Computer Sciences Department

Recency-Abstraction for Heap-Allocated Storage

Gogul Balakrishnan
Thomas Reps

Technical Report #1548

December 2005

UNIVERSITY OF WISCONSIN MADISON

Recency-Abstraction for Heap-Allocated Storage

Gogul Balakrishnan and Thomas Reps

Comp. Sci. Dept., University of Wisconsin; {bgogul, reps}@cs.wisc.edu

Abstract. In this paper, we present an abstraction for heap-allocated storage, called the *recency-abstraction*, that allows abstract-interpretation algorithms to recover non-trivial information for heap-allocated data objects. As an application of the recency-abstraction, we show how it can resolve virtual-function calls in stripped executables (i.e., executables from which debugging information has been removed).

1 Introduction

A great deal of work has been done on algorithms for flow-insensitive points-to analysis [1, 24, 7] (including algorithms that exhibit varying degrees of context-sensitivity [9, 6, 10, 25]). However, all of the aforementioned work uses a very simple abstraction of heap-allocated storage, which we call the *allocation-site abstraction* [17, 4]:

> *All of the nodes allocated at a given allocation site s are folded together into a single summary node n_s.*

In terms of precision, the allocation-site abstraction can often produce poor-quality information because it does not allow strong updates to be performed. A strong update overwrites the contents of an abstract object, and represents a definite change in value to all concrete objects that the abstract object represents [4, 22]. Strong updates cannot generally be performed on summary objects because a (concrete) update usually affects only one of the summarized concrete objects. If allocation site s is in a loop, or in a function that is called more than once, then s can allocate multiple nodes with different addresses. A points-to fact "p points to n_s" means that program variable p may point to *one* of the nodes that n_s represents. For an assignment of the form `p->selector1 = q`, points-to-analysis algorithms are ordinarily forced to perform a weak update: that is, selector edges emanating from the nodes that p points to are *accumulated*; the abstract execution of an assignment to a field of a summary node cannot kill the effects of a previous assignment because, in general, only *one* of the nodes that n_s represents is updated on each concrete execution of the assignment statement. Because imprecisions snowball as additional weak updates are performed (e.g., for assignments of the form `r->selector1 = p->selector2`), the use of weak updates has adverse effects on what a points-to-analysis algorithm can determine about the properties of heap-allocated data structures.

To mitigate the effects of weak updates, many pointer-analysis algorithms in the literature side-step the issue of soundness. For instance, when performing flow-sensitive pointer analysis, the initial points-to sets for each pointer variable is assumed to be $\emptyset$ (rather than $\top$). For local variables and malloc-site variables, the assumption that the initial value is $\emptyset$ is not a safe one—it does not over-approximate all of the program's behaviors. The program shown in Fig. 1 illustrates this issue. In Fig. 1(a), `*pp` is not initialized on all paths leading to

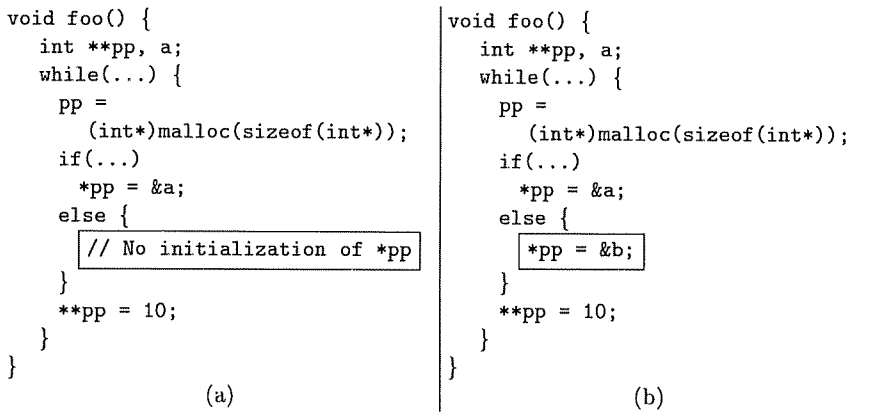


Fig. 1. Weak update problem for malloc blocks.

"**pp = 10", whereas in Fig. 1(b), *pp is initialized on all paths leading to "**pp = 10".

A pointer-analysis algorithm that makes the unsafe assumption mentioned above will not be able to detect that the malloc-block pointed to by pp is possibly uninitialized at the dereference **pp. For Fig. 1(b), the algorithm concludes correctly that "**pp = 10" modifies either a or b. But, for Fig. 1(a), the algorithm concludes incorrectly that "**pp = 10" only modifies a, which is not sound.

On the other hand, assuming that the malloc-block can point to any variable immediately after the call to `malloc` leads to sound but imprecise points-to sets in both versions of the program in Fig. 1. The problem is as follows. When the pointer-analysis algorithm interprets statements "*pp = &a" and "*pp = &b", it performs a weak update. Because *pp is assumed to point to any variable, doing a weak update does not improve the points-to sets for *pp. Therefore, the algorithm concludes that "**pp = 10" may modify any variable in the program.

