IMDEA SOFTWARE TECHNICAL REPORT
TR-IMDEA-SW-2012-001

ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions

Juan Caballero*, Gustavo Grieco*, Mark Marron*,
Zhiqiang Lin‡, and David Urbina‡

*IMDEA Software Institute ‡University of Texas at Dallas

August 30, 2012

Abstract

Data structure signatures can be used for finding instances of data structures holding sensitive data in memory, a crucial capability for many security applications such as memory forensics, rootkit detection, online games cheat analysis, reverse engineering, and virtual machine introspection. Manually generating data structure signatures is a tedious and error-prone process. Prior work automatically generates data structure signatures from the type definitions in the program's source code, but unfortunately for many programs their source code is not publicly available.

In this paper we present ARTISTE, the first tool for automatically generating data structure signatures without access to the program's source code or debugging symbols. The salient features of ARTISTE are: (1) it generates hybrid signatures that minimize false positives during scanning by combining points-to relationships, value invariants, and cycle invariants; (2) it uses a novel dynamic shape analysis to recover recursive data structures, classifying them by their shapes (e.g., doubly linked-list or tree); (3) it identifies data structures of the same type allocated at different program points; and (4) it accumulates data structure information over multiple executions, increasingly improving its accuracy. Our experimental results on a number of binary programs show that the hybrid signatures generated by ARTISTE accurately identify instances of the data structures in memory with no false positives or false negatives in 80% of the programs, while prior signature types produce large false positive rates.

1 Introduction

Data structures are central to programs because they store and organize the program's data. Some data structures are of special importance because they store sensitive data, such as the running processes in OS kernels, unit and resource information in online games, and credentials and contact information in Instant Messengers (IM). Locating instances of these sensitive data structures in memory is crucial for many security applications such as memory forensics [31, 32], rootkit detection [10, 15], online games cheat analysis [5], reverse engineering [28, 33, 50], and virtual machine introspection [19].

One approach for locating instances of data structures in memory is scanning using data structure signatures [4, 49, 51], which are either manually generated [30], a tedious and error-prone process, or automatically generated from the definitions in the program's source code [2, 15, 31, 32, 41]. Unfortunately, for many programs their source code (and debugging symbols) are not publicly available.

Another approach is to build a path signature that captures the pointer traversals needed to reach an instance of a data structure from a global variable [10]. However, this approach is fragile as any unresolved points-to information cuts the path, missing the data structure instance.

Thus, it is pressing to automatically generate data structure signatures without the program's source code or debugging symbols. Prior work has proposed to recover data structure information from low level data such as memory snapshots [12] or program binary code [28, 33, 50]. But, those approaches only address a subset of the process required to generate data structure signatures. For example, sensitive data is often stored in recursive data structures (RDS) such as lists and trees (e.g., the Windows Vista SP1 kernel uses over 1,500 doubly linked list types to store critical OS information [10]). However, none of the above approaches recover RDS, or automatically generate data structure signatures.

In fact, recovering high level data structures such as RDS from low level data is a challenging process. State-of-the-art approaches only partially address this problem. In particular, TIE [28] only infers primitive types (e.g., pointer, integer) for the program's variables. REWARDS [33] and HOWARD [50] excavate data structures, but do not identify data structures of the same type and do not identify RDS. LAIKA [12] identifies data structures in a memory snapshot and groups those of the same type, but it does not identify fields inside data structures, does not classify the type of RDS, and more importantly cannot generalize information from one snapshot to another.

In this paper we present ARTISTE, the first tool for *A*utomatically gene*R*a*TI*ng data *S*tructure signa*T*ur*E*s without access to the program's source code or debugging symbols. ARTISTE uses dynamic analysis to recover the data structures used by the program in a number of executions, and builds signatures to locate them in memory.

The salient features of ARTISTE are that it automatically: (1) generates *hybrid signatures* that capture points-to relationships among data structures, value invariants (e.g., constant value or value range), and cycle invariants (e.g., cycles between data structure instances), significantly minimizing false positives during scanning, whereas prior approaches use exclusively value invariants (e.g., [15]), just points-to relationships (e.g, [31,32]), or only cycle invariants (e.g., [30]); (2) identifies recursive data structures, classifying them by their shapes (e.g., doubly linked-list or tree); (3) merges data structures of the same type allocated at different program points, producing a closer approximation of the source-level types; and (4) accumulates data structure information over multiple executions, increasingly improving its accuracy.

We have implemented ARTISTE, and evaluated it on 5 Windows programs. The results show that the generated hybrid signatures produce no false positives and no false negatives for 4 of the 5 programs. For those 4 programs, value invariant only and points-to only signatures produce a large number of false positives, demonstrating the need for hybrid signatures that combine as much discriminating information as possible. For the remaining program, our signature performs no worse than a manually generated signature, showing that some data structures signatures are challenging to build even when source code, debugging symbols, and profiling information are available.

This paper makes the following contributions:

- We present the first approach for automatically generating data structure signatures without the program's source code or debugging symbols (§2). Our approach generates hybrid signatures that minimize false positives during scanning by combining points-to relationships, value invariants, and cycle invariants (§6).
- We design two online primitive type inference algorithms (§3) and show that they are faster and more accurate than an online version of REWARDS.
- We propose a technique for identifying data structures of the same type allocated at different program points (§4) by leveraging pointer target type, structure, and profiling information.
- We present a novel dynamic shape analysis (§5) for identifying recursive data structures and classifying them into useful shapes such as lists or trees.
- We design and implement ARTISTE, a data structure recovery and signature generation tool that works directly on program binaries and applies to Windows programs. We evaluate ARTISTE on 5 representative binary programs showing that its hybrid signatures outperform prior approaches (§7).

2 Overview

In this paper a *data structure* is a composite type (e.g., a record) defined in the program's source code and a *recursive*

```
Root: callsite_0x0100a7d2
000 012 struct(callsite_0x0100a7d2)
000 004 num32 FIELD.IN-RANGE.[0,53]
004 004 ptr32((struct(callsite_0x0100a7d2))) (DLL,F)
008 004 ptr32((struct(callsite_0x0100a7d2))) (DLL,B)
```

Figure 1: Example signature. Each line corresponds to a node in the tree, with the field offset, the field length, the field type, and the field's invariants.

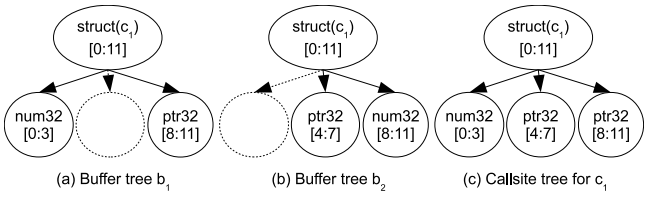

Figure 2: Merging two compatible buffer trees into a callsite tree produces a more refined representation of the data structure allocated at the callsite.

data structure is a data structure that contains at least one recursive pointer. A recursive data structure typically forms a *shape* such as a doubly-linked-list (DLL) or a tree.

A *data structure signature* captures information on a *root* data structure to be located in memory, as well as all data structures reachable by transitively following pointers from the root. Each data structure in a signature is summarized by a *format tree* that captures its layout (i.e., which fields exist at which offsets), the type of each field (e.g., pointer, integer array), and which other data structures or variables it points to (i.e., the target type of its pointers fields). The format tree may differ from the data structure definition as it captures the layout produced by the compiler, which may store an integer or pointer using 4 or 8 bytes depending on the architecture, add new fields (e.g., a vtable pointer in C++ objects), align fields with padding, and inline the layout of a parent class into a child class.

A *hybrid signature* is a data structure signature that annotates the fields in the format trees with value-invariants (e.g., a field has constant value or is always non-zero) and recursive pointers with the shape they form.

The shape information is added because some shapes have implicit *cycle invariants*. A cycle invariant describes a cycle in memory, i.e., a sequence of pointer traversals that start and end at the same object. For example, in a DLL self→forward→back=self. Cycle invariants have large discriminating power but are difficult to find because they are not visible in the source code and a cycle in the heap may not always hold, so prior work identifies them manually [30]. On the other hand, ARTISTE automatically identifies cycle invariants that are implicit in some of the recovered shapes.

Figure 1 shows the signature produced by ARTISTE for the lists of cards in the Spider game shipped with Windows. This signature contains only a single root tree. Each line corresponds to a field with its offset, length, type, and invariants. There are 3 fields: the card identifier at offset zero which the analysis infers as a 32-bit number in the range [0,53] (52 cards plus two special cards), and two recursive pointers at offsets 4 and 8 corresponding to the forward and back pointers of a DLL. Figure 2c depicts the corresponding format tree. Each node represents a field in the data structure with its range and type. The type of a node can be *structure*, *primitive*, *array*, and *dynamic array* (a variable-length array such as a string or a vector container).

