Computer Sciences Department

WHATSAT: Dynamic Heap Type Inference for Program Understanding and Debugging

Marina Polishchuk
Ben Liblit
Chloë W. Schulze

Technical Report #1583

December 2006

UNIVERSITY OF WISCONSIN MADISON

WHATSAT: Dynamic Heap Type Inference for Program Understanding and Debugging*

Marina Polishchuk
Microsoft Corporation †
marinapo@microsoft.com

Ben Liblit
University of Wisconsin–Madison
liblit@cs.wisc.edu

Chloë W. Schulze
Oracle Corporation †
chloe.schulze@oracle.com

December 2006

Abstract

C programs can be difficult to debug due to lax type enforcement and low-level access to memory. We present a dynamic analysis for C that checks heap snapshots for consistency with program types. Our approach builds on ideas from physical subtyping and conservative garbage collection. We infer a program-defined type for each allocated storage location or identify "untypable" blocks that reveal heap corruption or type safety violations. The analysis exploits symbolic debug information if present, but requires no annotation or recompilation beyond a list of defined program types and allocated heap blocks. We have integrated our analysis into the GNU Debugger (gdb), and describe our initial experience using this tool with several small to medium-sized programs.

1 Introduction

Suppose that a programmer notices an incorrect variable value during the execution of a C program. While debugging, the programmer may try to observe values of the variable at various points during execution, either by setting a watchpoint in the debugger or by inserting print statements. However, both of these techniques may be inadequate. Debugger watchpoints can be prohibitively slow. Adding print statements may be ineffective in cases of memory corruption, as the affected data structure may have no apparent relation to the code that corrupts it. The programmer may also run a static pointer analysis to check for erroneous memory accesses. However, typical analyses [1, 24] do not model dependencies between neighboring memory blocks. In C, many bugs are caused by buffer overruns and pointer mismanagement, so physical proximity of memory blocks is an important factor.

We have designed and implemented an automated tool to help programmers understand and debug program behavior at the physical memory level. Our tool offers the programmer a low-level, but typed, view of memory. Each allocated chunk is presented either according to its inferred overall data type or as "untypable" if no defined program type is compatible with its imposed constraints. A common scenario where this view is useful is in the case of a buffer overrun: rather than attempt to deduce what data structure lies near the corrupted location from raw memory values or printed variables, the programmer uses our tool to discover that a nearby memory block

*Supported in part by NSF under grant CCR-0305387.

†This work performed while at the University of Wisconsin–Madison.

is being used as an array of a particular type. The mere presence of an array makes "buffer overrun" a good hypothesis, and the array's type helps the programmer identify relevant code to examine for possible array bounds violations.

When memory has been corrupted, our tool may detect that values stored in one or more heap blocks do not match any feasible type for the block. In order to reason about the cause of the corruption, the programmer may want to know exactly when the heap first became corrupted. For this task, we support a binary search debugging strategy often used to find ill-behaving code: set a breakpoint at the location where a variable was last known to be correct, and another where it has the wrong value, then iteratively narrow the interval until the bad assignment is exposed. Our tool gives the programmer the ability to treat the entire heap as a memory value that is either in a good or bad state, and search for the time at which the heap was first corrupted using her usual dynamic debugging techniques, such as the binary search described above.

This paper makes the following contributions:

- We introduce the idea of a *consistent typing* for the heap at any given point during execution. Each block of allocated storage is assigned a type from among those used in the program. The type assignment satisfies a set of constraints imposed by the values stored in memory, the size of each allocated block, and a type conformance relation ($\preceq$) based on physical subtyping. Additional constraints may be imposed by declared program variables when debugging information is available.

- We give an algorithm that finds a consistent typing for a snapshot of the heap. When no consistent typing exists, we report the locations and causes of conflicts to the user.

- We present a memory visualization that focuses on physical layout and provides a lucid representation of how typed data is stored.

The remainder of this paper is organized as follows. We give an overview of our system and define basic terms in Section 2. Section 3 defines a subtyping relation on types used for individual bytes. Section 4 specifies the constraints that must be satisfied for a set of typed memory locations to be considered consistent, and Section 5 gives an algorithm to find a consistent typing for the entire heap. Section 6 discusses visualization and presents several case studies using our tool. Section 7 places our system in context with related work. Section 8 outlines future directions for our technique and concludes.

2 Preliminaries

Here, we describe the scenario under which we solve the problem of finding a consistent typing for the heap, outline our solution, and provide key definitions and notation used throughout the paper.

2.1 Definitions

A *typing* is a map $T : \mathit{Store} \rightarrow \mathit{Types}$ from each storage location ($\mathit{addr} \in \mathit{Store}$) to its corresponding type ($\tau \in \mathit{Types}$). Types are all those defined or used by the program, including structures, unions, pointers, arrays, and functions. Storage locations include:

- a fixed set of addresses holding global variables

- the set of locations that hold all local variables and function arguments on the stack at the current program point, or on multiple stacks for multithreaded programs

- all memory blocks dynamically allocated since the start of the program, which we refer to as *heap storage*

Globals, locals, and function arguments may have associated type information if symbolic debug information is present. However, memory blocks from heap storage never have associated dynamic type information. Unions are also untagged per standard C.

In the manner of conservative garbage collectors [3], we define a *valid pointer* as a block of memory whose value is in *Store* (i.e., is a storage location). A valid pointer may also point immediately after the

end of a block of storage; this is a common programming idiom that is explicitly allowed by the C standard [14].

A *type constraint* restricts the types that may reside at a given storage location. Our analysis imposes constraints on the types of individual bytes of memory, termed *byte types*. Informally, a byte type indicates that a byte holds the start of some program type or any subtype thereof. Byte types may also indicate that a byte is part of the interior of a multi-byte value that starts at an earlier location.

2.2 Overview

After establishing the *Store* and *Types*, our analysis proceeds as follows. First we assign byte types to all storage locations that hold valid pointers, and also to their corresponding pointed-to locations. One of the key problems that our tool tries to address is the fact that C programs often manipulate memory values in a way that disregards their declared types, which complicates debugging. Hence, our analysis treats the values that arise in the program as a foremost source of byte type constraints. Next, if symbolic debug information is available, byte types may also be transitively propagated from variables declared in the program, for which the exact type is known. However, even in the presence of this information, many bytes may remain wholly or partly unconstrained. After all available constraints are established on individual bytes, we systematically consider possible overall types for each block until the typing map is fully defined, or all typing alternatives for the memory blocks are exhausted. An overall type is assigned to a block when it is consistent with the individual byte types at each offset within the block as well as the constraints imposed by connections (via pointers) to or from other blocks. In some cases, constraints suffice to determine a unique consistent typing. Otherwise, when several consistent typings are possible, we use a search ordering heuristic to choose the most descriptive type for each heap block.

Whenever our algorithm applies a byte type to a location, the new type may conflict with an existing byte type at the same address. If a conflict arises when considering memory values and symbolic debug information if present, then we have found specific instances of pointer or declared variable usage that do not adhere to our notion of consistency (as detailed in Section 4). Such conflicts are reported to the user, and blocks containing them are marked as "untypable" and omitted during whole-block type assignment.

