Generating Sound and Effective Memory Debuggers

Yan Wang Iulian Neamtiu Rajiv Gupta

Department of Computer Science and Engineering
University of California, Riverside, CA, USA
{wangy,neamtiu,gupta}@cs.ucr.edu

Abstract

We present a new approach for constructing debuggers based on declarative specification of bug conditions and root causes, and automatic generation of debugger code. We illustrate our approach on several classes of bugs, memory or otherwise. For each bug class, bug conditions and their root cause are specified declaratively, in First-order logic, using 1 to 4 predicates. We employ a low-level operational semantics and abstract traces to permit concise bug specification and prove soundness. To facilitate locating bugs, we introduce a new concept of value propagation chains that reduce programmer burden by narrowing the fault to a handful of executed instructions (1 to 16 in our experiments). We employ automatic translation to generate the debugger implementation, which runs on top of the Pin infrastructure. Experiments with using our system on 7 versions of 4 real-world programs show that our approach is expressive, effective at finding bugs and their causes, and efficient. We believe that, using our approach, other kinds of declaratively-specified, provably-correct, auto-generated debuggers can be constructed with little effort.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids, Monitors; D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.4 [*Processors*]: Debuggers; I.2.2 [*Automatic Programming*]: Program synthesis

General Terms Languages, Reliability, Theory, Verification

Keywords Debugging; Fault Localization; Logic Specification; Operational Semantics; Runtime Monitoring

1. Introduction

Debugging is a tedious and time-consuming process for software developers. Debugging-related tasks (i.e., understanding and locating bugs, and correcting programs) can take up to 70% of the total time of software development and maintenance [18]. Therefore, providing effective debugging tools is essential for improving productivity. To assist in the debugging task, both general-purpose debuggers [10, 15, 18, 36], and specialized tools targeting memory bugs (e.g., buffer overflows [6, 25], dangling pointer dereferences [5], and memory leaks [29, 35]) have been developed.

These current debugging approaches have several shortcomings which are more pronounced in the context of memory-related bugs. First, detection of memory-related bugs is tedious using general-purpose debuggers, so programmers have to use tools tailored to specific kinds of bugs; however, to use the appropriate tool the programmer needs to first know what kind of bug is present in the program. Second, when faulty code is encountered during execution, its impact on program execution might be observed much later (e.g., due to a program crash or incorrect output), making it hard to locate the faulty code. Third, debuggers are also written by humans, which has two main disadvantages: (a) adding support for new kinds of bugs entails a significant development effort, and (b) lack of formal verification in debugger construction makes debuggers themselves prone to bugs, which limits their effectiveness.

We propose a novel approach to constructing debuggers that addresses the above challenges, and provide an illustration and evaluation on memory-related bugs. We allow bugs[1] and their root causes to be specified declaratively, using just 1 to 4 predicates, and then use automated translation to generate an actual debugger that works for arbitrary C programs running on the x86 platform. We have proved that bug detection is sound with respect to a low-level operational semantics, i.e., bug detectors fire prior to the machine entering an error state. Our work introduces several novel concepts and techniques, described next.

Declarative debugger specification. In our approach, bugs are specified via *detection rules*, i.e., error conditions that indicate the presence of a fault, defined as First-order logic predicates on abstract states. In Section 2 we show how bug specifications can be easily written. Using detection rules as input, we employ automated translation to generate the debugger implementation; thanks to this translation process, explained in Section 4.1, from 8 lines of specification about 3,300 lines of C code are generated automatically.

Debugger soundness. We use a core imperative calculus that models the C language with just a few syntactic forms (Section 3.1) to help with specification and establishing correctness. We define an operational semantics (Section 3.2) which models program execution as transitions between abstract states Σ; abstract states form the basis for specifying debuggers in a very concise yet effective way. Next, we define error states for several memory bugs, and use the operational semantics (which contains transitions to legal or error states) to prove that the detectors are sound (Appendix A).

Value propagation chains. In addition to bug detection rules, our specifications also contain *locator rules*, which define value propagation chains pointing to the root cause of the bug. These chains drastically simplify the process of detecting and locating the root cause of memory bugs: for the real-world programs we have applied our approach to, users have to examine just *1 to 16* instructions (Section 5.2).

Section 4 describes our implementation and online debugger usage. After a debugging session starts, a monitoring component

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.
Copyright © 2013 ACM 978-1-4503-2100-6/13/06. . . $10.00

[1] We define a *bug* to be a class of faults, for example, a double-free bug refers to all double-free faults present in the program.

maps the actual execution to the abstract machine state. The detectors generated from the bug specification perform online bug detection, i.e., while the program is running. Note that we detect bugs *before* they actually manifest, when the abstract machine is about to enter an error state—this prevents faults from silently propagating and accumulating. Moreover, when a fault is detected, we suspend the execution using a breakpoint, so that developers can examine the state and, with help from bug locators, get to the bug's root cause. Our approach works for arbitrary C programs. The task of monitoring is carried out by an automatically-generated Pin tool.

Experiments with using our system to detect actual bugs in real-world programs show that it is expressive, effective at finding bugs, and has acceptable performance overhead (Section 5).

Prior efforts in this area include memory bug detectors, algorithmic debugging, and monitoring-oriented programming; we provide a comparison with related work in Section 6. However, to the best of our knowledge, our work is the first to combine a concise, declarative debugger specification style with automatic generation of bug detectors and locators, while providing a correctness proof.

Our approach has the following advantages:

1. *Generality.* As we show in Section 2, bug specifications consist of 1 to 4 predicates per bug. Thus, specifications are easy to understand, scrutinize, and extend. Formal definitions of program semantics and error states show that bug detection based on these bug specifications is correct.
2. *Flexibility.* Instead of using specialized tools for different kinds of bugs, the user generates a single debugger that still distinguishes among many different kinds of bugs. Moreover, bug detectors can be switched on and off as the program runs.
3. *Effectiveness.* Bug detectors continuously evaluate error conditions and the user is informed of the error condition (type of bug) encountered before it manifests, e.g., via program crash. Bug locators then spring into action, to indicate the value chains in the execution history that are the root causes of the bug, which allow bugs to be found by examining just a handful of instructions (1 to 16), a small fraction of the instructions that would have to be examined when using dynamic slicing.

2. Bug Specification

Figure 1 provides an overview of our approach. As the program executes, its execution is continuously monitored and x86 instructions are mapped to low-level operational semantics states Σ (described in Section 3.2). For most memory bugs, programmers use an abstraction of the semantics (execution trace σ and redex e), to write bug specifications; the full semantics is available to specify more complicated bugs. Bug detectors and bug locators are generated automatically from specifications. During debugging, detectors examine the current state to determine when an error condition is about to become true, i.e., the abstract machine is about to enter an error state. When that is the case, locators associated with that error condition report the error and its root cause (location) to the programmer. Our debugger is able to simultaneously detect multiple kinds of bugs, as illustrated by the stacked detectors and locators in the figure.

We now present the user's perspective to our approach. Specification is the only stage where the user needs to be creatively involved, as the rest of the process is automatic, thanks to code generation. We first describe the specification process (Section 2.1). Next, we illustrate how our approach is used in practice for memory bugs (Section 2.2) and other kinds of bugs (Section 2.3). Later on (Section 5.2), we demonstrate the effectiveness of our approach by comparing it with traditional debugging and slicing techniques.

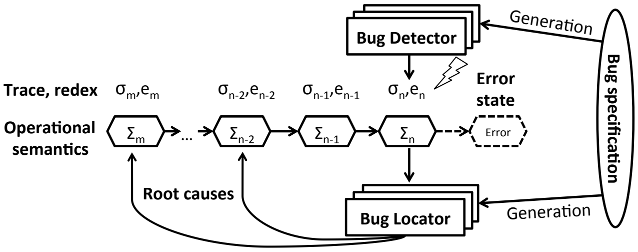


Figure 1. Overview of bug specification, detection and location.

2.1 Specifying Debuggers via Rules

Traces and redexes. To simplify specification, for most memory bugs, the programmers can describe bugs by just referring to traces σ and redexes e. The trace σ records the execution of relevant memory operation events—write for memory writes, malloc for allocation, free for deallocation—which are germane to memory bugs. Redexes e indicate the expression to be reduced next, such as function entry/exit, allocation/deallocation, memory reads and writes; when e is a memory operation, it contains a location r signifying the pointer to be operated on, e.g., freed, read from, or written to. The scarcity of syntactic forms for redexes and execution trace events provide a simple yet powerful framework for specifying C memory bugs.

Rules. To specify a bug kind, the user writes a rule (triple): $<\!detection\ point,\ bug\ condition,\ value\ propagation\!>$. The first two components, $detection\ point$ and $bug\ condition$, specify a bug detector, while the third component, $value\ propagation$, specifies a bug locator. Figure 2 shows how detection points, bug conditions and bug locators are put together to form rules and specify six actual classes of memory bugs. We now proceed to defining each component of a rule.

Detection points specify the reductions where bug detection should be performed, as shown below.

Detection point	**Next reduction** e	**Semantics**
$deref\_r\ r$	$*r$	memory read
$deref\_w\ r$	$r := v$	memory write
$deref\ r$	$*r/r := v$	memory access
$free\ r$	free r	deallocation
$call\ z\ v$	z v	function call
$ret\ z\ v$	ret z e	function return

The programmer only needs to specify the detection point (left column). Our debugger will then evaluate the bug condition when the operational semantics's next reduction is e (middle column). For example, if the programmer wants to write a detector that fires whenever memory is read, she can use $deref\_r\ r$ as a detection point. Detection points which can match multiple reduction rules, coupled with the simple syntax of our calculus, make for brief yet effective specification; for example, using a single detection point, $deref\ r$, the user will at once capture the myriad ways pointers can be dereferenced in C.

