Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Automatic Inspection of Program State in an Uncooperative Environment

José Wesley de Souza Magalhães[1] | Chunhua Liao[2] | Fernando Magno Quintão Pereira[1]

[1]Computer Science Department, UFMG, Belo Horizonte, Brazil
[2]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, California, United State of America

Correspondence
Antonio Carlos Avenue, 6627. Belo Horizonte, MG/Brazil. Email: fernando@dcc.ufmg.br

Summary

The program state is formed by the values that the program manipulates. These values are stored in the stack, in the heap, or in static memory. The ability to inspect the program state is useful as a debugging or as a verification aid. Yet, there exists no general technique to insert inspection points in type-unsafe languages such as C or C++. The difficulty comes from the need to traverse the memory graph in a so-called uncooperative environment. In this paper, we propose an automatic technique to deal with this problem. We introduce a static code transformation approach that inserts in a program the instrumentation necessary to report its internal state. Our technique has been implemented in LLVM. It is possible to adjust the granularity of inspection points trading precision for performance. In this paper, we demonstrate how to use inspection points to debug compiler optimizations; to augment benchmarks with verification code; and to visualize data structures.

KEYWORDS:
Inspection point, debugging, program state

1 | INTRODUCTION

The state of a program is the ensemble of values that the program manipulates. Said values are stored in memory regions that can be reached through program structures like pointers or activation records. The ability to inspect this state is useful for reasons including debugging, verification and visualization[1]. The identification of the memory regions that constitute the program state is a solved problem for type-safe languages that distinguish pointers from other types. As testimony to this fact, the identification of program state is the basis of mark and sweep garbage collectors[2,3]. However, identifying program state is difficult in type-unsafe languages. These languages include not only C, C++ and mainstream assemblies, but also the unsafe parts of otherwise type-safe languages such as Java, C# and Racket[4]. C and C++ are commonly known in the garbage collection community as *uncooperative*[5]. They own this qualifier to a weak type system, that neither associates size information with memory regions, nor distinguish pointers from scalars. Although there exist garbage collectors for languages like C or C++[6,7,8,9,10], such implementations are not mainstream. The more reliable these garbage collectors are, the heavier their overhead.

Contributions

We demonstrate that ideas previously used in the implementation of garbage collectors for type-unsafe languages can be used to inspect program state in these languages. With this goal in mind, we bring, from the systems community into the software engineering community techniques to create *Program Inspection Points*, a notion that we define in Section 2. Like in classic garbage collectors for C/C++[6,7,8,9,10], inspection points impose no overhead on programs unless they need to be inspected; Moreover, like in that line of work, we are willing to accept some imprecision: the inability to distinguish pointers from integers,

for instance, might prevent the inspection of some parts of the heap. Yet, in contrast to garbage collectors, we provide the program with the means to associate low-level data—heap and stack allocated memory, for instance—with high-level source-code information: user-defined names and line locations.

Customization

The goal of this paper is to show how to inspect the internal state of programs in settings that lack runtime type information. To harmonize precision and performance, we adopt *customization*: the ideas to be discussed in this paper can be adopted at different levels of granularity. These levels of granularity determine which forms of memory allocation can be tracked: static, stack or heap, and how deeply the graph formed by the points-to relations can be traversed. This paper shows how this customization is achievable within a unified framework, which we call WHIRO. WHIRO has been implemented on top of the LLVM compiler [11]. It relies on the compiler only—it does not depend on the operating system or the architecture. Thus, WHIRO can be used even in embedded devices that lack support of inspection tools like Valgrind [12], which is architecture-dependent, or GDB [13], which is operating system-dependent, or the OpenSmalltalk debugger [14], which relies on a virtual machine simulator.

As we explain in Section 3, WHIRO moves to compilation time as much computation as possible to preserve the performance of inspected programs. To resist the effects of compiler optimizations, WHIRO is implemented at the level of the compiler's intermediate representation—it requires no interventions on the source code of programs. Section 3.3 explains how WHIRO uses high-level debugging information to provide users with meaningful reports about the program's internal state. At its maximum granularity, WHIRO can inspect every block of memory allocated on the heap, without interfering with the original semantics of the instrumented program, as Section 3.4 explains.

Design Decisions

During the design of WHIRO, we were faced with several questions whose answers were not immediately evident to us. Section 4 discusses some of these challenges, explaining the design decisions taken throughout the implementation of WHIRO. In particular, we discuss how WHIRO deals with aspects of the C and C++ programming languages that make them uncooperative, such as pointer arithmetics, lack of size information associated with arrays and the weak type system, which does not tag values with type information. Some of our solutions to deal with these characteristics of C and C++ were motivated by pragmatism. For instance, WHIRO can traverse the graph formed by points-to relations, as long as pointers are declared as such. If scalar types are casted into pointers, then WHIRO is still able to inspect the contents of memory; however, inspection assumes that these pointers dereference a single cell, even when they refer to an array of multiple cells. Another difficulty that emerged during the project of WHIRO was related to compiler optimizations: in general, LLVM's debugging information is enough to ensure that the program state of optimized programs can be inspected; however, transformations such as array scalarization removes this information. Thus, WHIRO's implementation had to identify and repair the missing data in this case.

Applications

Section 5 shows several applications that can be built on top of WHIRO, including debuggers, benchmark synthesizers and program visualizers. These applications use different customizations of WHIRO, which are listed in Section 5.1. The possibility to customize the amount of program state that WHIRO tracks is key to producing practical applications. For instance, to debug compiler transformations (Sec. 5.2), we need to inspect every memory location affected by the program, regardless of its usage e.g., as a pointer or as a scalar, or its location e.g., static memory, stack or heap. To synthesize verification code for benchmarks (Sec. 5.3), we need to preserve their performance; for instance, printing only the state of local variables at the end of program execution. However, in scenarios where maintaining the performance of the original program is not essential, WHIRO can use much more precise memory-tracking techniques. As an example, to visualize data structures (Sec. 5.4), WHIRO can traverse the entire graph formed by pointers to locations in the heap.

Summary of Results

Section 6 evaluates the precision and the overhead of WHIRO. At its maximum precision, our implementation keeps track of every memory address that has a corresponding allocation point in the program's source code: static, local and heap allocated variables, including aggregates such as arrays and structs. At this level, we observe an average slowdown of 1.34x on MIBENCH programs [15], an increase of memory usage of 1.63x, and an increase of static executable size of 1.48x. However, precision is customizable. For instance, when used to synthesize verification code for benchmarks, we observe no performance regression, an increase of memory usage of 1.07x, and an increase in code size of 1.06x. WHIRO is open software, distributed under the GPL 3.0 License, and can be retrieved at https://github.com/JWesleySM/Whiro.

2 | CORE DEFINITIONS

The goal of this paper is to let users observe the state of the memory manipulated by a program at different points of its execution. A summary of our modus operandi follows. First, users select static inspection points in the source code of a program, like A-D in Figure 1. Second, WHIRO observes the variables at the different calling contexts (dynamic inspection points) that might exist at those locations. Figure 2 shows these dynamic contexts. At each one of these points, WHIRO produces an image of the program state, relating the contents of memory locations with the names of variables in the source code of the program. Figure 3 shows snapshots produced for two inspection points at different calling contexts. In the rest of this section, we define the concepts just mentioned, starting with the notion of a *Static Inspection Point*.

Definition 1 (Static Inspection Point – SIP)**.** Given a program P written as a sequence of statements, a static inspection point is a point following a statement of P.

```
01 int numNodes = 0;
02
03 struct Node {
04   int data;
05   struct Node* next;
06 };
07
08 struct Node* create(const int data, struct Node* next) {
09   struct Node* p = (struct Node*)malloc(sizeof(struct Node));
10   p->data = data;
11   p->next = next;
12   numNodes++;
   ------------------------------------------ [A]
13   return p;
14 }
15                                                  [B]
16 struct Node* incAll(struct Node *head) {         |
17   if (head) {                                    |
   -------------------------------------------------
18     return create(head->data + 1, incAll(head->next));
19   } else {
   ------------------------------------------ [C]
20     return NULL;
21   }
22 }
23
24 int main(int argc, char** argv) {
25   struct Node *n = NULL;
26   struct Node *m = NULL;
27   if(argc > 1){
28     for (int i = 0; i < argc; i++) {
29       n = create(i, n);
30     }
31   }
32   if(n){
33     m = incAll(n);
34   }
   ------------------------------------------ [D]
35   return 0;
36 }
```

FIGURE 1 A program with four static inspection points marked with letters A-D.

Example 1. The program in Figure 1 implements a function that recursively traverses a list duplicating its nodes while incrementing the values in said nodes. We have marked four inspection points in this program, labeled as A, B, C and D.

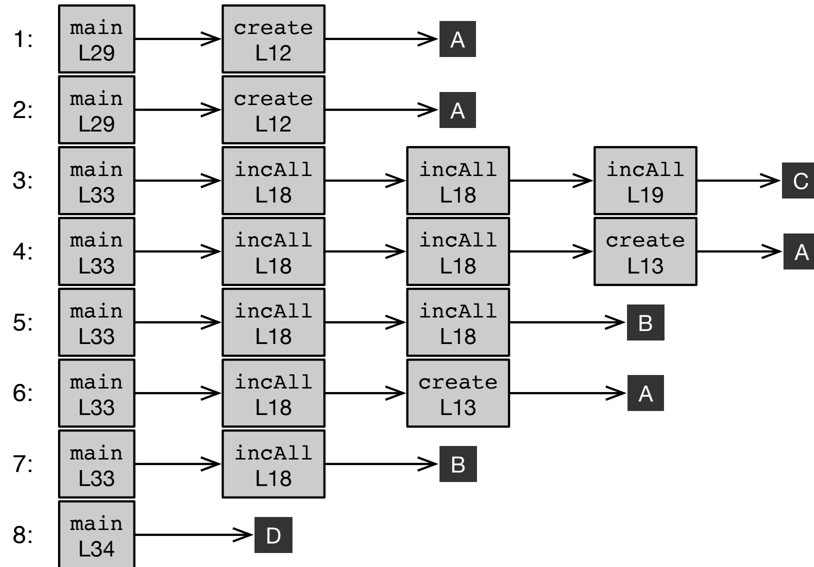

FIGURE 2 Dynamic contexts produced by the program in Figure 1, with the input "`main a`".

Definition 2 (Dynamic Inspection Point – DIP)**.** A dynamic inspection point is a static inspection point, plus a *calling context* when that point executes. The calling context of an invocation of a function f is determined by the list of functions active when f was invoked.

Example 2. Figure 1 shows four static inspection points. If we execute that program with the command line "`main a`", then we shall observe the eight dynamic inspection points seen in Figure 2. We have one dynamic inspection point for each time that a static inspection point is traversed during the execution of the program. Thus, the same static inspection point might yield a large number of different dynamic inspection points.

2.1 | Program States

We define as *state* the set of values of local, static and heap-allocated variables of a program at a given dynamic inspection point. The *Visible Program State* is the subset of the program state that is reachable from variables visible at the scope of that dynamic inspection point:

Definition 3 (Visible Program State)**.** The visible state of a program at a dynamic inspection point p is a map from the *program symbols* visible at the scope of p to values. *Program Symbols* are the names of program variables.

The notion of visible state differs from the notion of *Reachable State* commonly adopted in the description of trace-based garbage collectors. The reachable state of a program (see, for instance, Aho et al.[16] Sec.4.6.6) is formed by memory allocated statically or on the stack, plus the memory in the heap referenced by any value in the reachable state (including the heap itself). The visible state differs with regards to stack allocated variables. Only variables in the scope of the active function (the function on the top of the call stack) are considered visible.

Example 3. Figure 3 shows the visible program state at two dynamic inspection points during the execution of the "`main`" function from Figure 1, with string "`a`" as input. The visible program state is represented as a graph, in which nodes are memory locations, and edges denote points-to relations. The two graphs in Figure 3 correspond to the fourth and seventh dynamic inspection points seen in Figure 2. Concerning the latter, for instance, we have a variable that is statically allocated (`numNodes`); two variables that are allocated on the stack (`head` and `retn`); and a number of memory blocks allocated on the heap. These heap-allocated blocks form the two linked lists that exist once the function `incAll` returns, in Line 18 of Figure 1. The stack-allocated variable `retn` is not a visible program symbol, i.e., it is not a user-defined name in Figure 1. It represents the local variable that points to the value returned by `incAll`. Notice that the seventh dynamic inspection point also includes the activation

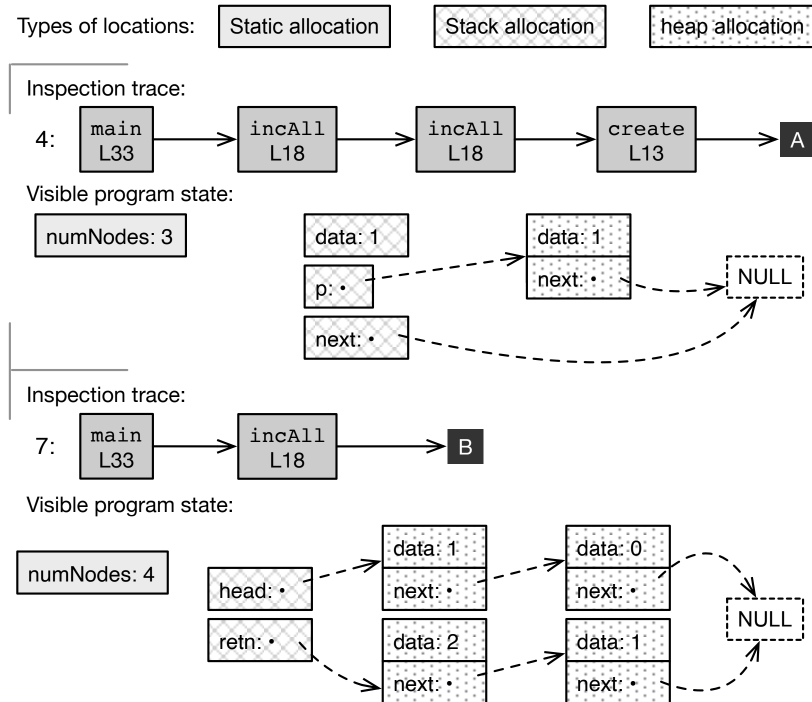

FIGURE 3 Visible program states at two different dynamic inspection points from Figure 2. Symbol "`retn`" is the auxiliary variable that holds the return value of function `incAll` at Line 18 of Figure 2.

of function `main`; however, the local variables of `main` are not part of the visible state in the static inspection point B (Line 18 of Figure 1). Therefore, variables like `m` and `n` (declared within `main`) are not depicted in the graphs that Figure 3 shows. Even though they are on the stack when `incAll` is active, they are not visible in the scope of that function. Had we shown the visible state at inspection point D (which is within function `main`), for instance, then these variables would be present.

