps

The analysis relies on proving the validity of VCs of the general form: \( \exists \bar{x} : P \models \exists \bar{y} : Q \). These are discharged in two steps:

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

**Building Witnesses** Existential variables \( \bar{y} \) are used by the VCs of the form \((\sigma \models \text{invariant}(\phi_{\text{inv}}, F))\) which expands to:

\[
\sigma \models \exists H', H'_0, ..., H'_n : \bar{H} \simeq H'^*F \land H' \simeq H'_0^*..H'_n^* : \phi_{\text{inv}}(H', F)
\]

Similarly for VCs generated for \( \phi_{\text{exit}} \). We need to build witnesses for \( H', H'_0, ..., H'_n \). Let \( \{H_0, ..., H_n\} \) be the original field heaps from \( \text{precond}(H, F) \), and \( p_0, ..., p_m \) be new pointers allocated by calls to \text{malloc} \( \text{sizeof}(\text{node}) \), then we build the witnesses as follows:

- the witness for \( H' \) is \( (\bar{H} - F) \) (heap difference as sets); and
If Proposition 1.

We can use the witness to derive a quantifier-free VC:

Essentially, the domain of \( H \)

Definition 4 and 5. Given a closed node constraint propagation, and rewrite rules (in the form Figure 2, and consists of two parts: inference rules for (positive) node and closed constraints.

We extend the \( H \)-solver from [7] with node\((p, H_0, ..., H_n)\) and closed\((H_0, ..., H_n)\) constraints. The closed heap solver schema is shown in Figure 2, and consists of two parts: inference rules for (positive) node/closed constraint propagation, and rewrite rules (in the form head \( \rightarrow \) body) for negated node/closed constraints.

The inference rules for the positive constraints are derived directly from Definitions 4 and 5. Given a closed\((H_0, ..., H_n)\) constraint, the solver will propagate a corresponding node\((v, H_0, ..., H_n)\) constraint for each \( \text{in}(H_i, p, v), i \in \text{Ptrs}(\text{node}) \) propagated by the \( H \)-solver. Likewise, the node\((p, H_0, ..., H_n)\) propagates a disjunction: constraining \( p = 0 \), or the constraining each field to be in the appropriate field heap, i.e. \( p + i \in \text{dom}(H_i) \) for \( i \in 0..n \).

The rewrite rules are derived from the negation of Definitions 4 and 5. Unlike the inference rules, the rewrite rules are used to eliminate negated node/closed from the goal \( P \land W \land \neg \text{Q} \). This is a two step process: (1) first the negation \( \neg \text{Q} \) is pushed inwards, and (2) next any resulting negated node/closed literals are eliminated by applying the rewrite rules from Figure 2.

The inference rules from Figure 2 can be directly implemented in Constraint Handling Rules (CHR) [8] using the following simple schema:

\[
\text{closed}(H_0, ..., H_n) \land \text{in}(H_i, p, v) \Rightarrow \text{node}(v, H_0, ..., H_n) \quad \text{(for } i \in \text{Ptrs}(\text{node}))
\]

\[
\text{node}(p, H_0, ..., H_n) \Rightarrow p \neq 0 \lor \bigwedge_{i \in 0..n} p + i \in \text{dom}(H_i)
\]

- the witness for \( H'_i \) is \( \{(p_j + i, \bar{H}(p_j + i) \mid j \in 0..m) \cup H_i \} \).

Essentially, the domain of \( H_i \) and \( H'_i \) are equal plus any additional nodes allocated via malloc. We can express the witness in terms of \( H \)-constraints:

\[
\text{witness}(F) \overset{\text{def}}{=} \left( \bar{H} \simeq H^* \land H' \simeq H'_0 \lor \ldots \lor H'_n \land \bigwedge_{i \in 0..n} H'_i \subseteq H_i \land \bigwedge_{j \in 0..m} p_j + i \in \text{dom}(H_i) \right)
\]

We can use the witness to derive a quantifier-free VC:

**Proposition 1.** If \( (\sigma \land \text{witness}(F)) \models \phi_{\text{wv}}(H', F) \) is valid then (6) is valid.

**Closed Heap Solver** We extend the \( H \)-solver from [7] with node\((p, H_0, ..., H_n)\) and closed\((H_0, ..., H_n)\) constraints. The closed heap solver schema is shown in Figure 2, and consists of two parts: inference rules for (positive) node/closed constraint propagation, and rewrite rules (in the form head \( \rightarrow \) body) for negated node/closed constraints.