Even the use of multiple summary nodes per allocation site, where each summary node is qualified by some amount of calling context (as in [19, 13]), does not overcome the problem; that is, algorithms such as [19, 13] must still perform weak updates.

At the other extreme is a family of heap abstractions that have been introduced to discover information about the possible shapes of the heap-allocated data structures to which a program's pointer variables can point [22]. Those abstractions generally allow strong updates to be performed, and are capable of providing very precise characterizations of programs that manipulate linked data structures; however, the methods are also very costly in space and time.

The inability to perform strong updates not only causes less precise points-to information to be obtained for pointer-valued fields, it also causes less precise numeric information to be obtained for `int`-valued fields. For instance, with interval analysis (an abstract interpretation that determines an interval for each variable that over-approximates the variable's set of values) when weak updates must be applied to `int`-valued fields, wider intervals are obtained.

In this paper, we present an abstraction for heap-allocated storage, referred to as the *recency-abstraction*, that is somewhere in the middle between the extremes of one summary node per malloc site [1, 24, 7] and complex shape abstractions [22]. The recency-abstraction enables strong updates to be performed in many cases.

The specific technical contributions of the paper are as follows:

- We propose an inexpensive abstraction for heap-allocated data structures that allows us to obtain some useful results for objects allocated in the heap.
- We show the effectiveness of the abstraction in a particularly challenging context: We measure how well it resolves virtual-function calls in x86 executables obtained from C++ code. In particular, for allocation sites that arise because the source code contains a call `new C`, where `C` is a class that has virtual methods, the recency-abstraction generally permits our tool to recover information about virtual-function tables. Using the recency-abstraction, our tool was able to resolve a large fraction of virtual-function calls accurately. Existing tools such as IDAPro [15] resolved none of the virtual-function calls in our set of examples.

The remainder of the paper is organized as follows: §2 provides background on the issues that arise when resolving virtual-function calls in executables. §3 describes our recency-abstraction for heap-allocated data structures. §4 provides experimental results evaluating these techniques. §5 discusses related work.

2 Resolving Virtual-Function Calls in Executables

In recent years, there is an increasing need for tools to help programmers and security analysts understand executables. For instance, commercial companies and the military increasingly use Commercial Off-The Shelf (COTS) components to reduce the cost of software development. They are interested in ensuring that COTS components do not perform malicious actions (or cannot be forced to perform malicious actions). Therefore, resolving virtual-function calls in executables is important: (1) as a code-understanding aid to analysts who examine executables, and (2) for recovering Intermediate Representations (IR) so that additional analyses can be performed on the recovered IR (*à la* Engler et al. [8], Chen and Wagner [5], etc.). Poor information about virtual-function calls can lead to portions of the program's state space not being explored, which is a source of false positives. In this section, we discuss the issues that arise when trying to resolve virtual-function calls in executables.

Consider an executable compiled from a C++ program that uses inheritance and virtual functions. The first 4 bytes of an object contains the address of the virtual-function table. We will refer to these 4 bytes as the virtual-function pointer. In an executable, a call to `new` results in two operations: (1) a call to `malloc` to allocate memory, and (2) a call to the constructor to initialize (among other things) the virtual-function pointer. A virtual-function call in source code gets translated to an indirect call through the virtual-function pointer (see Fig. 2).

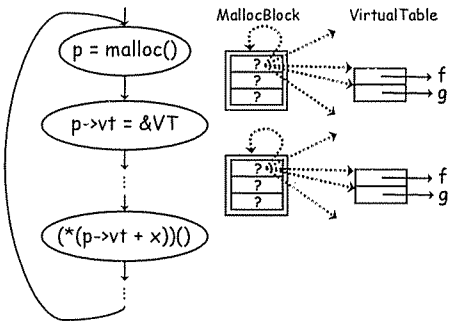


Fig. 2. Resolving virtual-function calls in executables.

When source code is available, one way of resolving virtual-function calls is to associate type information with the pointer returned by the call to `new` and then propagate that information to other pointers at assignment statements. However, type information is usually not available in executables. Therefore, to resolve a virtual-function call, information about the contents of the virtual-function pointer needs to be available. For a static-analysis algorithm to determine such information, it has to track the flow of information through the instructions in the constructor. Fig. 2 illustrates the results if the allocation-site abstraction is used. Using the allocation-site abstraction alone, it would not be possible to establish the link between the object and the virtual-function table: because the summary node represents more than one block, the interpretation of the instruction that sets the virtual-function pointer can only perform a weak update, i.e., it can only join the virtual-function table address with the existing addresses, and not overwrite the virtual-function pointer in the object with the address of the virtual-function table. After the call to malloc, the fields of the object can have any value (shown as ?); computing the join of ? with any value results in ?, which means that the virtual-function pointer can point to anywhere in memory (shown as dashed arrows). Therefore, a definite link between the object and the virtual-function table is never established, and a client of the analysis can only conclude that the virtual-function call may resolve to any possible function.