The reason to distinguish visible from reachable state is pragmatic. The difference between visible and reachable states does not compromise WHIRO's ability to inspect values created by recursive functions. Variables that are not in scope cannot be modified (within the defined semantics of C, for instance[17]). Once these invisible variables become active, e.g., when their activation record reaches the top of the calling stack, they can be inspected. Nevertheless, the static memory and the entire heap (including parts reachable only via invisible variables) are still tracked.

3 | TRACKING STATE

To track the visible program state, we use a data structure henceforth called *memory monitor*. Definition 4 introduces the several parts that constitute a memory monitor. Before discussing each one of these parts, one explanation is in order. WHIRO instruments programs in the Static Single Assignment (SSA) format[18]. SSA is an intermediate representation adopted in many different compilers, including LLVM, the underlying infrastructure on top of which WHIRO is built. The core characteristic of SSA-form programs is the fact that each variable name has only one definition site. In other words, each one of the multiple definition sites that a variable might have in the source code gives origin to a new variable name in LLVM's intermediate representation. Therefore, for each source variable v there might exist several SSA definitions, which are henceforth called $def(v)$.

Definition 4 (Memory Monitor)**.** The memory monitor is a data structure given by a tuple (G, S, H, T), such that:

G, S**:** Maps *of* global or stack `Symbols` *to* (`MetaVar`, `Trace`)

H**:** Set *of* `Heap Addresses`

T**:** Set *of* `Type Descriptors`

Where:

`MetaVar` consists of name and type of a variable;

`Trace` is List *of* (`ProgramLoC`, `SSADef`);

`SSADef` is the definition point of some expression in an SSA-form program;

`ProgramLoC` is a program point, i.e., a program instruction or declaration in the LLVM intermediate representation.

`Type Descriptor` is a metadata that specifies the name and the format specifier of a type in the source code.

In Definition 4, G maps global variables to debugging information. S is analogous, except that it maps stack-allocated variables to debugging information. There exists one table S per program function—each table stores information related to variables in the scope of a function. Thus, we represent each table S as S_f, where f is the name of the function that the table represents. H is the set of addresses of memory blocks allocated in the heap. T holds metadata describing every type in the target program.

3.1 | Static Components of the Memory Monitor

The G (global) and S (stack) components of the memory monitor exist only at compilation time. They do not exist when the target program executes. Their goal is to guide the insertion of instrumentation code in the program. Said code serves two purposes: (i) report program data to the user; and (ii) update the heap map H. The following parsing events cause updates of these structures (at instrumentation time):

1. Declaration of global variable v with type t: an entry $(def(v), t)$ is created in G. The term $def(v)$ is the SSA-form name of the variable v. It distinguishes this definition of v from other uses of the same name to declare different variables, e.g., local variables also called v. Since global variables have their address determined at compilation time, $def(v)$ for a global variable v represents its address. Such addresses are visible in every program point, thus the assignments to global variables in the program are not tracked. To inspect a variable statically allocated, the Memory Monitor always uses the address definition.
2. Declaration of a local variable v with type t within function f: an entry $(def(v), t)$ is created in S_f, where S_f is the table of stack symbols associated with function f. The term $def(v)$ is the SSA-form name of variable v. Notice that the same variable name can be redefined multiple times within the same function; however, each redefinition will have a different SSA-form name.
3. Assignment $v = u$ at line ℓ of function f: the pair $(\ell, def(u))$ is appended to $(def(v), t)$. The label ℓ is the line, in source code, where variable v is defined, and the term $def(u)$ is the SSA-form name of the variable that appears on the right side of the assignment. If u is a constant, then its SSA-form name is the constant itself; otherwise, it is a unique representation of the program location where u is defined.

Example 4. Figure 4 shows S and G for the program in Figure 1. G contains data for global variable `numNodes`. Table $S_{\texttt{main}}$ refers to function `main`. We omit the other two stack tables, e.g., $S_{\texttt{create}}$ and $S_{\texttt{incAll}}$, to save space. Notice that what we call an SSA definition, once mapped back into the source code, might not correspond to a variable name. For instance, the SSA definition associated with the global variable `numNodes` is the constant zero, and the SSA definition associated with the local variable `n` is the expression `create(i, n)`. This expression, in the low-level LLVM representation, will have a name, which is the name of the auxiliary variable used to store its result. Thus, the SSA definition associated with `create(i, n)` is the SSA name of that auxiliary variable. WHIRO uses debugging information to map this SSA definition back to the expression `create(i, n)`, which is present in the source code of the program.

3.2 | Dynamic Components of the Memory Monitor

The heap table H and the type table T exist during program execution. We call these two tables the *Auxiliary Program State*. Table T is constructed statically and is immutable. It contains *type descriptors*. Descriptors are *flyweights*, meaning that there

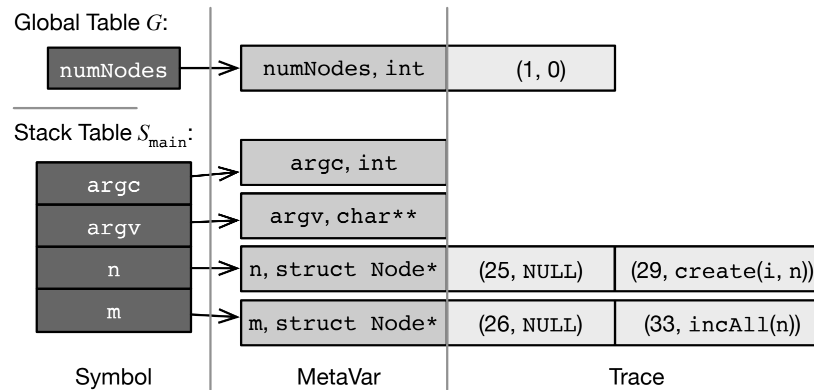

FIGURE 4 Global and stack tables for the program in Fig. 1. We show only the stack table for `main`.

exists one per type in the program's source code. T lets WHIRO print non-scalar variables, for it associates every type with an output format. It also allows WHIRO to navigate the program's visible state, indicating fields of aggregate types that are pointers. For product types (e.g., C-like `structs`) the descriptor consists of the name, format, and the offset of each field. For pointers and qualified types (`const`, `volatile`, etc), the descriptor also contains an index to access the descriptor of the base type. This recursive nature can represent any composite type.

Example 5. Figure 5 shows the type table for the program earlier seen in Figure 1. The table contains seven entries, one for each type used in Figure 1, namely: `int`, `struct Node*`, `struct Node`, `const int`, `char**`, `char*` and `char`. Each entry is associated with a type descriptor. Type descriptors for product types, e.g., `struct`, contain multiple blocks, one per field in the product. Blocks contain four attributes: `N`, `F`, `O` and `B`. The first attribute, `N`, is the name of the field within a product type. For non-product types, `N` is the empty string. The second attribute, `F`, is the format used to print information associated with variables of that type. We adopt the nomenclature used in the C programming language. For instance, integer variables are printed using the `%d` format, and pointers use the `%p` format. The third attribute, `O`, is the offset of a field within a product type. `O` is non-zero only for `struct` fields other than the first field. Finally, the last attribute, `B`, is the base type of a qualified type. For instance, the third entry in Table T in Figure 5 refers to `const int`; thus, its base type is `int`. This base type is associated with the first entry of T (`B = 0`).

	0	1	2	3	4	5	6
	int	pointer	Node	const	pointer	pointer	char
N:	""	""	"data"	""	""	""	""
F:	%d	%p	%d	%d	%p	%p	%c
O:	0	0	0	0	0	0	0
B:	0	2	0	0	5	6	6
N:			"next"				
F:			%p				
O:			8				
B:			2				

FIGURE 5 Type Table T created—at compilation time—for the program in Figure 1. The keys in the type table are N = name; F = format; O = offset and B = base type.

The *Heap Table* H, in contrast to the type table, changes during the execution of the program. Instrumentation inserted in the target program updates H. Each entry in the heap table contains an index to a descriptor in the type table. H also keeps track of

freed heap addresses, which are considered unreachable data. If a freed address is re-allocated by the memory manager, WHIRO sets the corresponding entry in H as reachable again, and updates the reference to T according to the type being allocated. The following events cause the heap table to be updated (at running time):

1. Memory is allocated, e.g., $v = \texttt{malloc(Tp)}$[1]: the monitor creates an entry $H[v]$ (if one does not exist), sets it as reachable and associates this descriptor with $T(\texttt{Tp})$. Notice that v contains a *memory address*—in this case the address returned by `malloc`. Hence, the Heap Table is indexed by addresses.
2. Memory is freed via $\texttt{free}(v)$. WHIRO sets $H[v]$ as unreachable. Again, notice that the contents of v—a memory address—is used to index H. Thus, a sequence like $v = \texttt{malloc(Tp)}; u = v; \texttt{free}(u)$ will have the same effect upon H as $v = \texttt{malloc(Tp)}; \texttt{free}(v)$.

Example 6. Figure 6 shows the auxiliary state after the program in Figure 1 executes the inspection point B for the second time. H contains four heap-allocated blocks corresponding to the nodes of two linked lists. Each block contains an index to access the type table; hence, WHIRO is able to find the type of memory chunks. The "Vis" attribute indicates whether that block was visited when traversing the heap graph, as section 3.3 shall explain.

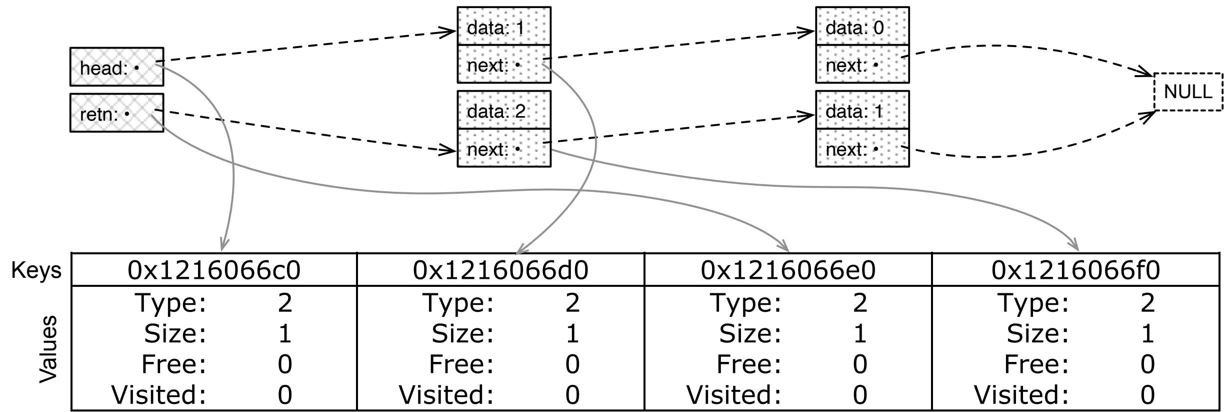

FIGURE 6 Heap Table H that exists at the seventh dynamic inspection point in Figure 3. The field "Visited" in the heap table is used only during information retrieval. It indicates if a node has been traversed when reporting the internal state of a program. This precaution ensures that the contents of blocks pointed to by multiple variables are only reported once.

3.3 | Information Retrieval

WHIRO inserts code at inspection points to retrieve the visible program state. Data is printed as either a textual log, or as a DOT graph (see Section 5.4). The rest of this section explains how WHIRO retrieves this information.

Inspection Traces. When the inspection point is logged, only one SSA definition $def(v)$ is valid for a given variable name v. However, the global map G or the stack table S_f associated with a function f can contain multiple definition sites for the same variable. Given a variable v, let $\texttt{Trace}(v)$ be the set of SSA definitions bound to v, either on the global table or on the stack table (see Definition 4). When reporting the state of v at a static inspection point p, WHIRO finds the definition $(\ell, def(v)) \in \texttt{Trace}(v)$ using dominance. Given two program points q and p within the same function f, we say that q dominates p if every path from the entry point of f to p goes across q. In a well-formed SSA-form program, the definition point of a variable dominates all the sites where the variable is used; and every use of a variable is reached by a single definition site. Therefore, the definition site that corresponds to a use of a variable at a given program point is unique, and can be statically determined.

[1]We use the same assumption as Banerjee *et al.*[10]: heap-allocated addresses only come out as the return value of particular functions like `malloc`, `calloc`, etc. Section 4 provides the complete list.

Extending Live Ranges. An inspection point reports the state of all the automatic variables declared in the function where the SIP (Def. 1) exists. A problem ensues if a variable v is not alive at the SIP, and the SIP can be reached through multiple SSA definitions of v. In this case, WHIRO would not know which $def(v)$ to use. To deal with this issue, WHIRO inserts, at the SIP, a ϕ-function joining all the definitions of v that reach that program point. This special notation, a ϕ-function, is part of the SSA format[18]. It joins the live ranges of different names in the low-level code that correspond to the same name in the high-level program. Example 7 shows how ϕ-functions let us extend the live ranges of variables.

Example 7. In Figure 1, `n` and `m` are promoted from the stack to virtual registers. Two definitions of `n` can reach point D, coming from lines 25 and 29. WHIRO shall create a ϕ-function right after the conditional test, to join the different definitions of `n`. This ϕ-function will define a new name for variable `n`. This new name will be inspected at the inspection point D. Figure 7 shows this new definition of `n`. Similarly, definitions of `m` can reach D coming from lines 26 and 33; however, for simplicity, Figure 7 omits the new definitions of this variable.