Bug conditions are First-order logic predicates which allow memory bugs to be specified in a concise, declarative manner, by referring to the detection point and the trace σ. First, in Figure 2 (bottom) we define some auxiliary predicates that allow more concise definitions for bug detectors. $Allocated(r)$ checks whether pointer r has been allocated. The low-level semantics contains mappings of the form $r \mapsto (bid, i)$, i.e., from pointer r to the block bid and index i it points to; $Bid(r)$ returns r's block in this mapping. Therefore, $Allocated(r)$ is true if the block r is currently pointing into a block bid that according to the trace σ has previ-

Rules	Detection point	Bug condition	Value propagation
[UNMATCHED-FREE]	$detect\langle\sigma; free\ r\rangle$:	$\neg Allocated(r) \vee r \neq Begin(r)$	$VPC(r)$
[DOUBLE-FREE]	$detect\langle\sigma; free\ r\rangle$:	$Allocated(r) \wedge Freed(r, r_1)$	$VPC(r), VPC(r_1)$
[DANGLING-POINTER-DEREF]	$detect\langle\sigma; deref\ r\rangle$:	$Allocated(r) \wedge Freed(r, r_1)$	$VPC(r), VPC(r_1)$
[NULL-POINTER-DEREF]	$detect\langle\sigma; deref\ r\rangle$:	$r = NULL$	$VPC(r)$
[HEAP-BUFFER-OVERFLOW]	$detect\langle\sigma; deref\ r\rangle$:	$Allocated(r) \wedge \neg Freed(r, \_) \wedge (r < Begin(r) \vee r \geq End(r))$	$VPC(r)$
[UNINITIALIZED-READ]	$detect\langle\sigma; deref\_r\ r\rangle$:	$\neg FindLast(\_, \mathsf{write}, r, \_, \_)$	

Auxiliary predicates

$$\begin{array}{lcl} Allocated(r) & \doteq & \exists\ (\_, \mathsf{malloc}, \_, bid)\ \in\ \sigma : bid = Bid(r) \\ Freed(r, r_1) & \doteq & \exists\ (\_, \mathsf{free}, r_1, bid)\ \in\ \sigma : bid = Bid(r) \end{array}$$

Figure 2. Bug detection rules and auxiliary predicates.

ously been allocated, i.e., it contains a malloc event for this bid; '_' is the standard wildcard pattern. $Freed(r, r_1)$ is true if the block bid that r is currently pointing into has been freed, i.e., the trace σ contains a free event for this bid. Note that free's argument r_1, the pointer used to free the memory block, is not necessarily equal to r, as r could be pointing in the middle of the block while r_1 is the base of the block (cf. Section 3.2).

With the auxiliary predicates at hand, we define First-order logic conditions on the abstract domain, as illustrated in the bug condition part of Figure 2. Note that $FindLast(ts, event)$ is a built-in function that traverses the trace backwards and finds the last matching event according to given signature. For example, a dangling pointer dereference bug occurs when we attempt to dereference r whose block has been freed before; this specification appears formally in rule [DANGLING-POINTER-DEREF], i.e., the bug is detected when the redex is $*r$ or $r := v$ and the predicate $Allocated(r) \wedge Freed(r, r_1)$ is true. Note that r_1 is a free variable here and its value is bound to the pointer which is used to free this block for the first time.

Bug locators. The last component of each rule specifies *value propagation chains* (VPC) which help construct bug locators. The VPC of variable *v* in a program state Σ is the transitive closure of value propagation edges ending at Σ for variable *v*. The VPC is computed by backward traversal of value propagation edges ending at Σ for variable *v*. Note that dynamic slicing does not distinguish data dependences introduced by computing values from dependences introduced by propagating existing values. Value propagation edges capture the latter—a small subset of dynamic slices.

For each bug kind, the VPC specifies how the value involved in the bug manifestation relates to the bug's root cause. For example, in [DOUBLE-FREE], the root cause of the bug can be found by tracing the propagation of r (the pointer we are trying to free) and r_1 (the pointer that performed the first free). In [NULL-POINTER-DEREF], it suffices to follow the propagation of the current pointer r which at some point became $NULL$.

2.2 Memory Debuggers in Practice

We now provide a comprehensive account of how our approach helps specify, detect and locate the root causes of memory bugs using three examples of actual bugs in real-world programs.

Double-free. Attempting to free an already-freed pointer is a very common bug. In Figure 2, the rule [DOUBLE-FREE] contains the specification for the bug: when the redex is free r and the predicates $Allocated(r)$ and $Freed(r, r_1)$ are both true, we conclude that r has already been freed.

The real-world program *Tidy-34132* contains a double-free memory bug which manifests itself when the input HTML file contains a malformed *font* element, e.g., $< font\ color\ =$“$green$”$<$ $?font >$. The relevant source code for this bug is presented in the left column of Figure 3. The program constructs a *node* structure for each element (e.g., *font*) in the HTML file. An element may contain multiple attributes corresponding to the *attributes* field of the *node* structure, which is a pointer to the *attribute* structure. The program pushes a deep copy of the *node* structure onto the stack when encountering an inline element (i.e., *font* in our test case) by calling *PushInline* (line 057). The deep copy is created by duplicating the dynamically allocated structure pointed to by each field in the *node* structure as well as fields of fields recursively. For example, the program duplicates the *node*'s *attributes* fields and fields of the *attributes* structures, as shown in line 092. However, the programmer makes a shallow copy of the *php* field in the *attribute* structure by mistake in line 033 because of a missing statement, as shown in line 039. All the copies of *node* structure pushed onto the stack by *PushInline* will be subsequently popped out in function *PopInline* (line 097), where all the allocated regions will be freed recursively. In some situations, due to the shallow copy, the *php* field of some *node* structures will contain dangling pointers. If some element in the HTML file is empty and can be pruned out, the program removes the node from the markup tree and discards it by calling *TrimEmptyElement* (line 309), which eventually calls *DiscardElement* at line 316. Node deletion is just a reverse process of node deep copy—it will free all the dynamically-allocated memory regions in the *node* structure in a recursive fashion, including the structures pointed to by the *php* fields. When providing certain HTML files as input, the program crashes when it tries to trim the empty *font* element because the *php* field of the *attributes* field of the *font* element has been freed in *PopInline*.

The second column of Figure 3 shows the events added to our trace σ during execution (irrelevant events are omitted). As we can see, the bug condition specified in rule [DOUBLE-FREE] is satisfied because σ contains events malloc, n, 1_H, and free, ptr, 1_H (1_H is the heap block id), indicating that block 1_H has been allocated and then freed, which makes $Allocated(r) \wedge Freed(r, r_1)$ true.

The root cause of the double-free bug is the shallow copy in line 033, and the fix (line 039 in istack.c) calls for far more program comprehension (why, when and how the two different pointers wrongly point to the same heap block) than just the positions of the two *free* calls (line 136 in parser.c), which is the best bug report that current automatic debugging tools (e.g., Valgrind) can achieve. With the help of our bug locators, programmers need to examine just 16 instructions to figure out how and when the two pointers used in *free* point to the same memory region by following the value propagation chains for the two pointers (the two pointers can be the same in some situations, in which case the two value propagation chains are exactly the same). We show the value propagation chains for this execution in the third column of Figure 3; in our actual implementation, this value chain is presented to the user. Note that the value of the pointer *ptr* used in the *free* function is first generated in function *malloc* and propagates to pointer *p* in function *MemAlloc*, and so on. The right child of node $\boxed{attrs \rightarrow php}$ is exactly the place where the shallow copy comes from (shallow copy

C code	Relevant events added to the trace σ
savedir.c: 76: char * savedir (const char *dir){ DIR *dirp; 85: dirp = opendir (dir); 86: if (dirp == NULL) 87: return NULL; 129: ...} increment.c: 173: get_directory_contents (char *path){... 180: char *dirp = savedir (path); ... 205: for (entry = dirp; entrylen = 206: strlen (entry))!= 0; **//crash** 207: entry +=entrylen +1)	write, $dirp$, _, savedir $retval$ write, $entry$, _, $dirp$ write, str, _, $entry$ **bug detected at** $\mathrm{strlen}(str)$

Value Propagation Chain

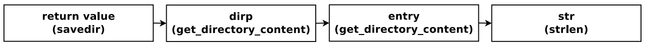


Figure 4. Detecting, and locating the root cause of, a NULL pointer dereference bug in *Tar-1.13.25*.

from $attrs \rightarrow php$ to $newattrs \rightarrow php$). Hence, with the help of our bug locators, programmers can quickly understand the root cause and fix the bug.

NULL pointer dereference. In Figure 2, the rule [NULL-POINTER-DEREF] is used to express and check for NULL pointer dereference bugs. The real-world program *Tar-1.13.25* contains a NULL pointer dereference bug which causes a crash when the user tries to do an incremental backup of a directory without having read access permissions to it. A source code excerpt containing the bug is shown in the first column of Figure 4. If the user does not have read access to the specified directories, the function *opendir* will return a NULL pointer. This causes the program to crash at line 206 when passing this pointer to function *strlen*.

With the help of our debugger, programmers can figure out the bug type, and get significant insight about the failure via bug locators. The trace of an execution which triggers this bug is shown on the right side of Figure 4. The NULL pointer bug detector will detect this bug when the NULL pointer is dereferenced in *strlen*. The value propagation chain of the NULL pointer, shown on the bottom of Figure 4, indicates where the NULL pointer originates (*line 87 in savedir.c*) and how it propagates to the crash point.

Unmatched free. Attempting to free an illegal pointer is a very common bug. In Figure 2, the rule [UNMATCHED-FREE] contains the declarative specification for the bug: whenever the evaluation reaches a point where the next expression is $\mathsf{free}\ r$, if at least one of two conditions is met, the rule fires. If $Allocated(r)$ is false, the program tries to free something that has not been allocated in the first place. If $r \neq Begin(r)$, the program attempts to free a pointer that has been allocated, but instead of pointing to the malloc'd block (i.e., the base), r points somewhere in the middle of the block.

The real-world Python interpreter *Cpython-870c0ef7e8a2*, contains an unmatched free bug (freeing something that has not been allocated) that leads to a crash. The bug manifests when the *type.__getattribute__* function is misused (e.g., *type.__getattribute__(str, int)*) in the input Python program. The *type.__getattribute__(typeName, attrName)* function finds the attribute associated with *attrName* in *typeName*'s attribute list. However, passing a type name, e.g., *int*, as attribute name crashes the program.

C code	Relevant events added to trace σ
unicodeobject.c: 1353:_PyUnicode_Ready(PyObject *unicode){... 1389: _PyUnicode_CONVERT_BYTES(...) 1405: free((PyASCIIObject*)unicode→wstr); 1479: ...} typeobject.c: 2483: type_getattro(type, PyObject* name){ **/*the following statements are missing in buggy code*/** 2488: **if (!PyUnicode_Check(name)) {...** 2492: **return NULL;}** 2517: attribute = _PyType_Lookup(type, name); 2551: PyErr_Format(PyExc_AttributeError, 2552: ”type object ’%.50s’ has no attribute ’%U’”, 2553: type→tp_name, name);	write, ptr, _, $unicode{\rightarrow}wstr$ **bug detected** **at** $\mathrm{free}(ptr)$

Value Propagation Chain

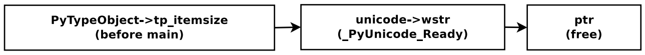

PyTypeObject->tp_itemsize (before main)
unicode->wstr (_PyUnicode_Ready)
ptr (free)

Figure 5. Detecting, and locating the root cause of, an unmatched free bug in *Cpython-870c0ef7e8a2*.

A source code excerpt containing the bug is shown in the first column of Figure 5. Encountering a *type.__getattribute__(typeName, attrName)* statement, the Python interpreter invokes the *type_getattro* function at line 2483 to find the attribute associated with *name* in *type*'s attribute list at line 2517. When no attribute is found, an error message will be printed at line 2551 by calling *PyErr_Format*; *PyErr_Format* will eventually call *_PyUnicode_Ready* to prepare an Unicode string and print it. *_PyUnicode_Ready* converts the Unicode string stored in *unicode→wstr buffer*, and then finally frees the buffer. However, the programmer has wrongly assumed that the *name* object at line 2483 must be an object of type *PyUnicodeObject* or subclass of it (e.g., *PyASCIIObject*), and has forgotten to add a type check at line 2488. When a type name is passed as the attribute name, the *unicode* at line 1405 is an object of type *PyTypeObject*, rather than *PyASCIIObject*. Thus, the programmer thinks *free* is invoked on *PyASCIIObject*'s *wstr* field when in fact it is invoked on *PyTypeObject*'s *tp_itemsize* field.