The key to resolving virtual-function calls in executables is to be able to establish that the virtual-function pointer definitely points to a certain virtual-function table. §2.1 describes the abstract domain used in Value-Set Analysis (VSA) [2], a combined pointer-analysis and numeric-analysis algorithm that can track the flow of data in an executable. The version of the VSA domain described in §2.1 (the version used in [2]) has the limitations discussed above (i.e., the need to perform weak updates); §3 describes an extension of the VSA domain that uses the recency-abstraction, and shows how it is able to establish a definite link between an object's virtual-function pointer and the appropriate virtual-function table in many circumstances.

2.1 Value-Set Analysis

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values or addresses that each variable holds at each program point. A key feature of VSA is that it takes into account pointer arithmetic operations and tracks integer-valued and address-valued quantities simultaneously. This is crucial for analyzing executables because numeric values and addresses are indistinguishable at runtime and pointer arithmetic is used extensively in executables. During VSA, a set of addresses and numeric values is represented by a safe approximation, which we refer to as a *value-set.*

Memory-Regions In the runtime address space, there is no separation of the activation records of various procedures, the heap, and the memory for global data. However, during the analysis of an executable, we break the address space into a set of disjoint memory areas, which are referred to as *memory-regions.* Each memory-region represents a group of locations that have similar runtime properties. For example, the runtime locations that belong to the activation record of the same procedure belong to a memory-region. For a given program, there are three kinds of regions: (1) the *global*-region that contains information about locations that correspond to global data, (2) the *AR*-regions that contain information about locations that corresponds to the activation-record of a particular procedure, and (3) the *malloc*-regions that contain information about locations that are allocated at a particular malloc site. We assume that the structure of each region is available either from debugging information or the structure-discovery mechanism described in [3]. We also treat each field of a struct as a separate variable. That is, a procedure with local variables

```
int a;
struct {
  int b;
  int c;
} d;
```

would be treated as having three int-valued variables `a`, `d.b`, and `d.c`. This allows us to treat fields of malloc-regions that hold a struct-valued quantity in the same way as variables in AR-regions. (For this reason, we use the term "variable" when referring to components of a malloc-region, and this is why the component of an AbsEnv (see below) that holds information about malloc-regions is of type $\text{AllocMemRgn} \rightarrow \text{VarEnv}_{\perp}$).

Value-Set A value-set is a safe approximation for a set of addresses and numeric values. Suppose that n is the number of regions in the executable. A value-set is an n-tuple of strided intervals of the form $s[l, u]$, with each component of the tuple representing the set of addresses in the corresponding region. For a 32-bit machine, a strided-interval $s[l, u]$ represents the set of integers $\{i \in [-2^{31}, 2^{31} - 1] | l \leq i \leq u, i \equiv l (\texttt{mod } s)\}$.

- s is called the *stride.*
- $[l, u]$ is called the *interval.*

– $0[l, l]$ represents the singleton set $\{l\}$.

VSA is a flow-sensitive, context-sensitive, abstract-interpretation algorithm (parameterized by call-string length [23]) that is based on an independent-attribute domain described below. To simplify the presentation, the discussion in this section uses the allocation-site abstraction for heap-allocated storage.

Let Proc denote the set of memory-regions associated with procedures in the program, AllocMemRgn denotes the set of memory regions associated with heap-allocation sites, and Global denote the memory-region associated with the global data area. We work with the following basic domains:

$$\begin{aligned} \text{MemRgn} &= \{\text{Global}\} \cup \text{Proc} \cup \text{AllocMemRgn} \\ \text{ValueSet} &= \text{MemRgn} \rightarrow \text{StridedInterval}_{\perp} \\ \text{VarEnv} &= \text{Var} \rightarrow \text{ValueSet}_{\perp} \end{aligned}$$

AbsEnv maps each region to its corresponding VarEnv and each register to a ValueSet:

$$\text{AbsEnv} = \begin{array}{l} (\text{register} \rightarrow \text{ValueSet}) \\ \times\ (\{\text{Global}\} \rightarrow \text{VarEnv}_{\perp}) \\ \times\ (\text{Proc} \rightarrow \text{VarEnv}_{\perp}) \\ \times\ (\text{AllocMemRgn} \rightarrow \text{VarEnv}_{\perp}) \end{array}$$

VSA associates each program point with an AbsMemConfig:

$$\text{AbsMemConfig} = (\text{CallString} \rightarrow \text{AbsEnv}_{\perp})$$

Example 1. We will illustrate VSA using the C program shown in Fig. 3(a). (In our implementation, VSA is applied to executables. We use C code for ease of understanding.) For this example, there would be three regions: `Global`, `AR_main`, and `malloc_M1`.

