

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

Permchecker: A Toolchain for Debugging Memory Managers with Typestate

KARL CRONBURG and SAMUEL Z. GUYER*, Tufts University, USA

Dynamic memory managers are a crucial component of almost every modern software system. In addition to implementing efficient allocation and reclamation, memory managers provide the essential abstraction of memory as distinct objects, which underpins the properties of memory safety and type safety. Bugs in memory managers, while not common, are extremely hard to diagnose and fix. One reason is that their implementations often involve tricky pointer calculations, raw memory manipulation, and complex memory state invariants. While these properties are often documented, they are not specified in any precise, machine-checkable form. A second reason is that memory manager bugs can break the client application in bizarre ways that do not immediately implicate the memory manager at all. A third reason is that existing tools for debugging memory errors, such as Memcheck, cannot help because they rely on correct allocation and deallocation information to work.

In this paper we present Permchecker, a tool designed specifically to detect and diagnose bugs in memory managers. The key idea in Permchecker is to make the expected structure of the heap explicit by associating *typestates* with each piece of memory. Typestate captures elements of both type (e.g., page, block, or cell) and state (e.g., allocated, free, or forwarded). Memory manager developers annotate their implementation with information about the expected typestates of memory and how heap operations change those typestates. At runtime, our system tracks the typestates and ensures that each memory access is consistent with the expected typestates. This technique detects errors quickly, before they corrupt the application or the memory manager itself, and it often provides accurate information about the reason for the error.

The implementation of Permchecker uses a combination of compile-time annotation and instrumentation, and dynamic binary instrumentation (DBI). Because the overhead of DBI is fairly high, Permchecker is suitable for a testing and debugging setting and not for deployment. It works on a wide variety of existing systems, including explicit malloc/free memory managers and garbage collectors, such as those found in JikesRVM and OpenJDK. Since bugs in these systems are not numerous, we developed a testing methodology in which we automatically inject bugs into the code using bug patterns derived from real bugs. This technique allows us to test Permchecker on hundreds or thousands of buggy variants of the code. We find that Permchecker effectively detects and localizes errors in the vast majority of cases; without it, these bugs result in strange, incorrect behaviors usually long after the actual error occurs.

Additional Key Words and Phrases: typestate, debugging, language implementation, memory management, memory layout, compiler extension

1 INTRODUCTION

In December of 2018, a bug was reported to Oracle’s OpenJDK [Oracle 2006] development team. In this bug, the Java Virtual Machine (JVM) would deadlock on an object monitor, even though no threads held exclusive access to the monitor. This failure only occurred when the jemalloc [Evans

Authors’ address: Karl Cronburg, karl@cs.tufts.edu; Samuel Z. Guyer, sguyer@cs.tufts.edu, Tufts University, 161 College Ave, Medford, Massachusetts, 02155-5593, USA, USA.

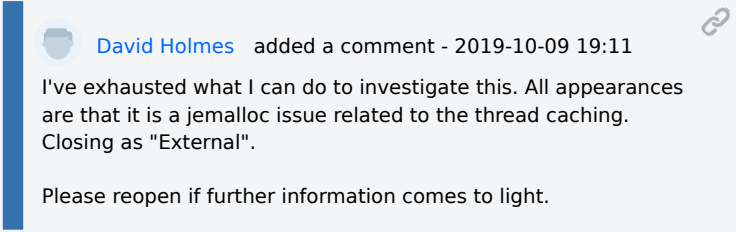
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

XXXX-XXXX/2021/9-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

2006] was used in place of the default malloc/free implementation, leading the developers to suspect a memory safety error. However, the failure was difficult to reproduce and caused knock-on effects leading to incomplete and misleading stack traces. After ten months of on-again off-again investigation, the team had all but given up, with one comment added to the bug report as follows:



David Holmes added a comment - 2019-10-09 19:11

I've exhausted what I can do to investigate this. All appearances are that it is a jemalloc issue related to the thread caching. Closing as "External".

Please reopen if further information comes to light.

In the Fall of 2019, this developer finally figured it out using carefully hand-coded instrumentation of object monitors. An address mix-up was causing the monitor in question to be owned by two different threads. After almost a year from the initial report, the bug was found and fixed: an address comparison that should have been a greater-than was using a greater-than-or-equal-to operation instead.

This kind of bug is among the most difficult and time-consuming to diagnose because it involves the very subsystem that we ordinarily rely on to *enforce* memory safety: the memory manager, and an object model describing how objects and their metadata are laid out in memory. For this reason, many of the existing tools and techniques for tackling memory errors are ineffective or cannot be easily applied to code in the memory manager. At the same time, the programming of such code is very challenging and error prone, involving sensitive pointer manipulations, implicit address invariants, and complex memory layouts accessed by disparate subsystems. We normally expect runtime system code to support a diverse array of hardware and software environments, where concurrency is the norm and performance is at a premium. Although the number of memory manager implementations is relatively small, every single program that dynamically allocates memory relies on one to run correctly.

In this paper we present Permchecker, a toolchain designed to aid in the debugging of memory managers, including explicit allocators and garbage collectors. Permchecker's design is based on the observation that all memory managers perform the same basic task: they take a large array of bytes and partition it into chunks, giving structure to memory and managing the lifecycle of objects for the client application. The specific partitioning strategy of a memory manager gives each chunk of memory meaning, much the way types give meaning to data in a programming language. An error in a memory layout looks much like a type safety error: a chunk of memory designated for one purpose is accessed improperly or used for another purpose.

Memory chunk types, however, have important differences from traditional types. For one, chunk types often cannot be explicitly defined in the type system of the implementation language. Instead, the role of a chunk is implied by its relative spatial location in memory. For example, a generational garbage collector might classify a page of memory as part of the nursery because it resides in the nursery's reserved address range. Similarly, the argument to a manual `free()` implementation is an untyped pointer, and this function might need to inspect the pointer value or metadata stored elsewhere in memory to determine, e.g., that the chunk belongs on a free list. Traditional types alone are a poor fit for these for these situations because the correct type information is not readily available from either type declarations or the operations performed.