2.1 Approach Overview

ARTISTE uses dynamic analysis, accumulating the (partial) information recovered over a number of program executions. It uses 3 classes of format trees at different stages of the recovery process: *buffer trees*, *callsite trees*, and *type trees*. For each execution, ARTISTE constructs a buffer tree for every heap allocation, loaded module, and function stack frame. Once all executions have been analyzed, it merges the buffer trees from buffers of the same type from all runs.

To identify the same module across executions it uses its name, and for stack frames it uses the function's entry address. To identify heap buffers of the same type the intuition is that buffers allocated at the same *callsite* (i.e., instruction that invokes an allocation) typically are of the same type. However, there are some exceptions. For example, the program could use a wrapper `my-malloc` and the callsite that invokes `malloc` inside the wrapper would return objects of different types. Multiple such wrappers could be nested. In this work we use the callsite as a first approximation of the type of a heap buffer, but only if the callsite proves to be *stable*, i.e., all the buffers it allocates have the same size and are compatible. If a site is not stable, we check if the upper callsite in the callstack is stable.

For stable callsites ARTISTE builds a callsite tree by merging all buffer trees allocated at the callsite. Since the same data structure can be allocated at different callsites, ARTISTE further merges the callsite trees of callsites that it considers equivalent into type trees.

Merging trees. Merging compatible trees produces a more refined tree because the information in each input tree may be partial. Merging two trees means inserting all fields of one tree into the other. When inserting a field, if its range does not partially overlap other fields (only disjoint and completely enclosed ranges are allowed), the field is added. If there is already a field in the tree with the same range, the type of the existing field is updated to be the most refined of both types. The merging succeeds if the following conditions hold: (1) node ranges do not partially overlap, (2) there are no incompatible types in overlapping nodes, (3) all primitive types are in leaf nodes, and (4) dynamic arrays are only present in the root node. Otherwise, the merging fails. We say that two trees are *compatible* if they can be merged without errors. Figure 2c shows the callsite tree that results

from merging buffer trees b_1 and b_2 (both from callsite c_1). The resulting callsite tree has 3 fields, compared to two for each buffer tree, and field [8:11] has type `ptr32`, rather than `num32` in buffer b_1. Since the Spider game has only one callsite, Figure 2c is also the type tree used in the signature.

Identifying the data structures of interest. To identify the unknown data structures holding the sensitive data, we monitor the application as we feed it instances of the sensitive data, as those will be inserted into the appropriate data structures. We use taint tracking [40] to taint the sensitive data (e.g., the nickname of a contact we add to our instant messaging account) and take a snapshot of the program after the data has been consumed. We scan the memory snapshot to identify tainted memory locations and the callsites of the heap buffers that contain the tainted data. Once ARTISTE has recovered the type trees, the taint information is used to select as signature root the type of the buffer that dominates most tainted buffers.

Code coverage. Producing a data structure signature requires significantly less coverage than recovering all data structures used by a program because signatures typically contain only a small number of data structures. For example, the signature needed to identify the user contacts in the Miranda IM client contains only 3 data structures, a small subset of the more than 1,500 that Miranda defines across 18 modules. To build that signature ARTISTE only needed 5 executions of Miranda operating on the relevant data structures. These were easily obtained by adding, editing, and removing contacts. These operations make the program use the relevant data structures even if they only cover a small part of the application. While ARTISTE can also be used to recover all data structures this requires an external input generation tool to produce inputs that traverse many different program paths [8,9,21] and may take a very long exploration time with large programs.

2.2 Architecture Overview

ARTISTE comprises four phases: *execution analysis* (§3), *global analysis* (§4), *dynamic shape analysis* (§5), and *signature generation* (§6).

Execution analysis. Figure 3 details the architecture of the execution analysis. First, the program is run inside a previously available *execution monitor* [**?**]. The execution monitor is a full system emulator that can run any PE/ELF program binary on an unmodified guest operating system (x86 Windows or Linux) inside another host operating system (Linux on x86). The execution monitor produces an *execution trace*, containing all executed instructions and the contents of each instruction's operands. It also produces an *allocation log* with the buffer address, size, and callsite for each allocation/deallocation operation (heap and memory-mapped files) invoked by the program during the run.

The execution trace and the allocation log are inputs to the analysis. The core of the execution analysis is the *type tracker*, which infers primitive types stored in registers and memory throughout the execution. For each buffer deallocated and each buffer alive at the end of the execution the *tree builder* produces a buffer tree using the primitive types inferred by the type tracker in the memory range of the buffer and array information from a previously available *array detection* module [6]. All buffer trees are output at the end of the execution.

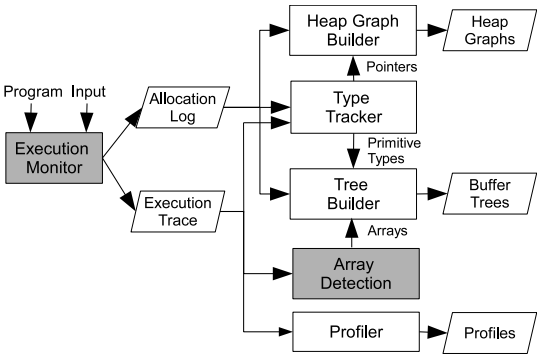

Figure 3: Execution analysis architecture. Gray boxes were previously available.

There are two additional modules. The *heap graph builder* outputs a graph of the program's heap (i.e., *a heap graph*) at a periodic interval during the execution. The *profiler* tracks for each buffer, the set of functions that operate on the buffer. If provided with a format tree, it collects the values assigned to its fields and produces value invariants for them.

Global analysis. Figure 4a details the architecture of the global analysis. All buffer trees are loaded in the *store*, which groups them by callsite. Then, the *dynamic array detector* classifies callsites as stable, dynamic arrays, or unstable. Next, the *callsite merger* merges all buffer trees for a stable callsite into a callsite tree. Finally, the *callsite clusterer* merges compatible callsite trees into type trees.

Dynamic shape analysis. For each heap graph produced during execution analysis, dynamic shape analysis identifies the shapes it contains. The accumulated shape information from all heap graphs is used to update the type trees output by the global analysis (Figure 4b).

Signature generation. The signature generator extracts the subset of the type trees output by the shape analysis that are reachable from the root data structure. Then, it uses the value invariants produced by the profiler to annotate the type trees, and outputs the final hybrid signature (Figure 4c).

3 Execution Analysis

Execution analysis builds a buffer tree for each buffer deallocated during a program run and for each buffer alive at the end of the run. The core of the execution analysis is the *type*

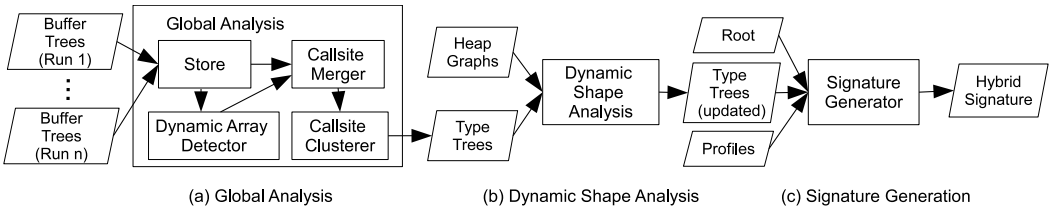

Figure 4: Global analysis, dynamic shape analysis, and signature generation architecture.

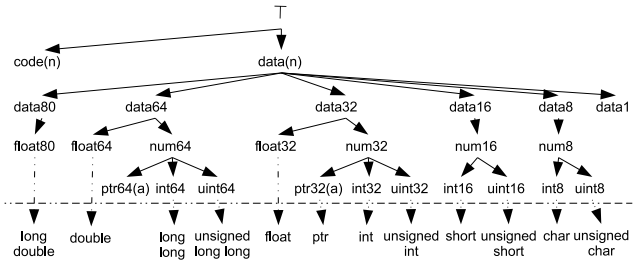

Figure 5: Our primitive type lattice. Above the horizontal line are the inferred primitive types and below the mapping to 32-bit C types.

tracker, which implements one of our two TIPO[1] dynamic data-flow-based primitive type inference algorithms, which are significantly more accurate and faster than an online version of REWARDS [33].

Our primitive type lattice. Figure 5 shows our lattice of primitive types, which is designed for the x86 architectures (16, 32, or 64-bit). In the lattice, $\top$ represents an unknown type. Not shown in the figure is $\bot$, which represents a type conflict. All types are connected to $\bot$. The dotted line marks the conversion from our primitive types to C types in a 32-bit architecture. The goal of the type tracker is to infer refined primitive types, i.e., as far down the lattice as possible without reaching $\bot$.