The second column shows the relevant events added to σ. As we can see, there is no event $\mathsf{malloc}, n, \_$ to make $Allocated(r)$ true. The value propagation chain of ptr (bottom of figure), shows how the wrong value of ptr is propagated from $unicode \rightarrow wstr$, which is a global variable and initialized before the execution of main (by the program loader), rather than dynamically allocated.

2.3 Other Classes of Bugs

While the core of our work is centered around the six classes of memory bugs we have just presented, programmers can use our approach to easily specify debuggers for other classes of bugs. We now proceed to briefly discuss examples of such classes; the bug specifications are presented in Figure 6.

Memory leaks. The rule [POSSIBLE-LEAK] specifies possible leaks as follows: if main is about to exit while the heap H contains one or more blocks that have not been freed, i.e., the heap domain is not empty, the rule fires.

With rule [DEFINITE-LEAK], we report leakages if, at the end of program execution, the heap H contains some blocks that no pointer in P points to. In other words, if there is no live pointer pointing to a block, we report the block as a definite leak.

The rule [LEAK-IN-TS] can be used to detect leaks in transactions. For simplicity, we assume that the scope of a transaction spans the entire body of a function denoted by metavariable ts. The programmer can easily specify that all the blocks allocated inside the transaction (body of ts) should be freed at the end of the transaction. We report leaks if, when function *ts* returns, the heap H contains some blocks which are allocated inside this function and have not been freed yet. Note that $FindLast(k, \mathsf{call}, ts, \_)$ matches the latest event which calls function *ts*, and the free variable *k* is bound

C code	Relevant events added to the trace σ	Value Propagation Chain

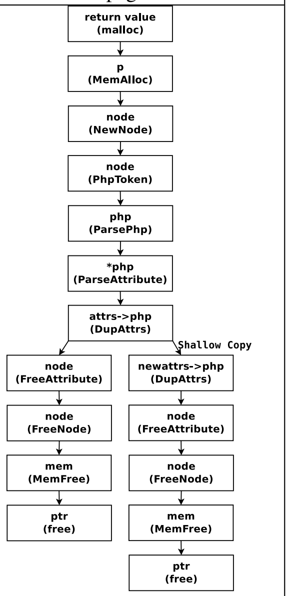
istack.c:
025: AttVal *DupAttrs(TidyDocImpl* doc, AttVal *attrs) {
032: newattrs = NewAttribute();
033: ***newattrs = *attrs;**
034: newattrs → next = DupAttrs(doc, attrs → next); ...
/*the following statement is missing in buggy code*/
039: **newattrs → php=attrs → php? \**
CloneNode(doc, attrs → php):NULL;
041:}
057: void PushInline(TidyDocImpl* doc, Node *node) {...
092: istack → attributes =
DupAttrs(doc, node → attributes);
094:}
097: void PopInline(TidyDocImpl* doc, Node *node) {...
147: while (istack→attributes){ ...
151: FreeAttribute(doc, av); }
parser.c:
128: Node*
DiscardElement(TidyDocImpl* doc, Node *element){
132: if (element){...
136: FreeNode(doc, element); }
140:}
309: Node *
TrimEmptyElement(TidyDocImpl* doc,Node *element){
311: if(CanPrune(doc, element)){...
316: return DiscardElement(doc, element);}
	$\mathsf{malloc}, n, \mathit{1}_H$
	$\mathsf{write}, p, \_, \mathit{return\ value\ of}$ malloc
	$\mathsf{write}, node, \_, p$
	$\mathsf{write}, node, \_, node$
	$\mathsf{write}, php, \_, node$
	$\mathsf{write}, *php, \_, php$
	$\mathsf{write}, attrs \rightarrow php, \_, *php$
	$\mathsf{write}, newattrs \rightarrow php, \_, attrs \rightarrow php$
	$\mathsf{write}, node, \_, newattrs \rightarrow php$
	$\mathsf{write}, mem, \_, node$
	$\mathsf{write}, ptr, \_, mem$
	$\mathsf{free}, ptr, \mathit{1}_H$
	$\mathsf{write}, node, \_, attrs \rightarrow php$
	$\mathsf{write}, mem, \_, node$
	$\mathsf{write}, ptr, \_, mem$
	
	bug detected at $\mathrm{free}(ptr)$
		 return value (malloc)
		p (MemAlloc)
		node (NewNode)
		node (PhpToken)
		php (ParsePhp)
		*php (ParseAttribute)
		attrs->php (DupAttrs)
		Shallow Copy
		node (FreeAttribute)
		node (FreeNode)
		mem (MemFree)
		ptr (free)
		newattrs->php (DupAttrs)
		node (FreeAttribute)
		node (FreeNode)
		mem (MemFree)
		ptr (free)

Figure 3. Detecting, and locating the root cause of, a double-free bug in *Tidy-34132*.

to the timestamp for this event. $Time(bid) > k$ checks whether this block is allocated inside this function (or transaction).

Garbage collector bugs. [GC-BUG] illustrates how to specify one of the basic correctness properties for garbage collector implementations, that the alive bits are set correctly. Consider, for example, a mark-and-sweep garbage collector that uses the least significant bit of each allocated block to mark the block as alive/reachable (bit = 1) or not-alive (bit = 0). We can check whether the alive bits are set correctly at the end of a GC cycle before resetting them (bit = 0), as shown in rule [GC-BUG]: all blocks in H are marked as alive and all blocks in F are marked as freed.

3. Formalism

We now present our formalism: a core imperative calculus that models the execution and memory operations of C programs. We introduce this calculus for two reasons: (1) it drastically simplifies programmer's task of expressing bugs in C programs, by reducing the language to a few syntactic constructs and the dynamic semantics to a handful of abstract state transitions, and (2) it helps prove soundness.[2]

3.1 Syntax

We adopt a syntax that is minimalist, yet expressive enough to capture a wide variety of bugs, and powerful enough to model the actual execution. The syntax is shown in Figure 7. A program consists of a list of top-level definitions d. Definitions can be main, whose body is e, global variables g initialized with value v, and functions f with argument x (which is a tuple in the case of multiple-argument functions) and body e.

Expressions e can take several syntactic forms: values v, explained shortly; variable names x (which represent local variables or function arguments, but not global variables); let bindings; stack allocations $\mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ e$, where variable x is either a local variable or a function argument, n is its size (derived from the x's storage size), and e is an optional initializer; sequencing $e; e$ and function application $e\ e$; function return $\mathsf{ret}\ \mathsf{z}\ e$; conditionals $\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$; $\mathsf{malloc}\ n$, allocating n bytes in the heap; $\mathsf{free}\ r$, deallocating a heap block; pointer dereference $*\,e$; assignment $e := e$; pointer arithmetic $e +_p e$, and integer arithmetic $e + e$. Values v can be integers n, global symbols z, or pointers r. Indexes, e.g., i, j, are integers and are used to specify the offset of a pointer in a memory block. Pointers r range over locations Loc, and are used as keys in a pointer map, as described next; note that we do not assume a specific type (e.g., integer, long) for pointers, as it is not relevant for defining the abstract machine.

$$
\begin{array}{llcl}
\text{Definitions} & d & ::= & \mathsf{main}\ e \\
 & & \mid & \mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d \\
 & & \mid & \mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d \\
\text{Expressions} & e & ::= & v \mid x \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ e \\
 & & \mid & e; e \mid e\ e \mid \mathsf{ret}\ \mathsf{z}\ e \\
 & & \mid & \mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & \mid & \mathsf{malloc}\ n \mid \mathsf{free}\ r \\
 & & \mid & *\,e \mid e := e \mid e +_p e \mid e + e \\
\text{Values} & v & ::= & n \mid \mathsf{z} \mid r \\
\text{Global symbols} & \mathsf{f}, \mathsf{g}, \mathsf{z} & \in & GSym \\
\text{Indexes} & i, j & ::= & n \\
\text{Pointers} & r & \in & Loc \\
\text{Integers} & n & & \\
\text{Variables} & x & &
\end{array}
$$

Figure 7. Syntax.

3.2 Operational Semantics

The operational semantics consists of state and reduction rules. The semantics is small-step, and evaluation rules have the form:

$$\langle H; F; \overline{S}; P; k; \sigma; f; e\rangle \longrightarrow \langle H'; F'; \overline{S}'; P'; k'; \sigma'; f'; e'\rangle$$

[2] Soundness refers to detectors being correct with respect to the operational semantics to help catch specification errors; it does not imply that we certify the correctness of auto-generated and manually-written code for the Pin-based implementation, which operates on the entire x86 instruction set.

Rules	Detection point	Bug condition
[POSSIBLE-LEAK]	$detect\langle H; \sigma; \mathsf{ret}\ main\ v\rangle :$	$dom(H) \neq \emptyset$
[DEFINITE-LEAK]	$detect\langle H; P; \sigma; \mathsf{ret}\ main\ v\rangle :$	$\exists\ bid \in H : \neg(\exists\ r \mapsto (bid, \_) \in P)$
[LEAK-IN-TS]	$detect\langle H; \sigma; \mathsf{ret}\ ts\ v\rangle :$	$\exists\ bid \in H : FindLast(k, \mathsf{call}, ts, \_) \wedge Time(bid) > k$
[GC-BUG]	$detect\langle H; F; \sigma; \mathsf{ret}\ gc\ v\rangle :$	$\neg((\forall\ bid \in H : IsAlive(bid)) \wedge (\forall\ bid \in F : \neg IsAlive(bid)))$
	Auxiliary predicates	$IsAlive(bid) \doteq bid\ \&\ 0x1 = 1$

Figure 6. Bug detection rules and auxiliary predicates for other classes of bugs.

which means expression e reduces in one step to expression e', and in the process of reduction, the heap H changes to H', the freed blocks set F changes to F', the stack $\overline{S}$ changes to $\overline{S}'$, the pointer map P changes to P', the timestamp changes from k to k', the trace changes from σ to σ' and the value origin f changes to f'. We now provide definitions for state elements and then present the reduction rules.