The inference rules for the positive constraints are derived directly from Definitions 4 and 5. Given a closed\((H_0, ..., H_n)\) constraint, the solver will propagate a corresponding node\((v, H_0, ..., H_n)\) constraint for each \( \text{in}(H_i, p, v), i \in \text{Ptrs}(\text{node}) \) propagated by the \( H \)-solver. Likewise, the node\((p, H_0, ..., H_n)\) propagates a disjunction: constraining \( p = 0 \), or the constraining each field to be in the appropriate field heap, i.e. \( p + i \in \text{dom}(H_i) \) for \( i \in 0..n \).

The rewrite rules are derived from the negation of Definitions 4 and 5. Unlike the inference rules, the rewrite rules are used to eliminate negated node/closed from the goal \( P \land W \land \neg \text{Q} \). This is a two step process: (1) first the negation \( \neg \text{Q} \) is pushed inwards, and (2) next any resulting negated node/closed literals are eliminated by applying the rewrite rules from Figure 2.

The inference rules from Figure 2 can be directly implemented in Constraint Handling Rules (CHR) [8] using the following simple schema:
Example 5 (Closed Heap Solver). Consider the following (simplified) VC from Example 2: \( (H \equiv H \ast F \land \text{node}(xs, H) = xs \notin \text{dom}(F)) \). This VC is valid iff the goal \( (H \equiv H \ast F \land \text{node}(xs, H) \land xs \in \text{dom}(F)) \) is unsatisfiable. The solver steps are shown in Figure 3. Here (H), (C), and (I) are inferences made by the \( H \)-solver, closed-heap solver, and an integer solver respectively. Here the \( H \)-solver assumes all pointers have a lower bound of \( B = 4096 \) (1 page). 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 \textit{false} the original goal is unsatisfiable, proving the VC is valid. \( \square \)

6 Experiments

We have implemented a prototype analysis tool as a LLVM [12] plug-in. The plug-in takes as input the LLVM Intermediate Representation (IR) and generates VCs as per Figure 1. The VCs are solved using an implementation of the closed heaps solver (specialized to a given data-structure) as described in Section 5. The solver itself is scripted using Constraint Handling Rules (CHR) [8], and executed on the Satisfiability Modulo Constraint Handling Rules (SMCHR) [6] system. In addition to the closed heap solver, we also load the SMCHR built-in \textit{linear} and \textit{bounds} solvers for respectively handling integer linear arithmetic constraints and propagating bounds.

All experiments were run on a Intel i5-2500K CPU at 3.3GHz. We compare our prototype tool against two Separation Logic-based systems:
- \textit{SLAyer} (Version 1.1) [2]: An automatic memory safety analysis tool; and
- \textit{Verifast} (Version 13.11.14) [9]: A manual verification system.

Our tool and Verifast runs on Linux, whereas SLAyer runs on Windows. We consider three main sets of benchmarks in C:
- \textit{GLib} selected functions from the GNU GLib library version 2.38.0.
- \textit{SLAyer} selected functions from the SLAyer distribution.
- \textit{Verifast} selected functions from the Verifast distribution.

All benchmarks manipulate various types of data-structures, including: Singularity Linked Lists (SLL), Doubly Linked Lists (DLL), Binary Trees (BT), Balanced Binary Trees (BBT) (of type \texttt{GTreeNode} from GLib), and Binary Graphs (BGs). All benchmarks are used unmodified with the following exceptions: (1) in \texttt{insert_internal} (g\_tree) the explicit stack was replaced by recursion; and (2) the SLAyer benchmarks were modified to avoid pointers to stack variables. These changes are not significant and are limitations of our prototype.

The experimental results are summarized in Figures 4 and 5. Detailed experimental data for Figure 4 is presented in Appendix A.

Verifying Safe Programs Figure 4 tests our prototype tool against several memory access safe functions that manipulate data-structures. Here, 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. Ideally both %Pass and \#Safe should be 100%.

Overall, our prototype tool verifies memory access safety for the majority (55/60) of the test suite. The results demonstrate that our approach can analyze “real-world” code such as in GLib. The tool failed to prove memory access safety for the following functions: (g_list) append, sort; (g_slist) insert, sort; and (SLL_ENTRY) copy. Each failed case is caused by a current limitation of our prototype. For example, the g_slist/insert function contains the pattern:

\[
\begin{align*}
\text{new_list} & = \text{malloc}(\text{sizeof(node)}); \text{new_list} -> \text{data} = \text{data}; \ldots \\
\text{while (} \text{position} -- > 0 \text{) && tmp_list} \{ \ldots \} \\
\text{new_list} -> \text{next} & = \text{prev_list} -> \text{next};
\end{align*}
\]

Here, new_list is allocated before a loop, but new_list->next is not initialized until after the loop. The loop therefore violates the assumed invariant that the heap is closed, causing the analysis to fail. This could be fixed by refining the analysis to allow for weaker invariants, or by modifying the code to explicitly initialize the next field (e.g. by zeroing) before entering the loop.