99 Another difference from traditional types is that a chunk of memory changes type during the
100 normal course of partitioning, allocation, deallocation, and coalescing. For example, when a page
101 is divided up into cells, all subsequent code must access memory in the page as cells, even though
102 they reside within a *page*. This behavior is different from dynamic typing, where a single *variable*
103 can refer to values of different types, but the values themselves do not change type. The most
104 closely related type feature is C/C++ union, which allows a single chunk of memory to be viewed
105 as different types, but provides no mechanism to check and maintain type safety.

106 The key idea in Permchecker is to use *typestates* [Strom and Yemini 1986] to specify and track
107 the structure of memory as it changes, and check that the memory manager accesses and updates
108 this structure according to the expected specification (the “*permissions*”). The system consists of
109 two main parts: (1) an API that developers can use to annotate memory management code with
110 the expected types and state transitions in each operation (e.g., alloc, free, scan, collect, etc), and
111 (2) a runtime system that associates typestates with real memory (a shadow map) and uses binary
112 instrumentation to check that at every memory access the permissions associated with the code
113 match the typestate of the memory. We show that this technique detects memory errors as soon
114 as they happen, rather than when they manifest, which can be much later in a program execution.

115 The term typestate, as we apply it to a dynamic setting, is slightly different from its static coun-
116 terpart. In a static analysis setting, a typestate requires three components. First, types describe the
117 shape of data and the allowable operations over a variable. Second, states and accompanying state
118 transitions narrow the allowable operations over the variable. Finally, a semilattice with a meet
119 function defined over it approximates the true state of the variable at each static program point
120 by defining how to combine state information at control-flow merge points.

121 With Permchecker **the first two components of typestate are defined, but the third is**
122 **not needed**. First, types describe the in-memory layout of a data structure and the allowable
123 operations over a memory location therein. Second, states and accompanying state transitions
124 restrict the memory location to being accessed by specific program instructions. Finally, program
125 annotations define code permissions which exactly track state information meaning no semilattice
126 approximation is necessary.

127 A goal of this work is to build tools that work with existing memory managers across a range
128 of platforms. Our approach is designed around this use case. We assume that developers cannot
129 reimplement their systems in a new language or using a new methodology. Therefore, our API is
130 generated from a separate specification and added to the memory manager code. To avoid high up-
131 front costs, developers can introduce these annotations gradually, starting with a simple, but coarse
132 model of memory, and increasing the precision as necessary with more fine-grained annotations.

133 *Contributions.* In this work, we apply *typestate* to a dynamic setting for the purpose of describing
134 how a memory manager partitions and recycles memory. The goal of Permchecker is to provide a
135 way for the developer of a runtime system to specify and check the lifecycles of memory typestates
136 across all abstraction levels of the system. A relatively simple tracking API allows the developer
137 to associate a typestate with a chunk of memory. A permissions API and annotations can then be
138 used to declare access permissions. This formulation of dynamic typestate is a natural extension of
139 the static typestate program analysis techniques developed in the 1980’s [Strom and Yemini 1986].

140 The contributions presented in this paper are as follows:
141

- 142 • A technique for specifying the hierarchical layout and decomposition of a memory man-
143 ager’s memory.
 - 144 • A tool for generating an API from a layout specification. The API tracks structures of mem-
145 ory via explicit typestates transitions.
- 146
147

- Compile-time support, implemented in the Clang C/C++ compiler, for automatically translating common tpestate transitions from function annotations to API calls.
- A Pin-based dynamic binary instrumentation (DBI) tool that tracks the tpestate of every byte of memory and ensures that memory accesses from the anywhere in the program (both the memory manager and the application) respect the expected structure.
- An experimental methodology that includes both real bugs and injected bugs. Since bugs in memory managers are relatively uncommon, we use a technique in which we extract the essence of a real bug as a bug pattern, and then run hundreds of experiments in which similar defects are introduced in different places in the code.

2 DESIGN OVERVIEW

Permchecker takes a step towards verifiable memory management by providing a way to explicitly annotate the structure and state of a heap, and then check that the implementation of the memory manager properly maintains and respects that structure. Our design is inspired by Valgrind’s Memcheck tool, but generalized to an arbitrarily hierarchical memory manager. At the application level, memory is either allocated or not, and Memcheck checks that application code only accesses allocated memory. Inside a memory manager, there are often numerous intermediate states of memory apart from allocated and free. Memory is obtained from the operating system in large chunks and carved into successively smaller pieces. Each partitioning is typically managed by an intermediate allocator which services requests from higher-level allocators and requests memory from lower-level allocators. Additionally, multiple independent allocators often service requests for specific categories of values, contributing further to the rich landscape of memory subsystems that compose a modern runtime system.

Allocation Policies & Alternative Hierarchies. As an example, a generational garbage collector might contain three independent allocators: a bump pointer for the nursery, a free-list allocator for the mature space, and a reference-counted allocator and collector for large objects. For instance, the free-list allocator has at least two allocation levels. First, a low-level block allocator carves off a large chunk of the virtual address space from the operating system. Second, the free-list’s cell allocator extracts from the large chunk an object-sized cell and doles it out to the application. This process is further complicated by a myriad of allocation and collection policies at both the block and cell level. As a result a fully automated technique, like Memcheck, is infeasible in the absence of definitive memory type and allocator state information.

Tpestate Permissions. It is crucial for runtime system code to only ever access the kind of memory it is supposed to. For example, a free-list allocator should not access memory currently allocated by the nursery’s bump pointer allocator. Given explicit expectations of these two subsystems, we can begin to verify that memory is managed safely by ruling out cross-contamination.