2.3 Notation

Figure 1 establishes a concise notation for C types, derived from that used by Siff *et al.* [22]. An array of n elements of type τ is written $\tau[n]$, while τ *ptr* is a pointer to τ. A tuple of the form $s\langle m_1,\ldots,m_k\rangle$ denotes a `struct`, while $u\{m_1,\ldots,m_k\}$ denotes an untagged union. Each m_j is a triple (τ,l,i) giving the type, name, and starting offset of one field within a structure or union. Structure fields are ordered by offset, with m_1 starting at offset zero. Union fields are unordered and all start at offset zero.

Our subtyping relation is given by $\prec$ and its reflexive closure $\preceq$. We use type notation $addr : \tau$ in place of the subtyping constraint $T(addr) \preceq \tau$ when the type mapping T is evident from context.

In the discussion that follows, byte offsets within a memory block are represented in terms of addition: if *block* is the start address of some block of storage, $block + i$ denotes the address of the i^{th} byte of *block*, starting from zero. We abbreviate the address $block + 0$ as simply *block*. The predicate $validPointer(addr)$ asserts that *addr* holds the start of a valid, non-null pointer value.

3 Byte Type Lattice

Our analysis may yield multiple distinct types for the same memory location. In some cases this reveals a conflict and likely misused memory. In other cases the types may, in fact, be compatible. This section explains how we construct a lattice from the set of C program types to model type compatibility in our analysis.

The data types in Figure 2 form a running example throughout the paper; hereafter, we omit the explicit

τ ::=
  atomic // no internal substructure
 | $\tau[n]$ // array of type τ of size n
 | $s\langle m_1,\ldots,m_k\rangle$ // `struct`
 | $u\{m_1,\ldots,m_k\}$ // untagged `union`
 | $(\tau_1,\ldots,\tau_k)\rightarrow\tau_0$ // function returning τ_0

m ::= (τ,l,i)
  // field labeled l of type τ at offset i

atomic ::=
  $e\langle id_1,\ldots,id_k\rangle$ // `enum`
 | τ *ptr* // pointer to τ
 | `char` | `int` | `double` | ...

Figure 1: Concise C type grammar

```
struct Point {
  double x;
  double y;
};

struct Shape {
  char *name;
  FILE *fptr;
};

struct Part {
  struct Point center;
  struct Shape *shape;
  struct Assembly *owner;
};

struct PartNode {
  struct Part *part;
  struct PartNode *next;
};

struct Assembly {
  struct Point center;
  struct PartNode *nodes;
  struct Assembly *owner;
};
```

Figure 2: Example data types for assembly-building program

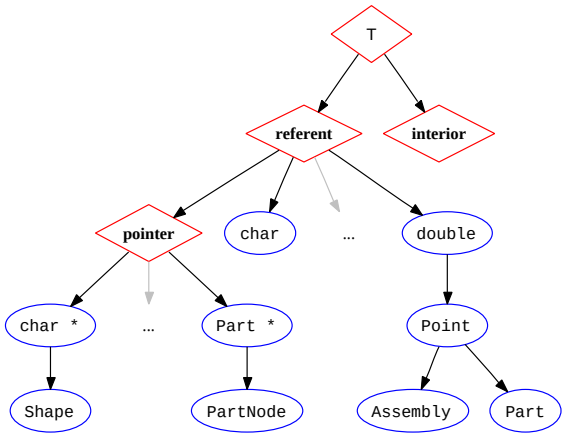

Figure 3: Byte type lattice corresponding to the data types in Figure 2

"`struct`" keyword. Given these types, the remainder of this section defines the subtyping relation used to construct the byte type lattice shown in Figure 3. For clarity, we omit $\perp$ in Figure 3 and from the subtyping definitions below. $\perp$ should be assumed as the *meet* of any two lattice elements for which no other lower bound is defined; it denotes that the corresponding two types may not be consistently stored at the same address.

3.1 Structures

A structure and its initial (offset 0) field have the same physical address but distinct types. Therefore the immediate supertype of a structure is the type of its initial field:

$$s\langle(\tau_1,l_1,0),\ldots,m_k\rangle \prec \tau_1$$

For example, the relationships `Part` $\prec$ `Point` $\prec$ `double` are read as "If an address holds the 0^{th} byte of a `Part`, then it also holds the 0^{th} byte of a `Point` and the 0^{th} byte of a `double`." By contrast, a structure containing three `double`s would not be considered a subtype of `Point`, even though they have similar layouts.

This definition is stricter than the physical subtyping of Chandra and Reps [5], in which a structure $s\langle m_1,\ldots,m_k\rangle$ is a subtype of any of its prefixes

$s\langle m_1,\ldots,m_i\rangle, i < k$. Our design allows only those pointer aliases that may arise in a program that does not use casts to evade the type system. This is merely a policy choice. More permissive alternatives could be used with no change to the rest of the analysis, and may be desired for use with certain programming idioms.

3.2 Special and Atomic Types

The four diamond-shaped nodes at the top of Figure 3 are always present. $\top$ denotes an unconstrained memory address that may hold any type. **referent** describes all types that can be the referent of a pointer, while **interior** describes the non-initial bytes of multi-byte atomic values. Pointer targets must be **referent** subtypes and can never be **interior**. For example, if v holds an eight-byte `double`, then byte $v+0$ has type `double` but bytes $v+1$ through $v+7$ all have type **interior** and may not be pointed to directly. Bitfields are also typed as **interior**, for their addresses cannot be taken.

pointer indicates that a storage location holds the 0^{th} byte of a valid pointer, and thus is potentially consistent with any pointer type. Pointers to pointers are allowed, so **pointer** $\prec$ **referent**. **pointer** is not the same as `void*`; the former represents all pointers, while the latter is a specific program type.

The oval nodes in the lattice correspond to actual types that may be used in a C program. Notice that the primitive atomic types are all sibling immediate subtypes of **referent**. This stipulates that two or more atomic types may not be simultaneously stored at the same address.

As a special case, we treat `void` as a zero-length type that is identical to **referent**. Although no byte should ever have type `void` in the final result, this convention allows transparent handling of aliases between `void*` and more fully typed pointers: pointers to τ and `void` may refer to the same address in our system as though `void` were a zero-length prefix of every **referent** subtype.

The subtyping relation is not extended across pointers. That is, $\tau_1 \preceq \tau_2 \not\Rightarrow \tau_1\ ptr \preceq \tau_2\ ptr$. This is the standard restriction for subtyping with updatable references, and any program that requires this form of subtyping to describe its heap must necessarily have used casts or other measures to violate type safety.

3.3 Arrays

We define the immediate supertype of an array type as the type of its elements:

$$\tau[n] \prec \tau$$

$\tau[n]$ may be viewed as $s\langle m_1,\ldots,m_n\rangle$, where all m_i have type τ, so the reasoning for this relation is analogous to that for structures.