Definitions. In Figure 8 we present the semantics and some auxiliary definitions. In our memory model, memory blocks b of size n are allocated in the heap via $\mathsf{malloc}\ \ n$ or on the stack via $\mathsf{salloc}\ \ n$. Block id's bid are keys in the domain of the heap or the stack; we denote their domain Bid, and represent elements in Bid as $1_H, 2_H, 3_H, \ldots$ (which indicates heap-allocated blocks) and $1_S, 2_S, 3_S, \ldots$ (which indicates stack-allocated blocks). Memory blocks are manually deallocated from the heap via $\mathsf{free}\ r$ and automatically from the stack when a function returns (the redex is $\mathsf{ret}\ \ \mathsf{z}\ \ v$). All the deallocated heap and stack blocks are stored in F—the "freed" set—as (bid, h) and (bid, s) respectively. Block contents b are represented at byte granularity, i.e., $\boxed{v_0, \ldots, v_{n-1}}$; a freshly-allocated block is not initialized, and is marked as $\boxed{\text{junk}}$. The heap H contains mappings from block id's bid to tuples (b, n, k); tuples represent the block contents b, the block size n and the timestamp k when the block was created. A stack frame S consists of mappings $bid \mapsto (b, n, k)$, just like the heap. The stack $\overline{S}$ is a sequence of stack frames.

We keep a pointer map P with entries $r \mapsto (bid, n)$, that is, a map from references to block id bid and offset n. Timestamps k are integers, incremented after each step. The trace σ records timed events ν, i.e., (timestamp, event) pairs. Events ν can be memory writes write, r, v, f which indicate that value v, whose origin was f, was written to location r; $\mathbf{if}$-conditions $n == n'$ which indicate that the value of the $\mathbf{if}$ guard n was n', allocations malloc, n, bid, deallocations free, r, bid, function calls call, z, v and function return ret, z, v. At each step we keep a value origin f that tracks where the last value v comes from: a constant, a global variable, or the prior step(s), as explained shortly. Runtime expressions e are the expressions defined in Figure 7.

We use several notational shorthands to simplify the definition of the rules; they are shown in the top-middle part of Figure 8. Given a pointer r, we can look it up in the heap H or stack $\overline{S}$, extract its bid and index i, and contents $\boxed{v_0, \ldots, v_{n-1}}$. We now explain the shorthands: $Bid(r)$ is the block id; $Idx(r)$ is the pointer's offset; $Begin(r)$ is the beginning address of a block r refers to; $End(r)$ is the end address of a block; $Size(r)$ is the size of the block; $Time(bid)$ is the timestamp at which the block was allocated; $Block(r)$ is the whole block contents; $Value(r)$ is the value stored in the memory unit pointed to by r.

Several other shorthands are defined in the top-right part of Figure 8 as follows: "bid fresh" means the bid is not in the domain of H, F, and $\overline{S}$, and bid has never been used before; $popStack(S, F)$ is used to deallocate all the blocks in the stack S, i.e., for all $bid \in dom(S)$, add (bid, s) to F.

We define a notion of the origin of a value v, denoted $orig(v, f)$, as follows: given a prior origin f, if v is a constant n, then the origin of value v is gen (value v is newly *generated* here); if v is a variable z, then the origin of value v is z (value v is propagated from variable z); otherwise, the origin of value v is f, i.e., the prior origin, indicating it is the result of a prior computation. This origin information is instrumental for constructing bug locators, as it helps track value propagation and hence bug root causes.

We use evaluation contexts $\mathbb{E}$ to indicate where evaluation is to take place next; they are modeled after expressions, and allow us to keep reduction rule definitions simple.

Evaluation rules. Further down in Figure 8 we show the reduction rules. The rule [LET] is standard: when reducing $\mathsf{let}\ x = v\ \mathsf{in}\ e$, we perform the substitution $e[x/v]$. The rule [LET-SALLOC] is used to model the introduction of local variables and function arguments; it is a bit more complicated, as it does several things: first it allocates a new block bid of size n on the stack, initialized to junk, then it picks a fresh r and makes it point to the newly allocated block bid and index 0, and finally substitutes all occurrences of x with r. The allocation rule, [MALLOC], is similar: we model allocating n bytes by picking a fresh bid, adding the mapping $[bid \mapsto (\boxed{\text{junk}}, n, k)$ to the heap, creating a fresh pointer r that points to the newly-allocated block at offset 0, recording the event $(k, \mathsf{malloc}, n, bid)$ in the trace σ, and updating the f to gen, meaning r is newly generated at this step. The deallocation rule, [FREE], works as follows: we first identify the bid that r points to, and then remove the $bid \mapsto (b, n, k_1)$ mapping from the heap, and add the (bid, h) tuple to F; we record the event by adding $(k, \mathsf{free}, r, bid)$ to the trace.

The function call rule, [CALL], works as follows: create an empty stack frame S and push it onto the stack, then rewrite $\mathsf{z}\ \mathsf{v}$ to be $\mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ (x := v; e)$, which means we allocate a new block for the function argument x on the stack, and set up the next reductions to assign (propagate) the value v to x, and then evaluate the function body e; we record the call by adding (k, call, z, v) to the trace, and propagate v's origin; we assume each function body e contains a return expression $\mathsf{ret}\ \mathsf{z}\ e'$. The converse rule, [RETURN], applies when the next expression is a return marker; it pops the current frame S off the stack, deallocates all the blocks allocated in S before, record the return by adding (k, ret, z, v) to the trace, and updates the f to $orig(v, f)$.

Dereferencing, modeled by the rule [READ], entails returning the value pointed to by r, and updating the f to be r, denoting that the origin of value v comes from r. When assigning value v to the location pointed to by r (which resides at block id bid and index i), modeled by the rule [ASSIGN], we change the mapping in the heap or stack (whichever r points to) to b', that is the block contents value at index i is replaced by v; we also record the write by adding $(k, \mathsf{write}, r, v, orig(v, f))$ to the trace, and record the assignment-induced value propagation by setting f to $orig(v, f)$.

Integer arithmetic ([INT-OP]) does the calculation, and updates the f to gen to mark the fact that n_3 is newly generated here; actually this rule is only necessary for purposes of value propagation, as most components of Σ remain unchanged. Pointer arithmetic

Definitions

Block id	bid	$\in$	Bid
Block contents	b	$::=$	$\boxed{v_0, \ldots, v_{n-1}}$
Heap	H	$::=$	$\emptyset$
			$\mid bid \mapsto (b, n, k), H$
Freed blocks	F	$::=$	$\emptyset \mid (bid, h), F$
			$\mid (bid, s), F$
Stack frame	S	$::=$	$\emptyset$
			$\mid bid \mapsto (b, n, k), S$
Stack	$\overline{S}$	$::=$	$\emptyset \mid \overline{S}, S$
Pointers	P	$::=$	$\emptyset \mid r \mapsto (bid, i), P$
Timestamp	k	$::=$	n
Events	ev	$::=$	write, r, v, f
			$\mid n == n'$
			$\mid \mathsf{malloc}, n, bid$
			$\mid \mathsf{free}, r, bid$
			$\mid \mathsf{call}, z, v$
			$\mid \mathsf{ret}, z, v$
Timed events	ν	$::=$	(k, ev)
Traces	σ	$::=$	$\emptyset \mid \nu \cup \sigma$
Value origin	f	$::=$	$gen \mid z \mid r$
Expressions	e	$::=$	...

Shorthands

Given $P[r \mapsto (bid, i)]$

$$\begin{array}{lcl} Bid(r) & \doteq & bid \\ Idx(r) & \doteq & i \end{array}$$

Given $H[bid \mapsto (b, n, k)] \vee \overline{S}[bid \mapsto (b, n, k)]$
and $P[r \mapsto (bid, i)], b = \boxed{v_0, \ldots, v_{n-1}}$

$$\begin{array}{lcl} Begin(r) & \doteq & bid \\ End(r) & \doteq & bid + n \\ Size(r) & \doteq & n \\ Time(bid) & \doteq & k \\ Block(r) & \doteq & b \end{array}$$

Given $H[bid \mapsto (b, n, k)] \vee \overline{S}[bid \mapsto (b, n, k)]$
and $P[r \mapsto (bid, i)], b = \boxed{v_0, \ldots, v_{n-1}}$
and $Begin(r) \leq r < End(r)$
$Value(r) \doteq v_i$

Shorthands (contd.)

$$\begin{array}{rl} bid\ \text{fresh} \doteq & bid \notin Dom(H) \wedge \\ & bid \notin Dom(F) \wedge \\ & bid \notin Dom(\overline{S}) \end{array}$$

$$popStack(S, F) \doteq F \cup \Big(\bigcup_{bid \in dom(S)} (bid, s)\Big)$$

$$orig(v, f) \doteq \begin{cases} gen, & \text{if } v \text{ is a const. } n \\ z, & \text{if, } v \text{ is a gvar. } \mathsf{z} \\ f, & \text{otherwise} \end{cases}$$

Evaluation contexts

$$\begin{array}{lcl} \mathbb{E} & ::= & [\,] \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e \\ & \mid & \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathsf{ret\ z}\ \mathbb{E} \\ & \mid & \mathbb{E};\ e \mid v;\ \mathbb{E} \\ & \mid & \mathsf{malloc}\ \mathbb{E} \mid \mathsf{salloc}\ n\ \mathbb{E} \\ & \mid & \mathsf{free}\ \mathbb{E} \\ & \mid & \mathbb{E} := e \mid r := \mathbb{E} \mid *\mathbb{E} \\ & \mid & \mathbb{E} +_p e \mid r +_p \mathbb{E} \\ & \mid & \mathbb{E} + e \mid n + \mathbb{E} \\ & \mid & \mathsf{if0}\ \mathbb{E}\ \mathsf{then}\ e\ \mathsf{else}\ e \end{array}$$