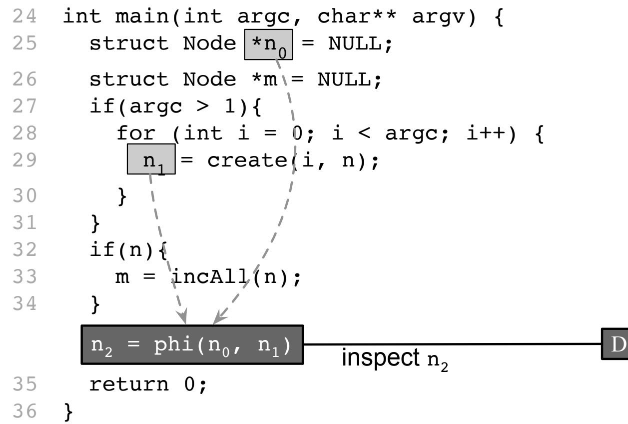

FIGURE 7 The SSA definitions of variable `n`. The instruction `n2 = phi(n0, n1)` copies `n1` into `n2` if the definition of `n1` at Line 29 reaches the inspection point. Otherwise, it copies `n0` into `n1`. In the actual instrumentation, new definitions would be created also for variable `m`; however, we omit them to keep the figure simple.

Function Counters and Calling Context. The names reported in an inspection point are defined statically. Nevertheless, DIPs (Def 2) run in different calling contexts. Hence, the same name may be associated with different values at running time. WHIRO distinguishes data at DIPs by the calling context. A static counter is associated with every instrumented function. The counter is incremented each time the function is called. To report the program state, the current value of the function counter is also informed.

Dealing with Multi-Dimensional Arrays. WHIRO can print the contents of arrays of primitive types; however, in its default mode, WHIRO produces a hashcode that summarizes that data. For arrays of aggregate types, WHIRO inspects each cell individually. To find the hash of an array v of primitive types, WHIRO still traverses v entirely, even if the array is multidimensional. It is to note that a change in any array position yields a different hashcode; hence, hashing does not harm the use of WHIRO as a debugging tool. If the dimensions of the array are determined by constants known at compilation time, then no additional instrumentation is inserted in the program to permit this traversal: the necessary loop is hardcoded at compilation time. Otherwise, WHIRO inserts instructions to compute those values at runtime. These values are kept in the S table. For arrays of pointers, WHIRO treats each contiguous block of memory independently.

Traversing the Heap Graph. The blocks allocated in the heap form a graph. Edges exist whenever a block contains a pointer to another heap address. One of the inspection modes of WHIRO traverses this graph (see Section 5.1) in depth-first fashion. To avoid cycles, WHIRO adds a bit to every node in the Heap Table H, indicating if that node has been visited. Tracing-based garbage collectors implement a similar approach[3,2].

The keys in the Heap Table are memory addresses. Memory allocation or deallocation change H, as explained in Example 6, because these actions create or remove valid addresses that can be dereferenced. However, aliasing among variables of pointer

type bears no influence on H. In other words, if v is a pointer, whose address is stored in H, an assignment such as $u = v$ does not change it. After the assignment, both u and v will dereference an address that is logged in H. Information retrieval will visit the block dereferenced by one of these variables once. To prevent multiple visits, we set every node as unvisited right after the traversal ends, through the "Visited" field in Figure 6.

Notice that WHIRO does not add meta information per variable in the target program. This lack of runtime information forces WHIRO to inspect pointers assuming the type used to declare those pointers—information readily available in the Type Table T. Therefore, WHIRO cannot recover the original type of a variable after a cast, for instance. Example 8 illustrates this limitation of WHIRO.

Example 8. Figure 8 shows a program that performs casts between a struct and an array of characters. In this example, if WHIRO inspects the contents of pointers `n0` or `c0`, then these pointers are treated as references to `struct Node`. However, if pointers `c1` or `n1` are inspected, then these pointers are treated as references to arrays of 16 characters.

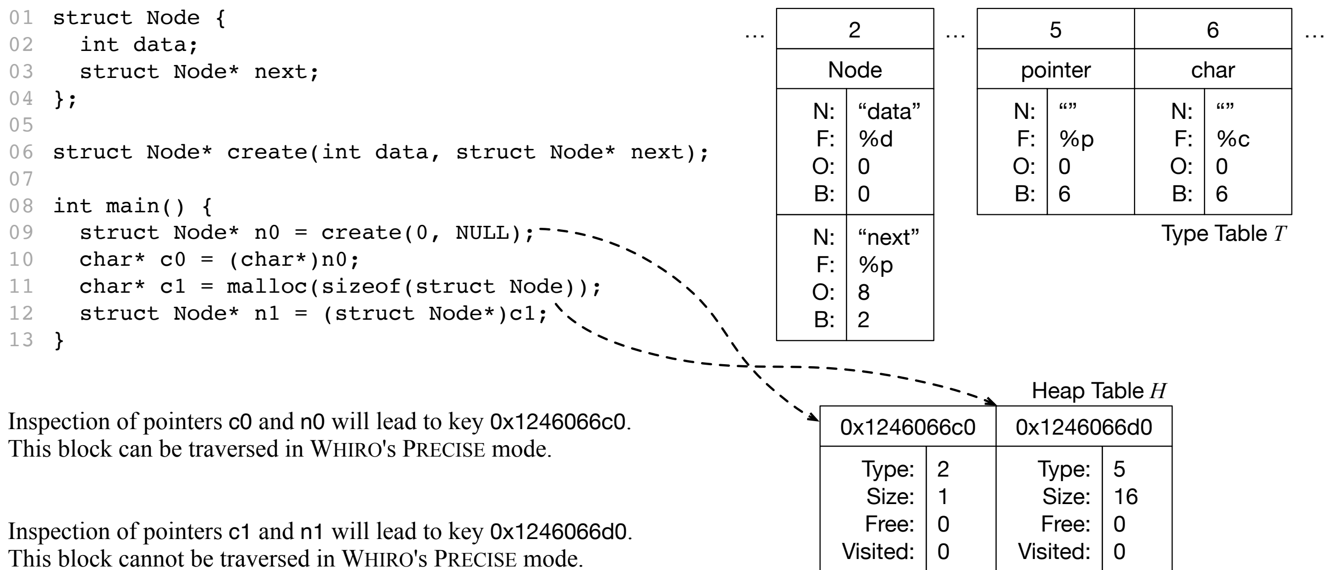

FIGURE 8 How WHIRO represents pointers and casts between pointers. See Figure 5 for the entire Type Table T.

As a consequence of not associating runtime information with references, WHIRO is not able to traverse the graph formed by objects allocated into a slab, for instance. A slab is an array where objects are stored in serialized format. This approach is typical in Bonwick's *Slab Allocators* [19]. References to objects stored in a slab would be treated like pointer `n1` in Example 8—as a sequence of bytes, and not as a structured value. A sound solution to this shortcoming would force WHIRO to track types per reference (or to track all the references to an allocation). Any of these approaches would incur additional overhead of monitoring pointer operations including assignments and arithmetic: a path that we opted not to follow.

3.4 | Properties of the Memory Monitor

A number of properties ensue from our implementation. We provide sketches of proofs of each property. All these properties have been verified experimentally, as we explain in Section 6.

Property 1 (Non-interference)**.** The memory monitor does not alter any value originally computed by the program.

Proof (Sketch): WHIRO does not update memory originally allocated by the program. Notice that WHIRO is not free of side-effects: if users inspect a program point, data shall be output from the program. □

The Non-Interference Property regards exclusively the values of variables manipulated by the program. Thus, a correct construction of WHIRO does not modify neither the program's runtime values, nor its memory allocation behavior, e.g., the order in which the different calls to `malloc` or `calloc` happen. The relative position of the fields within a C-like `struct` is also preserved. However, WHIRO's static instrumentation modifies the original memory layout of the target program by introducing stack-local and heap-allocated monitor data-structures. These interventions caused by WHIRO can modify the security properties of a program; for instance, with regards to exploits that leverage static knowledge of the memory layout. As a consequence, code modifications that rely on relative positions within, for instance, a function's activation record, must run after WHIRO. An example of such modification would be the canaries used to prevent buffer overrun attacks[20].

Property 2 (Time Complexity)**.** When retrieving or updating information, stack and global variables are located in $O(1)$; heap-allocated blocks are retrieved in $O(log_n)$, where n is the number of blocks currently allocated in the heap.

Proof (Sketch): Stack and global variables are accessed via their addresses. WHIRO uses a hash table to record heap blocks. Collisions are handled via a balanced binary tree. □

Property 3 (Space Complexity)**.** The sizes of the components G, S and T of the memory monitor are determined statically. G and S exists only at compilation time; T exists at compilation time and also at running time. The size of G is proportional to the number of global variables in the program; the size of S is proportional to the number of local variables declared in the program; the size of T is proportional to the number of types declared in the program.

Proof (Sketch): These facts are consequence of WHIRO's implementation: the sizes of G, S and T are determined statically, based on the number of global and automatic (i.e., stack-allocated) variables, and on the number of types declared in the program. □

Property 4 (Ordering)**.** When inspecting visible state, first stack-allocated names and heap-blocks are read, then global names. Global and stack data are reported in lexicographical order on the names of variables in source code. Heap blocks in the root of the reachable graph are reported in the order of updates. Within a connected component, the heap graph is reported in the order of updates.

Proof (Sketch): Global and stack variables are kept in G and S maps respectively. The contents of such maps are sorted by the key values, which in this case are the program symbols. Newly allocated heap blocks are pushed to the end of H, which means that it can be traversed as a list □

Notice that Property 4 is valid only for sequential programs. WHIRO, like most on-the-fly garbage collectors[21], does not synchronize access to the heap table H by default. Thus, it cannot guarantee Property 4 for multi-threaded programs. As many of our decisions, this one is also pragmatic: between performance and ordering, we opted for the former. Reversing this decision is a small change in the framework's implementation.

Property 5. (Coverage) At maximum precision, every block that has been allocated in the heap and has not been freed is traversed at least once if this memory is reachable.

Proof (Sketch): Every allocated block is kept in the Heap Table. When inspecting a variable that points to heap-allocated data, WHIRO checks if such data points to another valid memory location. In the positive case, WHIRO will also report the data in said location and perform the reachability checking again. By doing that, WHIRO is able to traverse the entire valid heap. By keeping the *Free* flag in the entries of H, we are able do distinguish which addresses are valid in the program. Notice that this property assumes that the functions that allocate blocks in the heap are known, for the invocation sites of these routines need to be instrumented. □

4 | IMPLEMENTATION DECISIONS

While developing WHIRO, we took a series of engineering decisions (also known as "hacks") to ensure that the framework could handle general C/C++ programs. These decisions, in our opinion, are not necessarily of scientific interest: they are motivated by us choosing LLVM as the underlying development ecosystem. Nevertheless, from a coding standpoint, they might hold some consequence. In this section, we go over some of these decisions.

4.1 | Dealing with pointer arithmetics

Languages like C, C++ and assembly allow arbitrary pointer arithmetics, meaning that it is possible to obtain a reference to an arbitrary index of memory, be that memory valid (allocated) or not. WHIRO does not keep track of which memory addresses are valid—that would be too costly. Blocks of memory allocated in the heap are reported as a hash code of the contents of their bytes, unless the type of these blocks contain pointers, e.g., like an array of arrays. Blocks of memory with a pointer type are traversed recursively in the precise-heap mode. Pointers to the first address of a heap block are reported as hash codes (unless the block contains pointers). Pointers to addresses past that point are reported as single-cell arrays with the type declared for the pointer. Example 9 will clarify these decisions.

Example 9. In the Precise mode with heap tracking, WHIRO will print for the program in Figure 9 the following output:

```
*c main 1: 0x37...5F // Hash of a 16-byte array
*d0 main 1: 42 // Value stored in 8-byte word of type long
d1 main 1: 42 // Value of a variable of type long
c main 1: 42 // Int value of a variable of type char
```

The first line, for `*c`, is the hash code of a memory block stored in the heap. To find out this hash code, WHIRO verifies that the address stored in `*c` has an entry in the Heap Table H. During the instrumentation, the Memory Monitor inserts code to update H after Line 2 of Figure 9. This instrumentation updates H at runtime, right after the allocation is evaluated at Line 2. The other variables point to addresses that do not have an entry in H; hence, they are printed as single cell arrays. Each of these single cell arrays is printed with the base type of the variable that points to it. Notice that WHIRO does not do anything in regards to unaligned memory accesses. If the program casts a pointer to an unaligned word in memory, that address will find no correspondence within the Heap Table. Thus, the contents of that location will be printed as is. In this case, WHIRO will not fill up unaligned bits with zeros, for instance.

```
01 int main() {
02     char* c = malloc(16);
03     long *d0 = (long*)(c+8);
04     *d0 = 42;
05     long d1 = (long)*(c+8);
06     char *c1 = (char*)&d1;
07     printf("%ld\n", d1);
08     return 0;
09 }
```

An entry for the address pointed by `c` will be created in the Heap Table H

The address `c+8` has no entry in H; hence, `d0` is inspected as `long[1]`

`c1` is inspected as `char[1]`, because its address has no entry in the H

FIGURE 9 Program that casts pointers to the middle of a memory block with a different type. Variable `c` will be inspected as `char[16]`, because its address has an entry in the Heap Table H. Because it points to an array, it will be printed as a hash code.

To summarize WHIRO's approach to deal with addresses pointing out to the middle of memory blocks, the following protocol applies, when reporting data about symbol v of type t_v in precise heap-tracking mode:

1. If t_v is a primitive type, or an aggregate type without pointer fields, report it by value;
2. If t_v is a pointer, or contains fields that are pointers, for each address p in v, do:
   (a) If p contains an entry in H, inspect it recursively.
   (b) Otherwise, let t_p be the base type of p. WHIRO inspects the contents of p, e.g., $*\,p$, as an array of type $t_p[1]$.

Currently, WHIRO has special treatment for `stdlib` functions that allocate memory (`calloc`, `free`, `malloc`, `realloc`, `reallocf`, `valloc` and `aligned_alloc`). These functions update the Heap Table H. Notice that these functions have not been modified in any way: WHIRO only understands that they perform memory allocation and deallocation. This combination of a Heap Table H and a Type Table T lets WHIRO handle arrays that contain structs, even if said structs contain pointers

to other arrays. WHIRO is able to traverse these data structures. Example 10 shows how WHIRO inspects a 3D-matrix in its precise/heap-tracking mode.