3.4 Unions

Untagged unions require special treatment, because a union may be used as any of its fields, but a consistent typing requires that every address be assigned a unique type. For each untagged union type $u\{m_1,\ldots,m_k\}$, we extend the type grammar to include one tagged case $u@m_r\{m_1,\ldots,m_k\}$ for each $1 \le r \le k$. Unions and their tagged cases adhere to the following subtyping relations:

$$u\{m_1,\ldots,m_k\} \prec \textbf{referent}$$
$$u@(\tau_r,l_r,0)\{m_1,\ldots,m_k\} \prec u\{m_1,\ldots,m_k\}$$
$$u@(\tau_r,l_r,0)\{m_1,\ldots,m_k\} \prec \tau_r$$

These relations forbid aliased pointers to differently-typed union fields. Each union must be used in a single consistent manner at any given point during execution. For example, a tagged union storing a `Point` can be the target of pointers to the untagged union as well as pointers to `Point` and `double`, but could not be the target of a pointer to `Part`. When two or more fields of a union have a common supertype, additional cases can be introduced that represent a subset of possible tagged cases rather than a single case. This preserves uniqueness of the lattice *meet* operation.

Note that only tagged unions and $\bot$ have multiple immediate supertypes. The byte type lattice without these becomes a tree.

3.5 Functions

Every function may be pointed to:

$$(\tau_1, \ldots, \tau_k) \rightarrow \tau_0 \prec \textbf{referent}$$

No other proper subtyping relations exist among function types. We allow neither return type covariance nor argument type contravariance, as these are not part of standard C. Calling a function with too many arguments, while safe in many C implementations, is also not endorsed by the standard and therefore not admitted here. This is merely a policy choice. More permissive alternatives could be used with no change to the rest of the analysis.

3.6 Finite Type Space

The byte type lattice contains an unbounded number of types, including arrays of arbitrary length and pointers of arbitrarily deep nesting. In practice, we consider only the following finite subset of types that are likely to be meaningful and useful for a given program:

- program-declared structures, unions, and enumerations
- tagged variants of unions or types containing unions
- arrays used by the program, e.g. `int[3][5]` if and only if at least one field or variable has exactly this type
- pointers used by the program, e.g. `int****` if and only if at least one field or variable has exactly this type
- pointers to known types up to two more levels of indirection

The number of tagged variants of unions and union-containing types is potentially exponential. In our experience, multi-union structures and nested unions are unusual, and therefore the number of tagged variants is typically linear.

Additional array types are synthesized as needed during the analysis to satisfy size constraints (Section 4.2), but only using element types appearing in the original program. For example, a block of 32 bytes may be typed as `int[8]` if `int` is a known 4-byte type. We do not consider `int[2][4]` unless the corresponding element type (`int[4]`) appeared in the original program.

```
  void main() {
    ...
    carAssm = create_assembly();
    ...
  }

  Assembly *create_assembly() {
1   Assembly *assm =
      malloc(sizeof(Assembly));
2   PartNode *node =
      malloc(sizeof(PartNode));

3   node->part = malloc(sizeof(Part));
4   node->next = node;
5   assm->nodes = node;

    // build part's shape and set name
6   init_part(node->part, "door", assm);
    ...
7   return assm;
  }
```

Figure 4: Program that builds a simple assembly

4 Consistency Constraints

In this section, we specify four kinds of constraints on storage locations that restrict the possible types for heap blocks. We then show how these constraints are combined to derive a consistent heap typing at one point in an example program. For simplicity, we assume a 32-bit architecture with 4-byte pointers and 8-byte `double`s. Our ideas generalize to 64-bit or other architectures as well.

The program excerpt in Figure 4 creates a part for a simple assembly and initializes it with its shape and owner assembly. Figure 5 shows the heap after the call to `init_part()` on line 6 of Figure 4. Blocks are labeled **A**–**E** arbitrarily, with the line number of each block's allocation given next to its label. Valid pointer values are shown in a C-style syntax, and the rest of memory is assumed to be set to zero when returned by `malloc()`. Bracketed numbers indicate

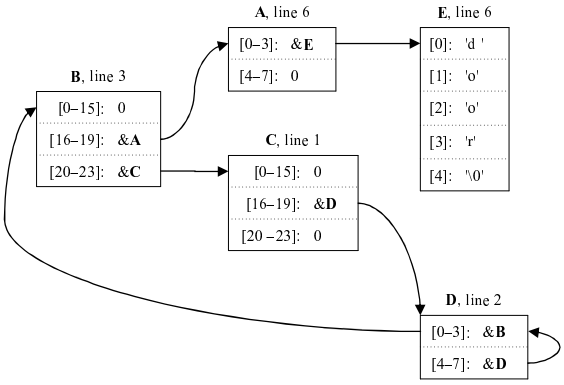

Figure 5: Allocated blocks and values after `init_part()` call in Figure 4

byte ranges within each block.

4.1 Value Constraints

Value constraints arise from concrete data values in memory at the instant the analysis is applied. They reflect the fact that some data types have limited domains that are much smaller than the set of all possible values that can fit in the allotted memory. The following generic value constraints are useful across a wide variety of C programs:

- valid pointer constraint

  $$\mathit{validPointer}(\mathit{addr}) \Rightarrow T(\mathit{addr}) \preceq \textbf{pointer}$$

  Valid pointer constraints are used to infer basic type information: if a value looks like a valid pointer, we require that it be typed as a pointer (Section 5.1). This assumes that no non-pointer ever takes on a valid pointer value by chance, a strategy widely employed by conservative garbage collectors [3].

- enumeration constraint

  A byte of type `enum` is consistent only if the value starting at that byte is equal to one of the defined constants for the enumerated type. Enumeration constants are not uniquely identifying in general, so this rule cannot be used to infer basic type information from values alone. However, this rule can be used as a filter, to reject inferred or hypothesized types that are definitely inconsistent with observed values.

- function pointer constraint

  In the presence of shared libraries without debug information, not all functions' start addresses are known. Therefore we treat a function pointer type as consistent if its value is any word-aligned address that may contain executable code. The mechanism for identifying code pages is platform specific. Due to the difficulty of reliably distinguishing code from data, function pointer constraints are best used as filters in the manner of enumeration constraints.

- character constraint

  In a program that manipulates ASCII text, if a block is otherwise unconstrained and ASCII character values are stored at every offset in the block, then the block should be typed as `char` or `char[`n`]` rather than any other consistent primitive or primitive array type (e.g. `double[`$n/8$`]` or `short[`$n/2$`]`). Character constraints are used to change the type search order rather than to infer or reject types. When this constraint is not applied, character arrays are considered as a last resort after arrays of other primitive types have been rejected (Section 5.3.3). [1]

Programmers may wish to define additional value constraints to reflect application-specific types and invariants. Our heap typing algorithm can accommodate arbitrary predicates that approve or reject the type proposed for a given location and value. For example, the data structure consistency specifications of Demsky *et al.* [6] could be applied as additional value constraints for selected types.

Table 1 shows valid pointer value constraints for the heap snapshot in Figure 5, and the possible types whose physical layouts are consistent with each constraint. An important detail is that, while locations

[1] Similar constraints can be used in other, non-ASCII locales when the character repertoire is known in advance.