Evaluation

[LET]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle \longrightarrow \langle H; F; \overline{S}; P; k+1; \sigma; f; e[x/v]\rangle$	
[LET-SALLOC]	$\langle H; F; \overline{S}, S; P; k; \sigma; f; \mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ e\rangle \longrightarrow$ $\langle H; F; \overline{S}, S[bid \mapsto (\boxed{\mathsf{junk}}, n, k)]; P[r \mapsto (bid, 0)]; k+1; \sigma; f; e[x/r]\rangle$	$r \notin Dom(P)$ $\wedge\ bid$ fresh
[MALLOC]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{malloc}\ n\rangle \longrightarrow$ $\langle H[bid \mapsto (\boxed{\mathsf{junk}}, n, k)]; F; \overline{S}; P[r \mapsto (bid, 0)]; k+1; \sigma, (k, \mathsf{malloc}, n, bid); gen; r\rangle$	$r \notin Dom(P)$ $\wedge\ bid$ fresh
[FREE]	$\langle H \uplus bid \mapsto (b, n, k_1); F; \overline{S}; P[r \mapsto (bid, 0)]; k; \sigma; f; \mathsf{free}\ r\rangle \longrightarrow$ $\langle H; F \cup (bid, h); \overline{S}; P; k+1; \sigma, (k, \mathsf{free}, r, bid); f; \mathsf{0}\rangle$	
[CALL]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{z}\ v\rangle \longrightarrow$ $\langle H; F; \overline{S}, S; P; k+1; \sigma, (k, \mathsf{call}, z, v); orig(v, f); \mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ (x := v; e)\rangle$	$z = \lambda x.e, S = \emptyset$
[RETURN]	$\langle H; F; \overline{S}, S; P; k; \sigma; f; \mathsf{ret\ z}\ e\rangle \longrightarrow$ $\langle H; F'; \overline{S}; P; k+1; \sigma, (k, \mathsf{ret}, z, v); orig(v, f); v\rangle$	$F' = popStack(S, F)$
[READ]	$\langle H; F; \overline{S}; P; k; \sigma; f; *r\rangle \longrightarrow \langle H; F; \overline{S}; P; k+1; \sigma; r; v\rangle$	$Value(r) = v \wedge v \neq junk$
[ASSIGN]	$\langle H[bid \mapsto (b, n, k_1]; F; \overline{S}[bid \mapsto (b, n, k_1]; P[r \mapsto (bid, i)]; k; \sigma; f; r := v\rangle \longrightarrow$ $\langle H[bid \mapsto (b', n, k_1]; F; \overline{S}[bid \mapsto (b', n, k_1]; P; k+1;$ $\sigma, (k, \mathsf{write}, r, v, orig(v, f)); orig(v, f); v\rangle$	$b' = b[i \mapsto v]$ $\wedge\ v \neq junk$
[INT-OP]	$\langle H; F; \overline{S}; P; k; \sigma; f; n_1 + n_2\rangle \longrightarrow \langle H; F; \overline{S}; P; k+1; \sigma; gen; n_3\rangle$	$n_3 = n_1 + n_2$
[PTR-ARITH]	$\langle H; F; \overline{S}; P; k; \sigma; f; r +_p n\rangle \longrightarrow$ $\langle H; F; \overline{S}; P[r_2 \mapsto (bid, i+n)]; k+1; \sigma; gen; r_2\rangle$	$Bid(r) = bid,$ $Idx(r) = i, r_2 \notin Dom(P)$
[IF-T]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{if0}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow$ $\langle H; F; \overline{S}; P; k+1; \sigma, (k, n == 0); f; e_1\rangle$	$n = 0$
[IF-F]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{if0}\ n'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow$ $\langle H; F; \overline{S}; P; k+1; \sigma, (k, n == n'); f; e_2\rangle$	$n' \neq 0$
[CONG]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathbb{E}[e]\rangle \longrightarrow \langle H'; F'; \overline{S}'; P'; k'; \sigma'; f'; \mathbb{E}[e']\rangle$	$\langle H; F; \overline{S}; P; k; \sigma; f; e\rangle \longrightarrow$ $\langle H'; F'; \overline{S}'; P'; k'; \sigma'; f'; e'\rangle$

Error rules

[BUG-UNMATCHED-FREE]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; \mathsf{free}\ r\rangle \longrightarrow Error$	$(bid \notin Dom(H)$ $\wedge (bid, h) \notin Dom(F))$ $\vee\ r \neq Begin(r)$
[BUG-DOUBLE-FREE]	$\langle H; F; \overline{S}; P[r \mapsto (bid, 0)]; k; \sigma; f; \mathsf{free}\ r\rangle \longrightarrow Error$	$(bid, h) \in Dom(F)$
[BUG-DANG-PTR-DEREF]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; *r\rangle \longrightarrow Error$	$(bid, h) \in Dom(F)$
[BUG-DANG-PTR-DEREF2]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; r := v\rangle \longrightarrow Error$	$(bid, h) \in Dom(F)$
[BUG-NULL-PTR-DEREF]	$\langle H; F; \overline{S}; P; k; \sigma; f; *r\rangle \longrightarrow Error$	$r = NULL$
[BUG-NULL-PTR-DEREF2]	$\langle H; F; \overline{S}; P; k; \sigma; f; r := v\rangle \longrightarrow Error$	$r = NULL$
[BUG-OVERFLOW]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; *r\rangle \longrightarrow Error$	$bid \in Dom(H) \wedge$ $(r < Begin(r) \vee r \geq End(r))$
[BUG-OVERFLOW2]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; r := v\rangle \longrightarrow Error$	$bid \in Dom(H) \wedge$ $(r < Begin(r) \vee r \geq End(r))$
[BUG-UNINITIALIZED]	$\langle H; F; \overline{S}; P; k; \sigma; f; *r\rangle \longrightarrow Error$	$Value(r) = junk$

Figure 8. Operational semantics (abstract machine states and reductions).

([PTR-ARITH]) is a bit more convoluted: we first find out bid and i—the block id and index associated with r, create a fresh r_2 that now points to block bid and index $i + n$ and add it to P and finally update the f to gen, to record that r_2 is newly generated here.

The conditional rules [IF-T] and [IF-F] are standard, though we record the predicate value and timestamp, i.e., $(k, n == 0)$ and $(k, n! = n')$, respectively, into the trace; predicate values serve as a further programmer aid. The congruence rule, [CONG], chooses where computation is to be applied next, based on the shape of $\mathbb{E}$.

Error rules. The bottom of Figure 8 shows the error state reduction rules. When one of these rules applies, the abstract machine is about to enter an error state—in our implementation, the debugger pauses the execution (breakpoint) just before entering an error state. These rules are instrumental for proving soundness (Appendix A). as they indicate when bug detectors should fire. For brevity, we only define error rules and prove soundness for the bugs in Figure 2. We now proceed to describing the error rules. [BUG-UNMATCHED-FREE] indicates an illegal free r is attempted, i.e., r does not point to the begin of a legally allocated heap block. [BUG-DOUBLE-FREE] indicates an attempt to call free r a second time, i.e., the block pointed to by r has already been freed. [BUG-DANG-PTR-DEREF] and [BUG-DANG-PTR-DEREF2] indicate attempts to dereference a pointer (for reading and writing, respectively) in an already-freed block. Similarly, [BUG-NULL-PTR-DEREF] and [BUG-NULL-PTR-DEREF2] indicate attempts to dereference (read from/write to) a null pointer. Rules [BUG-OVERFLOW] and [BUG-OVERFLOW2] indicate attempts to access values outside of a block. Rule [BUG-UNINITIALIZED] applies when attempting to read values inside an uninitialized block (allocated, but not yet written to).

3.3 Soundness

Intuitively, our soundness property states that bug detectors fire when the underlying machine is about to enter an error state. The proof can be found in Appendix A.

4. Implementation

We now describe our implementation; it consists of an offline translation part that generates the detectors and locators from a bug specification, and an online debugger that runs the program and performs detection/location.

4.1 Debugger Code Generation

From bug specification rules, described in Section 2.1, automated translation via Flex[13] and Bison[14] is used to generate a **detector** and **locator** pair. We illustrate this process using Figure 9 which contains the full bug specification text for double-free bugs as written by the developer.

The translator first generates two helper functions for the *Allocated* and *Freed* predicates, respectively. The *Allocated* helper function parses the tracked event trace (realized by the *state monitoring* runtime library, explained shortly) to find out whether the block associated with r is allocated in the heap. The generated detector checks whether the block pointed to by pointer r is allocated in the heap and freed later whenever the program's execution reaches the start of the *free* function.

Each generated locator computes several value propagation chains based on the bug specification. For example, as shown in Figure 9, two value propagation chains are computed for the two pointers (r and r1) which are used to deallocate the same memory block. Each write event write, r, _, z in the captured trace represents a value propagation edge from z to r. Value propagation chains are computed by traversing the value propagation edges back starting from the error detection point, until gen is encountered.

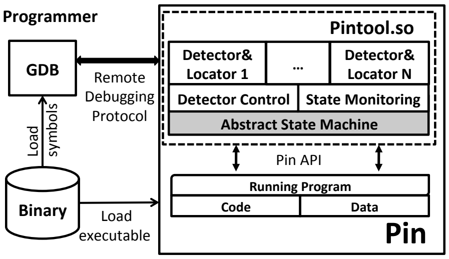


Figure 10. Online debugging process.

4.2 Online Debugging

Figure 10 shows an overview of the online debugger. The implementation runs as two separate processes (GDB and Pin) and consists of several parts: a GDB [15] component, that provides a command-line user interface and is responsible for interpreting the target program's debugging information; a *state monitoring* component, that tracks program execution and translates it into the abstract machine state of our calculus; and a *detector control* component that helps programmers turn detectors on and off on-the-fly. The generated bug detectors, together with the state monitoring and detector control component are linked and compiled to a pintool (a shared library) which is dynamically loaded by the Pin [20] dynamic binary instrumentation tool. Both our state monitoring component and automatically-generated bug detectors are realized by instrumenting the appropriate x86 instructions in Pin. The GDB component communicates with the Pin-based component via GDB's remote debugging protocol.

The *detector control* module allows programmers to turn detectors on and off at runtime. When the program's execution reaches a detection point, all the detectors associated with that detection point are evaluated in the specified order. Whenever any specified bug condition is satisfied, i.e., a bug is detected, our implementation first calls `PIN_ApplicationBreakpoint` to generate a breakpoint at the specified statement, and then generates a bug report which consists of all the concerned events in the bug specification, as a well as the source file name and line number.

The *state monitoring* component, a runtime library, observes the program execution at assembly code level and maps it back to transitions and state changes in the abstract machine state (e.g., H, P, σ) described in Section 3.2. Figure 11 shows a null pointer dereference bug to illustrate how the native x86 execution is mapped to the abstract state transitions in our calculus, as well as the detection points in the detection rules. The first three columns show the code in C, in our calculus, and assembly. Because C implicitly uses dereferenced pointers for stack variables (e.g., *p=1* in C is really *(&p)=1), and our calculus make the implicit dereference explicit, code in our calculus needs one more dereference than code in C (e.g., w:=***p in our calculus corresponds to w=**p in C). In the second column of Figure 11 we append the *_addr* suffix to variables from the first column (e.g., *p* becomes *p_addr*) to avoid confusion.

As we can see, the x86 execution has a straightforward mapping to the state transition in our calculus. For example, the execution of the first *mov (%eax),%eax* instruction is mapped back to the [READ] evaluation rule with r being *p_addr* (where r is stored in register *eax* here), while the second *mov (%eax),%eax* is mapped back to the same rule with r being *p_addr in our calculus. Meanwhile, each binary instruction has a natural mapping to the detection points (shown in the fourth column). For example, the first *mov (%eax),%eax* instruction corresponds to both $deref\_r\ p\_addr$ and $deref\ \ p\_addr$ detection points. That is, all the bug detectors associated with $deref\_r\ \ r$ or $deref\ \ r$ detection points are evaluated when the program is about to execute this instruction.

```
define  Allocated(r)  =  exists event(_, malloc, _, bid) in Trace suchthat (bid == Bid(r))
define  Freed(r, r1)  =  exists event(_, free, r1, bid)  in Trace suchthat (bid == Bid(r))
[double_free] detect <Trace; free r>:      Allocated(r) && Freed(r, r1)      :VPC(r), VPC(r1)
```

Figure 9. Actual bug specification input for double-free bugs.

We generate the recording infrastructure after parsing the specifications, and only activate the required event trackers (e.g., we only activate *malloc* and *free* event trackers for double-free bugs).