Example 10. Figure 10 shows a program that allocates and initializes a three-dimensional matrix stored in the heap as an array of arrays of arrays. Every invocation of the `malloc` function adds an entry to WHIRO's Heap Table H. Only the allocations at Line 13 of Figure 10 will not contain pointers. These blocks will be reported by the hash codes of their contents. The other blocks will be traversed recursively.

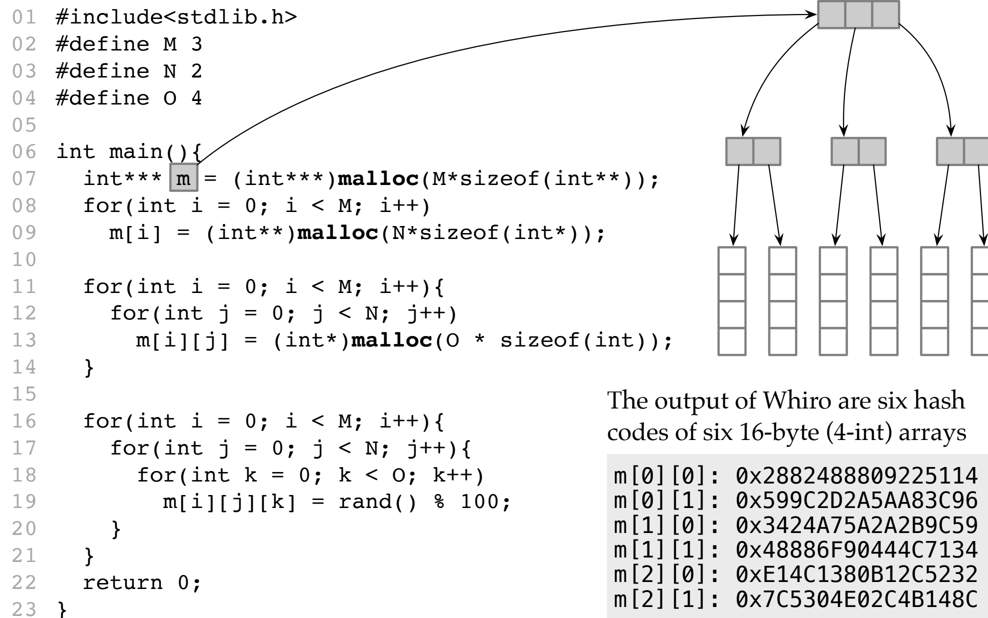

FIGURE 10 The output of WHIRO when inspecting a 3D-matrix stored as an array of arrays of arrays in the heap. Every gray box represents an entry in the Heap Table H.

4.2 | Reading one element past the end of an array

The C Standard allows a comparison between a pointer and the first address past the end of an array ([22] 6.5.6/8). However, this element cannot be dereferenced. Referring to the next address past the last address of an array is a common approach to implement C++ iterators, for instance. Example 11 shows a program that uses this trick. This possibility poses a problem to WHIRO, for our implementation cannot know if a pointer is valid or not. If the pointer is mentioned in the source code, this pointer will be printed as part of the program's visible state, once information is retrieved from the inspection point. In this case, undefined behavior might ensue.

Example 11. Figure 11 shows a program that reads one element past the last address of an array to iterate through the array. Pointer `e` cannot be dereferenced; however, it can be compared against another pointer.

WHIRO does not keep a list of valid memory blocks. Keeping this list would be overly expensive at runtime, for every memory address would have to be tagged. Thus, if an invalid address is mentioned in the source code, it will be printed as part of the data in a dynamic inspection point. Two problems emerge from this shortcoming. First, debuggers, like the one discussed in Section 5.2, might try to compare these pointers, and the result of this comparison is meaningless. Second, when tracking heap data, WHIRO might jump to whatever address it recognizes in such a pointer—possibly incurring into a segmentation fault. Our implementation does not try to deal with the first problem: it is up to WHIRO's users to handle false positives. To deal with the second problem, WHIRO only prints pointers within specific ELF program segments: `stack`, `heap`, `bss`, `data`, and `text`. The

```
01 int main() {
02   int v[4];
03   int *i;
04   int *e;
05   for (i = v, e = v + 4; i < e; ++i) {
06     *i = 1;
07   }
08   return 0;
09 }
```

FIGURE 11 A program that reads a pointer past the last address of an array.

boundaries of these segments are given by global variables. This solution works for ELF binaries, but it is not portable across different formats.

4.3 | Dealing with Union Types

Programming languages like C and C++, or the LLVM intermediate representation, provide users with "union types". These types are sets formed by the union of two other types. Union types in C and C++ are defined by the `union` key word. Unfortunately, these unions are not tagged, as it happens, for instance, with datatypes in functional programming languages like Haskell and SML/NJ. Therefore, at runtime, WHIRO cannot know what is the intended meaning of a union type: any of its composing types could be a valid representation. Example 12 illustrates this issue.

Example 12. Figure 12 shows a program written in C that defines a union of two types; chars and 32-bit floating point numbers.

```
01 void default_init(union element* e, char* str, int size) {
02   if (is_float(str, size)) {
03     e.f = -1.0625;
04   } else if (is_char(str, size)) {
05     e.c = 'a';
06   }
07   // ...
08 }
```

A: 0xBF880000 — 1 | 01111111 | 00010000000000000000000

B: 0x00000061 — 00000000 | 00000000 | 00000000 | 01100001

FIGURE 12 Program that defines a union type. WHIRO prints instances of union types as sequences of bits. For clarity, we show the sign, exponent and mantissa of the float pointer number, and split the char representation into four bytes (which is the size of the largest element of this union type).

To allow the comparison between union types across two different versions of the same program, WHIRO prints them as sequences of bits. In other words, the inspection of a value declared as a union type always yields the binary representation of that type, using its largest composing part. Similarly, whenever a union type is inserted into the heap table H, it is stored as its largest component, even if that is not its intended meaning.

Example 13. Any instance of the union `element` in Figure 12 will be inspected in the same way: as a sequence of bits, regardless of how this union has been initialized. Thus, even though WHIRO does not keep track of the type used to initialize instances of unions, WHIRO still supports comparing unions by inspecting their contents. In this case, the bits stored in the union can be compared.

4.4 | Shadowing Stack Variables

As mentioned in section 3.3, WHIRO reports all the variables local to the function where the inspection point is defined. For variables which are not alive at said point, we extend their live range by injecting ϕ-functions in the program. However, in some

cases there are no definitions of a variable v from a function f in the predecessors of the inspection block. In this case, extending the live range of v might involve the creation of many ϕ-functions from the available definitions of v until the inspection point—some of them with UNDEF values. To simplify this process, WHIRO *shadows* v to correctly track its value.

Shadowing a variable means duplicating in memory the value of that variable. The duplicate must be updated whenever the original variable suffers an assignment. To implement shadowing, we allocate a slot $shadow(v)$ in f's activation record. This slot has the same type as v. Then the trace of v is traversed and the memory monitor injects code in f to store all the definitions in $shadow(v)$ right after they are computed. Since $shadow(v)$ is created at the entry point of f with a special "undefined" value, the shadow value is visible in all the basic blocks of that function. Therefore, that value can be read at any point within f. Example 14 illustrates this modus operandi.

Example 14. Variable b in Figure 13 belongs into the local scope of function main. This variable is assigned at two different program points which reach the static inspection point at Line 11. Furthermore, its declaration point, with an undefined value, can also reach Line 11. Inspecting the value of b at Line 11 would require the insertion of three ϕ-functions in the program (between Lines 6-7, 8-9 and 10-11). Instead, WHIRO duplicates the value of b in memory.

```
01 int main(int argc, char** argv){
02   int a = 10;
03   int b; // alloca(shadow_b)
04   for(int i = 0; i < 5; i++){
05     if(argc > 1){
06       b = argc; // *shadow_b = arcg
07       while(b < a)
08         b++; // *shadow_b += 1
09     }
10   }
11   SIP // print(*shadow_b);
12   return 0;
13 }
```

FIGURE 13 Variable b would be shadowed. The value of the duplicated variable will be printed at the inspection point.

We shadow variables whenever undefined values can reach an inspection point. A special **undef** token is stored in the shadow location. This token is unique for every occurrence of what, in LLVM jargon, is known as an "immediate undefined behavior"[23] Sec.2. In this way, when comparing two different versions of the same program, WHIRO will match two instances of the same **undef** token. We opted for this strategy for simplicity, as it makes it unnecessary to update the SSA-form representation of the program. A more mature version of WHIRO probably will not resort to shadowing. Instead, keeping variables in SSA-like virtual registers for as long as possible.

4.5 | Inspecting State of Optimized Code

WHIRO is able to track the state of program variables in the face of some program optimizations. Such is possible as long as the code optimization satisfies the following requirements:

- It preserves the existence of the inspection point. Inspection points refer to regions in the source code of a program. These regions might not be preserved after optimizations such as dead-code elimination, which might erase sequences of basic blocks in the low-level representation of the program.
- It preserves the existence of meta-information associated with high-level variables. Some optimizations might eliminate any reference to variables present in the source code of the program. In this case, these variables will have no associated state, and WHIRO will not show them when generating reports.

In our experience, many code optimizations implemented in LLVM preserve the metadata associated with program symbols. Therefore, they pose no problems to WHIRO. However, optimizations such as dead-code elimination might eliminate this

metadata. In the case of array scalarization, an optimization that removes metadata, we have been able to augment WHIRO with enough information to still inspect the state of modified code. Array scalarization replaces a cell within an array with a temporary register. Readings and updates meant to happen over the array are diverted to the register. Example 15 shows an example of this optimization in action.

Example 15. Figure 14 (Top) shows a program that performs updates on an array `r`. The update depends on the values of arrays `a` and `b`. It is possible to scalarize `r[i]`. Figure 14 (Bottom) shows the program after scalarization takes place.

```
01 void sum0(int* a, int* b, int*restrict r, int N) {
02   int i;
03   for (i = 0; i < N; i++) {
04     r[i] = a[i];
05     if (!b[i]) {
06       r[i] = b[i];
07     }
08   }
09 }
10
11 void sum1(int* a, int* b, int*restrict r, int N) {
12   int i;
13   for (i = 0; i < N; i++) {
14     int tmp = a[i]; // scalarized:tmp/r
15     if (!b[i]) {
16       tmp = b[i]; // scalarized:tmp/r
17     }
18     SIP // inconsistent_hash:r
19     r[i] = tmp; // scalarized:tmp/r
20   }
21   SIP // consistent_hash:r
22 }
```

FIGURE 14 (Top) Program before scalarization. (Bottom) Program after scalarization.

Scalarization renders the state of an array invalid at some program regions. As an example, the contents of array `r` differ within the conditionals in Figure 14. If an inspection point exists in a region that contains scalarized arrays, then the contents of these arrays cannot be compared. To deal with this problem, WHIRO adds metadata to the program, to indicate the places in the code that contain scalarized arrays. When traversing the data at an inspection point that exists at such a place, WHIRO still prints the hash of the scalarized array, albeit with a message to the user, indicating that the hash was taken from stale memory.

5 | APPLICATIONS OF INSPECTION POINTS

This section presents three applications of WHIRO, each using a different customization of the framework. We define the possible customizations in Section 5.1, and go over the case studies in Sections 5.2–5.4.

5.1 | Customizations

WHIRO can be customized along three dimensions: memory allocation, tracking graph, and SIP granularity. These dimensions trade precision for performance. In this context, "precision" is ranked by the amount of information stored in the memory monitor, and "performance" is ranked by the running-time overhead imposed by instrumentation.

Memory Allocation. This customization determines which memory region of the program will be tracked by inspection points. WHIRO recognizes any combination of three regions:

STATIC: Tracks memory allocated statically.

STACK: Tracks memory allocated on the stack.

HEAP: Tracks memory allocated in the heap.

Tracking Graph. WHIRO can either treat program symbols as isolated entities, or can relate them via the pointers present in the instrumented code. These two possibilities give us the following customization modes:

FAST: Only local and static variables are tracked as isolated locations. Hence, this mode does not follow pointers when presenting the program state.

PRECISE: WHIRO shows the graph formed by relations between pointers. Contrary to the previous mode, this customization requires building the heap table.

WHIRO, in the PRECISE mode, works as a dynamic shape-analysis[24] able to track, at running time, the context-sensitive information that state-of-the-art analyses[25,26] approximate statically—a fact that we state in Property 6.

Property 6 (Shape). Let π_s be a static inspection point, and let π_d be any related dynamic inspection point. WHIRO will produce for π_d an image of the heap with no more edges than a "may" version of shape-analysis would summarize for π_s. Similarly, its image should contain for any π_d no less edges than a "must" version of shape analysis summarizes.

Proof (Sketch): WHIRO draws edges between heap blocks only if such blocks are reachable. Although the heap table maintains freed addresses, edges that target such addresses are never reported □

SIP Granularity. WHIRO allows the customization of static inspection points. The current implementation of WHIRO allows the user to add a static inspection point before any LLVM instruction. However, for the sake of pragmatism, the experiments in Section 6 only consider three possible locations for SIPs:

MAIN-RET: the program point immediately before the return point of the `main` routine.

ANY-RET: the program point immediately before the return point of any routine.

ANY-STORE: the program point immediately after any store instruction.

5.2 | Debugging Aid

The construction of tools to debug compiler optimizations has been a common source of research in programming languages[27,28,29]. The ability to compare states in an inspection point that is common across different versions of a transformed program lets us contribute along this direction.

Purpose: Given a program P plus a set of inputs, and an optimized version P' of it, compare the internal state of P and P' when these programs run with the given inputs. Use the result of this comparison to pinpoint bugs in the optimization. States are compared via the dynamic inspection points (see Definition 2) that represent them. The comparison is possible for every DIP that exists in P and in P'. Finding equivalent DIPs is simple: two DIPs, one from P, the other from P', are equivalent if they are originated by the same static inspection point. To compare the program state from two equivalent DIPs, static and stack-allocated variables are matched according to their names in the source code of the subject program. Heap variables are matched according to their allocation sites. Multiple nodes from the same allocation site are matched by the allocation order. It is possible that an optimization removes variables or heap nodes from the program state. Thus, the comparison happens only for those variables and nodes that exist simultaneously in P and in P'.