Block	Valid Pointers	Value-Consistent Types
A	$\mathbf{A}+0$	`PartNode`, `Shape`
B	$\mathbf{B}+16$, $\mathbf{B}+20$	`Part`, `Assembly`
C	$\mathbf{C}+16$	`Part`, `Assembly`
D	$\mathbf{D}+0$, $\mathbf{D}+4$	`PartNode`, `Shape`

Table 1: Valid pointer value and possible value-consistent types for heap in Figure 5

Block	Size	Size-Consistent Types
A	8	`PartNode`, `Shape`, `char[8]`, ...
B	24	`Part`, `Assembly`, `Shape[3]`, `PartNode[3]`, `float[6]`, ...
C	24	`Part`, `Assembly`, `Shape[3]`, `PartNode[3]`, `float[6]`, ...
D	8	`PartNode`, `Shape`, `int[2]`, ...
E	5	`char[5]`

Table 2: Size constraints and possible size-consistent types for heap in Figure 5

holding zero are unconstrained, zero is consistent with either a pointer type or most primitive types. In this example, both `PartNode` and `Shape` are value-consistent with block **A** if the value at $\mathbf{A}+4$ is viewed as a null pointer.

4.2 Size Constraints

The overall type for a block must fill exactly the number of bytes allocated for that block. For any address *block* which is the start of an allocated block,

$$T(block) = \tau \;\Rightarrow\; sizeof(block) = sizeof(\tau)$$

In C, dynamically allocated arrays tile multiple copies of the element type one after the other. A block holding a dynamic array with n elements of type τ must satisfy

$$sizeof(block) = n \times sizeof(\tau)$$

for some whole number of array elements n.

Table 2 shows size constraints and a few illustrative size-compatible types for blocks in the example heap snapshot.

4.3 Type Constraints

Type constraints relate multiple locations, either between blocks (for pointers) or within a single block (for multi-byte structures):

(*i*) If $T(addr) \preceq$ **pointer** and $*addr$ is within an allocated block (not one past the end), then $T(*addr) \prec$ **referent**.

(*ii*) If $T(addr) \preceq \tau$ for any atomic type τ as defined in Figure 1, then $T(addr+i) =$ **interior** for all $1 \le i < sizeof(\tau)$. Combined with rule (*i*) and the incompatibility of **interior** and **referent**, this forbids pointers into the interior of atomic values.

(*iii*) For any currently allocated block starting at *block*, $T(block) \prec$ **referent**. While similar to rule (*i*), this rule also affects leaked blocks to which nothing points.

(*iv*) If $T(addr) \preceq \tau\ ptr$ then $T(*addr) \preceq \tau$. Pointers and pointed-to types must be compatible modulo subtyping.

(*v*) If $T(addr) \preceq s\langle(\tau_1, l_1, i_1), \ldots, (\tau_k, l_k, i_k)\rangle$, then $T(addr + i_n) = \tau_n$ for all $1 < n \le k$. Structure fields must be compatible with the structure as a whole.

(*vi*) If $T(addr) = \tau[n]$, then $T(addr + i \times sizeof(\tau)) = \tau$ for all $1 \le i < n$. Array elements must be compatible with the array as a whole.

Implied constraints may imply additional constraints. A consistent heap typing must satisfy all transitively implied type constraints.

Untagged unions induce no additional type constraints. Any tagged union type $u@(\tau, l, 0)\{\ldots\}$ is also a subtype of τ and will pick up any appropriate constraints per the above rules.

4.4 Debug Constraints

If symbolic debug information is available for in-scope variables and function arguments, then this information may be added to the type map in the obvious manner. Equality constraints are appropriate

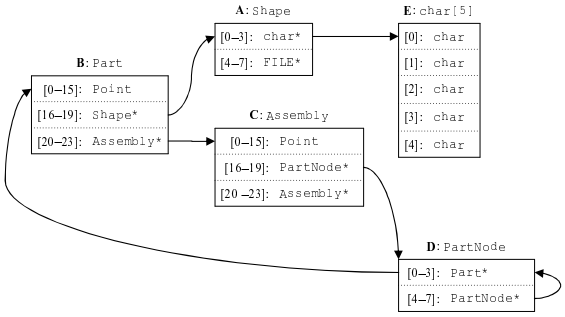

Figure 6: Fully constrained heap and derived typing for heap in Figure 5

here: if `x` is known to be an integer variable, then its type must be `int`, not some `int` subtype. Our main algorithm (Section 5) does not require debug information to find consistent typings, but takes debug constraints into account if present.

4.5 Example Solution

We now combine value, size, and type constraints to informally derive the consistent heap typing shown in Figure 6. Section 5 presents a systematic algorithm for finding consistent typings automatically. For clarity, we consider user-defined types before other possible matches (such as arrays of primitives). Assume that debug constraints are unavailable.

From valid pointer value and size constraints, block **A** must have type `Shape` or `PartNode`. If `Shape` is considered first, we propagate a `char` constraint to block **E** via rule (*iv*).

Value and size constraints require that block **E** have type `char[5]`. Because `char[5]` $\preceq$ `char`, block **E** can have type `char[5]` and still be consistent with the `char` pointer from block **A**. If we had tried **A** : `PartNode` first, then a conflict would have arisen; we discuss conflicts in detail in Section 5.

From size and value constraints, block **B** must have type `Assembly` or `Part`. If we try **B** : `Assembly`, then **B** $+ 16$: `PartNode*` via rule (*v*) and so **A** : `PartNode` via rule (*iv*), but this conflicts with **A** : `Shape` established earlier. Choosing **B** : `Part` is consistent with **A** : `Shape`, and also requires **C** : `Assembly`. This is consistent with value and size constraints on block **C**. Lastly, block **D** must have type `PartNode` due to the pointer field at **C** $+ 16$.

When there are few initial constraints or many identically-structured types, a consistent typing is not necessarily unique. For example, if all bytes in block **D** were zero, then the following would also be a consistent heap typing:

A : `char*[2]`	**D** : `char[8]`
B : `char**[6]`	**E** : `char[5]`
C : `char*[6]`	

5 Heap Typing Algorithm

In this section, we present an algorithm for assigning types to all storage locations, if a consistent typing is possible. Inputs to the algorithm consist of a snapshot of all values in the program's memory; the start addresses and sizes of all allocated heap blocks; a list of defined program types; and optional symbolic debug information giving the locations, sizes, and types of in-scope variables. The output is a typing T giving consistent byte types for all allocated bytes. The byte type for the 0^{th} byte of an allocated block gives the overall type for that block. In some (but not all) cases when no globally consistent typing exists, the algorithm can identify, describe, and eliminate untypable blocks while still producing a partial typing for the remaining blocks.

The algorithm begins by assigning types to individual bytes of storage, using the values that arise in the program (e.g., valid pointers and their pointed-to locations) as a foremost source of byte type constraints (Section 5.1). Next we transitively propagate byte types from variables declared in the program (for which exact types are known), reporting any type constraint violations to the user (Section 5.2). Finally, we systematically consider possible overall types for each memory block until the typing map is fully defined or all typing alternatives for the blocks are exhausted (Section 5.3).

5.1 Pointer Constraint Gathering

First we establish valid pointer value constraints on individual bytes as described in Section 4.1. Type constraint rules (*i*)–(*iii*) induce additional constraints where appropriate. Conflicts between **referent** and **interior** may arise during this stage. For example if blocks **X** and **Y** hold valid pointers, but **Y** points to byte $\mathbf{X}+2$, then $\mathbf{X}+2$ cannot simultaneously be the referent of the pointer in **Y** and the interior of the pointer in **X**. In this situation, it is difficult to know which block is truly erroneous. We describe the conflict to the user, then mark both blocks as untypable and disregard them for the remainder of the algorithm.