Comparison Versus SLAyer and Verifast In Figure 5 we compare our prototype against the Separation Logic-based tools SLAyer and Verifast. In contrast to our prototype tool:

- SLAyer and Verifast enforce full memory safety, including some memory error types we do not detect, such as null-pointer access, double frees, etc.;
- SLAyer’s recursively-defined data-structure reasoning is limited to linked-lists (e.g. SLL_ENTRY), whereas our prototype tool can handle a variety of different data-structures; and
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>-</td>
<td>0.10</td>
<td>✓</td>
<td>4.75</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>0.10</td>
<td>×</td>
<td>2.11</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>0.09</td>
<td>×</td>
<td>1.67</td>
<td>×</td>
</tr>
<tr>
<td>copy</td>
<td>-</td>
<td>0.09</td>
<td>✓</td>
<td>1.15</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>0.09</td>
<td>×</td>
<td>1.14</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>0.16</td>
<td>×</td>
<td>0.69</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>0.11</td>
<td>×</td>
<td>0.44</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>0.11</td>
<td>×</td>
<td>1.78</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>0.11</td>
<td>×</td>
<td>0.56</td>
<td>×</td>
</tr>
<tr>
<td>create</td>
<td>-</td>
<td>0.05</td>
<td>✓</td>
<td>0.22</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>0.04</td>
<td>×</td>
<td>0.22</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>0.04</td>
<td>×</td>
<td>0.23</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>0.03</td>
<td>×</td>
<td>1.28</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>0.04</td>
<td>×</td>
<td>0.64</td>
<td>×</td>
</tr>
<tr>
<td>filter</td>
<td>-</td>
<td>0.15</td>
<td>✓</td>
<td>2.67</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>0.14</td>
<td>×</td>
<td>T.O.</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>0.12</td>
<td>×</td>
<td>0.67</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>W</td>
<td>0.15</td>
<td>×</td>
<td>4.63</td>
<td>×</td>
</tr>
</tbody>
</table>

- Verifast can handle the memory safety and functional verification of a larger set of data-structures. However, Verifast is not automatic, and is reliant on user annotations, hence, we use their already annotated benchmarks. Thus, the comparison in Figure 4 takes into account the restrictions of the compared tool.

For the benchmarks in Figure 5 we consider several versions of the same function with different memory access bugs introduced. The bugs are:
- (−) = no bug, the function is safe.
- A = wrong sized allocation, e.g. p = malloc(sizeof(node)/2);
- O = increment pointer by offset, e.g. p = p + 5;
- W = cast integer to a wild pointer, e.g. p = (node *)12345;
- U = uninitialized pointer, e.g. node *p; p->next = q; and
- F = uninitialized field(s), e.g. p = malloc(sizeof(node)); return p;
Such bugs may be obfuscated at the LLVM IR-level which we handle.

The results are shown in Figure 5. Here Time is the running time (in seconds), where (T.O.) represents a timeout (of five minutes), and Safe? is the analysis result according to the tool. Here (✓) indicates the tool correctly detects safety, (✗) indicates the tool correctly detects (potential) unsafety, (?) indicates the tool gave a wrong answer (e.g. safe when unsafe) or the tool times-out. Note that the copy example from Figure 4 was trivially modified to pass the analysis.

The results show that SLAyer fails to detect some kinds of memory errors altogether, namely those caused by allocation (A) and offset (O) bugs. This is
likely caused by bugs and/or limitations of the SLAyer tool. Excluding such benchmarks, we see that our tool, though more general, is significantly faster (total = 1.61s) than SLAyer (total = 28.28s).

Verifast is a manual verification tool not designed for automated analysis. Nevertheless, for the tested benchmarks, Verifast (with user annotations provided) is of similar speed to our prototype (excluding schorr). For the schorr_waite benchmark, the (†) indicates that Verifast only proves safety w.r.t. binary trees (BT). Our tool is more liberal, proving memory access safety for binary graphs (BG) which is the intended use (i.e. garbage collectors). Note that Verifast can not be used to confirm that a program is unsafe. As such, we do not compare against Verifast for unsafe programs.

7 Related Work

This work is built on top of the $\mathcal{H}$-language and solver from [7]. The $\mathcal{H}$-language is related to Separation Logic [15], but with some key differences. The main problem is that Separation Logic enforces full memory safety, which is too strong for this work. By using the $\mathcal{H}$-language, we can encode memory access safety directly. Furthermore, the $\mathcal{H}$-solver is readily extensible to handle the node and closed properties from Section 4.

In general, memory safety is a very well researched topic [16]. The static component of our work is most closely related to Separation Logic-based tools such as SLAyer [2], Smallfoot [1], Verifast [9], etc. Unlike our approach, such tools do not assume a protected environment, and attempt to prove full memory safety. However, a tradeoff exists: full memory safety analysis is more difficult, and thus the above tools inherit significant limitations, such as (lack of) automation, or limits on the type of data-structures they can handle.