Challenge: We are comparing two potentially very different versions of the same program. There is not a perfect match between inspection points: optimizations like inlining remove some inspection points in the ANY-RET mode. There is not a perfect match between program symbols neither, as optimizations like constant propagation remove symbols.

Instrumentation mode: To maximize the likelihood that differences in program state are observed, we instrument every memory region (STATIC+ STACK+ HEAP), using the PRECISE mode, at maximum granularity (ANY-RET).

Results: We deliver an approximate solution for the problem of bug finding: our debugger only matches program points that are equivalent in both versions of the program. We are able to highlight program symbols that have been erased in the optimized version of the program, to reduce false positives. Auxiliary variables created by the compiler are not considered in this comparison, for they are not associated with high-level debugging information, nor do they have exact correspondence between P and P'. In Section 6.6 we show that this usage of WHIRO has been able to detect bugs injected in LLVM's implementation of constant propagation.

5.3 | Adding Verification Outputs to Benchmarks

The synthesis of benchmarks is an important program in the construction of predictive compilers[30,31,32]. A common technique to synthesize benchmarks is to mine code from open-source repositories[33]. Predictive compilation often requires thousands of benchmarks. As an example, EXEBENCH contains 700K executable C functions[34]. As pointed out by Armengol *et al.*[34], for verification purposes, a benchmark must have some output. By comparing said output with a reference, compiler engineers have confidence in the validity of any test produced with that benchmark. However, adding code to produce some output into thousands of benchmarks is not practical. Such tasks must be accomplished automatically, and WHIRO can provide such support.
Purpose: Given a program P, add output to it, by printing the state of static and local variables after execution.
Challenge: If instrumentation is too intrusive, the results obtained through benchmarking P might be subject to "probe effects". These effects emerge when inspecting the program provokes unintended alterations of its behavior. Therefore, the performance of P must be preserved as much as possible.
Instrumentation mode: To reduce probe effects, we use WHIRO to instrument MAIN-RET's SIPs. It reads STATIC and STACK memories in the FAST mode.
Results: We have applied the above customization of WHIRO onto 137 executable programs downloaded from ANGHABENCH. Each benchmark was tested with 10 different inputs. Outputs were produced for all the 137 programs, in 1,359 executions (out of a total of 1,370). Property 4 was verified when repeating this experiment. Each benchmark consists of a driver plus a function (which is the benchmark proper). Not counting the driver, which always gives us the same outputs, on average WHIRO inspects 7.8 variables per function, with a minimum of 2 variables and a maximum of 32 variables. The instrumented benchmarks have been returned to the maintainers of ANGHABENCH, and are now publicly available.

5.4 | Data Visualization

Tools able to provide graphic representation of the heap are useful for program understanding and debugging[35]. Viewing data structures and other program elements in a graphical format makes it easier to analyze the state of memory, recognize patterns, and observe the relation between different allocated blocks.
Purpose: Adapted WHIRO to render a visual representation of the graph determined by relations between pointers in the heap in a program.
Challenge: To the best of our knowledge, all the techniques discussed in the literature to visualize program state deal with type-safe languages with managed memory. Java is the usual target[35,36,37], although there are also DSLs conceived to this end[38,39]. Tracking relations between pointers in C or C++ is non-trivial, due to the difficulty to distinguish memory addresses from scalar types. Previous work that attacked this task required users to use special annotations to indicate which software events should be visualized[40].
Instrumentation mode: We customize WHIRO with the following configurations: the tracking graph is PRECISE; the SIP granularity is ANY-RET; and considering the HEAP memory allocations.
Results: Figure 15 shows heap snapshots produced with our adaptation of WHIRO. To ease visualization, we are showing only nodes stored in the heap. Currently, we can visualize the heap of all the programs in the MIBENCH collection, for instance. Graphs are produced in DOT format. Users can render them using different graph visualization algorithms. We are not able to distinguish pointer relations created by non-pointer types, like it happens in the infamous doubly-linked XOR list. As an example, when given the program in Figure 1 of Banerjee et al.[10], WHIRO prints a series of unconnected blocks with the `xorlist` type.

6 | EVALUATION

This section evaluates the ideas presented in this paper. To this effect, we investigate the following research questions:

RQ1: What is the overhead that our instrumentation adds to the compilation time?

RQ2: What is the runtime overhead imposed by our transformation on the instrumented programs?

RQ3: How much extra memory does WHIRO use?

RQ4: How does the size of the instrumented program grow in relation to its original size?

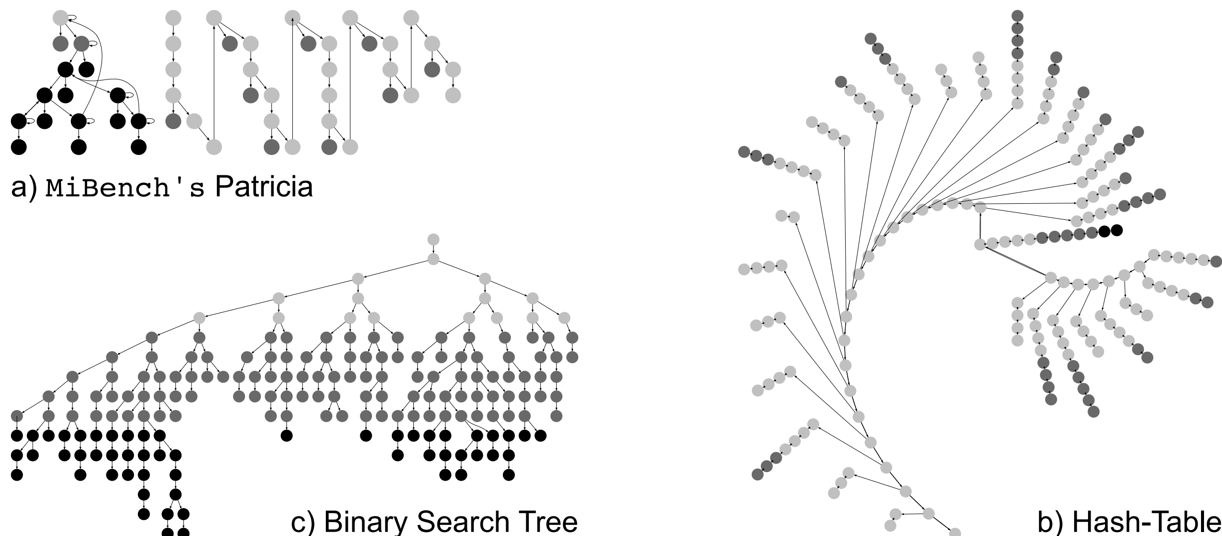

FIGURE 15 (a) Snapshot showing the two disjoint data structures in MIBENCH's Patricia, after the first invocation of `pat_search` returns with the test input. (b-c) The two disjoint graphs in the heap of a program that copies a binary tree into a hash table. Collisions are stored in a linked list.

RQ5: How does the number of static inspection points impact WHIRO's instrumentation time, or the running time and code size of instrumented programs?

RQ6: Can our technique be used to detect actual bugs in compiler optimizations?

Software: Instrumentation is implemented on LLVM version 10.0.0. Data structures that store the auxiliary state are implemented in C. They are linked statically with the bytecode that LLVM produces. The experimental evaluation uses Linux Ubuntu 64-bit version 18.04.5 LTS. Running time and memory usage are collected via Linux' built-in `time` command. Numbers of LLVM instructions are collected using LLVM's `-instcount`. The other statistics reported in Section 6.4 are gathered directly by the instrumentation pass.
Hardware: Experiments run on an 8-core Intel i7-8565U CPU at 1.80 GHz, and 8GB of RAM (DDR4) at 2,400MHz.
Benchmarks: We chose MIBENCH [15] to evaluate our techniques. We use the version of MIBENCH available in the LLVM test repository, which contains 16 benchmarks. We have constructed inputs for these programs using a synthesizer available at https://github.com/ekut-es/mibench, to ensure that each benchmark runs for more than 1.0 second when compiled with `clang -O0` (v10.0). We failed to meet this criterion for MIBENCH's `Patricia`, which runs for about half-a-second with the largest input that we found.
Methodology: Experiments in sections 6.1, 6.2, and 6.3 report averages of 10 executions. For each benchmark, we measure its running time 12 times, and discard the fastest and slowest executions. The elimination of the slowest and fastest samples per benchmark allows us to mitigate the impact of random effects on the experiments, such as cold-start execution. For instance, the first sample usually runs more slowly, due to the time to load shared libraries in memory. The programs were compiled with `mem2reg` and with `mergereturn`. The first pass maps memory locations to virtual registers. We have used it to speed up the execution of the experiments: we had to run each program multiple times per configuration of WHIRO; `mem2reg` tends to shorten our running times by almost half. The second pass, `mergereturn`, ensures that every function will have a single exit point. This property simplifies the task of finding static inspection points automatically. Nevertheless, none of these passes is necessary to apply WHIRO. Results are considered statistically significant within a confidence level of 99% via a Student T-Test.

6.1 | RQ1: Compilation Overhead

One of the design principles of WHIRO is to move to compilation time as much instrumentation overhead as possible. On the one hand, this strategy reduces the overhead that our inspection points impose onto executable programs, as we shall demonstrate in Section 6.2. On the other hand, it extends compilation time. This section analyzes this impact.

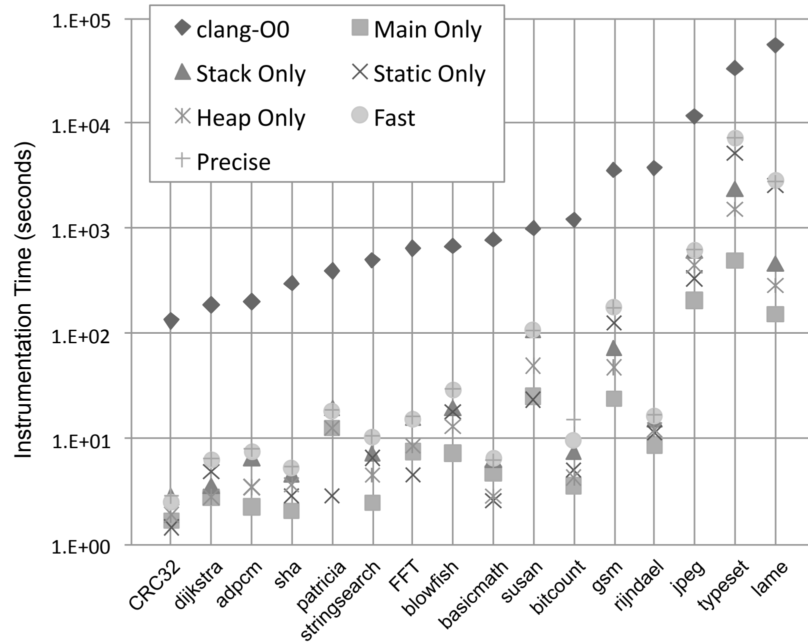

FIGURE 16 Time to instrument programs (in seconds). For reference, we provide the time of compiling each program with `clang -O0 -g`.

Discussion

Figure 16 shows the time required to instrument the different programs in MiBench. For reference purposes, we also show the time taken to compile each program with `clang -O0 -g`. The instrumentation time is shorter than standard compilation time in every case[2]. The overhead that instrumentation adds onto compilation depends primarily on the number of static inspection points. The more SIPs a program contains, the larger the number of interventions that Whiro carries out in the code of said program. Thus, the larger the program, the larger the instrumentation overhead. Consequently, Main-Ret incurs the smallest slowdown in compilation time, because it inspects only one function, e.g., `main()`. Notice that the instrumentation overhead, even at the Main-Ret granularity, is not constant: Whiro still must traverse the `main` function looking for return points. Furthermore, the number of local variables within this function bear an impact on instrumentation time, as we explain next.

Number of Variables to be Inspected. The instrumentation time depends also on the number of variables that must be inspected. For instance, Static tends to have a small impact on the compilation time, because programs usually have less static variables than local and heap variables. As another example, `lame` is about 6x smaller than `typeset`, considering the number of LLVM bytecodes that these programs contain; however, instrumenting the `main` routine of `lame` takes longer, for it contains more local variables.

Fast and Precise. According to Figure 16, the slowest absolute instrumentation time was 7.26 seconds using the Precise mode in MiBench's `typeset`. Yet, just to compile `typeset` without any optimization already takes 33.55 seconds. Figure 25 provides complete ratios. Notice that the Fast and the Precise instrumentation modes increase compilation time by approximately the same amount: these two modes inspect local and static variables in the same number of SIPs. There are two differences between them. First, the Precise mode adds to each SIP a call to a routine to traverse the heap. Second, the Precise mode adds to each site where memory is allocated a routine to update the Heap Table.

[2]Any program instruction can be considered a static inspection point. When the number of SIPs approaches the size of the program, the instrumentation time might exceed the compilation time. We provide some evidence regarding this fact in Section 6.5.

6.2 | RQ2: Running Time Overhead

WHIRO imposes an overhead on executable programs due to: (i) keeping the auxiliary state; and (ii) shadowing stack-allocated variables. This section analyzes this impact.

Discussion

Figure 17 shows how the performance of executable programs varies depending on the instrumentation mode. Points in Figure 17 show the ratio between the execution time of the instrumented program and its original version. Overhead increases with the granularity of the inspection; hence, PRECISE yields the slowest executables. This outcome is to be expected, because PRECISE inspects all the variables in a program and updates the heap table during execution. STATIC is also slow in general, because this mode inspects static variables at every function call (unless MAIN-RET only is used). MIBENCH's `dijkstra` accounted for the largest overheads. The slowdown, in this case, was caused by excessive heap usage: the program manipulates a graph stored dynamically. There were cases in which the instrumented program was faster than its original version. However, for all these cases we found p-values greater than 0.01; hence, we cannot consider them statistically significant.