Our primitive pointer types (`ptr32`, `ptr64`) are parametrized by a set of target types the pointer can point to, i.e., they distinguish between a `struct a*` and a `struct b*`. Pointer target types are fundamental to identify callsites of the same type and recursive data structures. We need a set because pointers may point to objects from multiple callsites, and also for handling void pointers, and pointers to classes with inheritance (where the pointer may point to objects of the parent and child classes).

Compared to the lattice used by TIE [28] our lattice adds floating-point types (single, double, and extended precision) and 64-bit data, separates code from data, and considers pointers to be number subtypes (i.e., `ptr32` is a subtype of `num32`). The latter change was done so that arithmetic instructions (e.g., `add, sub`) always operate on numbers, regardless if performing integer or pointer arithmetic.

[1]TIPO stands for Type Inference from Primitive Operations.

Algorithms overview. We propose two online type inference algorithms which operate forward as the execution progresses, and compare them with an online version of REWARDS. The main difference between our TIPO-A (aggressive) and TIPO-C (conservative) algorithms is that TIPO-A types more but may generate some type conflicts, while TIPO-C types less but does not produce type conflicts in our experiments.

Both algorithms type *locations*, i.e., bytes in registers, memory, and immediate values. For each location (i.e., byte), they track the current inferred set of *byte types*, where a byte type is a pair of a primitive type in our lattice and an offset from zero to the size of the primitive type minus one. At any point in the execution (e.g., buffer deallocations) the algorithm can output the inferred primitive types from our lattice by examining sequences of consecutive locations.

The main difference between our algorithms and prior work [22, 33] is that our algorithms do not use unification. With unification, when a data movement instruction executes ($dst \leftarrow src$) the type of src propagates to dst *and* the type of dst (before the instruction executes) propagates to src. Unification is problematic when dst is used as temporary storage and can thus hold variables of different types over time. Untyped storage is uncommon at source-level but happens often at binary-level with registers and the stack. While REWARDS disables unification when dst is a register the stack may also be used as temporary storage (e.g., to save registers).

Type sinks. Our algorithms infer types from instruction and function type sinks using dynamic data flow approaches. Instruction type sinks define types for the operands of selected x86 instructions. For example, both operands of an `add` instruction can be typed as `num32`. Function type sinks define types for the arguments and return values of functions with a publicly available prototype (e.g., from the Windows API). The source-level types in the prototype are mapped to one of our primitive types. For example, the `strlen` function returns a `size_t` value in the EAX register, so on function return EAX can be typed as `uint32`. Both type sink classes return a list of (location, byte type) pairs and separate read type sinks (locations in operands read by an instruction or function parameters) from write type sinks (locations in operands written or return values).

A special instruction type sink is applied to any instruction that uses indirect memory addressing. This sink determines which memory addressing register contains a pointer, and

Algorithm 1 TIPO-A

```
1  def instrument_insn(insn) {
2    for (l_src,T) in read_type_sinks(insn)
3      idx=get_idx(l_src)
4      add_byte_type(idx,T)
5    for (l_dst,T) in write_type_sinks(insn)
6      idx = new_idx()
7      set_idx(l_dst,idx)
8      add_byte_type(idx,T)
9
10   if (moves_data(insn))
11     for (l_src,l_dst) in read_write_locations(insn)
12       idx = get_idx(l_src)
13       set_idx(l_dst,idx)
14   else
15     for l_dst      in write_locations(insn) and
16        (l_dst,_) not in write_type_sinks(insn)
17       clear(l_dst)
18
19   if is_alloc_call(insn) or is_alloc_ret(insn)
20     (addr,size) = get_alloc_range(insn)
21     for a in [addr,addr+size)
22       clear(memloc(a))
23
24   if is_dealloc_call(insn)
25     (addr,size) = get_dealloc_range(insn)
26     create_buffer_tree(addr,size)
27     for a in [addr,addr+size)
28       clear(memloc(a))
29 }
```

queries for the callsite of the buffer the pointer points to. If the query returns a callsite (i.e., points-to a heap buffer), the callsite is added to the set of target types of the pointer, otherwise it is a pointer to $\top$. To efficiently handle the queries we use a red-black tree that stores the live buffer ranges and is updated at each allocation and deallocation.

Next, we describe TIPO-A in greater detail, and highlight the TIPO-C differences.

TIPO-A. Algorithm 1 describes our TIPO-A algorithm, which calls *instrument_insn* for each executed instruction. This function updates two maps: an *index map* from a location to an integer index, and a *byte type map* from an index to a set of byte types.

The byte type map is updated using the information from the type sinks (lines 2-8). For each location to type that is being read, the algorithm uses the `get_idx` function to obtain the current index of a location from the index map or create a fresh index for the location if it currently has no index. Then, it adds the byte type to the current set of byte types for the index. For locations to type that are being overwritten, a fresh index is created for the location and the byte type is added to the (empty) set for the new index.

The index map is updated in data movement instructions (e.g., mov, push, pop) by propagating the index from the locations in the source operand to the corresponding locations in the destination operand (lines 10-13). Note that `get_idx` creates a fresh index for the location if it has none. If the instruction does not move data but performs other operations the algorithm clears the index for the locations that are written by the instruction and have not been typed by a type sink associated with this instruction (lines 14-17).

If the instruction is the call or return of a memory allocation the algorithm clears the indices for all locations in the allocated memory range, as it may later be reused for a different heap object, module, or stack frame (lines 19-22).

When a heap deallocation is invoked, a buffer tree is created for the deallocated memory range and the memory range is cleared (lines 23-27). To create a buffer tree, the byte type set for each location in the range is first resolved to a single byte type by using the lattice meet operation and checking that indices are equal. For example, a location with a byte type set containing (num32,0) and (int32,0) resolves to (int32,0) but a set containing (num32,0) and (num32,1) resolves to ($\perp$,0). Then, sequences of consecutive locations that form complete primitive types are extracted from the range. For example, if the location at offset `o` in the buffer has a resolved byte type of (int16,0) and the location at `o+1` has (int16,1) then an int16 primitive type has been found at offset `o`. An empty buffer tree is created with the range of the deallocated buffer and for each primitive type found in the range, one field is added to the tree. Next, the tree is updated with information from the external array detection module. Finally, the buffer tree is added to the store tagged with the buffer's callsite.

```
struct{
  int a;
  float b;
} record;
```

```
1: mov &record,%ebx
2: xor %eax,%eax
3: mov %eax,(%ebx)
4: mov %eax,4(%ebx)
5: add (%ebx),%ecx
6: fldl 4(%ebx)
7: mov %ecx, (%ebx)
```

Figure 6: Example of a problem introduced by unification.

TIPO-C. Our conservative algorithm differs from TIPO-A in that it only updates a single type map from location to byte type set. In TIPO-C lines 3-4 are replaced by a call to `add_byte_type` which adds the byte type from the sink to the current byte type set of the location; lines 6-8 are replaced by a call to `set_byte_type` which sets the byte type set of the location to a singleton set; and at lines 12-13 rather than propagating an index from source to destination, the byte type set of the source is propagated. The rest of the algorithm is identical. A simple way of explaining the difference between algorithms is that TIPO-C only propagates types forward (starting at type sinks) but TIPO-A propagates also backwards because when a type sink types a location all other (previously seen) locations with the same index are also typed. This explains why TIPO-C is more conservative and thus types less.

We use Figure 6 to illustrate the differences between our algorithms and the online REWARDS. At line 1 the address of the `record` structure is moved into register `eax`. At line 2 `eax` is cleared. At lines 3–4 both fields in the structure are cleared using `eax`. At this point, with REWARDS fields `a` and `b` are unified with the value of `eax`. With TIPO-A `eax` and both fields have the same index. In all 3 algorithms `eax,a,b` are untyped. At line 5 `a` is added to the value

in the ecx register so a and ecx are assigned the num32 type by the add instruction type sink. At this point, with REWARDS and TIPO-A, eax, ecx, a, and b are typed as num32, which is incorrect for b. With TIPO-C, only a and ecx are typed as num32 as types only propagate forward. At line 6, the content of b is moved into a floating point register so b is assigned the type float32. At this point, with REWARDS and TIPO-A, eax, a, and b are all conflicted because num32 and float32 are not compatible in our lattice. With TIPO-C, a and ecx are correctly typed as num32 and b as float32. Finally, at line 7, a is assigned the content of ecx. REWARDS unifies ecx and a, propagating the conflict to ecx. Instead, TIPO-A propagates the index from ecx to a so that at this point both ecx and a are typed as num32. This removes the conflict in a. TIPO-C simply propagates the type from ecx to a so all locations stay unconflicted.