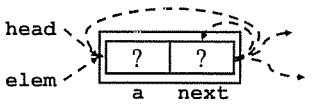
```
struct List {
  int a;
  struct List* next;
};

int main() {
  int i;
  List* head = NULL;
  for(i = 0; i < 5; ++i) {
    M1: List* elem =
          (List*)malloc(sizeof(List));
    elem->a = i;
    elem->next = head;
    head = elem;
  }
  return 0;
}
```

(a)

```
i → [(Global ↦ 1[0,4])]
head → [(malloc_M1 ↦ 0[0,0])]
elem → [(malloc_M1 ↦ 0[0,0])]
elem->a → ⊤
elem->next → ⊤
```

(b)



(c)

Fig. 3. Value-Set Analysis (VSA) example: (a) C program, (b) Value-sets at `M1`, and (c) Points-to information recovered by VSA (when the allocation-site abstraction is used) at the end of the loop (i.e., just after "`head = elem;`").

The value-sets that are obtained from VSA at the bottom of the loop body are shown in Fig. 3(b).

- "i → [(Global ↦ 1[0,4])]" indicates that i has a value (or a global address) in the range [0, 4].
- "elem → [(malloc_M1 ↦ 0[0,0])]" indicates that elem contains offset 0 in the malloc-region associated with malloc-site M1.
- "head → [(malloc_M1 ↦ 0[0,0])]" indicates that head contains offset 0 in the malloc-region associated with malloc-site M1.
- "elem->a → ⊤" and "elem->next → ⊤" indicate that elem->a and elem->next may contain any possible value. VSA could not determine better value-sets for these variables because of the weak-update problem mentioned earlier. Because malloc does not initialize the block of memory it returns, VSA assumes (safely) that elem->a and elem->next may contain any possible value after the call to malloc. Because malloc_M1 is a summary memory-region, only weak updates can be performed at the instructions that initialize the fields of elem. Therefore, the value-sets associated with the fields of elem remain ⊤.

Fig. 3(c) shows the information pictorially. The double box denotes a summary object. Dashed edges denote may-points-to information. In our example, VSA has recovered the following: (1) head and elem may point to at least one of the objects represented by the summary object, (2) "elem->next" may point to any possible location, and (3) "elem->a" may contain any possible value. □

3 An Abstraction for Heap-Allocated Storage

This section describes the *recency-abstraction*. The recency-abstraction is similar in some respects to the allocation-site abstraction, in that each abstract node is associated with a particular allocation site; however, the recency-abstraction uses two memory-regions per allocation site s:

AllocMemRgn = {MRAB[s], NMRAB[s] | s an allocation site}

- MRAB[s] represents the **m**ost-**r**ecently-**a**llocated **b**lock that was allocated at s. Because there is at most one such block in any concrete configuration, MRAB[s] is *never* a summary memory-region.
- NMRAB[s] represents the **n**on-**m**ost-**r**ecently-**a**llocated **b**locks that were allocated at s. Because there can be many such blocks in a given concrete configuration, NMRAB[s] is generally a summary memory-region.

In addition, each MRAB[s], NMRAB[s] ∈ AllocMemRgn is associated with a "count" value, denoted by MRAB[s].count and NMRAB[s].count, respectively, which is a value of type SmallRange = {[0, 0], [0, 1], [1, 1], [0, ∞], [1, ∞], [2, ∞]}. The count value records a range for how many concrete blocks the memory-region represents. While NMRAB[s].count can have any SmallRange value, MRAB[s].count will be restricted to take on only values in {[0, 0], [0, 1], [1, 1]}, which represent counts for non-summary regions. Consequently, an abstract transformer can perform a strong update on a field of MRAB[s].

In addition to the count, each MRAB[s], NMRAB[s] ∈ AllocMemRgn is also associated with a "size" value, denoted by MRAB[s].size and NMRAB[s].size, respectively, which is a value of type StridedInterval. The size value represents an over-approximation of the set of sizes of the concrete blocks that the memory-

region represents. This information can be used to report potential memory-access violations that involve heap-allocated data.