With Permchecker, each location in memory is assigned a type, or more precisely a tpestate since the type can change over the course of execution. After assignment, code which accesses the location must have permission to access the assigned tpestate. An error is thus reported as soon as a memory access occurs over a tpestate of memory the code was not expecting to access. The resulting error message is a detailed report of the impermissible memory access, and is described in the terminology of the particular system. For example, a garbage collector’s nursery allocator might incorrectly attempt to allocate memory recently added to the mature space’s free-list. The error reported by Permchecker would be something like the following: “*Observed type FREELIST_CELL expecting type NURSERY_CELL.*”

197 *Incrementality.* A primary goal of this work is to provide practical debugging tools and techniques
198 that can help diagnose bugs in existing systems. To this end, Permchecker is built as a lightweight
199 C/C++ API and accompanying heavyweight instrumentation tool runnable on virtually any mem-
200 ory manager. We present our use of this tool to debug OpenJDK’s Hotspot VM, Oracle’s open-
201 source version of their industrial-strength JVM. Not discussed experimentally in this paper is our
202 use of Permchecker to debug memory bugs found in other manual and automatic memory man-
203 agers. These include dlmalloc [Lea 1991], Doug Lea’s implementation of the standard malloc/free
204 interface, and the Jikes Research Virtual Machine [IBM 2005], another implementation of a JVM.
205 Without the trial-and-error process of studying this multitude of diverse systems, we would not
206 have landed upon an *incremental* usage of Permchecker where the programmer starts out with a
207 simple model of their heap, adding more and more typestate information over time.
208
209

210 3 BACKGROUND & RELATED WORK

211 Memory safety is not a new concern. Over the years, a wide array of systems, tools, techniques, and
212 languages have been developed to help diagnose and prevent these potentially catastrophic errors.
213 One of the most successful and widely deployed techniques is automatic memory management
214 (garbage collection) which, together with runtime bounds checking, has practically eliminated the
215 most common coding errors that lead to memory corruption. But a nagging problem has remained:
216 how to find and fix bugs in the memory manager itself. In this section, we discuss how this concern
217 is different from ordinary memory safety and why that makes it such a hard problem to deal with.
218

219 *Debugging an Abstraction.* Memory safety ensures that a memory access respects the boundaries
220 and lifetimes of objects and values in memory. At the level of raw memory (virtual addresses),
221 however, there are no boundaries and lifetimes; memory is just one giant array of bytes. These ab-
222 stractions are created and maintained by the memory manager, and unfortunately existing mem-
223 ory safety tools require this information in order to perform their checks. Valgrind [Nethercote
224 and Seward 2007], for example, can perform highly detailed memory safety checks on an applica-
225 tion, but it must be able to intercept (shim) calls to malloc and free in order to know when objects
226 are allocated and freed, and their bounds. When the memory manager itself has an error, however,
227 the memory checker becomes unreliable because it is using faulty information.

228 *Missing Information.* A memory manager bug is similar in spirit to a compiler bug: a rare “bug
229 of last resort” that is often identified only after extensive application-level debugging fails. In a
230 compiler bug, however, it is possible to inspect the assembly code output and verify that it correctly
231 implements the input source code (or not, as the case may be). No such opportunity exists for errors
232 in the heap layout of a memory manager. Most implementations do not include a formal description
233 of what correct states of the heap can look like. Therefore, we cannot merely inspect the resulting
234 contents of memory to determine if the memory manager has functioned correctly. For this reason,
235 a preprocessing tool in the Permchecker toolchain is inspired by LoCal [Vollmer et al. 2019] and
236 Floorplan [Cronburg and Guyer 2019], both of which are memory layout description languages
237 intended to provide special purpose heap access abstractions.

238 *GC Debugging.* Recent research targeting, among others, the multicore OCaml garbage collector
239 (GC) tackles the issue of unreliability in debugging tools for concurrent GC bugs. While memory
240 corruption is one source of unreliability in producing accurate error reports, another source of un-
241 reliability is the non-deterministic nature of many GCs. The RR [O’Callahan et al. 2017] debugger,
242 an extension to the GNU Debugger, alleviates nondeterminism while debugging. It does this by
243 recording and replaying program executions with relatively low overhead. The developers of RR
244
245

246 report that a principle motivation in developing it the way they did was “to improve the [expe-
247 rience] of [using] user-space record-and-replay while being less invasive than [e.g.] a full kernel
248 implementation”. These principles are shared by our approach to debugging garbage collectors. We
249 studied the practicality of adding small amounts of GC instrumentation for the purpose of reliably
250 reporting the violation of unobservable program invariants.

251 *Static Solutions.* Tpestate finds numerous applications in both static analysis [Strom and Yem-
252 ini 1986] and static programming language features. For instance tpestate in Rust [Weiss et al.
253 2019] derives from the language’s support for linear and affine types at compile-time, by enforcing
254 memory movement semantics statically in the context of a single variable. In contrast tpestates
255 supported by Permchecker derive from the toolchain’s tracking of shadow memory at run time,
256 and enforcing memory access permissions dynamically in the context of an individual memory
257 location. The key difference here is that a single variable cannot be aliased by constructing it from
258 dynamic calculations, whereas a memory location *must* be aliasable via pointer operations in the
259 implementation of a memory manager.
260

261 *Smart Pointers.* Much like linear and affine types for enforcing usage constraints at compile-time on
262 variables, a smart pointer can check or enforce certain usage properties at run time *on a variable* as
263 well. One canonical example is a reference-counted smart pointer for distinguishing live allocated
264 objects from garbage. Such a reference counting scheme tracks the number of known pointers
265 stored in memory in places where they can, in the future, be loaded into variables (registers, at
266 run time) and referenced as such with a load or store instruction. This ability to access memory
267 indicates the memory remains allocated. A memory manager however not only manages allocated
268 memory: it also manages and requires access to any and all freed memory. Thus, without reach-
269 ability as a proxy for whether or not a memory location should be accessible, reference-counted
270 smart pointers alone cannot sufficiently check memory management code for safety.

271 *Compiler Extensions.* A compiler extension known as AddressSanitizer [Serebryany et al. 2012],
272 like Valgrind’s Memcheck, is able to detect out-of-bounds accesses to memory locations with
273 shadow memory. Unlike Memcheck, it relies on unaddressable “poisoned” padding to be added
274 to heap, stack, and global chunks of memory. As a result this mechanism is able to detect a subset
275 of inter-chunk corruptions, but not any intra-chunk corruptions. Detecting intra-chunk corrup-
276 tion requires a richer allocation distinction than just allocated/free, and fewer assumptions about
277 where memory comes from. The key problem we face, in the domain of memory management, is
278 that multiple memory allocation schemes share the same single virtual address space.
279

280 *Bug Injection Methodologies.* The process of generating a corpora of bugs is often motivated by a
281 specific kind of mistake a programmer can make: off-by-one errors, operator selection defects [Rice
282 et al. 2017], bounds-checking mistakes [Dolan-Gavitt et al. 2016], and more. These mistakes can
283 be constructed by source code mutation [Roy et al. 2018]. What this bug creation technique does
284 not take into account is the modeling of undesirable algorithmic operations of the system having
285 bugs injected into it.