After this phase, the typing constrains some bytes to be subtypes of **pointer**, **interior**, or **referent**, but many bytes remain unconstrained ($\top$) and no program types yet appear.

5.2 Debug Constraint Gathering

If symbolic debug information is available, debug constraints are applied next, then propagated transitively across pointers and into compound types using type constraint rules (*iv*)–(*vi*). Value and size constraints are checked where appropriate. Note, however, that size constraints are only partially enforced: a block must be at least large enough to contain the expected type, but may be larger. For example, the target of a `double*` must be at least eight bytes long, but may be longer if it is part of a larger structure, union, or array.

During this stage, conflicts may arise among debug constraints (e.g. `int` and `float` in the same location); between debug and size constraints (e.g. `int` in a two-byte block); between debug and value constraints; or between debug and **pointer**, **referent**, or **interior** constraints derived in the previous stage. If any such conflict occurs, we assume that execution has deviated from type safety and that the static type system therefore cannot be trusted to predict run-time types. We report the problem to the user and then back out all debug constraints. The remainder of the algorithm will operate using observed memory values only, without considering declared variable types. A more selective approach, which we leave as future work, would be to discard only a minimal subset of problematic debug constraints while keeping the remainder.

Barring conflicts, at the conclusion of this phase, the global typing includes program types for memory addresses that are (transitively) reached from pointers in program variables. However, these types are merely lattice upper bounds. For example, a pointer to `char` may actually point to an array of characters or to a structure with an initial `char` field. Some bytes within reachable blocks, and all bytes within unreachable blocks, still carry only the **pointer**, **referent**, and **interior** constraints added previously.

5.3 Completing the Heap Typing

Given the initial byte type constraints, we next assign an overall type to every heap block. Fully enforced size constraints ensure that the size of a block's overall type is equal to the block size, so all allocated bytes are constrained when the typing is completed and the 0^{th} byte of each block determines the block's overall type.

5.3.1 Typing Feasibility Check

Typing completion begins by verifying that every heap block may be assigned at least one program type, given the initial constraints. A block that cannot be assigned any known type may be corrupt or may have been allocated in a library whose internal types are unavailable. We describe the problematic block to the user, then mark it as untypable and disregard it in the type search phase that follows.

5.3.2 Search Algorithm

The main search phase considers the possible types for each block, backtracking in the event of conflicts. When a type is verified as consistent with all current value, size, type, and (optional) debug constraints on a particular block, we update the byte types for all bytes in this block to reflect the overall type, as well as propagate constraints one level forward across pointers in the block, and proceed to the next block. If no consistent type is found for some block,

Block	Type Considered	Outcome	Induced Constraints on Other Blocks
C	`FILE`	size conflict: sizeof(**C**) < sizeof(`FILE`)	
C	`Part`	✔	**D** + 0 : `Shape`
D	`Shape`	type conflict at **D** + 0: *meet*(`Shape`,`FILE`) = ⊥	
C	`Assembly`	✔	**D** + 0 : `PartNode`
D	`PartNode`	✔	**B** + 0 : `Part`
A	`PartNode`	size conflict: sizeof(**E**) < sizeof(`PartNode`)	
A	`Shape`	✔	**E** + 0 : `char`
E	`char[5]`	✔	

Table 3: Heap typing algorithm execution trace

we backtrack to the last block that still has remaining type alternatives, and resume the search from that point. The algorithm terminates either when the last considered block is assigned a consistent type, or when all possible types for all blocks have failed, in which case no consistent typing exists.

Naïvely implemented, this search is geometric in the number of blocks and exponential in the number of types. However, since an entire block's type must be a subtype of its 0^{th} byte's type, the search can be restricted to the sublattice below this bound. This optimization is especially effective with debug constraints enabled, as most pointers refer to the initial byte of an allocated block.

5.3.3 Heuristic Type Ordering

As discussed in Section 4.5, a heap may have several consistent typings. We prefer a typing that is most informative for the user, so we consider the possible types for each block in a particular order:

1. `struct`s, large to small
2. tagged `union`s, large to small
3. atomic types, large to small
   - non-`enum`s before `enum`s of the same size
4. arrays, recursively sorted by element type
5. pointers, recursively sorted by referent type

Ties in the above ordering are broken arbitrarily. Note that untagged unions are omitted from the order. We treat untagged unions as akin to abstract base types in object oriented languages. No block ever has an untagged union as its actual type; only specific tagged cases of a union may be "instantiated" as allocated blocks.

Intuitively, if a block can have a program-defined type, then showing that type rather than an array of primitives may be more informative. The size ordering addresses the same issue: we try to assign a large struct type to a block before considering an array of small structures. We place primitives before pointers because we find that an unconstrained block containing all zeroes is presented more naturally as an array of primitives than an array of null pointers. Recursive sorting of pointer types places type-specific pointers like `int*` before generic `void*`, and shallowly nested pointers like `void*` before deeply nested pointers like `void***`.

The order is designed to heuristically direct our solution toward more useful typings. We do not guarantee that the final heap typing is globally minimal or optimal with respect to this order, but we find that it yields good results in practice.

5.4 Example: Computing the Solution

We now illustrate the algorithm as applied to our running example, again considering the heap snapshot after the `init_part()` call on line 6 of Figure 4. To illustrate conflict handling, we modify the val-

ues shown in Figure 5 as follows: assume that bytes $\mathbf{B}+16\ldots\mathbf{B}+19$ have been corrupted, and no longer hold a valid pointer value. All other value constraints remain as shown in Table 1.

During debug constraint collection, an inconsistency arises when a $\mathbf{B}$: `Part` constraint is propagated from variable `node`, because `Part` requires a valid pointer or null at bytes $\mathbf{B}+16\ldots\mathbf{B}+19$. Since a conflict is found, debug constraints are discarded. The search continues using only value constraints. Block **B** does not pass the typing feasibility check, so it is omitted from the search. Table 3 summarizes steps taken during the backtracking search for complete types. The remaining blocks are considered in their allocation order, and the types are ordered according to our sorting heuristic. The algorithm is able to recover the types of the four remaining blocks using value and size constraints only, backtracking several times throughout the search. The final heap typing for the four remaining blocks is as shown in Figure 6.

5.5 Propagation Correctness

After assigning a type to a block, we update constraints for the block itself and also propagate all pointer constraints forward across one dereference. We claim that this is sufficient to ensure that no inconsistency between typed blocks is overlooked.

Proof. Without loss of generality, assume there is an inconsistency between blocks **R** and **S**, which are two levels of pointer indirection apart. Let **X** be the intermediate block, and let x_R (resp. b_X) denote the the byte of **X** (resp. **S**) that is constrained by **R** (resp. **X**). To prove the claim, it is enough to show that the inconsistency between **R** and **S** is detected without crossing pointers twice, regardless of the order in which our algorithm considers the blocks.

If blocks are ordered $\langle$**R, X, S**$\rangle$ or $\langle$**S, R, X**$\rangle$, then the constraint on x_R is taken into account when **X** is assigned an overall type, and is propagated directly to **S** at that time. In both orderings, our algorithm will be unable to assign a consistent type to the last block, and will eventually backtrack to consider a different type for **R**.