This example shows that (1) TIPO-C is more conservative, producing less conflicts; (2) the use of unification by REWARDS propagates the conflicts generating a snowball effect; and (3) TIPO-A limits the propagation of conflicts and can even recover from them. What the example does not show but our evaluation will is that TIPO-A significantly types more than TIPO-C (§7).

Performance. Our algorithms are significantly faster than REWARDS because they do not need to transitively update the constraint sets but simply update the maps. Our evaluation shows that in exchange for typing less, TIPO-C is a bit faster than TIPO-A. TIPO-A uses two optimizations. First, any location not in the index map has implicitly type $\top$, so there is no need to track untyped locations. Second, the clear function not only removes the index for the location from the index map, but also does garbage collection of unneeded indices in the byte type map.

4 Global Analysis

Global analysis runs once, after all executions have been analyzed. It comprises two steps. First, it merges the buffer trees from the same callsite, across all executions, into callsite trees, identifying dynamic arrays and unstable callsites in the process (§4.1). Then, it identifies compatible callsite trees and merges them into type trees (§4.2).

4.1 Callsite Trees

The first step in global analysis is classifying each callsite, from which a buffer was allocated in any execution, as *stable*, *dynamic array*, or *unstable*, and to produce callsite trees for stable callsites and dynamic arrays. For this, ARTISTE partitions callsites into those that always allocate buffers of the same size and those that do not. For callsites that allocate buffers of the same size all the buffer trees for the callsite are merged together. If the merging succeeds (i.e., the buffer trees are compatible), the callsite is stable and the merged tree is the callsite tree. Otherwise, the callsite is unstable and no callsite tree is output.

For each callsite that returns multiple buffer sizes, the dynamic array detector checks if the callsite allocates a variable-length array of objects of the same type, which are used by strings and in container implementations such as vectors or hash tables. For this, it first computes the greatest common denominator (gcd) of the sizes of all buffers allocated at that callsite. Then, it splits each buffer tree for the callsite into subtrees of gcd size and merges all subtrees together. If the merging produces no errors, it outputs a callsite tree of type dynamic array of objects with the structure of the merged subtree. Otherwise, the callsite is unstable and no callsite tree is output.

The above process could incorrectly consider a dynamic array or an unstable callsite to be stable, e.g., traces contain only one allocation of a callsite that allocates a variable-length string. These errors tend to disappear as coverage increases and may also be corrected when merging callsites by pointer target type (§4.2).

4.2 Type Trees

Multiple callsites may allocate objects of the same type. ARTISTE uses two steps to merge callsite trees of the same type into type trees. First, ARTISTE uses the pointer target type information to identify callsites that are pointed-to by the same pointer. Then, it clusters the resulting trees based on their format and profiling information.

Merge by pointer target types. Most pointers always point-to objects of the same type, but there are exceptions such as void pointers, pointers to classes with inheritance, and pointers to unions. ARTISTE merges callsites that are pointed to by the same pointer if and only if their callsite trees are compatible. First, it splits the callsite trees into equivalence classes using the pointer target type information. For this, ARTISTE folds over all the callsite trees transitively grouping pointers with a non-empty target type intersection. For example, if pointer p_1 in callsite A has a target set of callsites $\{C, D, E\}$ and pointer p_2 in callsite B has a target set of callsites $\{E, F\}$, then callsites $\{C, D, E, F\}$ form an equivalence class because they are likely to allocate objects of the same type.

For each equivalence class, ARTISTE merges its callsite trees. If the merging succeeds, the merged tree replaces the callsite trees, otherwise, they are left separate. Finally, it iterates over all the resulting trees, updating the pointers to point to the merged trees, i.e., pointers x and y now point to the tree resulting from merging callsites $\{C, D, E, F\}$.

Clustering. To cluster these groups of callsites into the final types, we define a distance metric between two callsite groups. The distance metric is based on the intuition that callsites of the same type have the same structure and that they are used by the program in similar fashion. Our distance metric first checks if the trees are compatible. If so, it

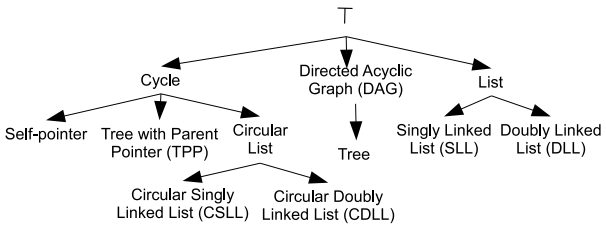

Figure 7: Our shapes lattice.

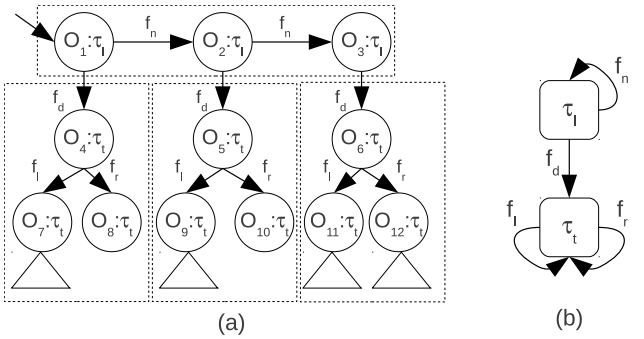

Figure 8: On the left, a heap graph with each node annotated with its type and each edge with the field holding the pointer. On the right, the corresponding type graph.

combines the following two features:

- *Format feature:* This normalized feature captures how different the formats of two trees are, with zero meaning identical and one completely dissimilar. The smaller this feature, the higher confidence the trees belong to the same type. This feature first computes the tree edit distance using the algorithm proposed by Zhang and Shasha [52] and then normalizes the distance using the metric normalization proposed by Li and Zhang [29].
- *Profiling feature:* This normalized feature captures how differently two groups of callsites are used by the program. It uses the information obtained by the profiler during execution analysis. In particular, it compares the sets of functions that have read or written buffers of those callsite groups across all executions. First, it unions the function sets of all callsites in the group. Then, to compare two function sets A and B, it employs the Jaccard index, which yields 0 when the sets are disjoint and 1 when they are identical [24].

The distance between two callsite groups is computed as:

$$d(g_1, g_2) = \begin{cases} \sum_i w_i \cdot d_i(g_1, g_2) & \text{if } \textit{compatible(g1,g2)} \\ 1 & \text{otherwise} \end{cases}$$

where d_i is the distance for feature i, w_i is the weight associated to the feature, and $\sum_i w_i = 1$. We address weight selection in §7.2.

Once a distance matrix is computed, ARTISTE applies the partitioning around medoids (PAM) algorithm to cluster the callsite groups [27]. Since the PAM algorithm takes as input the number k of clusters to output, the clustering is run with different k values, selecting the one which maximizes the Dunn index [16], a measure of clustering quality. Once clustered, all trees in a cluster are merged into a type tree. Finally, the pointers are adjusted to point to the new types.

5 Dynamic Shape Analysis

Dynamic shape analysis automatically identifies the shape of recursive data structures (RDS) present in a heap graph. It runs on each collected heap graph, accumulating the information. It comprises two steps. First, the heap graph is partitioned into disjoint regions (§5.1). Then, regions corresponding to RDS are classified as one of the shapes in the lattice in Figure 7 (§5.2).

The inputs to dynamic shape analysis are a heap graph, the type trees, and a type graph derived from the type trees. We describe them next.

Heap graph. The heap graph is a directed labeled graph $G_H = (V_H, E_H)$, where V_H is the set of objects (i.e., live buffers) in the heap and E_H is the set of pointers. Each object has three attributes: start address, size, and callsite. Each pointer is a tuple (src, dst, o), where $src, dst \in V_H$ are the source and destination nodes, and o is the offset of the pointer in the source object. Pointers pointing to the middle of an object and null pointers are not included in E_H.

Type graph. The type graph is derived from the type trees. It is a directed labeled graph $G_T = (V_T, E_T)$, where V_T is the set of types in the program's type trees, and an edge (n, m, f) represents that type n contains a pointer to type m at field f in the type tree.

Type trees. The type trees are also used to define two functions: $\mathsf{Type}(o)$ returns the type of object o by mapping the callsite of the object to the corresponding type, and $\mathsf{Fields}(\tau)$ returns the set of all pointers defined in type τ.

Dynamic shape analysis outputs for each shape: the shape type, the concrete objects in the shape, the pointers pointing into the shape (i.e., shape roots), and the role of the recursive pointers in the shape (e.g., forward, back, or parent). The accumulated information is used to annotate the pointers in the type trees with shape information.