286 In a memory manager, we know of a relatively fixed and small set of algorithmic operations
287 that are bad: use-after-free, overlapping allocations, among others. In contrast, the categories of
288 linguistic (syntactic and semantic) mistakes a programmer can make are limited by which symbols
289 the programmer types and the compiler accepts. The key difference is that injecting invalid algo-
290 rithmic operations into a system realistically stresses a broad category of resulting downstream
291 behaviors of the system, while injecting linguistic bugs does not. Injecting a linguistic bug requires
292 both a far larger enumeration of sources of linguistic mistakes and clever tactics for eliminating
293 repetitive bugs, such as ones which always crash the program.
294

```

295 1 <id> := [A-Z][_a-zA-Z0-9]*
296 2 <ts> := <id> // A tpestate is a unique identifier
297 3 <prim> := bytes | words
298 4 <stmt> := <ts> -> <ts>
299 5 | <ts> -> seq { <exp> , ... } // One or more ', '-sep exps
300 6 | <ts> -> union { <exp> | ... } // One or more '| '-sep exps
301 7 | # <exp> // Prefix notation for "zero-or-more" reps
302 8 <exp> := <ts> | <stmt> | [0-9]+ <prim>
303 9 <spec> := <stmt>+
304

```

Fig. 1. Syntax for describing the layout of memory in terms of tpestates. A layout <spec> is one or more layout statements describing a sequence of memory (seq), alternative views of the same memory (union), some amount of primitive memory (<prim>), or repetitions of a structure with (#).

4 TOOLCHAIN OVERVIEW BY EXAMPLE

In this section we present in modest detail the mechanisms by which a memory management programmer and the Permchecker toolchain orchestrate the checking of tpestates in a memory manager. In doing so we illuminate the role of the compiler, VM annotations, and dynamic instrumentation in tracking and checking the tpestates. The idea here is that each toolchain component is small or simple in nature and designed to serve a single purpose effectively: manage and check uniquely identifiable tpestates over a process' address space. To start, the following are some important high-level definitions clarifying the scope of tpestate in this work:

Tpestate. This is a uniquely identifiable state of a memory location, coupled with the valid value types that can reside in that location. A mutable tpestate on a heap memory location for the duration of a program's execution is similar to a dynamic type on a local variable for the duration of a function's execution. The key difference is that a type typically restricts a function call-site to referring to a specific set of variables, whereas a tpestate restricts an assembly instruction to accessing a specific set of memory locations.

Set of tpestates. This is a collection of tpestates, each of which can be accessed by the same piece of polymorphic code. Such a collection associated with a specific piece of code permits access to memory locations containing the same types of values, without unnecessarily restricting the code to a highly particular tpestate.

4.1 Allocation Hierarchies

In Figure 1 we define a minimal syntax for defining tpestates of memory. The idea is that tpestates are interrelated via the hierarchy by which they are allocated. For example in Hotspot, a page resource allocates for regions, the region manager allocates for objects, and object management code allocates object headers and various primitive application field types. To broadly describe this hierarchy with the syntax, we might write the following:

```

337 1 Pages -> # union { Region | ... }
338 2 Region -> seq { # Object, # words }
339 3 Object -> seq { Header, # Field } (L1)
340 4 Header -> union { 2 words | Array -> 3 words }
341 5 Field -> # words
342
343

```

Code L1 defines 6 tpestates: Pages, Region, Object, Header, Array, and Field. The idea is that each and every memory location in use by the memory manager is in one of these tpestates at some point during runtime (or implicitly in an UNMAPPED tpestate). For example, an allocated heap region with a single 10-word object allocated into it involves three tpestates. The first 2 words of the region are in the Header tpestate, the next 8 words are Field, and the remaining memory is all Region. In this example, the first # repetition in the Region statement effectively takes on a value of 1 (a single object) and the second # repetition consumes the remaining words in a (fixed-size) region. Also note that the ellipsis on the first line is a placeholder for other allocation schemes that we do not discuss here.

The repetition (#) operator means *some number of* chunks of memory. Intuitively, this operator flood fills the surrounding space by repeating in memory zero or more times the layout expression it is applied to. Formally, the semantics of repetition are equivalent to those of the Kleene star regular expression operator. For example, the (#) operator can describe gaps in the address space of statically unknown size inbetween known structures. This application enables the description of alignment padding at the beginning of an object, dead objects fragmented among live allocated objects, and more.

4.2 Layout Description

In addition to defining a hierarchy, we need to be able to declare what the intended contents (values) of a memory layout are. In a memory manager, navigating a memory layout to obtain values in memory is performed by pointer manipulations and offset calculations. These calculations ultimately lead us to the underlying contents of memory. To declare the underlying contents of objects from Code L1, we might for example write the following:

```

368 1 Object -> union {
369 2     Free  -> # words
370 3     | Alloc -> seq { Header, # words } }
371 4 Header -> union {                                     (L2)
372 5     Arr  -> seq { MarkWord, KlassPtr, Len, Gap } // 3 words
373 6     | Cls -> seq { MarkWord, KlassPtr } }           // 2 words
374

```

In this Code L2, we define the tpestates of a Hotspot Object and its Header. An object is either in the Free tpestate, or it is in the Alloc tpestate and therefore contains a header and some number of payload words of memory which we leave unrefined. The Header of such an object is either 3 or 2 words in size. This depends on whether or not the object is an array with an array length (Len) field. The Arr is this array type, whereas the Cls is an ordinary Java class object. A KlassPtr is a reference to C++ metadata about a Java class. The Gap field is unused alignment padding. Lastly, a MarkWord is a bit-field containing runtime metadata such as the object’s hash, locking bits, number of object promotions or evacuations, and a mark bit. In some cases these fields overlap and change depending on the state of the object, especially in relation to per-thread object locking. Apart from the mark bit and promotion count (“age”), no space is reserved in the mark word for the GC. Assuming a 64-bit architecture, the header fields take on the following sizes:

```

387 1 MarkWord -> 1 words
388 2 KlassPtr -> 1 words                                     (L3)
389 3 Len      -> 4 bytes
390 4 Gap      -> 4 bytes
391
392

```


Code L2 and L3 serve two purposes: (1) to define hierarchical relationships among named typestates, and (2) to associate sizes with them. With (1), the idea is that this code models where a piece of memory comes from in the context of a hierarchy of typestate mutators. One such typestate mutator is the code in Hotspot which stores a garbage object onto a free-list. This mutator code implicitly changes the typestate of the underlying memory from allocated to free. However, this is in some sense a lazy operation. Perhaps we could deem an object to be free as soon as a Java-level pointer update causes the object to no longer be accessible by the Java heap. While more precise, this definition is untrackable without invasive changes to the runtime system. The point is it is not always clear, based on a cursory analysis of the code, where a typestate mutation should occur. This state of affairs reflects our dynamic formulation of typestate for tracking memory *as implemented*, with minimal disruption to that implementation.