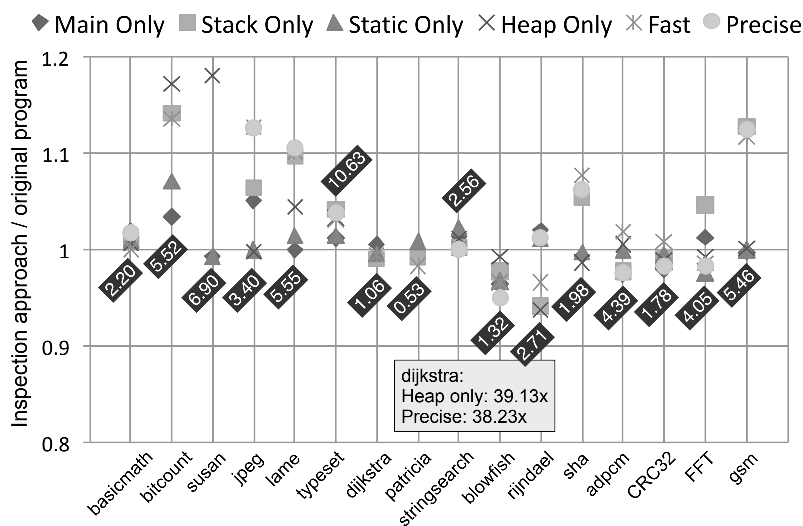

FIGURE 17 Increase of execution time (with relation to the original program). Numbers in boxes denote the running time of the original program (in seconds).

Overhead vs. Size: Unless the heap is inspected, the overhead of WHIRO tends to be constant: some state must be recorded for each local variable that the program uses; and the number of local variables in a program is proportional to the size of said program. Thus, most of the overheads in Figure 17 orbit around 1.0. We emphasize that there is no direct correlation between the impact of WHIRO on the running time of programs, and the size of said programs, measured in terms of the number of LLVM instructions or number of available static inspection points. To support this statement with data, Figure 18 shows the overhead[3] that WHIRO imposes on ten large[4] programs in the LLVM test suite when running at its highest granularity—the PRECISE mode.

Figure 18 contains two benchmarks from MIBENCH: `typeset` and `lame`, which are used to answer other research questions throughout this paper. The remaining programs come from different benchmark collections present in the LLVM test suite, and are only mentioned during our analysis of **RQ2** in this section. In particular, Figure 18 uses a version of the `jpeg` filter

[3]Figures 17 and 18 were produced using different methodologies. Figure 17 uses only programs from MIBENCH. There exists an input generator for MIBENCH, which we could use to ensure that most programs would run for more than one second. This apparatus does not exist for the other programs from the LLVM test suite. Thus, Figure 18 uses the original inputs that LLVM provides for each benchmark, including `typeset` and `lame`. Consequently, running times in Figure 18 are much shorter than in Figure 17. In both cases, numbers are averages of 10 executions.

[4]We chose the ten largest benchmarks in the LLVM test suite that we could run outside the LLVM test framework. Size is measured as the number of lines of C code. Following this criterion, we had to disregard four benchmarks: `perl`, `p2c`, `TimberWolfMC` and `sqlite3`. We could not compile the former two programs, even without WHIRO, and we did not find inputs to run the other two.

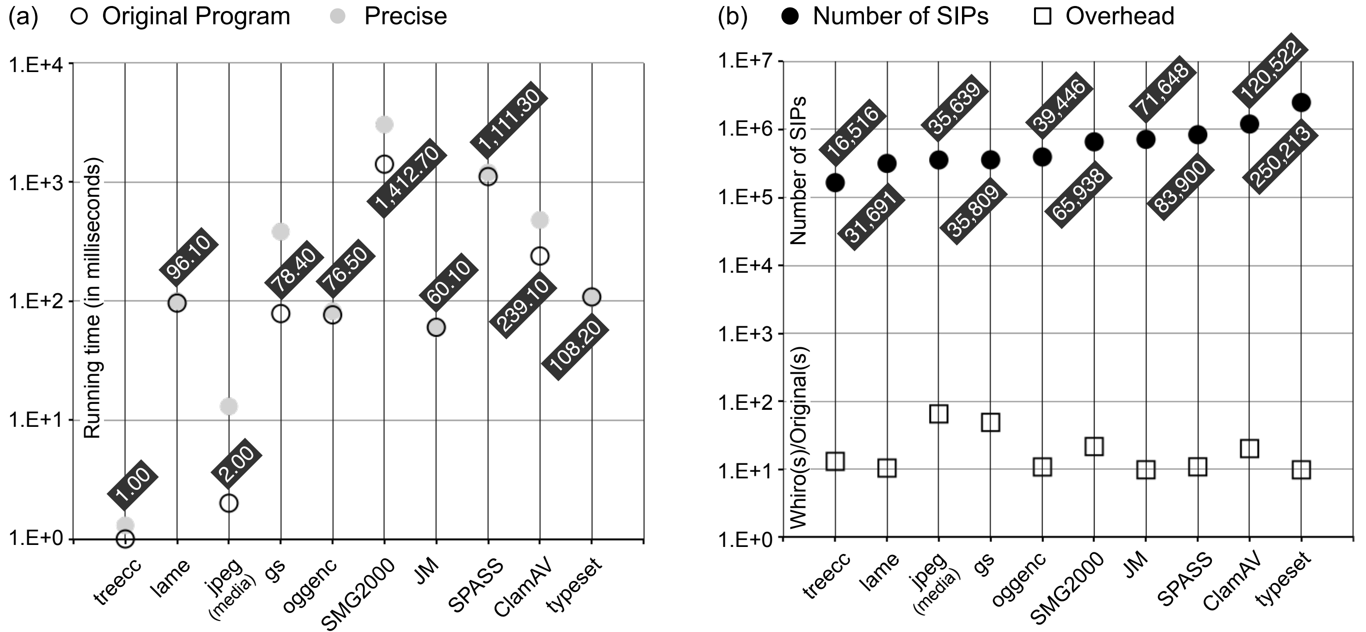

FIGURE 18 (a) Overhead that WHIRO's PRECISE mode imposes onto the ten largest benchmarks available in the LLVM test suite. Numbers in black boxes show the running time of the original programs (in milliseconds). (b) Number of static inspection points per benchmark, compared to the average overhead observed on these benchmarks when WHIRO runs in PRECISE mode.

from MEDIABENCH, which differs from the version available in MIBENCH, being considerably larger. WHIRO's overhead, when applied on its PRECISE mode, ranges from 0.98x, in `JM` to 6.50x, in MEDIABENCH's `jpeg`. Overhead is measured as the ratio between the running time of the instrumented program and the original program. In the case of JM we did not observe a speedup: rather, there is no statistically significant difference between the original and the instrumented benchmark. The geometric mean across running time variations is 1.70x; the median is 1.08x. These results are on a par with those seen in Figure 17.

Printing Data Overhead. The largest overhead observed during our experiments happened when MIBENCH's `dijkstra` was inspected in WHIRO's PRECISE mode: we recorded a slowdown of 44x. With only 267 LLVM instructions, `dijkstra` is the second smallest benchmark in the MIBENCH suite. However, it stores a large quantity of data in the heap: even printing this data at every static inspection point is not practical. To give the reader some perspective on this observation, we analyze the cost of printing all the data that WHIRO records for some small benchmarks. Figure 19 compares the running time of three benchmarks: `stringsearch`, `sha`, and `FFT` with and without counting the time necessary to print all the inspection traces. Printing obviously increases the running time in all inspection modes. This increase is more noticeable in the FAST and PRECISE modes, which inspect values stored in all memory regions. Time grows, in the worst case, by factors of 2.7x, 35.7x and 24.8x in `stringsearch`, `sha`, and `FFT`, respectively.

6.3 | RQ3: Memory Overhead

The dynamic components of the Memory Monitor exist during all the execution of a program. Therefore, inspection points are expected to increase memory consumption. To read peak memory usage, we use Linux' `time -v` and report memory consumption as the "maximum resident set size".

Discussion

Figure 20 plots the memory consumption ratio between instrumented and non-instrumented programs. In 81% of the benchmarks, memory consumption increased by no more than 1.3x, and in 75% of them, this overhead was less than 1.1x. The inspection modes which use more memory are PRECISE and HEAP. In addition to the type table T, these instrumentation modes keep the heap table H in memory. Large memory usage was observed in `dijkstra` when using either of these instrumentation modes. The `jpeg` program was the benchmark that experienced the largest growth in memory consumption not related to heap

	stringsearch		sha		FFT	
	No Print	Print	No Print	Print	No Print	Print
main	2.59	2.85	1.97	2.89	4.09	4.24
stack	2.56	4.35	2.09	57.44	4.23	3.62
static	2.62	6.75	1.97	2.31	3.95	92.03
heap	2.59	2.67	1.95	1.94	4.02	4.32
fast	2.56	5.65	2.13	57.80	3.99	85.90
precise	2.56	6.88	2.11	75.26	3.98	93.83

FIGURE 19 Time to run the benchmarks (in seconds) with and without counting the time to print inspection traces.

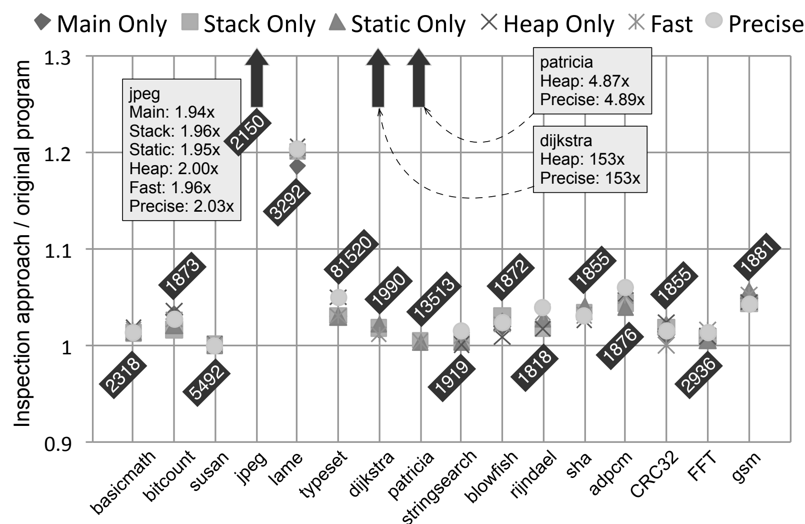

FIGURE 20 Increase in memory consumption (with relation to the original program). Numbers in boxes denote memory consumed by the original program, in Kilobytes.

allocation. Memory consumption has increased, in this case, due to the large number of entries in `jpeg`'s type table: 2,560 in total. Nevertheless, notice that this expansion is relative to the amount of memory that the benchmark requires without instrumentation. In absolute terms, this growth is small, when compared to the memory requirements of larger benchmarks, such as `typeset`, which uses almost 40x more memory than `jpeg`.

6.4 | RQ4: Code-Size Overhead

Instrumentation increases code. This boost is due to the new routines that access the T and H tables, due to the variables created to shadow stack-allocated data, and due to code to print the values of variables at inspection points. In this section, we analyze this growth. The size of code is measured in the number of LLVM instructions.

Discussion

Figure 21 shows the ratio between the sizes of instrumented and original programs in MiBench. The Fast and Precise instrumentation modes are, as expected, the most prodigal, increasing code size by factors of almost 4x in `lame`. Growth in the other benchmarks is more moderate, but typically above 2x for these modes. Static also tends to increase code-size by

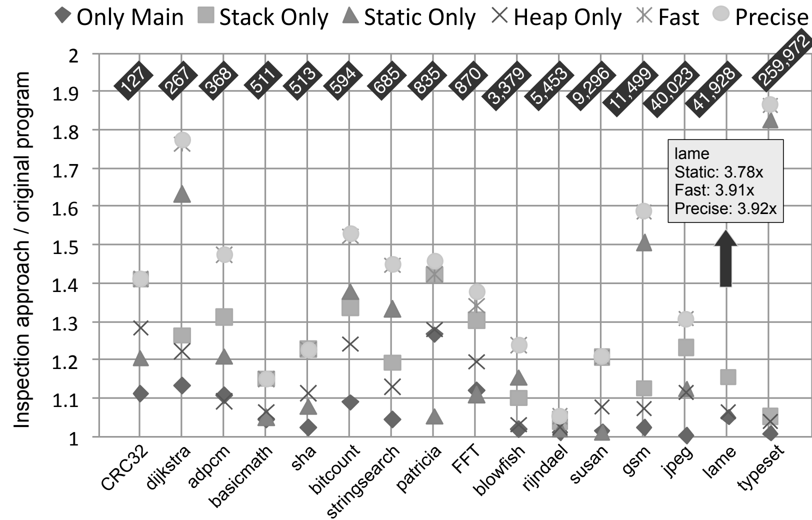

FIGURE 21 Code growth (with relation to the original program). Numbers in boxes denote the number of LLVM instructions in the original program, compiled with `mem2reg`. Benchmarks are sorted by the size of the original program.

substantial margins, because static variables are inspected in every function call. The MAIN-RET mode accounts for the smallest code increase, because an inspection point is created in only one function.

Factors of Code Growth. We conducted an experiment aiming to investigate whether there exist features of programs that correlate well with code growth. Figure 22 summarizes these results. MIBENCH contains 1,228 functions spread across 16 benchmarks. Most variables, 7,130, are stack allocated; 3,582 variables are non-static pointers and 788 are static (of any type). Figure 22 shows that the number of functions and variables is strongly correlated with code growth when inspecting only the stack or the heap. In all these cases, Pearson R^2 is above 0.95. When inspecting only static memory, the number of variables and the number of original instructions are the determining factors. In this case, R^2 is always above 0.85. When inspecting the entire program—with either the FAST or the PRECISE modes—instructions are the factor that determines growth, with an R^2 value of 0.86.

	Variables			Functions			Instructions		
	Total	Mean	R^2	Total	Mean	R^2	Total	Mean	R^2
main	891	52.4	0.80	1	1	X	388,003	22,823.7	0.29
stack	7130	419.4	0.97	1,228	76.75	0.97			0.73
static	788	46.4	0.87			0.57			0.85
heap	3582	210.7	0.95			0.97			0.91
fst/prc	7918	465.8	0.63			0.60			0.86

FIGURE 22 Relations between program features and code growth. We let fst = FAST and prc = PRECISE.

6.5 | RQ5: Number of Static Inspection Points

The impact of WHIRO on the behavior of programs depends on the number of static inspection points instrumented. This section evaluates WHIRO varying fractions of injected SIPs in programs to analyze its impact on compilation time, running time and code

size. To vary the number of static inspection points, we use the three different instrumentation modules currently implemented in WHIRO: MAIN-RET, ANY-RET and ANY-STORE, which are described in Section 5.1.