5.1 Identifying Regions

We define a *region* $R \subseteq V_H$ to be a subset of the objects in the heap graph that play the same role in the program. At a high level, two objects belong to the same region (i.e., play the same role) if they: (1) belong to the same recursive data structure (e.g., the internal list nodes), or (2) have the same type and are pointed to by equivalent fields of objects in the same region (e.g., the content objects hanging from the list

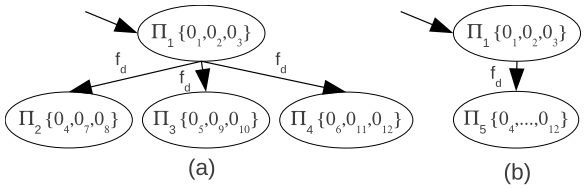

Figure 9: On the left, the heap graph from Figure 8a partitioned by applying the recursive structure relation. On the right, the final partition after applying also the equivalent successors relation.

nodes).

For example, Figure 8a shows a concrete heap graph, which contains a singly-linked list of trees. Nodes in the same recursive data structure have the same role. This includes the three upper nodes in the list and the nodes in each of the three trees. In addition, the 3 trees play the same role in that they represent the payload of the list. This section explains how our partition of the heap graph into regions separates these objects based on their different roles.

Our general approach to identify regions is to iteratively group objects together according to some notion of equivalence, a standard technique in heap abstraction [11, 14, 48]. Our particular approach formulates the identification of regions as a congruence closure computation [39]. To construct the closure we first build a map from objects to partitions using a Tarjan union-find structure. Formally, $\Pi : V_H \rightarrow \{\pi_1, \ldots, \pi_k\}$ where $\pi_i \subseteq V_H$ and $\{\pi_1, \ldots, \pi_k\}$ *partition* V_H. We start with one partition per object, then apply two equivalence relations (detailed below) on the partitions, until the partitions are closed under the relations. At this point the resulting partitions (i.e., equivalence classes) correspond to the regions.

The first equivalence relation identifies objects that are part of the same recursive data structure. Intuitively, two objects belong to the same recursive data structure if they have types which belong to a cycle in the type graph and there is a path between them in the heap graph. In particular, we say that two types are *mutually recursive* (i.e., $\tau_1 \sim \tau_2$) if they belong to the same strongly connected component in the type graph. The following relation iteratively identifies all mutually recursive types until a fixpoint is reached. We use $\sigma(o.f)$ to denote the target object that the pointer stored in field f of object o points-to, and $\mathsf{RegType}(R) = \{\,\mathsf{Type}(o) \mid o \in R\}$ to denote the set of the types of all objects contained in the region.

Relation 1 (Recursive Structure) *Given partitions* π_1 *and* π_2*, the* recursive structure *congruence relation is* $\pi_1 \equiv_r^{\Pi} \pi_2 \Leftrightarrow \exists \tau_1 \in \mathsf{RegType}(\pi_1), \exists \tau_2 \in \mathsf{RegType}(\pi_2)$ *s.t.* $\tau_1 \sim \tau_2 \wedge \exists o \in \pi_1, \exists f \in \mathsf{Fields}(\mathsf{Type}(o))$ *s.t.* $\sigma(o.f) \in \pi_2$.

In the example in Figure 8 the above relation first identifies in the type graph (Figure 8b) two types that are mutually recursive with themselves: $\tau_l \sim \tau_l$ and $\tau_t \sim \tau_t$. Then, it groups connected nodes in the heap graph from those two types into the same region. Figure 9a shows the results of applying this relation, where the 3 list nodes have been grouped into a single partition π_1 and the nodes in each of the 3 trees have been group into partitions π_2, π_3, π_4.

The second equivalence relation identifies whether two successor partitions π_2, π_3 of a given partition π_1 have equivalent roles and thus π_2, π_3 should be merged together. For two successor partitions to have equivalent roles they should be successors on the same pointer field and they should be compatible. Formally, we say that partition π_2 is a *successor* of π_1 on f if there is a pointer field f pointing from an object in π_1 to an object in π_2, i.e., iff $\exists o \in \pi_1, \exists f \in \mathsf{Fields}(\mathsf{Type}(o))$ s.t. $\sigma(o.f) \in \pi_2$. We say that two (successor) partitions are compatible if they share some type, i.e., $\mathsf{Compatible}(\pi_2, \pi_3) \Leftrightarrow \mathsf{RegType}(\pi_2) \cap \mathsf{RegType}(\pi_3) \neq \emptyset$.

Relation 2 (Equivalent Successors) *For the partition* π_1 *with successors* π_2 *on label* f_1 *and* π_3 *on* f_2*,* π_2 *and* π_3 *are* equivalent successors $\pi_2 \equiv_s^{\Pi} \pi_3 \Leftrightarrow (f_1 = f_2) \wedge \mathsf{Compatible}(\pi_2, \pi_3)$.

In the example in Figure 9 the above relation merges together partitions π_2, π_3, π_4 since they are all successors of π_1 on the same pointer field f_d and they all contain type τ_t. Figure 9b shows the final partition of the heap graph into regions after applying Relation 2 on the results of Relation 1 shown in Figure 9a. Note that going from Figure 9a to Figure 9b may require multiple intermediate steps until a fixpoint is reached by Relation 2.

Using the above two relations the congruence closure can be efficiently computed in $O(E_H + V_H)$ space and $O(E_H * log V_H)$ time. The partitions correspond to the regions.

5.2 Shapes

Once the heap graph has been partitioned into regions, the analysis identifies which regions correspond to shapes and their type. For each region, the analysis computes a *region graph*, which is the subgraph of the heap graph that contains all nodes in the region and the edges between them, i.e., $R_\pi = (\pi, \{(src, dst, o) \in E_H | src, dst \in \pi\})$. Region graphs that contain at least two connected nodes correspond to shapes. In Figure 9b both regions found contain connected nodes and thus correspond to shapes.

To identify the shape type, the analysis looks at the structure of the region graph. First, the analysis identifies the *root* objects in each region, which are objects in the region with incoming pointers from objects in other regions. Then, the analysis performs a depth-first-search (DFS) traversal of each region graph starting from the roots. The DFS traversal classifies each edge (src, dst, o) in the region graph as: *forward*, if *dst* has not yet been visited; *cross*, if *dst* has already been visited and the traversal of all children of *dst* has completed; and, *back*, if *dst* has been visited but the traversal of its children has not completed.

Using the edge classification, the analysis performs a preliminary, coarse-grained, classification of the shape regions using graph-theoretic definitions. Each shape region is marked as: a *Cycle* if any edges in R_π are back edges, a *Dag* if any edges in R_π are cross edges (and there are no back edges), or a *Tree* if all edges in R_π are forward edges.

To refine this preliminary classification the analysis checks if any of the following predicates holds for the shape's region graph R_π:

- **Singly linked list:** contains forward pointers at the offset o_f and all edges in R_π have this offset.
- **Doubly linked list:** contains forward pointers at offset o_f and back pointers at offset o_b, and $\forall(src, dst, o_f)$, $\exists(dst, src, o_b)$. We report o_f and o_b as the *forward* and *back* references in the list, respectively.
- **Circular singly linked list:** contains both forward and back pointers at offset o.
- **Circular doubly linked list:** contains both forward and back pointers at offsets o_1, o_2, and $\forall(src, dst, o_1)$, $\exists(dst, src, o_2)$. We select the smallest offset as the *forward* reference and the other as the *back* reference.
- **Tree with parent pointer:** contains back pointers at offset o_p, all other pointers are only seen on forward edges, and for all edges (src, dst, o_p) there exists a forward edge (dst, src, o). We select the pointer at o_p as the *parent* reference in the tree and all other as the *child* references.
- **Self pointer:** contains a back edge to itself (src, src, o).

If any of the above predicates holds, the shape type is refined. Otherwise, the previously identified coarse-grained type (tree, dag, cycle) is reported.

In the example in Figures 8–9, the region graph for π_1 (upper dotted box in Figure 8) contains two edges that the DFS traversal marks as forward. Thus, the coarse-grained shape type for π_1 is *Tree*. Since both edges are from the same pointer field f_n the region graph for π_1 satisfies the *Singly linked list* predicate, and the shape type for π_1 is singly linked list on the forward pointer f_n. For region π_2, the DFS traversal marks all edges as forward, so the coarse-grained shape type is also *Tree*. In this case, there are two pointer fields (f_l and f_r in type τ_t) and no specific predicate is satisfied. Thus, the shape type for π_2 is *Tree* with children pointers at f_l and f_r.

6 Signature Generation & Matching

Given the recovered type trees and the name of the root tree, the signature generation module first extracts the subset of type trees reachable by transitively following pointers starting from the root type. Then, it annotates the fields in the type trees with value invariants produced by the profiler.