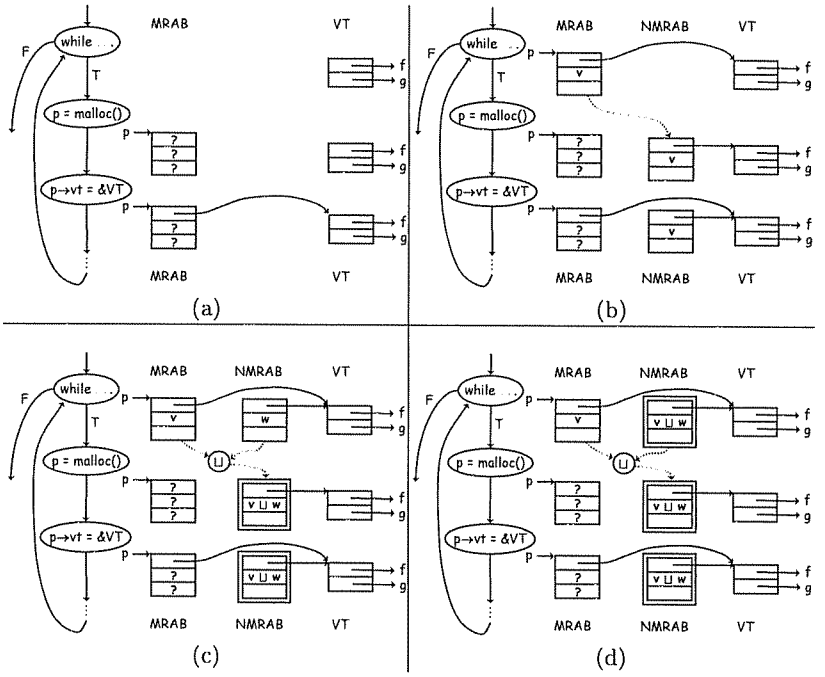


Fig. 4. A trace of the evolution of parts of the AbsEnvs for three instructions in a loop.

Example 2. Fig. 4 shows a trace of the evolution of parts of the AbsEnvs for three instructions in a loop during VSA. It is assumed that there are three fields in the memory-regions MRAB and NMRAB (shown as the three rectangles within MRAB and NMRAB). Double boxes around NMRAB objects in Fig. 4(c) and (d) are used to indicate that they are summary memory-regions.

For brevity, in Fig. 4 the effect of each instruction is denoted using C syntax; the original source code in the loop body contains a C++ statement "`p = new C`", where `C` is a class that has virtual methods `f` and `g`. The symbols `f` and `g` that appear in Fig. 4 represent the addresses of methods `f` and `g`. The symbol `p` and the two fields of `VT` represent variables of the Global region. The dotted lines in Fig. 4(b)–(d) indicate how the value of NMRAB after the `malloc` statement depends on the value of MRAB and NMRAB before the `malloc` statement.

The AbsEnvs stabilize after four iterations. Note that in each of Fig. 4(a)–(d), it can be established that the instruction "`p->vt = &VT`" modifies exactly one field in a non-summary memory-region, and hence a strong update can be performed on `p->vt`. This establishes a definite link between MRAB and VT—and hence between NMRAB and VT. □

Example 3. Fig. 5 shows the improved VSA information recovered for the program from Fig. 3 at the end of the loop when the recency-abstraction is used. In particular, we have the following information:

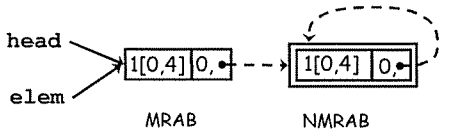


Fig. 5. Improved VSA information for the program from Fig. 3 at the end of the loop (i.e., just after "`head = elem;`") when the recency-abstraction is used. (The double box denote a summary region. Dashed edges denote may-points-to information.)

- `elem` and `head` definitely point to the beginning of the MRAB region.
- `elem->a` contains the values (or global addresses) $\{0, 1, 2, 3, 4\}$.
- `elem->next` may be 0 (NULL) or may point to the beginning of the NMRAB region.
- NMRAB.a contains the values (or global addresses) $\{0, 1, 2, 3, 4\}$.
- NMRAB.next may be 0 (NULL) or may point to the beginning of the NMRAB region.

□

This idea is formalized with the following basic domains (where underlining indicates differences from the domains given in §2):

$$\begin{aligned}
\text{MemRgn} &= \{\text{Global}\} \cup \text{Proc} \cup \text{AllocMemRgn} \\
\text{ValueSet} &= \text{MemRgn} \to \text{StridedInterval}_\perp \\
\text{VarEnv} &= \text{Var} \to \text{ValueSet}_\perp \\
\underline{\text{SmallRange}} &= \underline{\{[0,0],[0,1],[1,1],[0,\infty],[1,\infty],[2,\infty]\}} \\
\underline{\text{AllocAbsEnv}} &= \underline{(\text{SmallRange} \times \text{StridedInterval} \times \text{VarEnv})_\perp}
\end{aligned}$$

The analysis associates each program point with an AbsMemConfig:

$$\begin{aligned}
\text{AbsEnv} = {} & (\text{register} \to \text{ValueSet}) \\
& \times (\{\text{Global}\} \to \text{VarEnv}_\perp) \\
& \times (\text{Proc} \to \text{VarEnv}_\perp) \\
& \times (\text{AllocMemRgn} \to \underline{\text{AllocAbsEnv}}) \\
\text{AbsMemConfig} = {} & (\text{CallString} \to \text{AbsEnv}_\perp)
\end{aligned}$$

Let count, size, and varEnv, respectively, denote the SmallRange, StridedInterval, and VarEnv associated with a given AllocMemRgn. A given absEnv $\in$ AbsEnv maps allocation memory-regions, such as MRAB[s] or NMRAB[s], to $\langle$count, size, varEnv$\rangle$ triples.