Discussion

Figure 23 shows how the number of static inspection points impacts the behavior of three different benchmarks from the LLVM test suite: `JM`, `gs` and `SMG2000`[5]. The choice of benchmarks is arbitrary. These programs also appear in Figure 18. The number of SIPs impacts most directly WHIRO's instrumentation time. As already discussed in Section 6.1, the more inspection points a program contains, the longer WHIRO will take to instrument it.

JM	Inst. Time (sec)	Running Time (sec)	Program Size (bytes)	SIPs	I. Time	R. Time	Size
MAIN-RET	0.66	0.06	1,003,800	1			
ANY-RET	5.75	0.06	1,070,408	453	1.00	0.93	0.36
ANY-STORE	132.62	0.07	1,051,896	3,496			
gs	Inst. Time (sec)	Running Time	Program Size (bytes)	SIPs			
MAIN-RET	0.35	0.40	685,224	1			
ANY-RET	2.20	0.41	738,080	742	0.98	0.96	0.66
ANY-STORE	13.17	0.51	730,776	2,289			
SMG2000	Inst. Time (sec)	Running Time	Program Size (bytes)	SIPs			
MAIN-RET	0.52	2.88	840,880	3			
ANY-RET	9.41	4.83	939,168	400	1.00	-0.28	0.34
ANY-STORE	135.86	3.06	907,120	2,442			

FIGURE 23 Impact of the number of static inspection points on different costs associated with the usage of WHIRO. The last three columns show the Pearson Coefficient (R^2) between the number of SIPs and the instrumentation time, the running time and the size of programs, respectively.

Correlations. The last three columns of Figure 23 show the Coefficient of Determination (using Pearson's coefficient) between the number of static inspection points and the different metrics evaluated in this section. The correlation between the number of SIPs and the instrumentation time is almost a perfect 1.0 in all the three benchmarks. The correlation between the number of SIPs and the running time of the instrumented programs is also strong, except for `SMG2000`. In this case, it is cheaper to instrument stores than return points. This result is not statistically noisy: it is statistically significant with a confidence level of 0.01. Code growth is related to the number of SIPs; however, the correlation is weaker. Even considering ANY-STORE, the number of SIPs is relatively small if compared to the number of potential SIPs in the program (which Figure 18 shows). Nevertheless, the more SIPs we have, the larger the code growth; hence, rank correlation is high: the Kendall Coefficient relating number of SIPs and code growth is a perfect 1.0 in the three cases.

6.6 | RQ6: Potential as Debugging Aid

As mentioned in Section 5.2, WHIRO can be used to pinpoint errors in program transformation techniques like compiler optimizations. A state mismatch between two versions of a program processing the same input is a strong indication of a bug. To analyze the effectiveness of WHIRO as a debugging aid, we manually inserted a bug into the LLVM's sparse conditional constant propagation pass. We altered the optimization lattice by changing the semantics of its meet operator. Whenever we have two constants $c_1 \wedge c_2$, with $c_1 = c_2$, we propagate a random value instead of the constant. Programs were instrumented using PRECISE mode.

[5]In some cases, LLVM's `mergereturn` unifies the exit points in a function by creating an unreachable basic block that post-dominates all the other blocks in the function. This may happen in face of multiple routines that halt the program execution, such as C standard `exit`. In this case, WHIRO creates SIPs before each one of the halting calls to ensure that the state will still be reported. Therefore, there might be more than one inspection point when instrumenting the return point of a function. We can observe that in `SMG2000`, in which WHIRO creates 3 SIPs even when inspecting only `main`.

	basicmath	susan	jpeg	FFT	gsm
Number of different constants found in the program's IR	8	4	1	2	1
Miscomputations of the meet operator	21	1	1	2	1
Named variables with divergence states reported by WHIRO.	2	0	10	2	0

FIGURE 24 Let P be a program, P' a version of P optimized by the correct version of constant propagation, and P'' a version of P optimized by the buggy implementation of constant propagation. The first line shows how often the intermediate representations of P' and P'' contained different constants. The second line shows how often *phi*-functions were wrongly evaluated by the buggy implementation of constant propagation. The third line shows how many named variables, i.e., a variable with a name in P's source code showed different values at corresponding DIPs in P' and P''.

Discussion

The unsound implementation of constant propagation caused bugs in five, out of 16 MiBench programs. Opportunities for injecting bugs did not occur in the other benchmarks. Figure 24 summarizes results. The first row reports the number of different instructions found in a `diff` between programs compiled with the correct and the buggy implementation of constant propagation. The second row reports how often a wrong value was computed in the incorrect implementation of constant propagation. We say that a value was "incorrectly computed" whenever the buggy implementation of constant propagation evaluates a ϕ-function[6] such as $v = \phi(v_1, \dots, v_n)$, where every variable v_i, $1 \leq i \leq n$ is bound to the same constant c. In this case, the wrong implementation of the meet operator will bind variable v to a random constant, instead of c. In total, such wrong propagations happened 26 times throughout programs in MiBench.

The last row shows the number of program variables identified by Whiro with incorrect values due to the bug. Whiro has correctly identified 14 mismatches. The correspondence between mismatches identified by Whiro and wrong evaluations of the meet operator is not perfect because: (i) Whiro only prints out results for *named variables*, that is, variables that have a name in the program's source code. Thus, auxiliary locations created by the compiler are not tracked; and (ii) Whiro, in the Any-Ret mode, only compares state at the return point of functions. Thus, Whiro, in this experiment, will report a mismatch for a variable u that exists in the intermediate representation of LLVM if:

1. Variable u can be replaced by a constant known at compilation time.
2. Variable u depends on a variable $v = \phi(v_1, \dots, v_n)$ that joins variables that are constants known at compilation time.

We emphasize that every bug captured by Whiro is a true positive, meaning that a mismatch in the program state is caused by a miscomputation of the meet operator used in constant propagation. However, the same miscomputation can cause multiple divergences in the program state, because several variables can depend on the same miscomputed value. Such is the case, for instance, of `jpeg`, where a single miscomputation has caused divergences in ten variables that have names in the program's source code. On the other hand, it is also possible that miscomputations do not emerge in the visible program state. Such is the case in `basicmath`, `susan` and `gsm`. In this case, the values that were wrongly computed did not bear influence on the values of any variable that has a name in the program's source code.

6.7 | Summary of Results

Figure 25 summarizes the key results discussed in this section. For reference, the 16 programs in MiBench, together, give us 376,338 LLVM instructions when compiled with `clang -O0 -g` and optimized with two passes: `mem2reg` and `mergereturn`. It takes 115 seconds to compile these programs (with the above flags) in our setup.

[6] Section 3.3 explains the semantics of these special instructions.

	RQ1-Cmp	RQ2-Exe	RQ3-Mem	RQ4-Size	Size
Main	0.01	1	1.07	1.06	381,129
Stack	0.02	1.04	1.07	1.21	409,706
Static	0.01	1	1.07	1.32	720,339
Heap	0.01	1.31	1.63	1.12	395,882
Fast	0.03	1.04	1.07	1.48	747,094
Precise	0.03	1.34	1.63	1.48	747,439

FIGURE 25 Summary of results. **Cmp**: compilation time (Sec. 6.1). This column is the geometric mean of 16 ratios comparing the instrumentation time with the compilation time (`clang -O0 -g`). **Exe**: runtime overhead (Sec. 6.2); **Mem**: Memory consumption (Sec. 6.3); **Size**: code size (Sec. 6.4). These three columns report geometric means of 16 ratios between instrumented and original program. **Size**: number of LLVM instructions in the 16 benchmarks.

7 | RELATED WORK

In terms of purpose, the ability of inspecting program state supports the implementation of lightweight forms of program verification. In terms of implementation, the techniques discussed in this paper are heavily inspired by previous work on garbage collection for type-unsafe languages. In the rest of this section, we discuss how our work fits in these two subfields of the programming languages literature.

7.1 | Compiler Correctness

The area of program verification is immense. Although our work does not deliver formal guarantees about the behavior of programs, it helps developers to debug compilers in at least three ways: (i) automatically producing outputs for benchmarks; (ii) revealing program state to developers; and (iii) checking the behavior of compiler optimizations. Figure 26 outlines where such purposes fit in the broader area of program verification.

Some program generators add verification code to the benchmarks produced. CSmith[42], for instance, uses a checksum based on the values of global non-pointer variables, determined at the end of the program's execution. This checksum roughly corresponds to the lowest level of granularity that we provide: inspection at the end of the program based on statically allocated variables. Richards *et al.*[47] also insert verification code in their synthetic JavaScript benchmarks to record state at multiple points of program execution. Yang *et al.*[48] developed an approach which automatically instruments Register Transfer Level (RTL) modules to assist in debugging of high-level synthesis tools. Their technique reads the execution traces of a program to gather the expected values for the operations in that program. Yang *et al* then insert verification code during the generation of the RTL module, including information about the equivalence between the RTL and the LLVM IR of the program. Nevertheless, the set of programs that can be successfully verified by Yang *et al.*'s technique is substantially smaller than the set of programs that WHIRO handles, because Yang *et al.*'s implementation is restricted to a specific subset of RTL instructions. In contrast to our work, these techniques are tightly coupled with the generation of synthetic codes. Verification is inserted onto synthetic programs, at the time these programs are produced; not in general code, like Section 5.3 shows.

7.2 | Reverse Engineering of Low-Level Code

In many ways, by inspecting the internal state of programs, WHIRO supports a form of reverse engineering. There exists a vast literature on the reverse engineering of binary code[49,50,51,52]. The goal of this kind of techniques is to to recover high-level control structures and types out of the analysis of low-level binary code. In this sense, this work differs from that literature in several ways. First, WHIRO is a dynamic analysis tool. Second, WHIRO requires recompilation, whereas one of the ultimate goals of binary disassemblers is to reconstruct programs without looking into any source-code information.

Nevertheless, there exist dynamic analyses for reverse engineering of binary code[53,54]. These techniques typically build the so-called *dynamic slices* of a program[55,56]. Given a program P, plus an input I of it, the dynamic slice of P in regards to I is

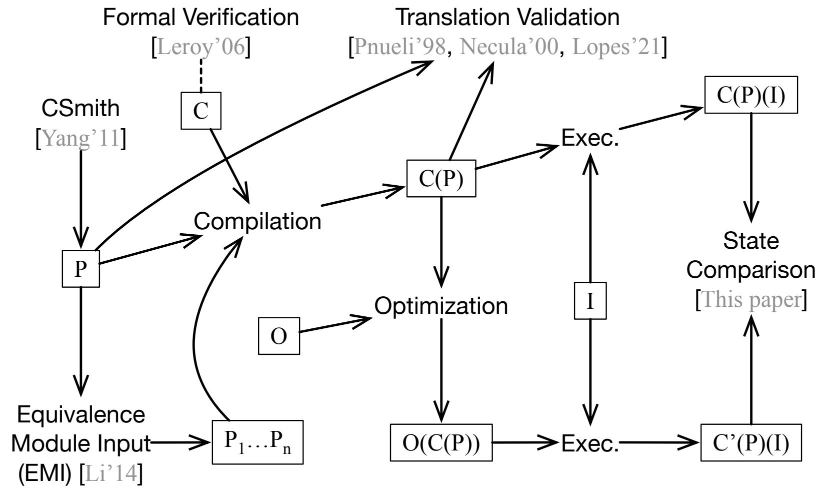

FIGURE 26 Comparison of different techniques to find or prevent bugs in compilers. A formally verified compiler (C) is correct by construction[41]. Program synthesizers such as `CSmith`[42] create programs (P) that are given as inputs to compilers to test them. Variations of P, e.g., $P_1, \ldots, P_n$, can be created via program mutation[43,44]. Translation validation[45,23,29,28,46] checks that the compiler's output, e.g., the compiled program $C(P)$, is correct. Our technique can be used to check the outcome of compiler optimizations (O). Given a compiled program $P_c = C(P)$, and its optimized version $P_o = O(C(P))$, inspection points let developers match the final state of P_c and P_o for any input I.

the subprogram formed by the instructions of P that run when P executes after reading I. The reports produced by WHIRO are not dynamic slices in this sense. However, they could be thought as *dynamic data slices*, because they contain the subset of data (in contrast to the subset of instructions) that is visible at a certain execution point.

7.3 | Garbage Collection in Uncooperative Environments

This work was inspired by implementations of garbage collectors for C and C++. These programming languages do not feature garbage collection by default. Nevertheless, different research groups have designed and implemented garbage collectors for them. This task is challenging because the type systems of these languages do not prevent values of numeric type from being treated as pointers. Probably, the most well-known implementation of garbage collection for C is due to Boehm *et al*[5,6]. Those ideas have been extended along different directions[7,8], and still motivate current research efforts[9,10]. All these implementations traverse the graph formed by heap-allocated pointers, although they differ on how they recognize said graph. *Heap Tracing* is also how we collect heap state. The ideas discussed in this paper bear at least two similarities with the work of Boehm *et al*[5,6], namely:

1. One of the key requirements of Boehm *et al*'s work was to limit the overhead imposed by the garbage collector. If the collector is not used, its overhead is barely perceived. Such is also the case of WHIRO, as empirically demonstrated in Section 6.2.
2. Boehm *et al* accept imprecisions. In particular, they do not try to identify integers used as pointers. In other words, neither integers nor pointers are tagged in any way; nor static analyses are required to separate them. Such is also the case of WHIRO, as explained in Section 5.4.

Nevertheless, in its maximum precision, WHIRO departs from the classic implementation of Boehm *et al*. Such divergence happens because, to trace the heap, we organize heap-allocated data into a table (the table H introduced in Definition 4). The heap table not only lets WHIRO traverse the graph formed by program pointers, but also associates these pointers with user-defined names and line locations, for presentation purposes. In contrast, the need for debugging information has never been a requirement in typical implementations of garbage collectors. Thus, at a high level of granularity, our tracing technique would not be a viable alternative to garbage collection in an uncooperative environment: the need to maintain and access a global table imposes a prohibitively large overhead onto programs, as seen in the last row of Figure 25.

8 | CONCLUSION

This paper presented a technique to inspect the internal state of programs. In terms of implementation, this technique borrows most of its ideas from previous research concerning the design and implementation of garbage collectors for C and C++. However, we have redirected these ideas: instead of doing memory management, we give users a human-observable "peephole" into the program state. This peephole can be used in various ways, as Section 5 demonstrates; furthermore, it is adjustable, going from slow/precise to fast/cursory modes.