Next we describe maintaining state transitions for the pointer mapping P. A block id is assigned to each allocated block, and the block id is increased after each allocation. Unique block ids ensure the detection of dangling pointer dereference bugs even when a memory block is reused. Each pointer is bound with the block id and index of the block pointed to by shadow memory. We implement the pointer mapping transition by propagating the shadow value of each pointer along with the pointer arithmetic operation. Although we only need the mapping for pointers, we temporarily maintain mapping information for registers. The fifth column in Figure 11 shows an example of how the pointer mapping is changed by propagating the shadow value for the execution of assembly code given in the third column. For example, the *malloc* function returns the address of the allocated block (e.g., the block id is 1_H) in the register *%eax*, we shadow *%eax* to (1_H, 0), denoted by $P[\%eax \mapsto (1_H, 0)]$ in Figure 11. The mapping info is propagated from register *%eax* into *p* after the execution of $mov\ \%eax, -0x10(\%ebp)$, denoted by $P[-0x10(\%ebp) \mapsto (1_H, 0)]$ in Figure 11, which means that pointer *p* points to the first element inside block 1_H. Suppose two bug detectors are generated based on the buffer overflow and null pointer dereference specifications in Figure 2. Then when the program's execution reaches the first *mov (%eax),%eax* instruction, we are at a `deref` *r* detection point (*r* is stored inside register *%eax*), and pointer mapping information for register *%eax* contains the pointer mapping information for *r* here($P[\%eax \mapsto (1_H, 0)]$). By evaluating the two detectors, none of the bug conditions are satisfied. The pointer mapping for register *%eax* is set to invalid (denoted by (x,x) in Figure 11) due to the assignment. The execution continues to the second *mov (%eax),%eax* instruction, and the null pointer dereference bug is reported because $r == 0$ is satisfied here (*r* is stored in register *%eax* and its value equals zero).

Value origin tracking is implemented similarly to pointer mapping. Each variable and register is tagged with a shadow origin of its value, and whenever the next expression to reduce is $r := v$, we update the origin (shadow value) of r to be the origin of v, and we record r and its new origin in the trace.

Storing all the tracked events and value propagations in memory may cause the debugger to run out of memory for long-running programs. Older events, which are unlikely to be accessed, can be dumped to disk and reloaded into memory if needed. However, we did not encounter this problem for our examined programs.

5. Experimental Evaluation

We evaluate our approach on several dimensions: *efficiency*, i.e., the manual coding effort saved by automated generation; *effectiveness/coverage*, i.e., can we (re)discover actual bugs in real-world programs; and *performance* overhead incurred by running programs using our approach.

5.1 Efficiency

We measure the efficiency of our debugger code generation by comparing the lines of code of the bug specification and the generated C implementation. For each kind of bug, we specify the bug detector and bug locator as shown in Figure 2. Table 1 shows the comparison of lines of codes for bug specification and generated debugger for each kind of bug and all bugs combined.

Program name	Traditional debugging	Dynamic slicing	VPC
Tidy-34132-double-free	28,487	4,687	16
Tidy-34132-null-deref	55,777	13,050	4
Bc-1.06	42,903	19,988	1
Tar-1.13.25	74	7	4
Cpython-870c0ef7e8a2	20,719	13,136	2
Cpython-2.6.8	1,083	444	10
Cpython-08135a1f3f5d	270,544	135,366	1
Cpython-83d0945eea42	11,916	7,285	2

Table 3. Debugging effort: instructions examined.

Since detectors use the same model (detection point and predicates on the abstract machine state), and share the code for the state monitoring library, the generated code for all detectors combined is 3.3 KLOC, while for a single detector, the code size ranges from 2.2 to 2.4 KLOC. Note that the generated implementations are orders of magnitude larger than the bug specifications.

5.2 Debugger Effectiveness

A summary of benchmarks used in our evaluation is shown in Table 2; each benchmark contains a real reported bug, with the details in columns 3–6. We now provide brief descriptions of the experience with using our approach to find and fix these bugs. Note that three of the bugs were presented in detail in Section 2.2, hence we focus on the remaining five bugs.

In addition to the double-free bug, *Tidy-34132* also contains a NULL pointer dereference which manifests when the input HTML file contains a nested *frameset*, and the *noframe* tag is unexpectedly included in the inner *frameset* rather than the outer one, which causes function *FindBody* to wrongly return a null pointer.

Bc-1.06 fails with a memory corruption error due to heap buffer overflow (variable *v_count* is misused due to a copy-paste error).

Cpython-2.6.8 has a double-free memory bug when there is a folder in the current directory whose name is exactly the same as a module name, and this opened file is wrongly closed twice, resulting in double-freeing a *FILE* structure. *Cpython-08135a1f3f5d* crashes due to a heap buffer overflow which manifests when the *imageop* module tries to convert a very large RGB image to an 8-bit RGB. *Cpython-83d0945eea42* fails due to a null pointer dereference when the *_pickle* module tries to serialize a wrongly-initialized object whose *write_buf* field is null.

It can be easily seen that the benchmark suite includes bugs from our detector list and that all the bugs come from widely-used applications. Thus, this benchmark suite is representative with respect to debugging effectiveness evaluation.

All the bugs were successfully detected using the debuggers generated from the specifications in Figure 2. However, we did find several cases of false positives. Because our approach is based on Pin, which cannot track code execution into the kernel for system calls, our generated debuggers detected some false positives (uninitialized reads). This limitation can be overcome by capturing system call effects [26], a task we leave to future work.

We now quantify the effectiveness of our approach by showing how locators dramatically simplify the process of finding bug root causes. We have conducted the following experiment: we com-

C	Our calculus	Assembly code	Detection points	Tracked pointer mapping	Additions to σ

int w;
int **p;
p=(**int****)
malloc(4);
*p=0;
 	
w=**p;
	let w_addr=salloc 4 in
	let p_addr=salloc 4 in
	p_addr:=malloc 8;
	
	*p_addr:=0;
	
	w_addr:=***p_addr;
		$call\ malloc$
		$mov\ \%eax, -0x10(\%ebp)$
		$mov\ -0x10(\%ebp), \%eax$
		$movl\ \$0x0, (\%eax)$
		$mov\ -0x10(\%ebp), \%eax$
		$mov\ (\%eax), \%eax$
		$mov\ (\%eax), \%eax$
		$mov\ \%eax, -0xc(\%ebp)$
			deref_w/deref p_addr
			deref_r/deref p_addr
			deref_w/deref *p_addr
			deref_r/deref p_addr
			deref_r/deref *p_addr
				$P[\%eax \mapsto (1_H, 0)]$
				$P[-0x10(\%ebp) \mapsto (1_H, 0)]$
				$P[\%eax \mapsto (1_H, 0)]$
				$P[(\%eax) \mapsto (x, x)]$
				$P[\%eax \mapsto (1_H, 0)]$
				$P[\%eax \mapsto (x, x)]$
					(malloc, n, 1_H)
					(write, p_addr, _, malloc $retval$)
					(write, $*p\_addr$, _, gen)
					bug detected at
					deref $*p\_addr$

Figure 11. State transition for a null pointer dereference bug.

Lines of code	Unmatched Free	Double Free	Dangling Ptr. Deref.	Null Ptr. Deref.	Heap Buffer Overflow	Uninitialized Read	**Total**
Specification	2	3	3	1	3	1	8
Generated debugger	2.3K	2.4K	2.4K	2.2K	2.3K	2.2K	3.3K

Table 1. Debugger code generation efficiency: comparison of lines of specification and generated debuggers for different bugs.

Program Name	**LOC**	**Bug type**	**Bug location**	**Bug source**	**Program description**
Tidy-34132	35.9K	Double Free	istack.c:031	BugNet [27]	Html checking & cleanup
Tidy-34132	35.9K	Null Pointer Dereference	parser.c:161	BugNet [27]	Html checking & cleanup
Bc-1.06	17.0K	Heap Buffer Overflow	storage.c:176	BugNet [27]	Arbitrary-precision Calculator
Tar-1.13.25	27.1K	Null Pointer Dereference	incremen.c:180	gnu.org/software/tar/	Archive creator
Cpython-870c0ef7e8a2	336.0K	Unmatched Free	typeobject.c:2490	http://bugs.python.org	Python interpreter
Cpython-2.6.8	336.0K	Double Free	import.c:2843	http://bugs.python.org	Python interpreter
Cpython-08135a1f3f5d	387.6K	Heap Buffer Overflow	imageop.c:593	http://bugs.python.org	Python interpreter
Cpython-83d0945eea42	271.1K	Null Pointer Dereference	_pickle.c:442	http://bugs.python.org	Python interpreter

Table 2. Overview of benchmark programs.

pute the number of instructions that would need to be examined to find the root cause of the bug in three scenarios: *traditional debugging*, *dynamic slicing*[37], and *our approach*. We present the results in Table 3. Traditional debugging refers to using a standard debugger, e.g., GDB, where the programmer must trace back the execution starting from the crash point to the point that represents the root cause. For the bugs considered, this would require tracing back through the execution of 74 to 270,544 instructions, depending on the program. When dynamic slicing is employed, the programmer traces back the execution along dynamic dependence edges, i.e., only a relevant subset of instructions need to be examined. Breadth-first traversal of dependence chains until the root cause is located leads to tracing back through the execution of 7 to 135,366 instructions, depending on the program. In contrast, in our approach, the programmer will trace back through the execution along value propagation chains which amounts to the examination of just 1 to 16 instructions. Hence, our approach reduces the debugging effort significantly, compared to traditional debugging and dynamic slicing.

5.3 Performance

The focus of our work was efficiency and effectiveness, so we have not optimized our implementation for performance. Nevertheless, we have found that the time overheads for generated monitors and locators are acceptable for interactive debugging. When measuring overhead, we used the same failing input we had used for the effectiveness evaluation. We report the results in Table 4. We use the "Null Pin" (the program running under Pin without our debugger) time overhead as the baseline, which is shown in the first column, and the time overhead with all detectors on is in the second column. The third column shows the time overhead with all detectors on and value propagation on. All experiments were conducted on a DELL PowerEdge 1900 with 3.0GHz Intel Xeon processor and 3GB RAM, running Linux, kernel version 2.6.18.

Program name	**Null Pin** seconds	**Bug detect** seconds (factor)	**Bug detect&VP** seconds (factor)
Tidy-34132-double-free	0.77	6.05 (7.9x)	7.62 (9.9x)
Tidy-34132-null-deref	0.62	4.52 (7.3x)	5.58 (9.0x)
Bc-1.06	0.62	4.61 (7.4x)	5.70 (9.2x)
Tar-1.13.25	1.08	5.89 (5.5x)	7.43 (6.9x)
Cpython-870c0ef7e8a2	3.95	59.21 (15.0x)	80.84 (20.5x)
Cpython-2.6.8	3.31	33.16 (10.0x)	41.35 (12.5x)
Cpython-08135a1f3f5d	2.95	32.03 (10.9x)	40.13 (13.6x)
Cpython-83d0945eea42	3.17	54.21 (17.1x)	63.83 (20.1x)

Table 4. Program execution times (from start to bug-detect), when running inside our debugger.