The transformers for various operations are defined as follows:

- At the entry point of the program, the AbsMemConfig that describes the initial state records that, for each allocation site s, the VarEnvs for both MRAB[s] and NMRAB[s] are $\perp$.
- The transformer for allocation site s transforms absEnv to absEnv$'$, where absEnv$'$ is identical to absEnv, except that all ValueSets of absEnv that contain $[..., \text{MRAB}[s] \mapsto si_1, \text{NMRAB}[s] \mapsto si_2, ...]$ become $[..., \emptyset, \text{NMRAB}[s] \mapsto si_1 \sqcup si_2, ...]$ in absEnv$'$. In x86 code, return values are passed back in register `eax`. Let *size* denote the size of the block allocated at the allocation site. The value of *size* is obtained from the value-set associated with the parameter of the allocation method. In addition, absEnv$'$ is updated on the following arguments:

$$\begin{aligned}
\text{absEnv}'(\text{MRAB}[s]) &= \langle [0,1], \mathit{size}, \lambda \text{var}.\top_{\text{ValueSet}} \rangle \\
\text{absEnv}'(\text{NMRAB}[s]).\text{count} &= \begin{array}{l} \text{absEnv}(\text{NMRAB}[s]).\text{count} \\ + \text{absEnv}(\text{MRAB}[s]).\text{count} \end{array} \\
\text{absEnv}'(\text{NMRAB}[s]).\text{size} &= \begin{array}{l} \text{absEnv}(\text{NMRAB}[s]).\text{size} \\ \sqcup \text{absEnv}(\text{MRAB}[s]).\text{size} \end{array} \\
\text{absEnv}'(\text{NMRAB}[s]).\text{varEnv} &= \begin{array}{l} \text{absEnv}(\text{NMRAB}[s]).\text{varEnv} \\ \sqcup \text{absEnv}(\text{MRAB}[s]).\text{varEnv} \end{array} \\
\text{absEnv}'(\texttt{eax}) &= [(\text{MRAB}[s] \mapsto 0[0,0]), (\texttt{Global} \mapsto 0[0,0])]
\end{aligned}$$

In the present implementation, we assume that an allocation always succeeds; hence, in place of the first and last lines above, we use

$$\begin{aligned}
\text{absEnv}'(\text{MRAB}[s]) &= \langle [1,1], \mathit{size}, \lambda \text{var}.\top_{\text{ValueSet}} \rangle. \\
\text{absEnv}'(\texttt{eax}) &= [(\text{MRAB}[s] \mapsto 0[0,0])]
\end{aligned}$$

Consequently, the analysis only explores the behavior of the system on executions in which allocations always succeed.

- The join $\text{absEnv}_1 \sqcup \text{absEnv}_2$ of $\text{absEnv}_1, \text{absEnv}_2 \in \text{AbsEnv}$ is performed pointwise; in particular,

$$\begin{aligned}
\text{absEnv}'(\text{MRAB}[s]) &= \begin{array}{l} \text{absEnv}_1(\text{MRAB}[s]) \\ \sqcup \text{absEnv}_2(\text{MRAB}[s]) \end{array} \\
\text{absEnv}'(\text{NMRAB}[s]) &= \begin{array}{l} \text{absEnv}_1(\text{NMRAB}[s]) \\ \sqcup \text{absEnv}_2(\text{NMRAB}[s]) \end{array}
\end{aligned}$$

In all other abstract transformers (e.g., assignments, data movements, interpretation of conditions, etc.), MRAB[s] and NMRAB[s] are treated just like other memory regions—i.e., Global and the AR-regions—with one exception:

- During VSA, all abstract transformers are passed a *memory-region status map* that indicates which memory-regions, in the context of a given call-string suffix cs, are summary memory-regions. Whereas the Global region is always non-summary, to decide whether a procedure P's memory-region is a summary memory-region, first call-string cs is traversed, and then the call graph is traversed, to see whether the runtime stack could contain multiple pending activation records for P.

  The summary-status information for MRAB[s] and NMRAB[s] is obtained differently—from the values of AbsMemConfig(cs)(MRAB[s]).count and AbsMemConfig(cs)(NMRAB[s]).count, respectively.

The memory-region status map provides one of two pieces of information used to identify when a strong update can be performed. In particular, an abstract transformer can perform a strong update if the operation modifies (a) exactly one register, or (b) exactly one variable in a non-summary memory-region.