4.3 Typestate Identifiers: Code Generation

In this section we discuss how we annotated the Hotspot VM with typestate layout annotations generated from a version of Code L1. The idea is that we can generate much of the repetitive boilerplate necessary to track typestates at run time. This boilerplate includes a consistent centralized naming scheme for managing hierarchies of allocable typestates.

First, a preprocessing tool processes a layout description, generating a series of named typestates, among other code snippets to reduce boilerplate during annotation of a memory manager. From Code L2, we get out a series of typestates defined as follows:

```

1   #define PCK_Object ((TypeState)1)
2   #define PCK_Object_Free ((TypeState)2)
3   #define PCK_Object_Alloc ((TypeState)3)
4   #define PCK_Object_Alloc_Header ((TypeState)4)           (C1)
5   #define PCK_Object_Alloc_Header_Arr_MarkWord ((TypeState)5)
6   #define PCK_Object_Alloc_Header_Arr_KlassPtr ((TypeState)6)
...
30  #define PCK_MarkWord ((TypeState)30)
31  #define PCK_KlassPtr ((TypeState)31)
...

```

In this series of typestates in Code C1, `TypeState` is a class for unique identifiers. Each identifier is referenceable with a unique numeric value, as indicated by the implicit cast constructors referenced above. The number zero (0) is reserved for `PCK_UNMAPPED` memory. Any top-level defined `<stmt>` gets associated with it a nested hierarchy of addressable typestates. For example, a memory location can be part of a free (`PCK_Object_Free`) or allocated (`PCK_Object_Alloc`) object. Notably this means that no typestates of the form `PCK_Alloc_*` are generated, but indeed `PCK_Object_*` are generated. The key is that one can choose which typestate contexts are diagnostically helpful to track based on how memory is managed.

One important distinction is between the generated typestates `PCK_Object_Alloc_Header_Arr_MarkWord` and `PCK_MarkWord`. The latter, `PCK_MarkWord`, represents memory for a top-level C++ variable not located in the Java heap. The former is a fully-qualified mark word of an allocated array object header *located on the Java heap*. This typestate represents memory which was allocated by the proper Java object allocation code. The distinction here is necessarily self-enforced, based on annotation usage, but it is a diagnostically helpful distinction. The idea is that by default internal memory allocation context is all-but lost once an initialization is complete. With the two

```

442 1 namespace pck {
443 2   class Map;
444 3   class TypeState; // Includes an identifying member field of u64_t
445 4   Map create();
446 5   void destroy(Map m);
447 6   TypeState Map::get(Address a);
448 7   void      Map::put(Address a, TypeState ts); }
449

```

Fig. 2. The VM-level interface to initialize and track a memory to tpestate mapping. All functions here are in the pck (Permchecker) namespace.

tpestates however, we can now retain the context for validation or diagnosis purposes if and when a corruption occurs.

However, the tpestate distinction comes at a price. In order to specify when the distinction does *not* matter, we must annotate the code with all of the allowable tpestates. Instead of doing this manually, the code generator does this for us. The generator creates names for collections of tpestates based on all the qualified names by which any given “leaf” tpestate can be allocated. For example, for the mark word tpestates from Code L2 we get the following collection:

```

461 1 #define PCK_ALL_MarkWord PCK_MarkWord \
462 2   PCK_Object_Alloc_Header_Cls_MarkWord \
463 3   PCK_Object_Alloc_Header_Arr_MarkWord \
464 4   PCK_Header_Arr_MarkWord PCK_Header_Cls_MarkWord
465

```

(C2)

In this macro we have (1) a C++ variable mark word, (2) an allocated class’s or (3) array’s mark word on the Java heap, and (4) a C++ array variable’s or (5) Java class variable’s mark word. This collection is a fairly common idiom in memory managers. That is, a mark word accessor function typically does not care which variant it accesses. By annotating code with this PCK_ALL_MarkWord tpestate collection, we get the benefit of a coarse polymorphic tpestate when the precise distinction does not matter. At the same time, we still get the benefit of a detailed taint analysis when Permchecker reports an error in terms of the precise tpestate observed at run time.

4.4 Tpestate Tracking

The VM’s interface to the tpestate tracker is shown in Figure 2. Apart from its connection to dynamic instrumentation, this interface operates how one would expect it to. An individual shadow map can be created and destroyed, with each map maintaining an injective key-value mapping from addresses to tpestates. The special UNMAPPED tpestate is the default value for unaddressable memory, and UNMAPPED can be explicitly referenced as a tpestate. The interface further uses array-access overloading to provide the following map update and query syntax:

```

482 1 Address a; TypeState ts;
483 2 map[a] = ts;
484 3 pck::assert(map[a] == ts)
485

```

(C3)

In this Code C3, we have defined some address *a* and some tpestate *ts*. On the second line, an assignment operation calls `Map::put` to assign the r-value *ts* to the map location referenced by address *a*. On the third line, an equality comparison operation calls `Map::get` to query the tpestate at address *a* and check that it matches *ts*. Note that the equality and map-access operations return

491 proxy object types tracking what address was accessed, and the arguments to any comparisons.
 492 This proxy information allows `pck::assert` to report a helpful error message in terms of the ad-
 493 dress, observed tpestate, and expected tpestate of the comparison, rather than simply reporting
 494 that the assertion failed.