Value invariants. Given a signature and a set of execution traces, the profiler collects the values assigned across the executions for every field in the type trees in the signature. For each field it tries to produce a value invariant. Compared to other tools like DAIKON [17], our profiler generates only the following small number of invariants:

- Zero: (field == 0)
- Constant: (field == c)
- Non-zero: (field != 0)
- Set: (field $\in$ {a,b,c})
- Range: (field $\in$ [a,b])

The *profiler* applies the invariant templates in the above order, e.g., a constant invariant will not also produce a value-set invariant.

Cycle invariants. Doubly-linked lists, as well as all shapes that are subtypes of Cycle in Figure 7 contain implicit cycle invariants. During signature matching, ARTISTE uses the shape information annotated in the pointer fields to verify the implicit cycle invariants. For example, self→forward→back=self for CDLL and DLL, and self→child→parent=self for TPP. Acyclic shapes like DAG, Tree, or List have implicit non-cycle invariants such as self→forward→forward≠self, although these typically discriminate less.

Snapshots. Our execution monitor can take snapshots of the memory state of a process by walking the page table of the process. For each page mapped to the process, it stores the virtual address range of the page, its contents, and any associated taint information.

Signature matching. We have built a scanner that checks a hybrid signature on each memory address in the snapshot, reporting addresses that match. Given an address and a type tree the scanner first reads from the snapshot the range of the data structure. Then, it checks if the value of each field in the data structure satisfies any value or cycle invariants defined for the field. For primitive type fields, it also checks some additional constraints on the field value. Specifically, if the field is a pointer it checks that the target address exists in the memory snapshot; if the field is a signed or unsigned integer (but not a generic number) it checks that its value is not an address present in the snapshot (i.e., not a pointer); and for floating point fields we use a heuristic proposed in DIMSUM [31] that checks if the exponent is within a certain range since really small and really large values are uncommon in most non-scientific applications.

If all fields satisfy these checks, then for each pointer in the type tree, the scanner recurses into matching the type tree corresponding to the pointer's target type at the (candidate) pointer's target address unless: it is a null pointer, the target address has already been checked against that type, or the pointer target type is unknown. The scanner limits the recursion depth (by default to 5) to avoid infinite loops when matching cyclic data structures. The scanning starts at the

	Execution Analysis			Global Analysis
	TIPO-A	TIPO-C	REWARDS *	
healthbmk	26.1 min	22.6 min	2.5 days	2.9 sec
bhbmk	8.5 min	7.4 min	2.5 hours	1.4 sec
miranda	23.9 min	21.6 min	1.4 days	9.9 sec
pidgin	2.2 hours	1.5 hours	> 2.5 days	20.7 sec
spider	7.1 min	6.5 min	56.1 min	0.1 sec

Table 1: Total runtime of the execution and global analysis over 5 program runs.

Program	Float64	Ptr32	Int32	Num32	Other
bhbmk	35,527	5,954	0	2,825	0
healthbmk	3,630	124,282	2,021	11,301	0
miranda	0	6,477	132	3,210	238
pidgin	304	55,733	35	80,394	12,493
spider	0	836	0	756	0
Total	39,461	193,282	2,188	98,486	12,731

Table 3: Primitive types inferred by TIPO-A over the 5 program runs.

lowest address in the snapshot with the root type and linearly traverses the process address space, skipping pages not present in the snapshot.

7 Evaluation

We have implemented ARTISTE using over 8,000 lines of Ocaml and 3,000 lines of C++ code. Our evaluation of ARTISTE comprises two parts. First, we evaluate the 3 phases involved in the signature generation process (§7.1–§7.3) Then, we evaluate the signatures by scanning a number of memory snapshots (§7.4). We use 5 programs in our evaluation. Bhbmk and Healthbmk are two C++ programs that are part of the Olden heap analysis benchmark [18] so we know their data structures and the shape of their recursive data structures. Pidgin [43] and Miranda [37] are two large and popular open-source IM clients written in C that are interesting because they contain sensitive data such as the user's contacts. For these 4 programs we have their source code and for all but Miranda also PDB symbol files. Source code and PDB files are not used by ARTISTE, but are needed as ground truth to evaluate the results. We also include the Windows Spider card game for which we have no source code or debugging symbols. All programs are run on a Windows XP SP3 guest OS.

Since our goal is to create signatures for the program's data structures rather than the internal data structures in Windows API libraries, we configure ARTISTE to ignore allocations by standard Windows libraries (e.g., `kernel32.dll` or `gdi32.dll`) focusing on the program's modules (executable and associated DLLs). The analysis comprises the following total modules: 1 for bhbmk and healthbmk, 4 for spider, 18 for Miranda, and 90 for Pidgin. Despite this, the data types implementing the containers from the C++ STL are still inferred because they are provided as C++ templates and inlined into every module that uses them.

7.1 Execution Analysis

In this section, we evaluate our type inference algorithms, comparing them with an online version of REWARDS. For a fair comparison, rather than using the REWARDS implementation (kindly provided by the authors), we re-implement it in our framework, so that we can use exactly the same type sinks and the same type lattice for all algorithms. This way, we focus the comparison in the algorithm and particularly in the use of unification. We term this version REWARDS*.

Table 1 shows the runtime of the 3 algorithms over 5 traces. REWARDS* on Pidgin did not finish running on any of the 5 traces in the 60 hours that we had allocated for each trace, so the runtime is left as at least 2.5 days. The results show that in the 4 programs that complete, the fastest algorithm is TIPO-C, 12% faster than TIPO-A, and that REWARDS* is two orders of magnitude slower than both. This is due to REWARDS* needing to transitively update the constraint sets for every move-like instruction.

Table 2 shows the execution analysis results over 5 executions. The second column shows the total number of heap bytes (above) and buffers (below) allocated by the program over the 5 executions. Then, for each algorithm, the table shows the number and percentage of allocated bytes that are typed, untyped ($\top$), and conflicted ($\bot$). The numbers for REWARDS* on Pidgin are missing because the analysis did not complete, as explained above. The results show that TIPO-A types the most locations, followed by TIPO-C, with REWARDS* typing the least. All 3 algorithms leave similar numbers of bytes untyped so bytes that are typed with TIPO-A and TIPO-C are conflicted with REWARDS*. This happens because with unification once a conflict is found it propagates causing a snowball effect. While TIPO-A types more than TIPO-C, it also produces some conflicts for Pidgin and Miranda, while TIPO-C does not produce any conflicts in these experiments.

Table 3 presents the total number of primitive types inferred in the 5 runs using TIPO-A. Pointers dominate most programs except for bhbmk which uses a large number of double floating point numbers. It also shows that we infer many more generic numbers than signed or unsigned integers. While a few of these are pointers that were not dereferenced, this number illustrates the difficulty of determining signedness at the binary level.

Overall, the execution analysis results show that TIPO-A and TIPO-C significantly outperform REWARDS* in performance and accuracy. We believe TIPO-A is preferable for most applications since it types more with only a small number of conflicts and similar performance to TIPO-C.

			TIPO-A			TIPO-C			REWARDS*		
	Runs	Bytes / Buf.	Typed	⊤	⊥	Typed	⊤	⊥	Typed	⊤	⊥
bhbmk	5	330,884 12,993	319,012 96.4%	11,872 3.6%	0 0%	307,424 92.9%	23,460 7.1%	0 0%	291,560 88.1%	10,353 3.1%	28,971 8.8%
health-bmk	5	806,424 63,310	579,456 71.9%	226,968 28.1%	0 0%	567,136 70.3%	239,288 29.7%	0 0%	84,536 10.5%	231,136 28.7%	490,752 60.8%
pidgin	5	1,479,670 49,213	558,966 37.8%	907,182 61.3%	13,522 0.9%	284,712 19.2%	1,194,958 80.8%	0 0%	- -%	- -%	- -%
miranda	5	5,268,186 47,118	39,583 0.7%	5,225,272 99.2%	3,331 0.1%	5,990 0.1%	5,262,196 99.9%	0 0%	16,828 0.3%	5,205,512 98.8%	45,846 0.9%
spider	5	9,072 756	6,368 70.2%	2,704 29.8%	0 0%	6,312 69.6%	2,760 30.4%	0 0%	4,000 44.1%	2,704 29.8%	2,368 26.1%

Table 2: Primitive type inference results. Out of the total number of bytes allocated during 5 executions, how many were typed, untyped (⊤), and conflicted (⊥), using the 3 different typing algorithms. The missing numbers take over 60 hours to analyze each trace.