4 Experiments

This section describes the results of our preliminary experiments. Tab. 1 shows the characteristics of the set of examples that we used in our evaluation. These programs were originally used by Pande and Ryder in [20] to evaluate their algorithm for resolving virtual-function calls in C++ programs. The programs

in C++ were compiled without optimization[1] using the Microsoft Visual Studio 6.0 compiler and the .obj files obtained from the compiler were analyzed. The column labeled "Coverage" is discussed below.

	x86 Insts	Procs	Indirect Calls	Coverage (%)	Time (s)
NP	252	5	6	98.99	1
primes	294	9	2	75.79	<1
family	351	9	3	98.71	1
vcirc	407	14	5	78.23	<1
fsm	502	13	1	87.57	5
office	592	22	4	60.34	<1
trees	1299	29	3	68.08	9
deriv1	1369	38	18	56.11	4
chess	1662	41	1	84.21	16
objects	1739	47	23	37.12	2
simul	1920	60	3	22.84	6
greed	1945	47	17	72.87	10
shapes	1955	39	12	64.38	10
ocean	2552	61	5	44.65	17
deriv2	2639	41	56	32.26	2

Table 1. Characteristics of the example programs. "Coverage" denotes the part of the program that is reachable during analysis (see §4).

Tab. 2 shows the results of applying VSA to resolve virtual-function calls. The column labeled $\bot$ shows the number of unreachable indirect call-sites. The column labeled $\top$ denotes the number of reachable indirect call-sites at which VSA could not determine the targets. The other columns show the distribution of the number of targets at the indirect call-sites. For example, the column labelled 1 denotes the number of indirect-call sites that had a single target.

A non-zero value in the $\top$-column means that at some indirect calls VSA could only determine that the virtual-function call could resolve to any procedure. VSA reports such call-sites to the user, but does not explore any procedures from that call-site. The coverage column in Tab. 1 shows the fraction of the executable code that was analyzed during VSA. Note that for programs for which the $\top$-column is 0, the coverage reported in Tab. 1 is an over-approximation of the actual runtime coverage for all possible runs of the executable. However, for programs with non-zero values in the $\top$-column, some parts of the program may not have been explored. For programs with a 0 value in the $\top$-column, the number of calls reported as unreachable are ones that are definitely unreachable. For instance, the eight calls that are flagged as unreachable in `deriv1` are definitely unreachable (see the $\bot$-column).

We see that our algorithm resolves virtual-function calls for most of the call-sites. It is important to realize that all these virtual-function calls are resolved solely by tracking the flow of data through memory (including the heap). The

[1] Note that unoptimized programs generally have more memory accesses than optimized programs; optimized programs make more use of registers, which are easier to analyze than memory accesses. Thus, for static analysis of stripped executables, unoptimized programs represent a *greater* challenge than optimized programs.

	$\bot$	1	2	≥ 3	$\top$
NP	0	0	6	0	0
primes	1	1	0	0	1
family	0	3	0	0	0
vcirc	0	5	0	0	0
fsm	0	1	0	0	0
office	0	4	0	0	0
trees	1	0	0	1	2
deriv1	8	8	2	0	0
chess	0	0	0	0	1
objects	18	0	4	0	1
simul	2	0	0	0	1
greed	6	10	0	0	1
shapes	4	4	3	0	1
ocean	3	0	0	0	2
deriv2	33	22	0	0	1

Table 2. Distribution of the number of callees at each indirect call.

analysis algorithm does not rely on symbol-table or debugging information; instead it uses the structure-discovery mechanism described in [3]. Existing tools for analyzing executables, such as IDAPro, resolve none of the virtual-function calls.

VSA could not resolve other indirect call-sites mostly because it could not establish that all the elements of an array are definitely initialized in a loop. The problem is as follows. In some of the example programs, an array of pointers to objects is initialized via a loop. These pointers are later used to invoke a virtual-function call. Even if VSA were successful in establishing the link between the virtual-function pointer and virtual-function table, VSA could not establish that all elements of the array are definitely initialized by the instruction in the loop. Hence, the values of the pointers in the array remain $\top$. Note that this issue is orthogonal to the problem at hand. Even if one were to use other mechanisms (such as the one described in [12]) to establish that all the elements of an array are initialized, the problem of establishing the link between the virtual-function pointer and the virtual-function table still requires mechanisms similar to the recency-abstraction.

We do not directly compare against the results from [20], because it would not be a fair comparison: [20] makes an unsafe assumption that elements in a array of pointers (say, locally allocated or heap allocated) initially point to nothing ($\emptyset$), rather than to anything ($\top$). Suppose that `p[]` is such an array of pointers and that a loop initializes every other element with `&a`. A sound result would be that `p`'s elements can point to anything. However, because in the algorithm used in [20] the points-to set of `p` is initially $\emptyset$, [20] would determine that `p`'s elements point to `a`, which is unsound.