From Table 4, we can see that the time overhead incurred by all bug detectors ranges from 5.5x to 17.1x compared to the baseline, while the time overhead incurred by all bug detectors and value propagation ranges from 6.9x to 20.5x. We believe this overhead is acceptable and a worthy tradeoff for the benefits of our approach.

When running the programs inside our debugger we have found that (1) running time increases linearly with the number of bug detectors enabled, and (2) even with the overhead imposed by our dynamic approach with all detectors and value propagation on, real-world programs took less than 81 seconds to crash on inputs that lead to bug manifestation. These results demonstrate that the overhead is acceptable and our approach appears promising for debugging tasks on realistic programs.

6. Related Work

Memory debuggers. A number of works aim to handle multiple kinds of memory bugs [2, 11, 28]. DieHard [2] is a unified algorithm for memory management for avoiding memory errors. Purify [11] and Valgrind Memcheck [28] detect memory bugs using dynamic binary instrumentation. Bond et al.'s approach [3] tracks the origins of unusable values; however, it can only track the origin of Null and undefined values while our VPCs capture not only origin, but propagation for any specified variable. VPCs are therefore much more useful. E.g., for a double-free bug, the origin of pointer *r* used in the second free is always the return value of *malloc* (which is not very informative). These approaches are specialized to find a reduced class of memory bugs, and the bug detection is "hard-coded". Our approach permits very easy extensibility to new kinds of bugs via specification and code generation; we also present a soundness proof to show that the debuggers specification are correct.

Advanced debugging and bug finding. MemTracker [34] provides a unified architectural support for low-overhead programmable tracking to meet the needs for different kinds of bugs. FindBugs [12] leverages bug patterns to locate bugs and Algorithmic (or declarative) debugging [32] is an interactive technique where the user is asked at each step whether the prior computation step was correct [33]. Program synthesis has been used in prior work to automatically generate programs from specifications at various levels: types [21], predicates or assertions/goals [22]; however no prior work on synthesis has investigated specification at the operational semantics level in the context of debugging.

Runtime verification and dynamic analysis. Monitor-oriented programming (MOP) [24] and Time Rover [16] allow correctness properties to be specified formally (e.g., in LTL, MTL, FSM, or CFG); code generation is then used to yield runtime monitors from the specification. Monitor-oriented programming (MOP) [4, 24] combines formal specification with runtime monitoring. In MOP, correctness properties can be specified in LTL, or as FSM, or as a CFG. Then, from a specification, a low-overhead runtime monitor is synthesized to run in AspectJ (i.e., use aspect-oriented programming [17] in JavaMOP [4]) or on the PCI bus (in BusMOP [30]) to monitor the program execution and detect violations of the specification. Time Rover [7, 16] combines LTL, MTL and UML specification with code generation to yield runtime monitors for formal specifications.

PQL [23] and PTQL [9] allow programmers to query the program execution history, while tracematches [1] allows free variables in trace matching on top of AspectJ. GC assertions [31] allow programmers to query the garbage collector about the heap structure. Jinn [19] synthesizes bug detectors from state machine for detect foreign function interface.

Ellison and Roşu [8] define a general-purpose semantics for C with applications including debugging and runtime verification; in our semantics we only expose those reduction rules that help specify memory debuggers, but our approach works for the entire x86 instruction set and sizable real-world programs including library code.

Compared to all these approaches, our work differs in several ways: the prior approaches are adept at specifying properties and generating runtime checkers (which detect what property has been violated), whereas ours points out *where*,*why*, and *how* a property is violated; also, we introduce value propagation chains to significantly reduce the effort associated with bug finding and fixing.

7. Conclusions

We have presented a novel approach to constructing memory debuggers from declarative bug specifications. We demonstrate that many categories of memory bugs can be specified in an elegant and concise manner using First-order logic; we then prove that bug specifications are sound, i.e., they do not miss bugs that manifest during execution. We show that from the concise bug specifications, debuggers that catch and locate these bugs can be generated automatically, hence programmers can easily specify new kinds of bugs. We illustrate our approach by generating debuggers for six kinds of memory bugs. Experiments with using our approach on real-world programs indicate that it is both efficient and effective.

Acknowledgments

This research is supported by the National Science Foundation grants CCF-0963996 and CCF-1149632 to the University of California, Riverside.

A. Soundness Proof

We use Σ as a shorthand for a legal state $\langle H; F; \overline{S}; P; k; \sigma; f; e\rangle$, and $Error$ as a shorthand for an error state. Hence, the condensed form of the reduction relation for legal transitions is $\Sigma \longrightarrow \Sigma'$, while transitions $\Sigma \longrightarrow Error$ represent bugs. We name state Σ as $\Sigma_e = \langle H_e; F_e; \overline{S}_e; P_e; k_e; \sigma_e; f_e; e\rangle$, and Σ' as $\Sigma'_e = \langle H'_e; F'_e; \overline{S}'_e; P'_e; k'_e; \sigma'_e; f'_e; e'\rangle$. If expression e generates an event, we name it as ν_e, that is, $\sigma'_e = \sigma_e, \nu_e$.

At a high level, our notion of soundness can be expressed as follows: if the abstract machine, in state Σ, would enter an error state next, which is the "ground truth" for a bug, then the user-specified bug detectors, defined in terms of just e and σ, must fire.

The proof of soundness relies on several key definitions and lemmas. We first define well-formed states, then prove that reductions to non-error states preserve well-formedness, and finally the soundness theorem captures the fact that the premises of error transition rules in fact satisfy the user-defined bug specification, hence bugs will be detected. Well-formed states are defined as follows:

Definition A.1 (Well-formed states). *A state $\Sigma = \langle H; F; \overline{S}; P; k; \sigma; f; e\rangle$ is well-formed if:*

1. $H \cap F = \emptyset$
2. $(H \cup F) \cap (\bigcup_{S \in \overline{S}} S) = \emptyset$

The first part says that block id's cannot simultaneously be in the heap H and in the freed set F, while the second part ensures that block id's cannot simultaneously be both in the heap H (or the freed set F) and in the stack $\overline{S}$.

Lemma A.2 (Empty initial state). *Suppose the initial state, denoted Σ_0, is $\langle H_0; F_0; \overline{S}_0; P_0; k_0; \sigma_0; f_0; a\rangle$. In this state the heap, stack, and freed set are empty, i.e.:*

1. $H_0 = \emptyset$
2. $\overline{S}_0 = \emptyset$
3. $F_0 = \emptyset$

Proof. The lemma holds by construction. □

Lemma A.3 (Well-formed initial state). Σ_0 *is well-formed.*

Proof. The lemma follows immediately from Definition A.1. □

Next, we introduce a lemma to prove that non-error transitions keep the state well-formed.

Lemma A.4 (Preservation of well-formedness). *If Σ is well-formed and $\Sigma \longrightarrow \Sigma'$, and Σ' is not an error state, then Σ' is well-formed.*

The proof is by induction on the reduction $\Sigma \longrightarrow \Sigma'$. Intuitively, this lemma states that, since the state always stays well-formed during non-error reductions, memory bugs cannot "creep in" and manifest later, which would hinder the debugging process.

Lemma A.5 (Trace prefixes). *If expression* **a** *is reduced before expression* **b***, then:*

1. *σ_a is a prefix of σ_b, denoted as $\sigma_a \sqsubseteq \sigma_b$.*
2. *$F_a \subseteq F_b$.*

Proof. By induction on the length of the reduction. □

Lemma A.6 (Order of events)**.** *Suppose the current state is $\Sigma_e = \langle H_e; F_e; \overline{S}_e; P_e; k_e; \sigma_e; f_e; e\rangle$, i.e., the redex is e, and there exists an event $\nu_a \in \sigma_e$. Then the expression a generating event ν_a has already been reduced (which implies a is reduced before e).*

Proof. By contradiction. □

Intuitively, Lemmas A.5 and A.6 show that reduction order is consistent with the order of corresponding events in the trace σ.

Lemma A.7. *Suppose we are in state*

$$\Sigma_e = \langle H_e; F_e; \overline{S}_e; P_e; k_e; \sigma_e; f_e; \mathsf{free}\ r_e\rangle$$

i.e., the redex is $e = \mathsf{free}\ r_e$, and we have $Bid(r_e) \notin Dom(H_e) \wedge (Bid(r_e), h) \notin Dom(F_e)$. Then there does not exist an event $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_e$ such that $bid_a = Bid(r_e)$.

Proof. Suppose there exists an event $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_e$ such that $bid_a = Bid(r_e)$. It is easy to see that the expression generating ν_a is $a = \mathsf{malloc}\ n_a$ and the reduction rule is [MALLOC]. $bid_a = Bid(r_e) \in Dom(H'_a)$ by rule [MALLOC]. Expression a is reduced before e by Lemma A.6. There are two subcases:

(a) there exists an expression $c = free\ r_c$ reduced after a and before e, where $Bid(r_c) = bid_a = Bid(r_e)$. Because expression c does not get stuck, the rule applied must be rule [FREE]. Then $(Bid(r_e), h) = (Bid(r_c), h) \in Dom(F'_c)$ by rule [FREE]. By Lemma A.5, $F'_c \subseteq F_e$, then $(Bid(r_e), h) = (Bid(r_c), h) \in Dom(F_e)$: a contradiction.