			TIPO-A					TIPO-C				
	Runs	Callsites (stb/dynarr/unstb)	Types (ptr)	Types (final)	Typed	⊤	⊥	Types (ptr)	Types (final)	Typed	⊤	⊥
bhbmk	5	23 (22/1/0)	23	19	83.0%	17.0%	0%	23	19	81.5%	18.5%	0%
healthbmk	5	16 (15/1/0)	11	9	81.0%	19.0%	0%	11	9	78.6%	21.4%	0%
miranda	5	126 (98/28/0)	112	82	0.7%	99.3%	0%	115	85	0.3%	99.7%	0%
pidgin	5	170 (129/41/0)	161	123	17.1%	82.8%	0.1%	161	125	11.6%	88.4%	0%
spider	5	1 (1/0/0)	1	1	100%	0%	0%	1	1	100%	0%	0%

Table 4: Global analysis results. It includes the number of unique callsites, as well as the number of type trees obtained by merging callsite trees by pointer target type.

7.2 Global Analysis

Table 4 presents the global analysis results on the buffer trees produced in the 5 program runs for each program in §7.1. The third column shows the number of callsite trees produced by merging the buffer trees by callsite and the split into stable, dynamic arrays, and unstable. It shows that the number of callsite trees is much smaller than the number of buffer trees and significantly larger for the programs where more modules are analyzed (Miranda and Pidgin). Surprisingly, there is a single callsite in the spider game. It also shows that most callsites are stable, i.e., they allocate buffers of a single size. The remainders are dynamic arrays, which mostly correspond to strings and containers such as vectors. The fourth column shows the number of trees produced by merging callsite trees using the pointer target type information. The number of trees reduces for 3 of the 5 programs. We have verified the correctness of the merging by pointer target type using the ground truth for those programs with debugging symbols.

The last step in global analysis is to cluster the remaining trees using their structure and profiling information. To select the best weights for clustering we use the ground truth from healthbmk and run the clustering with different weights by increasing the format weight from 0 to 1 in 0.01 increments. We evaluate each clustering using 3 standard external clustering validity measures: the Rand index, the Jaccard index, and the FM index [23]. All three metrics are maximized when the format weight is 0.86 and the profiling weight is 0.14. Columns 5 and 10 show the final number of type trees output by the global analysis for each algorithm. The clustering reduces the number of trees in all programs, except, obviously, spider. The other columns capture the number of bytes in the final type trees that are typed, untyped, and conflicted. For some programs, these numbers are smaller than the ones in Table 2. This means that there is a large number of buffers which were typed to a large extent. Once merged, the percentage of bytes typed in the remaining trees reduces. However, each merging step produces a finer tree as partial information is accumulated.

Our manual analysis concludes that the clustering is conservative: it does not merge trees of different types; however, it fails to merge some trees of the same type. For example, Figure 10 shows the inferred type graph for Healthbmk. The dynamic array corresponds to a vector container from the Microsoft STL implementation. Nodes with the same shade of gray should have been merged. The two shades correspond to the head and internal nodes of a doubly-linked list. The nodes that were not merged (the gray ones with *1 in their name) belong to lists that were always empty in the 5 runs. Thus, they are used differently, so the clustering does not merge them. However, the resulting type graph closely resembles the one obtained from the source code definitions.

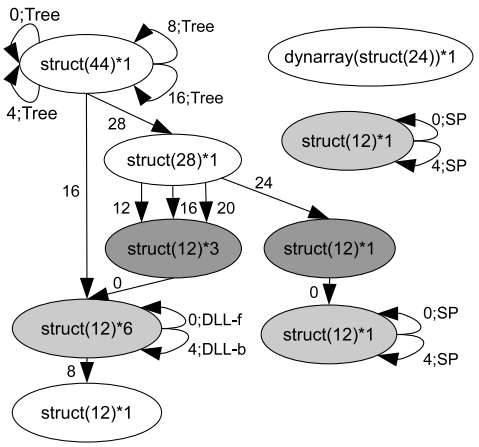

Figure 10: The inferred type graph for healthbmk. Each node is labeled with the root type times the number of callsites it corresponds to.

Program	SLL	DLL	CDLL	Tree	SP	Other
bhbmk	0	0	0	1	0	0
healthbmk	0	0	1	1	4	0
miranda	1	0	2	0	0	0
pidgin	2	5	0	6	1	2
spider	0	0	1	0	0	0

Table 5: Shape Analysis Results.

7.3 Dynamic Shape Analysis

Every 200,000 instructions, the execution analysis produces a heap graph using the inferred pointers. Once a type graph is inferred, all heap graphs for a program are fed to the dynamic shape analysis. Table 5 shows the inferred shapes from over 1,000 heap graphs. All 5 programs use recursive data structures. The most common shapes are lists and trees. Doubly-linked-lists (CDLL and DLL) are more common than singly-linked-lists. The 5 self-pointers (SP) correspond to recursive pointers in empty lists. In Pidgin, a cycle and a DAG are also identified. The only data type seen in the spider game turns out to be the node of a doubly-linked-list. Each card stack visible in the GUI corresponds to a DLL. We note that Healthbmk uses the List container from the Microsoft C++ STL, which our analysis identifies as being implemented using a CDLL.

7.4 Signatures

The signature generator creates the signature by extracting the set of types reachable from a given root type. To select the root type for both Pidgin and Miranda, we use the taint information in the snapshot (from the contacts that we manually introduced) and select as signature root the callsite (or the type that this callsite was merged into) of the buffer that dominates most tainted buffers. For bhbmk and healthbmk since they build large trees in memory we select the tree root type as signature root. For spider, we choose its only type. For bhmbk, healthbmk, and spider we clone the selected root type into another type where we add an invariant to avoid matching an empty tree or list.

Program	Trees	Value Inv.	Ptr	CI
bhbmk	9	6 (1/3/0/2)	32 (22/2/8)	0
healthbmk	8	18 (0/7/5/6)	25 (25/0/0)	1
miranda	3	3 (1/1/0/1)	4 (4/0/0)	0
pidgin	65	41 (10/13/7/11)	276 (223/27/26)	2
spider	2	4 (1/1/0/2)	4 (4/0/0)	1

Table 6: Summary of generated signatures.

Table 6 summarizes the generated signatures including the number of type trees in the signature (2^{nd} column), the number of value invariants and their split into "Constant/Non-Zero/Set/Range" (3^{rd} column), the number of pointers and their split into pointers with "one target / no target / multiple targets" (4^{th} column), and the number of implicit cycle invariants (CI).

Table 7 shows the results of matching the signatures on a snapshot taken at the end of an execution. It shows the number of pages in the snapshot, the number of instances of the root data structure in the snapshot (from the ground truth), and the number of detections, false positive rate, and false negative rate for 3 types of signatures: hybrid, value-invariant only, and points-to only. The results show that the hybrid signatures produce no false negatives, e.g., the Pidgin signature successfully identifies the 10 user contacts in memory and the Miranda signature identifies the data structure holding all properties of the ICQ protocol, including the ICQ contacts. They also show no false positives for 4 of the 5 programs (all but Miranda). For those 4 programs, the value invariant and points-to only signatures detect the instances but produce a large number of false positives. This shows that *signatures need to comprise as much discriminating information as possible; it is not enough to use only value-invariants, points-to relationships, or cycle invariants*.

The main reason for the false positives are areas of memory filled with zero values. Zero is problematic because it can match pointers, integers, and floats. As such it is important to have value-invariants that determine if a pointer has values different than zero for the points-to relationships and cycle invariants to work. For the Miranda signature all signatures produce a large number of false positives. We have built a signature manually from the source code for this data structure but achieved no better results. This shows that there are data structures for which signatures are difficult to build since there is not enough discriminating power.

Overall, this results shows that hybrid signatures work better than prior value-invariant and points-to signatures, minimizing false positives, and that some data structures signatures are challenging to build even when source code, debugging symbols, and profiling information are available.

Program	Pages	Inst.	Hybrid Signature			Value Invariant Signature			Points-to Signature		
			Det.	FP%	FN%	Det.	FP%	FN%	Det.	FP%	FN%
bhbmk	226	1	1	0%	0%	20	95%	0%	58,964	99.99%	0%
healthbmk	226	5	5	0%	0%	15	66.7%	0%	58,767	99.99%	0%
miranda	2,040	1	138,077	99.99%	0%	227,315	99.99%	0%	794,944	99.99%	0%
pidgin	3,697	10	10	0%	0%	798,170	99.99%	0%	1,217,394	99.99%	0%
spider	701	10	10	0%	0%	1,940	99.5%	0%	197,678	99.99%	0%

Table 7: Signature matching results.

8 Related Work

This section discusses related work on data structure recovery, data structure signatures, shape analysis, and typing the heap.