For the remaining four orderings, the order of constraint propagation does not directly correspond to block order. Consider the ordering $\langle$**X, S, R**$\rangle$. The algorithm advances to a new block only if all previous ones have been consistently typed, so consistency between **X** and **S** is guaranteed when **R** is considered. At this time, two cases arise when assigning an overall type to **R**. If the type assignment requires modifying x_R, then the type is rejected, since after an overall type for **X** is determined, all of its byte types are "frozen" to henceforth reflect the overall type. Otherwise, the overall type of **X** is consistent with **R**, but, since **X** and **S** are already consistent, this violates our initial assumption that **R** and **S** conflict. The argument for the three remaining orderings ($\langle$**X, R, S**$\rangle$, $\langle$**S, X, R**$\rangle$, $\langle$**R, S, X**$\rangle$) is similar.

An alternative scenario to consider is two disconnected blocks, **I** and **J**, that both refer to the same third block, **Q**. Here, the constraints imposed by **J** on **Q** will never invalidate the consistency of **I** and **Q**, because if **I** induces a constraint $\mathbf{Q}+i:\tau$, and **J** later modifies this constraint without conflict to $\mathbf{Q}+i:\tau'$, then $\tau' \preceq \tau$. □

6 Evaluation

We have implemented the above algorithm within the GNU Debugger (gdb), a popular symbolic debugger for C [11]. When the program is stopped at a breakpoint, the user may type "`whatsat <expr>`" to perform heap type inference and then display type-annotated memory beginning at the address computed by `<expr>`.

Implementing the *validPointer* predicate requires that the debugger probe the debuggee's current heap allocation state. We modify the debuggee's memory management routines to maintain a list of currently allocated blocks in a reserved global location known to the debugger; the debugger reads this list directly from the debuggee's address space as needed. We record the start address and size of each block, plus the address of the instruction that allocated the block. `whatsat` uses the latter in diagnostic messages to report the source file, line number, and function at which each untypable block was allocated.

This extra allocation tracking uses standard hooks exposed by the GNU `libc` implementation [10] and is contained within a shared library that may be preloaded into any program one wishes to debug without recompilation or relinking. Our allocation hooks also zero-initialize newly allocated blocks. This is done to avoid spurious typing errors due to random data values in uninitialized heap memory. However, it can be useful to disable this feature in order to verify that the program under study fully initializes all of its own heap storage under normal running conditions (Section 6.5).

Code	Category
`|`	heap, allocated and typed
`*`	heap, allocated but untypable
`X`	heap, freed
`?`	heap, never allocated
`S`	stack
`D`	static data
`P`	static code
`.`	static miscellaneous

Table 4: Memory category codes

6.1 Visualization of Typed Memory

Following heap type inference, `whatsat` displays memory contents augmented with derived type information. Visualization begins at any address of the user's choosing (e.g. "`whatsat 0x9275008`" or "`whatsat &foo[3]`") and continues forward through raw memory under user control.

Figure 7 shows part of a type-annotated heap for the assembly-building program used earlier. Each line shows a capitalized hexadecimal memory address (e.g. "`0X9275008:`"), up to one word of raw memory content at that address ("`0x00000000`"), and an interpretation of that memory typed according to our algorithm ("`x = (double) 0`"). Multi-word atomic types, such as `double`, extend over multiple lines in the memory visualization. Indentation and field labels ("`x =`") reflect nesting and compound types. Figure 7 shows five distinct but proximate memory blocks containing four structures and one character array.

The "`|`" and "`?`" labels to the left of each address mark locations that are currently allocated and have never been allocated, respectively. Table 4 shows the complete list of memory category codes.

6.2 Schedule

Schedule is small C application from the Siemens buggy program suite [13]. Given a list of jobs and their priorities as input, the application computes and prints a schedule for running the jobs. We seeded schedule with an argument-transposition bug, and temporarily disable debug constraints. Running schedule on one of its test inputs (`6 1 6 inputs/lu12`) leads to a crash at:

```
if (prio_queue[i]->mem_count > 0)
```

`prio_queue` is a global array of `List` pointers, the `i`'th element of which has become null. A heap consistency check using `whatsat` finds no untypable blocks, suggesting that outright heap corruption is unlikely. By rerunning the program in the debugger, we backtrack to the last point where `prio_queue[i]` had a non-null value. `whatsat` finds types for all blocks, and in particular infers that `prio_queue[i]` points to a block of type `Ele`. Yet `prio_queue` should be an array of `List` pointers, not `Ele` pointers. We continue rerunning the program, stopping at earlier and earlier points. Each time we use `whatsat` to test whether `prio_queue[i]` has become a `Ele` pointer instead of a proper `List` pointer. This brings us to the buggy call, where an `Ele` pointer argument and a `List` pointer argument were passed in the wrong order. (The compiler's type checker failed to catch the swap due to inadequate function prototyping, an unfortunate but not uncommon problem.) Execution actually continues well beyond the bad call, in part because the physical layouts of `Ele` and `List` are sufficiently similar that code intended for one can (incorrectly but non-fatally) manipulate the other. Heap type inference, however, can distinguish the two and correctly determines that `prio_queue[i]` is not what it seems. Each `whatsat` query ran in under 0.03 seconds.

```
                                (struct Assembly)
                                   center = (struct Point)
| 0X9275008:   0x00000000           x = (double) 0
| 0X927500C:   0x00000000
| 0X9275010:   0x00000000           y = (double) 0
| 0X9275014:   0x00000000
| 0X9275018:   0x09275040        nodes = (struct PartNode *) 0x9275040
| 0X927501C:   0x00000000        owner = (struct Assembly *) 0x0
? 0X9275020:   0x00000000
? 0X9275024:   0x00000019
? 0X9275028:   0x09275008
? 0X927502C:   0x00000018
? 0X9275030:   0x0074922e
? 0X9275034:   0x00000000
? 0X9275038:   0x00000000
? 0X927503C:   0x00000011
                                (struct PartNode)
| 0X9275040:   0x09275068        part = (struct Part *) 0x9275068
| 0X9275044:   0x09275040        next = (struct PartNode *) 0x9275040
? 0X9275048:   0x00000000
? 0X927504C:   0x00000019
? 0X9275050:   0x09275040
? 0X9275054:   0x00000008
? 0X9275058:   0x08048757
? 0X927505C:   0x09275028
? 0X9275060:   0x00000000
? 0X9275064:   0x00000021
                                (struct Part)
                                   center = (struct Point)
| 0X9275068:   0x00000000           x = (double) 0
| 0X927506C:   0x00000000
| 0X9275070:   0x00000000           y = (double) 0
| 0X9275074:   0x00000000
| 0X9275078:   0x092750a0        shape = (struct Shape *) 0x92750a0
| 0X927507C:   0x09275008        owner = (struct Assembly *) 0x9275008
? 0X9275080:   0x00000000
? 0X9275084:   0x00000019
? 0X9275088:   0x09275068
? 0X927508C:   0x00000018
? 0X9275090:   0x080486f7
? 0X9275094:   0x09275050
? 0X9275098:   0x00000000
? 0X927509C:   0x00000011
                                (struct Shape)
| 0X92750A0:   0x092750c8        name = (char *) 0x92750c8 "door"
| 0X92750A4:   0x00000000        file = (struct _IO_FILE *) 0x0
? 0X92750A8:   0x00000000
? 0X92750AC:   0x00000019
? 0X92750B0:   0x092750a0
? 0X92750B4:   0x00000008
? 0X92750B8:   0x08048709
? 0X92750BC:   0x09275088
? 0X92750C0:   0x00000000
? 0X92750C4:   0x00000011
                                (char [5])
| 0X92750C8:   0x      64        [0] = 100 'd'
| 0X92750C9:   0x      6f        [1] = 111 'o'
| 0X92750CA:   0x      6f        [2] = 111 'o'
| 0X92750CB:   0x      72        [3] = 114 'r'
| 0X92750CC:   0x      00        [4] = 0 '\0'
```