Dynamic systems for preventing or catching memory errors, such as Cyclone [11], CCured [14], and SoftBound [13], employ runtime mechanisms possibly with garbage collection. The dynamic component of our work, i.e. the “protected environment”, similarly assumes garbage collection. Some alternatives to garbage collection exist, such as [4], that could potentially be adapted to our work.

8 Conclusions and Future Work

This paper presents a memory access safety analysis for programs that manipulate recursively defined data-structures, such as lists, trees, etc. Experimental results are promising, and show that a prototype implementation of the analysis scales well, and can complete with existing analysis tools for full memory safety.

For future work we wish to extend the analysis instantiation to handle arrays and more complex data-structures over multiple types. There is potential for this work to be applied to binary analysis, i.e. to check the memory access safety of untrusted modules or libraries distributed in binary or LLVM IR form [5].

---

5 Verifast will merely report that the user annotations failed to prove safety.
References

## A Appendix: Detailed Experimental Data

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>#VC</th>
<th>Time</th>
<th>Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_list_append</td>
<td>26</td>
<td>16</td>
<td>0.14</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_concat</td>
<td>20</td>
<td>16</td>
<td>0.16</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_delete_link</td>
<td>39</td>
<td>126</td>
<td>0.48</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_find</td>
<td>16</td>
<td>17</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_index</td>
<td>22</td>
<td>14</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_insert_before</td>
<td>53</td>
<td>51</td>
<td>0.19</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_insert</td>
<td>37</td>
<td>32</td>
<td>0.12</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_insert_sorted</td>
<td>59</td>
<td>221</td>
<td>1.21</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_last</td>
<td>16</td>
<td>18</td>
<td>0.08</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_length</td>
<td>16</td>
<td>11</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_nth</td>
<td>15</td>
<td>25</td>
<td>0.10</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_nth_data</td>
<td>14</td>
<td>25</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_nth_prev</td>
<td>14</td>
<td>25</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_position</td>
<td>22</td>
<td>15</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_prepend</td>
<td>22</td>
<td>19</td>
<td>0.08</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_remove_all</td>
<td>28</td>
<td>52</td>
<td>0.16</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_remove_link</td>
<td>38</td>
<td>126</td>
<td>0.47</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_reverse</td>
<td>18</td>
<td>18</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_list_sort</td>
<td>74</td>
<td>148</td>
<td>3.71</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_append</td>
<td>24</td>
<td>13</td>
<td>0.05</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_concat</td>
<td>18</td>
<td>14</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_find</td>
<td>16</td>
<td>17</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_index</td>
<td>22</td>
<td>14</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_insert_before</td>
<td>44</td>
<td>54</td>
<td>0.13</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_insert</td>
<td>41</td>
<td>60</td>
<td>0.20</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_insert_sorted</td>
<td>58</td>
<td>130</td>
<td>0.36</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_last</td>
<td>16</td>
<td>16</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_length</td>
<td>16</td>
<td>11</td>
<td>0.05</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_nth</td>
<td>14</td>
<td>25</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_nth_data</td>
<td>15</td>
<td>33</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_nth_prev</td>
<td>14</td>
<td>25</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_position</td>
<td>22</td>
<td>14</td>
<td>0.05</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_prepend</td>
<td>13</td>
<td>5</td>
<td>0.03</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_remove_all</td>
<td>29</td>
<td>36</td>
<td>0.12</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_remove_link</td>
<td>38</td>
<td>126</td>
<td>0.47</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_reverse</td>
<td>18</td>
<td>18</td>
<td>0.07</td>
<td>✓</td>
</tr>
<tr>
<td>g_slist_sort</td>
<td>74</td>
<td>148</td>
<td>3.71</td>
<td>✓</td>
</tr>
<tr>
<td>g_tree_find_node</td>
<td>41</td>
<td>23</td>
<td>0.22</td>
<td>✓</td>
</tr>
<tr>
<td>g_tree_height</td>
<td>20</td>
<td>19</td>
<td>0.28</td>
<td>✓</td>
</tr>
<tr>
<td>g_tree_insert_internal</td>
<td>85</td>
<td>454</td>
<td>11.77</td>
<td>✓</td>
</tr>
<tr>
<td>g_tree_node_balance</td>
<td>20</td>
<td>30</td>
<td>1.24</td>
<td>✓</td>
</tr>
<tr>
<td>g_tree_node_rotate_left</td>
<td>38</td>
<td>52</td>
<td>0.50</td>
<td>✓</td>
</tr>
<tr>
<td>g_tree_node_rotate_right</td>
<td>38</td>
<td>52</td>
<td>0.50</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Total**: 618 LOC, 1148 #VC, 7.78 Time, 1146 (99.8%)