Known Limitations

This work, which exists today as the WHIRO framework, has limitations. We have discussed some of these limitations throughout this paper. We revisit them in this section as a way to inspire future developments of this project:

Ordering: Accesses to the heap table H are not synchronized; thus, in the face of concurrency, Property 4 is not guaranteed. Lack of ordering complicates matching states of different variations of the same parallel program, as we did in Section 6.6.

Optimizations: WHIRO relies on debugging data to report to users the state of program variables. If some optimization removes this meta-information (or removes the program point altogether), then it will not have any state to report.

Interference: WHIRO modifies the memory layout of a program. Thus, code transformations that rely on the relative position between program data must run after WHIRO. An example would be transformations that ensure control-flow integrity by checking that the return address of a function has not been modified[20].

Usability: as mentioned in Section 5.1, WHIRO provides three built in modules to insert inspection points in programs: MAIN-RET, ANY-RET and ANY-STORE. Currently, users do not have a way to specify particular return points, having to rely on those modules, which are selected through command-line flags.

Performance: as discussed in Section 6.2, at its maximum granularity, WHIRO might slow down a program to the point that running this program becomes only practical for debugging purposes.

These limitations are not fundamental to the goal of inspecting program state; however, they are inherent to our implementation decisions, and represent tradeoffs between practicality and effectiveness. For instance, to be language-agnostic, WHIRO modifies the intermediate representation of a program; yet, to be human-friendly, WHIRO outputs information with regards to the variables in the high-level representation of the program. Thus, optimizations that eliminate static inspection points will prevent it from generating reports concerning the SIP just deleted. Similarly, WHIRO is not an interpreter: its interventions are carried out via machine-code inserted in the target program. Therefore, WHIRO modifies the memory layout of programs. On the one hand, it is intrusive; on the other, its deployment requires only support from the compiler—the operating system and the target architecture are left untouched.

Future Work

The current limitations of WHIRO draw attention to potential extensions of it, the development of which might yield exciting future work. Delivering ordering guarantees in face of concurrency is an interesting challenge. Simply synchronizing access to the memory monitor might be a solution that is too restrictive to the purposes of WHIRO; first, due to the overhead of synchronized memory accesses; second, because this extra synchronization might force ordered execution of statements that were not subject to such constraints before. Thus, care must be taken when inspecting the state of concurrent code, lest the inspector changes the semantics of the subject program. In regards to usability, it is our view that WHIRO should be eventually integrated with a domain-specific language (DSL) that lets users specify the program points where SIPs must be inserted. This approach would decouple the specification of inspection points from the code that inserts them into the program. We believe that the design and implementation of such a DSL would yield a useful and interesting extension of the ideas discussed in this paper.

Software:

WHIRO is publicly available at https://github.com/JWesleySM/Whiro under the GPL-3.0 License. A short video-tutorial explaining how to use it to observe the internal state of programs is available at https://youtu.be/E8xj-KSqe4M.

Acknowledgement

We thank Breno Guimarães, Angélica Moreira and Luigi Soares for proofreading a draft of this paper. We thank Luciano Almeida for contributions made in WHIRO's code base. Fernando Pereira has been supported by CNPq (Grant 406377/2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PRINT). Wesley Magalhães was the recipient of a scholarship from the Brazilian Ministry of Education via CAPES. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-826460). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Program (ASCR SC-21).

References

1. Brusilovsky P. Program Visualization as a Debugging Tool for Novices. In: Association for Computing Machinery. ; 1993; New York, NY, USA: 29–30
2. Zorn B. Comparing Mark-and Sweep and Stop-and-Copy Garbage Collection. In: Association for Computing Machinery. ; 1990; New York, NY, USA: 87–98
3. Wilson PR. Uniprocessor Garbage Collection Techniques. In: Springer-Verlag. ; 1992; Berlin, Heidelberg: 1–42.
4. Mastrangelo L, Ponzanelli L, Mocci A, Lanza M, Hauswirth M, Nystrom N. Use at Your Own Risk: The Java Unsafe API in the Wild. In: Association for Computing Machinery. ; 2015; New York, NY, USA: 695–710
5. Boehm HJ, Weiser M. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 1988; 18(9): 807–820.
6. Boehm HJ. Space efficient conservative garbage collection. *ACM SIGPLAN Notices* 1993; 28(6): 197–206.
7. Henderson F. Accurate garbage collection in an uncooperative environment. *ACM SIGPLAN notices* 2003; 38(2): 256–262.
8. Rafkind J, Wick A, Regehr J, Flatt M. Precise garbage collection for C. In: ACM. ; 2009; New York, NY, USA: 39–48.
9. Lee D, Won Y, Park Y, Lee S. Two-Tier Garbage Collection for Persistent Object. In: Association for Computing Machinery. ; 2020; New York, NY, USA: 1246–1255
10. Banerjee S, Devecsery D, Chen PM, Narayanasamy S. Sound garbage collection for C using pointer provenance. *Proceedings of the ACM on Programming Languages* 2020; 4(OOPSLA): 1–28.
11. Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: IEEE. ; 2004; Washington, DC, USA: 75–.
12. Nethercote N, Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Association for Computing Machinery. ; 2007; New York, NY, USA: 89–100
13. Stallman RM, Pesch R, Shebs S. *Debugging with GDB: The GNU Source-Level Debugger*. USA: GNU Press . 2002.
14. Ingalls D, Miranda E, Béra C, Boix EG. Two decades of live coding and debugging of virtual machines through simulation. *Softw. Pract. Exp.* 2020; 50(9): 1629–1650. doi: 10.1002/spe.2841
15. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: IEEE. ; 2001; Washington, DC, USA: 3–14
16. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc. . 2006.
17. Hathhorn C, Ellison C, Roşu G. Defining the Undefinedness of C. In: Association for Computing Machinery. ; 2015; New York, NY, USA: 336–345

18. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. An Efficient Method of Computing Static Single Assignment Form. In: ACM. ; 1989; New York, NY, USA: 25–35

19. Bonwick J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In: USTC'94. USENIX Association. ; 1994; USA: 6.

20. Cowan C, Pu C, Maier D, et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: SSYM'98. USENIX Association. ; 1998; USA: 5.

21. Zakowski Y, Cachera D, Demange D, et al. Verifying a Concurrent Garbage Collector with a Rely-Guarantee Methodology. *J. Autom. Reason.* 2019; 63(2): 489–515. doi: 10.1007/s10817-018-9489-x

22. Schildt H. *The Annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990*. USA: McGraw-Hill, Inc. . 1990.

23. Lopes NP, Lee J, Hur CK, Liu Z, Regehr J. *Alive2: Bounded Translation Validation for LLVM*: 65–79; New York, NY, USA: Association for Computing Machinery . 2021.

24. Sagiv M, Reps T, Wilhelm R. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 1998; 20(1): 1–50. doi: 10.1145/271510.271517

25. Thakur M, Nandivada VK. Compare Less, Defer More: Scaling Value-Contexts Based Whole-Program Heap Analyses. In: Association for Computing Machinery. ; 2019; New York, NY, USA: 135–146

26. Thakur M, Nandivada VK. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In: Association for Computing Machinery. ; 2020; New York, NY, USA: 27–38

27. Dahiya M, Bansal S. Black-Box Equivalence Checking Across Compiler Optimizations. In: Chang BE., ed. *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. 10695 of *Lecture Notes in Computer Science*. Springer. ; 2017; Berlin, Heidelberg: 127–147

28. Pnueli A, Siegel M, Singerman E. Translation Validation. In: Springer-Verlag. ; 1998; Berlin, Heidelberg: 151–166.

29. Necula GC. Translation Validation for an Optimizing Compiler. In: Association for Computing Machinery. ; 2000; New York, NY, USA: 83–94

30. Armengol-Estapé J, O'Boyle MFP. Learning C to x86 Translation: An Experiment in Neural Compilation. *CoRR* 2021; abs/2108.07639.

31. Cummins C, Petoumenos P, Wang Z, Leather H. Synthesizing Benchmarks for Predictive Modeling. In: IEEE. ; 2017; Piscataway, NJ, USA: 86–99.

32. Cummins C, Petoumenos P, Murray A, Leather H. Compiler Fuzzing Through Deep Learning. In: ACM. ; 2018; New York, NY, USA: 95–105

33. Silva dAF, Kind BC, Souza Magalhães dJW, Rocha JN, Guimarães BCF, Pereira FMQ. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In: IEEE Computer Society. ; 2021; USA: 378–390

34. Armengol-Estape J, Woodruff J, Brauckmann A, Souza Magalhaes dJW, O'Boyle MFP. ExeBench: An ML-scale dataset of executable C functions. In: Association for Computing Machinery. ; 2022; New York, NY, USA: 1–10.

35. Aftandilian EE, Kelley S, Gramazio C, Ricci N, Su SL, Guyer SZ. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In: Association for Computing Machinery. ; 2010; New York, NY, USA: 53–62

36. Grech N, Fourtounis G, Francalanza A, Smaragdakis Y. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. *Proc. ACM Program. Lang.* 2017; 1(OOPSLA). doi: 10.1145/3133892

37. P. JK, Jayaraman S, Jayaraman B, Sethumadhavan M. Finite-state model extraction and visualization from Java program execution. *Softw. Pract. Exp.* 2021; 51(2): 409–437. doi: 10.1002/spe.2910

38. Brade K, Guzdial M, Steckel M, Soloway E. Whorf: A Visualization Tool for Software Maintenance. In: IEEE Computer Society. ; 1992: 148–154

39. Smith C, Strauss J, Maher P. Data Structure Visualization: The Design and Implementation of an Animation Tool. In: Association for Computing Machinery. ; 2010; New York, NY, USA

40. Stasko J. Animating Algorithms with XTANGO. *SIGACT News* 1992; 23(2): 67–71. doi: 10.1145/130956.130959

41. Leroy X. Formal Verification of a Realistic Compiler. *Commun. ACM* 2009; 52(7): 107–115. doi: 10.1145/1538788.1538814

42. Yang X, Chen Y, Eide E, Regehr J. Finding and Understanding Bugs in C Compilers. In: ACM. ; 2011; New York, NY, USA: 283–294

43. Le V, Afshari M, Su Z. Compiler Validation via Equivalence modulo Inputs. In: Association for Computing Machinery. ; 2014; New York, NY, USA: 216–226

44. Sun C, Le V, Su Z. Finding Compiler Bugs via Live Code Mutation. In: Association for Computing Machinery. ; 2016; New York, NY, USA: 849–863

45. Gupta S, Rose A, Bansal S. Counterexample-Guided Correlation Algorithm for Translation Validation. *Proc. ACM Program. Lang.* 2020; 4(OOPSLA). doi: 10.1145/3428289

46. Tristan JB, Govereau P, Morrisett G. Evaluating value-graph translation validation for LLVM. In: ACM. ; 2011; New York, NY, USA: 295–305

47. Richards G, Gal A, Eich B, Vitek J. Automated Construction of JavaScript Benchmarks. *SIGPLAN Not.* 2011; 46(10): 677–694.

48. Yang L, Gurumain S, Fahmy SA, Chen D, Rupnow K. JIT trace-based verification for high-level synthesis. In: IEEE. ; 2015: 228-231.

49. Bao T, Burket J, Woo M, Turner R, Brumley D. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In: SEC'14. USENIX Association. ; 2014; USA: 845–860.

50. Harris LC, Miller BP. Practical Analysis of Stripped Binary Code. *SIGARCH Comput. Archit. News* 2005; 33(5): 63–68. doi: 10.1145/1127577.1127590

51. Lee J, Avgerinos T, Brumley D. TIE: Principled Reverse Engineering of Types in Binary Programs. In: The Internet Society. ; 2011.

52. Meng X, Miller BP. Binary Code is Not Easy. In: Association for Computing Machinery. ; 2016; New York, NY, USA: 24–35

53. Collie B, O'Boyle MFP. Program Lifting using Gray-Box Behavior. In: Lee J, Cohen A., eds. *PACT*IEEE. ; 2021: 60–74

54. Rimsa A, Amaral JN, Pereira FMQ. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* 2021; 51(2): 353–384. doi: 10.1002/spe.2907

55. Agrawal H. *Towards Automatic Debugging of Computer Programs*. PhD thesis. Purdue University, USA; 1992. UMI Order No. GAX92-01293.

56. Agrawal H, Demillo RA, Spafford EH. Debugging with Dynamic Slicing and Backtracking. *Softw. Pract. Exper.* 1993; 23(6): 589–616. doi: 10.1002/spe.4380230603

AUTHOR BIOGRAPHY

José Wesley Magalhães is a Ph.D. Candidate in Computer Science at The University of Edinburgh (UoE). Currently, Wesley is working as a Research Postgraduate in the Institute for Computing and Systems Architecture (ICSA) under the supervision of Professor Michael O'Boyle. He also holds a Master's Degree and a Bachelor's Degree in Computer Science. His research interests include Compilers, Programming Languages Design, Static and Dynamic Analyses, and Benchmark Validation.

Chunhua "Leo" Liao is a senior computer scientist in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory. His research focus has been on software techniques to improve the performance and correctness of parallel programs. His research interests encompass parallel languages, especially OpenMP, optimizing compilers, runtime systems, and programming tools. Dr. Liao received his Ph.D. degree in Computer Science from University of Houston in Aug. 2007. He also holds M.E. and B.E. degrees in Computer Science from Sichuan University in China.

Fernando M. Q. Pereira, got his Ph.D. at the University of California, Los Angeles, in 2008. Since November of 2009 he is an associate professor at the Department of Computer Science of the Federal University of Minas Gerais. He does research in compilers, and is interested in the design and implementation of static analyses and code optimizations. At UFMG he teaches programming languages and compilation technology. Fernando's research is supported by public research agencies, such as INRIA, FAPEMIG, CAPES and CNPq, and by private enterprises, such as Intel, LGE, Google and Maxtrack.

How to cite this article: J. W. Magalhaes, L. Chunhua, and F. Pereira (2022), Automatic Inspection of Program State in an Uncooperative Environment, *SPE.*, *2022;00:0–0*.