Figure 7: Type-annotated heap excerpt for assembly-building program

```
? 0X804ABB8:   0x00000000
? 0X804ABBC:   0x00000000
                           (struct queue [4])
                              [0] = (struct queue)
D 0X804ABC0:   0x00000001        length = (int) 1
D 0X804ABC4:   0x0804b0d0        head = (struct process *) 0x804b0d0
                              [1] = (struct queue)
D 0X804ABC8:   0x00000001        length = (int) 1
D 0X804ABCC:   0x0804b0a8        head = (struct process *) 0x804b0a8
                              [2] = (struct queue)
D 0X804ABD0:   0x00000000        length = (int) 0
D 0X804ABD4:   0x00000000        head = (struct process *) 0x0
                              [3] = (struct queue)
D 0X804ABD8:   0x00000000        length = (int) 0
D 0X804ABDC:   0x00000000        head = (struct process *) 0x0
? 0X804ABE0:   0x00000002
D 0X804ABE4:   0x0804b008   (void *(*)(size_t, const void *)) 0x804b008
D 0X804ABE8:   0x00000000   (void (*)(void *, const void *)) 0
D 0X804ABEC:   0x00749380   (void *(*)(void *, size_t, const void *)) 0x749380 <realloc_hook_ini>
D 0X804ABF0:   0x007493d0   (void *(*)(size_t, size_t, const void *)) 0x7493d0 <memalign_hook_ini>
? 0X804ABF4:   0x0804b018
? 0X804ABF8:   0x00000000
```

Figure 8: Schedule2 global variables visualization

The preceding analysis was conducted without debug constraints, and therefore without `whatsat` having prior knowledge that `prio_queue` should contain only `List` pointers. If debug constraints are included, then the known type of `prio_queue` requires that all pointed-to elements have type `List`. For the bad `prio_queue[i]` pointer, this is incompatible with the `Ele` type required by value and size constraints. `whatsat` detects and reports the conflict. Thus, debug constraints can be especially useful when pointer misuse has broken type correctness without trashing the heap in the manner of a wild pointer bug or buffer overrun.

6.3 Schedule2

Schedule2 is a different implementation of a job scheduler, also part of the Siemens suite [13]. Version 8 of schedule2 contains a bug that causes the program to crash inside `malloc()`. A stack trace reveals that the crash is due to a bad pointer dereference: a function pointer, `__malloc_hook`, does not point to a function. `whatsat` confirms that the claimed type for `__malloc_hook` is inconsistent with its value, and therefore that debug constraints are not satisfiable. After debug constraints are discarded, `whatsat` infers that this block actually contains a `process` structure.

`__malloc_hook` is assigned from `old_malloc_hook`, which holds the same bad `process` pointer instead of a function pointer. Using `whatsat` to explore the physical memory neighborhood around `old_malloc_hook` reveals that a four-element structure array precedes `old_malloc_hook`. Figure 8 shows `whatsat`'s visualization of this area. `old_malloc_hook` appears at address `0X804ABE4`; the preceding array is clearly visible starting at address `0X804ABC0`. Observe that `old_malloc_hook` is perfectly positioned to receive an errant `process` pointer should the neighboring array overrun its bounds. Thus informed, we identify the array, the code that writes to it, and the missing bounds check that constitutes the true bug. All `whatsat` queries used in this case study completed within 0.03 seconds.

While a hardware watchpoint might also have been used to trap the bad write to `old_malloc_hook`, this would require rerunning the program and reproducing the bug. Many memory corruption bugs are difficult to reproduce on demand; not all bugs are amenable to the sort of iterative backtracking used in Section 6.2. We see here that `whatsat` can also provide useful postmortem information on the first instance of a bug.

6.4 Space

Space is an interpreter for an antenna array definition language (ADL) written for the European Space Agency [21, 26]. As distributed by the Galileo Subject Infrastructure Repository [7] it consists of a correct version in 9,564 lines of C code along with buggy variants and an extensive test suite. We ran the correct version on one of its test inputs (`inputs/gr120`) with a debugger breakpoint set at the very end of `main()`. At this point, 174 memory blocks are allocated in the heap. A `whatsat` query completes in 0.8 seconds and finds that the "correct" variant of space contains an untypable block:

```
untypable block of 168 bytes
at 0x805c088, allocated in
elemdef() at space.c:1880
```

Just after this block is allocated, `whatsat` finds no problems and types the block as `Elem`. An informal binary search as suggested in Section 1 reveals that the block is later corrupted by an assignment of an uninitialized local variable into one of its fields. The field is an `int` but the uninitialized value it receives happens to be a valid pointer left behind on the stack by earlier calls. No other type looks like an `Elem` with a pointer in place of this `int` field, so the block is untypable. This "correct" version of space runs correctly only because this improperly initialized field is not actually used by any other code. The bug described here was previously unknown to us and, to our knowledge, not previously reported in any published literature concerning the space test suite.

6.5 Exif

Exif is an open source utility for manipulating JPEG image metadata [9]. It consists of 10,375 lines of C code split into a shared library and a main driver program. We ran exif with a breakpoint set after `exif_loader_get_data()`, which builds an in-memory representation of a JPEG input file. We disabled zero-initialization of heap blocks to test whether exif performs its own initializations properly. `whatsat` identifies two untypable blocks allocated in `exif_content_add_entry()` at `exif-content.c:110`. The code in question performs a reallocation to grow an array of pointers to entry blocks:

```
entries = realloc(entries,
  sizeof(ExifEntry) * (count + 1));
```

The size calculation is incorrect. It reserves space for an array of `ExifEntry` structures, but `entries` is actually an array of *pointers* to `ExifEntry` structures. Because each `ExifEntry` is larger than a pointer, the program does not overrun this buffer. However, the extra space at the end of the array is wasted and, because it contains uninitialized random data that may not look like valid pointers, `whatsat` determines that arrays allocated here are untypable.

We initially identified this previously unreported bug in release 0.6.9 and 0.6.10 of the exif driver and library. We have confirmed that it persists in the latest development snapshot as of April 15, 2006 (12,410 lines of C code). Exif developers have since confirmed the bug and applied our suggested fix.