495 As we will discuss in the next section, the dynamic instrumentation effectively performs an
 496 assertion, like above, over the tpestate map for each and every memory access executed by the
 497 processor. To simplify our discussion, we focus on the (sufficient) use case of tracking a single
 498 tpestate map. In this case, the `create` and `destroy` functions get called near the beginning and
 499 end of program execution, respectively. In the case of Hotspot, some complication arises from the
 500 use of `fork/exec` system calls, requiring explicit map initialization after the main Java thread is
 501 created. A more careful integration with thread and process creation system calls could provide
 502 more automation of this process in the future. For now, this is unnecessary work to automate a
 503 very small amount of code that a memory management developer knows exactly where to put.

504
 505 *4.4.1 In Practice.* We initially inserted 52 assertions related to object mark-words and class point-
 506 ers in Hotspot. Of these, 11 assertions were subsequently replaced with annotations on simple
 507 accessor functions, to be discussed in Section 4.5.1. A typical pattern in a memory manager is for a
 508 simple one-line accessor function to access a specific set of tpestates, and so a function-level anno-
 509 tation is ideal. In contrast, manual assertions are ideal when instrumenting (1) compiled Java code,
 510 (2) other VM components that cannot easily rely on the C++ compiler, and (3) when substantial
 511 code refactoring would be necessary to systematically instrument certain memory accesses.

512 *4.4.2 Code Generation.* For some tpestates, Permchecker’s preprocessor generates helper macros.
 513 One such macro manages all the tpestate updates necessary to allocate a series of numerous
 514 nested components. For example, for tracking the nested tpestates associated with an array
 515 header the preprocessor generates the following macro based on Code L2:
 516

```
517 1 #define transition_Object_Alloc_Header_Arr (a) { \
518 2   map[words(a, 1)] = PCK_Object_Alloc_Header_Arr_MarkWord; \           (C4)
519 3   map[words(a + words(1), 1)] = PCK_Object_Alloc_Header_Arr_KlassPtr; \
520 4   map[bytes(a + words(2), 4)] = PCK_Object_Alloc_Header_Arr_Len; \
521 5   map[bytes(a + words(2) + bytes(4), 4)] \
522 6     = PCK_Object_Alloc_Header_Arr_Gap; }
```

523
 524 In this Code C4 we see a series of array-update operations over the shadow map, as explained
 525 earlier. The `words` and `bytes` functions in this code return a C++ proxy class indicating the location
 526 and size of a piece of memory to update in the tpestate map. For the two-parameter versions, the
 527 first parameter is a starting address and the second parameter is a number of bytes to assign the
 528 tpestate to. For the one-parameter version, the parameter is just a memory size for use in offset
 529 calculations. For example, the offset calculation in the second map update above computes the
 530 address of a offset by one word of memory. The key is that this code fragment conceptually (if not
 531 literally, for performance) boils down to a series of appropriate calls to `Map::put`.

532 4.5 Permission Tracking

533
 534 Next, we need to be able to assign permissions to a piece of code in terms of the tpestates the code
 535 is allowed to access. The primary mechanism the toolchain supports for this purpose is permission
 536 attributes on functions and classes in C++. The exact low-level mechanism to track this information
 537 is interception of specific function calls by a dynamic instrumentor. We conclude this section by
 538 discussing what bugs this dynamic instrumentor can and cannot detect as tpestate violations.
 539

```

540 1 namespace pck {
541 2 enum Perms { RW, R, W, NONE };
542 3 Map::protect(TypeState ts, Perms ps);
543 4 Map::unprotect(); }
544

```

Fig. 3. The VM-level interface to set memory access protections by tpestate, instead of a concrete address like with the POSIX `mprotect` system call. The `Map::protect` method here applies only to the calling thread, and temporarily overrides any previous protections for the given tpestate. These previous protections are restored by a subsequent matching call to `unprotect`.

		Total	Mark & Klass
550 1	<code>__attribute__((pck_R (TypeState...)))</code>		
551 2	<code>__attribute__((pck_W (TypeState...)))</code>	Lines added	1830
552 3	<code>__attribute__((pck_RW (TypeState...)))</code>	Transitions	144
553 4	<code>__attribute__((pck_NONE (TypeState...)))</code>	Fncn Annots	64
554		Typestates	60
555			16

Fig. 4. **Left:** Read, write, and read-write annotation forms which can be attached to either a function or a class to indicate that entity has permission to access the listed tpestates. The `pck_NONE` option allows one to explicitly *remove* access, notably on a member function where the class generally has permission but the member function should not. **Right:** Lines of code added/modified in the VM to annotate/instrument the most crucial parts of Hotspot (enabling Java object allocation). The “Mark & Klass” column counts lines of code relating to just `MarkWord` and `KlassPtr` variants. The bottom-most row is a count of unique tpestates referenced in the instrumentation.

The VM’s interface to the permission tracker is shown in Figure 3. This interface models memory access permissions by associating with each thread of execution a set of per-tpestate permissions. For example, each thread which calls an accessor function with read or write access to the mark word of an object header can temporarily be given permission to access any such mark word. This does not guarantee the correct mark word is accessed, just that the underlying memory was allocated as such. The key is that an individual function, class, or algorithmic operation typically either has permission to access the mark word, or it does not.

The `protect` and `unprotect` functions implement this permission model. These functions operate similarly to the `mprotect` POSIX system call, but with a few differences. A call to `protect` states that the currently executing thread now has permission to access any memory address in the given tpestate. A subsequent matching call to `unprotect` reverts the permissions to what they were before. These functions are thread-safe, because each thread has its own permissions. These functions are *not* safe with respect to non-reentrant control flow, such as runtime system exception handling. Future work could automatically support arbitrary control flow by associating permissions with instruction pointers instead of function entry and exit points.

In designing this part of `Permchecker`, a simpler model was considered where permissions could only be updated with `protect` and the previous permissions are forgotten. This simpler stateless model, however, fails to account for a nested call frame for which the parent and child frames execute functions both with permission to access the same tpestate. Calling `unprotect` during the child’s epilogue would therefore incorrectly wipe out the permissions the parent still needs to possess. Therefore the stateful implementation of `protect` in the dynamic instrumentor tracks a stack of bit-vectors corresponding to nested function annotations.