(b) there does not exist an expression $c = \mathsf{free}\ r_c$ reduced after a and before e, where $Bid(r_c) = bid_a = Bid(r_e)$, then $Bid(r_e) = bid_a \in Dom(H_e)$: a contradiction. □

Theorem A.8 (Soundness)**.** *Let the current state be Σ, where $\Sigma \neq Error$, the current trace be σ and the redex be $\mathbf{e}$. Suppose $\mathbf{p}$ is a bug detector, i.e., a predicate on σ and $\mathbf{e}$, and* [BUG-P] *is an error rule associated with the detector. If the machine's next state is an $Error$ state (*[BUG-P] $\Sigma \longrightarrow Error$*) then the detector fires, i.e., predicate $\mathbf{p}$ is true.*

Proof. By induction on the reduction $\Sigma \longrightarrow \Sigma'$. Proceed by case analysis. Suppose the last reduction rule applied is R, there are nine cases for R.

case [BUG-UNMATCHED-FREE] **:**
The redex must be an expression b such that $b = \mathsf{free}\ r_b$, and $(Bid(r_b) \notin Dom(H_b) \wedge (Bid(r_b), h) \notin Dom(F_b)) \vee r_b \neq Begin(r_b)$ is true by the premise of the rule [BUG-UNMATCHED-FREE]. There are two subcases:

case $Bid(r_b) \notin Dom(H_b) \wedge (Bid(r_b), h) \notin Dom(F_b)$ holds **:**
By Lemma A.7, there does not exist an event $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_b$ such that $bid_a = Bid(r_b)$, thus $\neg Allocated(r_b)$ holds, hence $\mathbf{p}$ holds and detector [UNMATCHED-FREE] fires.

case $r_b \neq Begin(r_b)$ holds **:**
It is easy to see that the predicate for the unmatched free bug specified in Figure 2 fires if this error state is encountered; expression b corresponds to the detection point $free\ r_b$, and $\neg Allocated(r_b) \vee r_b \neq Begin(r_b)$ holds, hence $\mathbf{p}$ holds and detector [UNMATCHED-FREE] fires.

case [BUG-DOUBLE-FREE] **:**
The redex must be $b = \mathsf{free}\ r_b$, and $(Bid(r_b), \mathrm{h}) \in Dom(F_b)$ holds by the premise of the rule [BUG-DOUBLE-FREE]. By Lemma A.2, $F_0 = \emptyset$, hence there must exist an expression a, reduced before b, that changes F. The only rule which changes F is rule [FREE]. That is, there exists an expression $a = free\ r_a$, and $\nu_a = (k_a, \mathsf{free}, r_a, bid_a) \in \sigma'_a$, such that $(bid_a, h) = (Bid(r_b), h) \in Dom(F_b)$, and $bid_a \in Dom(H_a)$ by rule [FREE]. By Lemma A.5, $\sigma'_a \sqsubseteq \sigma_b$, so $\nu_a \in \sigma_b$.
Similarly, by Lemma A.2, $H_0 = \emptyset$; hence there must exist an expression c reduced before a and its corresponding reduction rule expands H (the only rule is [MALLOC]). That is, $c = \mathsf{malloc}\ n_c$, and $\nu_c = (k_c, \mathsf{malloc}, n_c, bid_c) \in \sigma'_c$, such that $bid_c = bid_a \in Dom(H_a)$. By Lemma A.5, $\sigma'_c \sqsubseteq \sigma_b$, so $\nu_c \in \sigma_b$.
It is easy to see that the detector [DOUBLE-FREE] specified in Figure 2 fires if this error state is encountered (expression b corresponds to the detection point $free\ r_b$, and the $Allocated(r_b)$ and $Freed(r_b)$ auxiliary predicates are true, because of the existence of event ν_c and ν_a in σ_b respectively, hence $\mathbf{p}$ holds).

case [BUG-DANG-PTR-DEREF] **:**
The redex must be $b = *r_b$, and $(Bid(r_b), h) \in Dom(F_b)$ holds by the premise of the rule [BUG-DANG-PTR-DEREF]. Similarly to the previous case, there exists an expression $a = \mathsf{free}\ r_a$ reduced before b, where $\nu_a = (k_a, \mathsf{free}, r_a, bid_a) \in \sigma_b$, and $bid_a = Bid(r_b)$, and there exists an expression $c = \mathsf{malloc}\ n_c$ reduced before a, where $\nu_c = (k_c, \mathsf{malloc}, n_c, bid_c) \in \sigma_b$, and $bid_c = Bid(r_b)$.
It is easy to see that the detector [DANGLING-POINTER-DEREF] specified in Figure 2 fires if this error state is encountered (expression b corresponds to the detection point $deref\ r_b$, and the $Allocated(r_b)$ and $Freed(r_b)$ auxiliary predicate is true because of the existence of event ν_c and ν_a in σ_b respectively, hence $\mathbf{p}$ holds).

case [BUG-DANG-PTR-DEREF2] **:**
The redex must be $b = r_b := v_b$, and $(Bid(r_b), h) \in Dom(F_b)$ by the premise of the rule [BUG-DANG-PTR-DEREF2]. The proof is similar to case [BUG-DANG-PTR-DEREF], hence the detector [DANGLING-POINTER-DEREF] fires.

case [BUG-NULL-PTR-DEREF] **:**
The redex must be $b = *r_b$, and $r = NULL$ by the premise of the rule [BUG-NULL-PTR-DEREF]. It is easy to see that $\mathbf{p}$ holds and the detector [NULL-POINTER-DEREF] specified in Figure 2 fires if this error state is encountered (expression b corresponds to the detection point $deref\ r_b$).

case [BUG-NULL-PTR-DEREF2] **:**
The redex must be $b = r_b := v_b$, and $r = NULL$ by the premise of the rule [BUG-NULL-PTR-DEREF2]. The proof is similar to case [BUG-NULL-PTR-DEREF] hence the detector [NULL-POINTER-DEREF] fires (expression b corresponds to the detection point $deref\ r_b$).

case [BUG-OVERFLOW] **:**
The redex must be $b = *r_b$, and $Bid(r_b) \in Dom(H_b) \wedge (r_b < Begin(r_b) \vee r_b \geq End(r_b))$ by the premise of the rule [BUG-OVERFLOW]. Similarly, we can prove that there exists an expression $a = \mathsf{malloc}\ n_a$ reduced before b, where $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_b$, and $bid_a = Bid(r_b)$. Suppose there is an expression $c = \mathsf{free}\ r_c$ reduced after a and before b, where $\nu_c = (k_c, \mathsf{free}, r_c, bid_c) \in \sigma_b$, and $bid_c = bid_a$. Then $(Bid(r_b), h) = (bid_c, h) \in Dom(F'_c)$ by rule [FREE]. By Lemma A.5, $(Bid(r_b), h) = (bid_c, h) \in Dom(F_b)$. Because state Σ_b is well-formed by Lemma A.3 and A.4, then $F_b \cap H_b = \emptyset$ by Definition A.1. Thus $Bid(r_b) = bid_c \notin Dom(H_b)$: a contradiction.

Thus, there is no such expression c and no corresponding event ν_c in σ_b.

It is easy to see that the detector [HEAP-BUFFER-OVERFLOW] specified in Figure 2 fires if this error state is encountered (expression b corresponds to the detection point $deref\ r_b$, and the $Allocated(r_b)$ holds because of the existence of event ν_a, and $\neg Freed(r_b)$ is true due to the non-existence of ν_c in σ_b. Meanwhile, $(r_b < Begin(r_b) \vee r_b \geq End(r_b))$ is trivially satisfied.

case [BUG-OVERFLOW2] **:**

The redex must be $b = r_b := v_b$, and $Bid(r_b) \in Dom(H_b) \wedge (r_b < Begin(r_b) \vee r_b \geq End(r_b))$ by the premise of the rule [BUG-OVERFLOW2]. The proof is similar to case [BUG-OVERFLOW] hence the detector [HEAP-BUFFER-OVERFLOW] fires (expression b corresponds to the detection point $deref\ r_b$).

case [BUG-UNINITIALIZED] **:**

The redex must be $b = *r_b$, and $Value(r_b) = junk$ by the premise of the rule [BUG-UNINITIALIZED]. By the definition of $Value(r)$, we get $Bid(r_b) \in Dom(H) \vee Bid(r_b) \in Dom(\overline{S})$. There are two subcases:

case $Bid(r_b) \in Dom(H)$ **:**

By Lemma A.2, $H_0 = \emptyset$, there must be some expression a reduced before b and its corresponding reduction rule expands H (the only rule is [MALLOC]). That is, there is $a = \mathsf{malloc}\ n_a$, and $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma'_a$, such that $bid_a = Bid(r_b)$, and $H'_a[bid_a \mapsto (\boxed{\text{junk}}, n_a, k_a)]$. That is, $Value(r_b) = junk$ in state Σ'_a.

Suppose there exist intervening reductions which change the $Value(r_b)$ after a and before b. Because the only rule which can change the value of a memory block is rule [ASSIGN], all such expressions should have the form: $r := v$ and follow rule [ASSIGN]. Without loss of generality, let us consider one of such possible expression, e.g., $c = r_c := v_c$ and $\nu_c = (k_c, \mathsf{write}, r_c, v_c, f_c) \in \sigma'_c$, such that $r_c = r_b$. By rule [ASSIGN], $v_c \neq junk$, so $Value(r_b) \neq junk$ in state Σ_b: a contradiction.

Thus, there is no such expression c and no corresponding event ν_c in σ_b.

It is easy to see that the detector [UNINITIALIZED-READ] specified in Figure 2 fires if the uninitialized read error state is encountered (expression b corresponds to the detection point $deref_r\ r_b$, and $FindLast(\_, \mathsf{write}, r_b, \_,)$ is false because of the non-existence of event ν_c in σ_b; hence because $\neg FindLast(\_, \mathsf{write}, r_b, \_,)$ holds, $\mathbf{p}$ holds).

case $Bid(r_b) \in Dom(\overline{S})$ **:**

Similarly, there must be an expression $a = \mathsf{let}\ x = \mathsf{salloc}\ n_a\ \mathsf{in}\ e$, such that $bid_a = Bid(r_b)$, and $\overline{S}'_a[bid_a \mapsto (\boxed{\text{junk}}, n_a, k_a)]$; the proof is similar to the prior subcase.

□

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. OOPSLA '05, pages 345–364.

[2] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. *PLDI'06*, pages 158–168.

[3] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. OOPSLA '07, pages 405–422.

[4] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA'07*, pages 569–588.

[5] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN '06*, pages 269–280.

[6] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *ICSE '06*, pages 162–171, 2006.

[7] D. Drusinsky. The temporal rover and the atg rover. In *SPIN 2000*.

[8] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL '12*, pages 533–544.

[9] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. OOPSLA '05, pages 385–402.

[10] R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *ASE '10*, pages 143–146.

[11] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *USENIX Winter Tech. Conf.*, pages 125–136, 1992.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. *OOSPLA'04*, pages 92–106.

[13] `http://flex.sourceforge.net/`. Flex homepage.

[14] `http://www.gnu.org/software/bison/`. Bison homepage.

[15] `http://www.gnu.org/software/gdb/`. Gdb homepage.

[16] `http://www.time-rover.com`. Time Rover homepage.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, pages 220–242.

[18] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. *ICSE'08*, pages 301–310.

[19] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. PLDI '10, pages 36–49.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, pages 190–200.

[21] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05*, pages 48–61.

[22] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, pages 90–121, 1980.

[23] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. OOPSLA '05, pages 365–383.

[24] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.

[25] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *PLDI'09*, pages 245–258.

[26] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *SIGMETRICS'06*, pages 216–227.

[27] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *ISCA'05*, pages 284–295, 2005.

[28] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *PLDI'07*, pages 89–100.

[29] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI '09*, pages 397–407.

[30] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *RTSS'08*, pages 481–491.

[31] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the gc compute efficiently?: a language for heap assertions at gc time. OOPSLA '10, pages 256–269.

[32] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.

[33] J. Silva. A survey on algorithmic debugging strategies. *Adv. Eng. Softw.*, pages 976–991, 2011.

[34] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Trans. Archit. Code Optim.*, pages 5:1–5:33, 2009.

[35] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. In *PLDI '11*, pages 270–282.

[36] C. Zhang, D. Yan, J. Zhao, Y. Chen, and S. Yang. Bpgen: an automated breakpoint generator for debugging. In *ICSE '10*, pages 271–274.

[37] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *ICSE '03*, pages 319–329, May 2003.