After `whatsat` identifies these untypable blocks, it ignores them for the remainder of the analysis. That analysis, however, does not find a valid heap typing for exif in a timely manner. It is possible that no valid typing exists even though all individual blocks match at least one known type. It is also possible that a valid typing exists, but is pathologically mismatched with our heuristic search order. Improving the diagnostic capabilities of our analysis when unresolvable conflicts arise late in the search is an important area for future study.

7 Related Work

Chandra and Reps [5] and Siff *et al.* [22] introduce an alternate type system for C that allows subtyping based on the physical layouts of data structures. They describe static type checking and inference rules that test program conformance with this alternate type system. In contrast, our approach is dynamic: we examine a frozen snapshot of a running program's heap, rather than the space of all possible program heaps. This allows us to use concrete

memory values and allocated block sizes to refine our analysis. As is typical for dynamic analyses, we focus on specific bugs triggered during a run without guaranteeing that all possible bugs will be detected.

The subtyping relation induced by our byte type lattice is more restrictive than the Chandra/Siff physical subtyping relation. Both allow subtyping between a structure and its first field, but we disallow more general structure prefixing or the use of `char` arrays as storage placeholders. These are merely policy choices. Our approach can use permissive Chandra/Siff subtyping or a variety of other relations with no changes to constraint collection or the core heap typing algorithm. However, not all subtyping relations are sensible in this context. For example, Cardelli's structural record subtyping relation [4], disregards field order and is therefore needlessly permissive for our scenario, where field orders are fixed.

As a dynamic heap-walking tool, our system shares some qualities with a garbage collector or leak detector, and a list of unreachable (leaked) memory blocks could easily be extracted from our analysis. Traditional garbage collectors require data structure layout information for the root set and possibly for allocated blocks as well. Conservative garbage collectors [3] relax this requirement by assuming that any location holding a valid pointer value is indeed a pointer. Our approach moves flexibly between these extremes. We use type information for global and stack storage if available, but can operate without it by making pointer/value assumptions in the manner of a conservative collector. Ultimately, the information we recover is richer than that produced by garbage collectors: we find not only the size and embedded pointers of each allocated block, but also complete program types that are globally consistent both within and between all blocks.

Zimmermann and Zeller present strategies for extracting C heaps and displaying them to highlight key relationships [27]. Their system depends on debugger-provided type information augmented with a few C-specific heuristics also used by `whatsat`, such as pointer validity testing and dynamic array size computation. These heuristics consider only isolated blocks, though, and have no notion of global consistency. Zimmermann and Zeller comment that "While such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules– either hand-crafted or inferred from the program." Dynamic heap type inference generalizes and improves upon these heuristics by defining a notion of global heap typing that considers not just local values within isolated blocks, but also the relationships between interlinked blocks. This lets `whatsat` find globally consistent heap typings and reduces or eliminates the need for hand-crafted disambiguation rules. We also note that the visualization of Zimmermann and Zeller abstracts away physical block locations in favor of box-and-arrow diagrams, whereas `whatsat`'s visualization focuses on physical layout and proximity. Both representations may be of interest, depending on the debugging task at hand.

The problem of heap corruption due to pointer and cast abuse is longstanding, and has inspired solutions ranging from static analysis [8, 19] to run-time instrumentation [12, 16, 17, 18, 20, 25] and the design of safer language dialects [2, 15, 23]. Our approach performs programmer-directed heap validity checks in an interactive debugging context, and does not attempt to prevent or trap errors as they occur. This allows us to be significantly less invasive: we require no changes to the C language; no recompilation or source annotation beyond a compiler-provided list of program types; no run-time instrumentation beyond a list of allocated blocks; and no dynamic type tagging or other changes to data structure layouts. Additionally, our analysis depends only on the instantaneous state of the program heap at a given moment in time: other than maintaining a list of currently allocated blocks, we do not record any trace information while the program runs.

In this sense `whatsat` can be seen as an experiment in minimalism. Rather than monitor every potentially interesting action, we ask how much information can be recovered with only the bare minimum imposition at run-time. We believe that both highly invasive and minimally invasive approaches have benefits. Exploring the extremes helps illuminate potential strategies to improve debugging tools all along the instrumentation and analysis spectrum.

8 Future Work and Conclusion

Extensions to our work are possible both for improved efficiency as well as enhanced user experience. Backtracking can be reduced by treating blocks and pointers as graph nodes and edges, and traversing strongly connected components of the heap graph in topological order. Richer error reporting could include a detailed trace of the constraint conflicts surrounding untypable blocks; we expect this would be a valuable diagnostic aid. Static program information, such as the types of casts that immediately follow most `malloc()` calls, can be treated as an additional source of constraints or as an independent "second opinion" with which dynamically observed types should (but may not) agree. The general algorithm can accommodate a variety of subtyping policy choices and application-specific consistency constraints; provisions for end-user customization and extension of the analysis should allow the tool to be more helpful for a wider variety of programs and programming styles.

Low-level programming languages sometimes require low-level debugging. However, one need not completely abandon the type system even when working with non-type-safe languages. A low-level but type-annotated view of the heap can help in debugging and more general program understanding tasks. We have presented an algorithm that infers program-defined types for memory locations. Solution consistency is defined in terms of constraints that use a novel blend of ideas from physical subtyping and conservative garbage collection. When no consistent typing exists due to heap corruption or pointer abuse, we offer focused diagnostic information to help identify the cause. Our implementation works for general C programs and requires no source annotation, no recompilation, no run-time instrumentation beyond heap allocation tracking, and no changes to physical data structure layouts. Experiences with the tool, while limited in scope, suggest that dynamic heap type inference may be a useful addition to the programmer's toolkit.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 290–301, New York, NY, USA, 1994. ACM Press.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice & Experience*, 18(9):807–820, 1988.

[4] Luca Cardelli. Structural subtyping and the notion of power type. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–79, New York, NY, USA, 1988. ACM Press.

[5] Satish Chandra and Thomas W. Reps. Physical type checking for C. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 66–75, 1999.

[6] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, Portland, ME, USA, July 18–20 2006.

[7] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[8] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press.

[9] EXIF tag parsing library. `http://libexif.sf.net/`.

[10] Free Software Foundation, Inc., Boston, MA, USA. *The GNU C Library*, 0.10 edition, July 6 2001.

[11] John Gilmore and Stan Shebs. *GDB Internals*, February 2004.

[12] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Conference*, pages 125–138, San Francisco, CA, USA, 1992. USENIX Association.

[13] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, May 1994.

[14] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999.

[15] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[16] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, pages 13–26, 1997.

[17] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C: An interpreter-based programming environment for the C language. In *Proceedings of the USENIX Summer Conference*, pages 161–171, San Francisco, CA, USA, June 1988. USENIX Association.

[18] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Debugging via run-time type checking. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 217–232, London, UK, 2001. Springer-Verlag.

[19] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[20] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[21] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[22] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas W. Reps. Coping with type casts in C. In Oscar Nierstrasz and M. Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 180–198. Springer, 1999.

[23] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1-3):49–72, 1998.

[24] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[25] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice & Experience*, 22(4):305–316, 1992.

[26] Filippos I. Vokolos and Phyllis G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 44, Washington, DC, USA, 1998. IEEE Computer Society.

[27] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20–25, 2001, Revised Lectures*, volume 2269 of *Lecture Notes in Computer Science*, pages 191–204. Springer, May 2001.