4.5.1 Function & Class Annotations. In Figure 4, on the left, we have a series of tpestate permission annotations for functions and classes. On a function, an annotation compiles to calls to

589 protect and unprotect in the prologue and epilogues of the function. On a class, the annotation
 590 gets distributed across functions in that class to the same effect. Based on this model, an annotated
 591 function calling some other function conservatively grants the callee the same permissions it has.

592 Also in Figure 4, on the right, is a breakdown of modifications made to the Hotspot VM to
 593 achieve permission checking of object header words. The modifications reported are the ones at-
 594 tributable to the memory management developer, with the **Total** column including modifications
 595 necessary to support tracking of the Java heap across the entire allocation hierarchy. These num-
 596 bers represent a reasonable upper bound on the annotation burden of dealing with a small number
 597 of the most complex tpestates in a memory manager.

598 The lines of code added to Hotspot include modifications to the JIT compiler, assembler, argu-
 599 ment parsing, imports, error reporting, foreign function interface, and write barrier. Apart from
 600 these tasks, 144 and 64 lines were respectively added to track tpestates and to apply permissions
 601 to code. Of those tpestate management lines of code, 49 deal with just a variety of 16 tpestates
 602 pertaining to the mark or klass word of an object. The remaining 44 of 60 tpestates provide di-
 603 agnostically helpful allocation context for when the mark or klass word is accessed erroneously.
 604 Such tpestates include allocated object fields, free regions, reserved pages of memory, and others.

605 The amount of annotation effort required to validate layout safety for a tpestate correlates
 606 with the number of places in the code memory changes to/from the tpestate, plus the number
 607 of instructions allowed to access it. As a result simpler kinds of memory, like the page allocator
 608 in Hotspot, require relatively few annotations. The desired level of specificity then governs how
 609 many different kinds of memory are annotated, leading to a higher annotation effort.

610 Permchecker allows for progressively annotating each tpestate in isolation while still being
 611 effective for diagnosing violations of the annotated tpestate(s). Thus the value of Permchecker
 612 is directly proportional to the number of completely annotated tpestates. Annotating the appro-
 613 priate places in the code for a single tpestate is the smallest unit of work for a developer to
 614 accomplish when getting started with Permchecker. In our experience such a task can take any-
 615 where from a few minutes, to a few hours, to at most a few days of a single developer’s time. In the
 616 next few subsections we go on to discuss how Permchecker reports helpful tpestate mismatches
 617 in the presence of either a specification or an implementation error.

619 *4.5.2 True Negatives.* Consider a VM function intended to access the mark word of an object,
 620 and we wish to give the corresponding assembly code permission to access mark words therein.
 621 Therefore we want the function to compile to an instruction sequence that looks like the following:

```
622
623 1 // Begin perms: stack (RW)
624 2 mov $0x21,%esi
625 3 mov $0x2,%edx
626 4 // Begin perms: stack (RW) & mark word (W)
627 5 call protect // Mangled name: _ZN3pck3Map7protectEyNS_5PermsE (A1)
628 6 mov -0x28(%rbp),%rax
629 7 movl $0x1,(%rax) // Write to a mark word
630 8 call unprotect // Mangled name: _ZN3pck3Map9unprotectEv
631 9 // Ending perms: stack (RW)
```

632
 633 Code A1 executes as follows. We store immediate values for a mark word’s tpestate (0x21) into
 634 register esi and for write access permission (0x2) into edx. On line 5 we call protect to register
 635 this permission. Next on line 6, we load an address to an object off the stack and into register rax.
 636 Finally, on line 7, we set four bytes of the object header to 0x1 to indicate it is a regular unlocked

637

638 object. After this, a call to `unprotect` reverts the thread's permissions to what they were at lines
 639 2 and 3. The idea is that if line 7 writes to 4 bytes in a valid mark word typestate then the program
 640 does not have a memory error – a true negative check by Permchecker.

641

642 **4.5.3 True Positives.** Now consider a bug where the mark word annotation is in the wrong place,
 643 or the C++ compiler erroneously reorders a memory access to the object's mark word. Thus the
 644 memory access occurs outside the scope of the `protect` call as follows:

645

```
646 1 // Begin perms: stack (RW)
647 2 mov $0x21,%esi
648 3 mov $0x2,%edx
649 4 call protect // Begin perms: stack (RW) & mark words (W) (A2)
650 5 call unprotect // Begin perms: stack (RW)
651 6 mov -0x28(%rbp),%rax
652 7 movl $0x1,(%rax) // Violation!
```

653

654 In this Code A2 Permchecker will report an access violation because line 7 here did *not* have
 655 permission to write to the mark-word even though line 7 is supposed to be able to do so. The key
 656 is that Permchecker does not make a claim of truth as to which permission was correct: the code
 657 or the memory. By doing so, Permchecker can effectively report localized memory errors without
 658 needing to assume *any* code is necessarily correct. Thus, not only is a bug in the implementation
 659 reported as a typestate mismatch, but a specification (or compiler) bug is reported as such too.

660

661 **4.5.4 Benign Checks.** In the two snippets of assembly code above, we in fact accessed two different
 662 kinds of memory: a mark word, and a pointer to that mark word found on the stack. For the true-
 663 negative, both memory accesses occur within the scope of our call to `protect`, but we only needed
 664 the mark word access to be checked. In checking both, we made a tradeoff in how Permchecker was
 665 designed. The idea is that it is often sufficient to check a memory access against a set of possibly
 666 unrelated typestates when doing so simplifies our permission specification.

667

668 **4.5.5 Exceptions Limitation.** When Permchecker instruments a function with calls to `protect` and
 669 `unprotect`, it does not handle the presence of throw-catch style C++ exceptions. Nor does it reason
 670 about any other dynamic feature which alters ordinary program control flow in a permission-
 671 annotated function. A long term goal for Permchecker is to obviate this limitation by directly
 672 relating typestate permissions with individual memory access assembly instructions in the code
 673 spaces of a process. This high level of individual detail, however, increases by default the upfront
 674 annotation cost required to annotate a memory manager. Therefore some care must ultimately go
 675 into the design of assembly instruction permission techniques for Permchecker to support.