Data structure recovery. One approach to data structure recovery uses static analysis. Aggregate Structure Identification [45] decomposes aggregate data structures in Cobol programs using the program's access patterns and the type information from functions with a known prototype. Since source code and debugging symbols may not available, other work applies static analysis on x86 binaries [1,47]. More recently, TIE [28] infers the type of the program's variables from its binary using instruction type sinks. We draw inspiration from TIE for our primitive type lattice but in our dynamic approach solving a constraint system at each deallocation would be too expensive.

LAIKA [12] uses a machine learning approach to identify data structures in a memory snapshot and cluster those of the same type. This information is used to compare how similar two snapshots are. Our work differs on the use of dynamic analysis, which enables aggregating information from multiple executions and memory snapshots.

Another approach recovers types and data structures using dynamic analysis. Guo et al. [22] propose an algorithm for inferring abstract types by partitioning variables into equivalence classes. Our primitive type inference algorithm is similar in that it assigns each used variable a fresh index, but it does not use unification and recovers primitive types. DISPATCHER [7] follows ASI in leveraging the type information from functions with a known prototype for type inference, but uses a dynamic data-flow approach. It reverse-engineers the structure of the buffer holding a message about to be sent on the network, which corresponds to the message structure. REWARDS [33] uses a similar approach but proposes an algorithm that types the program's internal data structures, rather than a single buffer. Our primitive type inference algorithm differs from REWARDS in that it works online, does not use unification, uses a well-defined lattice, and parametrizes pointers by their target type. HOWARD [50] recovers data structures and arrays using pointer stride analysis, but does not recover type information other than pointers. POINTERSCOPE [53] infers pointer and non-pointer types using a variant of the traditional W algorithm [13] that constrains unification. ARTISTE differs from prior dynamic techniques in that it addresses the full data structure recovery process from primitive field types up to recursive data structures.

Data structure signatures. Tools like PTFINDER [49], VOLATILITY [51], and MEMPARSER [4] scan memory images using manually generated value invariant signatures to identify instances of data structures of interest. Baliga et al. [2] propose automatically generating value invariant signatures from the program's type definitions using an approach introduced in DAIKON [17]. Gavitt et al. [15] harden value invariant signatures by removing fields that do not affect the program's processing. SIGGRAPH [32] proposes graph signatures, which capture information about the points-to relationships between data structures. DIMSUM [31] uses probabilistic inference to build graph signatures that work in physical memory snapshots for which there is no page mapping available. Liang et al. [30] manually build signatures that capture cycle invariants.

ARTISTE differs from these prior work in that it does not require access to the type definitions in the source code and that it automatically generates hybrid signatures, which combine points-to relationships, value invariants, and cycle invariants, offering higher discriminating power.

Shape analysis. There has been a wealth of prior work on static shape analysis [3, 11, 20, 34, 35, 48]. These techniques require flow and context-sensitive analysis, which makes them expensive and necessarily conservative. They also require access to the program's source code. Our partitioning of the heap graph into regions is based on the work of Marron et al. [35] and related to other approaches that summarize the state of a heap snapshot [36, 38].

Another approach identifies recursive data structures that appear in a particular program execution. Raman and August [46] profile the memory access behavior of programs that use recursive data structures, but do not classify them into shapes. Pheng and Verbrugge [42] visualize the evolution of data structures classifying a recursive data structure as Tree, DAG, or cycle, which corresponds to our coarse-grained classification. The most related work on dynamic shape analysis is by Jump et al. [26]. Their approach to classify recursive data structures into shapes first manually builds degree signatures for a number of prevalent recursive data structures. These signatures are matched against degree summary graphs built for each program class by piggybacking the Java garbage collector during heap traversal. Our dynamic shape analysis differs in that we partition the heap and identify the shape of each region, rather than aggregat-

ing information per class. Thus, our approach can identify different lists in a heap graph, not only the fact that a certain class implements a list.

Typing and traversing the heap. Polishchuk et al. [44] address the problem of obtaining a consistent typing for the objects in the program's heap. This problem is relevant in the context of rootkit detection to traverse the kernel's dynamic memory [2, 10, 25]. These works rely on the availability of the program's type definitions. ARTISTE can enable such traversal when definitions are not available.

9 Conclusion

We have presented ARTISTE, the first tool for automatically generating data structure signatures without the program's source code or debugging symbols. Unlike prior work that generates signatures containing only points-to relationships, value invariants, or cycle invariants, ARTISTE generates hybrid signatures that combine all of them, significantly minimizing false positives. ARTISTE incorporates a number of novel techniques for data structure reverse engineering including: (1) two new online algorithms that improve the accuracy and speed for primitive type inference, (2) a clustering approach that uses points-to, structural, and profiling information for identifying objects of the same type allocated at different program points, and (3) a novel dynamic shape analysis that precisely recognizes the recursive data structures a program uses, classifying them by their shapes. Our experimental results with a number of binary programs show that ARTISTE's hybrid signatures achieve close-to zero false positives and false negatives when used to find the data structure instances of interest in memory.

References

[1] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, Mar. 2004.

[2] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, Aug. 2008.

[3] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, July 2007.

[4] C. Betz. http://sourceforge.net/projects/memparser/.

[5] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. Openconflict: Preventing real time map hacks in online games. In *S&P*, May 2011.

[6] J. Caballero. *Grammar And Model Extraction For Security Applications Using Dynamic Program Binary Analysis*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, September 2010.

[7] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *CCS*, Nov. 2009.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, Dec. 2008.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *CCS*, Oct. 2006.

[10] M. Carbone, W. Cui, Long, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *CCS*, Nov. 2009.

[11] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, June 1990.

[12] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *OSDI*, Dec. 2008.

[13] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, Jan. 1982.

[14] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond *k*-limiting. In *PLDI*, June 1994.

[15] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *CCS*, Nov. 2009.

[16] J. C. Dunn. Well-separated clusters and optimal fuzzy partitions. *Journal of Cybernetics*, 4(1), 1974.

[17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3), Dec. 2007.

[18] B. U. G. Lomow, J. Cleary and D. West. A performance study of time warp. In *SCS Multiconference on Distributed Simulation*, February 1988.

[19] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, Feb. 2003.

[20] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL*, Jan. 1996.

[21] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, June 2005.

[22] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA*, July 2006.

[23] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17(2-3), Dec. 2001.

[24] P. Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2), 1912.

[25] N. L. P. Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS*, Oct. 2007.

[26] M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM*, June 2009.

[27] L. Kaufman and P. J. Rousseeuw. *Finding Groups In Data: An Introduction To Cluster Analysis*, volume 4. Wiley-Interscience, 1990.

[28] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, Feb. 2011.

[29] Y. Li and Z. Chenguang. A metric normalization of tree edit distance. *Frontiers of Computer Science in China*, 5(1), Mar. 2011.

[30] B. Liang, W. You, W. Shi, and Z. Liang. Detecting stealthy malware with inter-structure and imported signatures. In *ASIACCS*, Mar. 2011.

[31] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu. Dimsum: Discovering semantic data of interest from unmappable memory with confidence. In *NDSS*, Feb. 2012.

[32] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, Feb. 2011.

[33] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, Feb. 2010.

[34] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, Jan. 2005.

[35] M. Marron, D. Kapur, and M. Hermenegildo. Identification of logically related heap regions. In *ISMM*, June 2009.

[36] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting runtime heaps for program understanding. In *arxiv*, 2012.

[37] Miranda im. http://www.miranda-im.org/.

[38] N. Mitchell. The runtime structure of object ownership. In *ECOOP*, July 2006.

[39] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2), Apr. 1980.

[40] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, Feb. 2005.

[41] N. Petroni, A. Walters, T. Fraser, and W. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation Journal*, 3(4), Dec. 2006.

[42] S. Pheng and C. Verbrugge. Dynamic data structure analysis for java programs. In *ICPC*, June 2006.

[43] Pidgin: The universal chat client. http://www.pidgin.im/.

[44] M. Polishchuk, B. Liblit, and C. W. Schulze. Dynamic heap type inference for program understanding and debugging. In *POPL*, Jan. 2007.

[45] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, Jan. 1999.

[46] E. Raman and D. I. August. Recursive data structure profiling. In *Workshop on Memory System Performance*, June 2005.

[47] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *CC*, Mar. 2008.

[48] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, Jan. 1999.

[49] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. In *DFRWS*, Aug. 2006.

[50] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, Feb. 2011.

[51] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. http://www.volatilesystems.com/default/volatility.

[52] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6), Dec. 1989.

[53] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin. Identifying and analysing pointer misuses for sophisticated memory-corruption exploit diagnosis. In *NDSS*, Feb. 2012.