5 Related Work

Other work on pointer analyses The recency-abstraction is similar in flavor to the allocation-site abstraction [17, 4], in that each abstract node is associated

with a particular allocation site; however, the recency-abstraction is designed to take advantage of the fact that VSA is a flow-sensitive, context-sensitive algorithm. Note that if the recency-abstraction were used with a flow-insensitive algorithm, it would provide little additional precision over the allocation-site abstraction: because a flow-insensitive algorithm has just one abstract memory configuration that expresses a *program-wide* invariant, the algorithm would have to perform weak updates for assignments to MRAB nodes (as well as for assignments to NMRAB nodes); that is, edges emanating from an MRAB node would also have to be *accumulated*.

With a flow-sensitive algorithm, the recency-abstraction uses twice as many abstract nodes as the allocation-site abstraction, but under certain conditions it is sound for the algorithm to perform strong updates for assignments to MRAB nodes, which is crucial to being able to establish a definite link between the set of objects allocated at a certain site and a particular virtual-function table.

If one ignores actual addresses of allocated objects and adopts the fiction that each allocation site generates objects that are independent of those produced at any other allocation site, another difference between the recency-abstraction and the allocation-site abstraction comes to light:

- The allocation-site abstraction imposes a *fixed partition* on the set of allocated nodes.
- The recency-abstraction shares the "multiple-partition" property that one sees in the shape-analysis abstractions of [22]. An MRAB node represents a *unique* node in any given concrete memory configuration—namely, the most recently allocated node at the allocation site. In general, however, an abstract memory configuration represents multiple concrete memory configurations, and a given MRAB node generally represents different concrete nodes in the different concrete memory configurations.

Hackett and Rugina [14] describe a method that uses local reasoning about individual heap locations, rather than global reasoning about entire heap abstractions. In essence, they use an independent-attribute abstraction: each "tracked location" is tracked independently of other locations in concrete memory configurations. The recency-abstraction is a different independent-attribute abstraction.

The use of count information on (N)MRAB nodes was inspired by the heap abstraction of Yavuz-Kahveci and Bultan [26], which also attaches numeric information to summary nodes to characterize the number of concrete nodes represented. The information on summary node u of abstract memory configuration S describes the number of concrete nodes that are mapped to u in any concrete memory configuration that S represents. Gopan et al. [11] also attach numeric information to summary nodes; however, such information does not provide a characterization of the number of concrete nodes represented: in both the present paper and [26], each concrete node that is combined into a summary node contributes 1 to a *sum* that labels the summary node; in contrast, when concrete nodes are combined together in the approach presented in [11], the effect is to create a *set* of values (to which an additional numeric abstraction may then be applied).

The size information on (N)MRAB nodes can be thought of as an abstraction of auxiliary size information attached to each concrete node, where the concrete size information is abstracted in the style of [11].

Strictly speaking, the use of counts on abstract heap nodes lies outside the framework of [22] for program analysis using 3-valued logic (unless the framework were to be extended with counting quantifiers [16, Sect. 12.3]). However, the use of counts is also related to the notion of active/inactive individuals in logical structures [21], which has been used in the 3-valued logic framework to give a more compact representation of logical structures [18, Chap. 7]. In general, the use of an independent-attribute method in the heap abstraction described in §3 provides a way to avoid the combinatorial explosion that the 3-valued logic framework suffers from: the 3-valued logic framework retains the use of separate logical structures for different combinations of present/absent nodes, whereas counts permit them to be combined.

References

1. L. O. Andersen. Binding-time analysis and the taming of C pointers. In *Part. Eval. and Semantics-Based Prog. Manip.*, pages 47–58, 1993.
2. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
3. G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. Tech. Rep. 1533, Comp. Sci. Dept., Univ. of Wisconsin, Madison, US., September 2005.
4. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.
5. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, pages 235–244, November 2002.
6. B.-C. Cheng and W.W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.
7. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.
8. D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Op. Syst. Design and Impl.*, pages 1–16, 2000.
9. M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, 2000.
10. J.S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS*, 2000.
11. D. Gopan, F. DiMaio, N.Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algs. for the Construct. and Anal. of Syst.*, pages 512–529, 2004.
12. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
13. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *3nd IEEE/ACM Int. Symp. on Code Gen. and Opt.*, pages 291–302, 2005.
14. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.

15. IDAPro disassembler, http://www.datarescue.com/idabase/.
16. N. Immerman. *Descriptive Complexity.* Springer-Verlag, 1999.
17. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, pages 66–74, 1982.
18. T. Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000.
19. A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *TOSEM*, 2005.
20. H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *SAS*, pages 238–254, 1996.
21. S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. In *Symp. on Princ. of Database Syst.*, 1994.
22. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 24(3):217–298, 2002.
23. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.
24. B. Steensgaard. Points-to analysis in almost-linear time. In *POPL*, 1996.
25. J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *PLDI*, 2004.
26. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *SAS*, 2002.