676

677 **4.5.6 Failure Modes.** A false positive is generally not possible with Permchecker. This is because a
 678 typestate error indicates the presence of a bug in the memory manager, the layout specification, or
 679 both. Such a bug can even be present and reported as an error when the application runs correctly.
 680 Such a report indicates an at-least benign bug that *should* be fixed.

681

682 In contrast, a false negative *is* generally possible with Permchecker. For example, a false negative
 683 will occur in Code A1 when attempting to write the mark-word actually writes a value to stack
 684 memory. Permchecker does not report this as an error because the executing thread had permission
 685 to write to the stack when the purported mark word instruction clobbers the stack. To mitigate
 686 this kind of source of false negative, we have come up with a dynamic heuristic to understand the
 sensitivity of the specified code permissions.

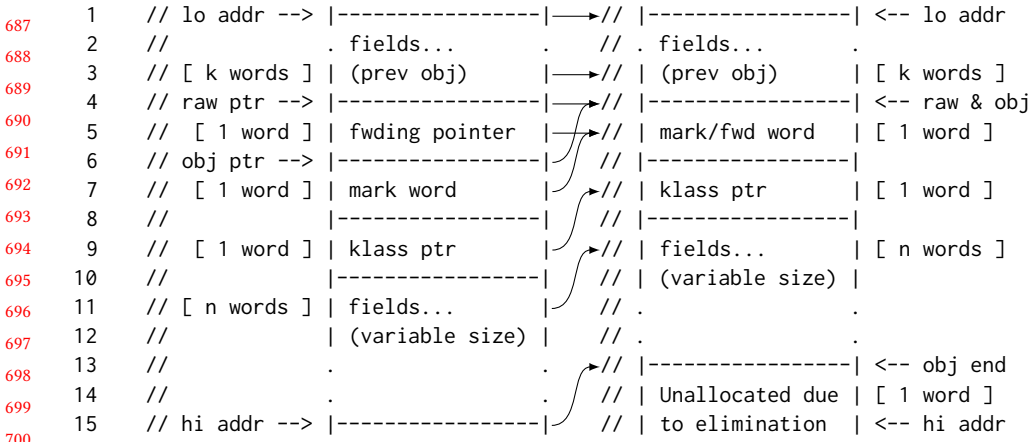


Fig. 5. ASCII-art diagram depicting the difference between two memory layouts central to an off-by-one bug affecting memory safety.

The heuristic Permchecker provides is to generate a table of typestate usage statistics at the end of a program’s execution. This table includes which typestates each instruction¹ was observed to have accessed, and whether or not any permissible typestates were never accessed. The idea is that any given instruction in a mostly-bug-free memory manager will access a fixed set of typestates after sufficient testing. Then, any remaining permissible typestates are suspicious. The permission annotation could have been overly broad, blanketing a number of unrelated memory access instructions. Or, the instruction was insufficiently tested. In either case, there exist remedial actions the memory management developer can take which either incrementally improve the precision of the annotations, or improve the typestate coverage of the testing benchmarks.

4.6 Runtime Debugging

With typestate tracking and permissions in place, we now need to check the validity of each memory access. The goal is to ensure that for each address the only load and store instructions operating over it occur when the code has permission to access it. This section discusses how we achieve this goal by presenting an example where a header’s mark word is incorrectly accessed.

4.6.1 Mark Word Example. In Figure 5 we see two distinct memory layouts. On the right is the layout of an allocated Java object, as used in the production version of Hotspot. On the left is a modified version of this layout, with a word of memory containing a forwarding-pointer added to the beginning of the object’s header. In the middle of the diagram arrows pointing left-to-right indicate how the different typestates shift in the address space when the author of the left layout modified a part of the memory manager to use the layout on the right.

In Figure 6, we see the error observed and reported by Permchecker when two different pieces of code disagree on the expected layout of objects in memory. In this case, the VM allocates and initializes the contents of objects according to the production version of the VM where non-array objects consume 2 words of memory for the header. When the VM proceeds to access memory according to the development version with an extra header word, Permchecker reports that an offending memory access read a Klass pointer but expected (had permission to access) mark word typestates. The idea here is that Permchecker does not assume that *either* the observed layout or

¹Requires debugging symbols to get source locations.

```

736 [error][pck] /jdk13/src/.../oop.inline.hpp:123
737 Permchecker Violation in Thread #2:
738   Expected tpestates: PCK_MarkWord, ...
739   Observed tpestate: PCK_Object_Header_Alloc_KlassPtr
740   Address = 0x00007ffb1c43e008
741   Address is preceded by 1 words of PCK_Object_Header_Alloc_MarkWord
742   Address starts 1 words of PCK_Object_Header_Alloc_KlassPtr
743   This is followed by 6 words of PCK_Object_Alloc
744   Dumping tpestates to /memdbg/permchecker.3952.log
745   Dumping core file to /memdbg/core.3952

```

Fig. 6. The error reported by Permchecker when code assuming the left-layout of Figure 5 attempts to access the mark word of an object allocated and initialized by code assuming the layout on the right of that figure.

the expected layout are necessarily correct. In fact, both could be incorrect. Instead, we label code with intentions and Permchecker treats those annotations not as a ground-truth specification of expected behavior, but as a mechanism for pin-pointing inconsistencies.

4.6.2 Lightweight Debugging. Permchecker supports both a lightweight and a heavyweight mode. In the lightweight mode, only memory locations explicitly assigned a tpestate are checked for access permissions. In this mode, a program with no updates to the tpestate map will never produce a violation because all memory locations remain in the UNMAPPED tpestate. The idea is that the lightweight mode eliminates a lot of error detection noise when the memory manager is initially being annotated, favoring local sensitivity over system-wide sensitivity. It does this by only telling the developer when an unexpected piece of code accesses memory in a known tpestate. Memory accesses to locations with unknown tpestates are ignored.

4.6.3 Heavyweight Debugging. In the heavyweight mode, Permchecker reports an impermissible memory access for all tpestates, including UNMAPPED, as an error. In this mode, a program with no updates to the tpestate map will produce a violation for *every single* memory access because no instruction has permission to read or write unmapped memory. The idea is that the heavyweight mode increases the number of true positives once a related set of tpestates have been annotated in the memory manager, thus increasing system-wide sensitivity. While the lightweight mode allows the developer to build the lifecycle of an individual tpestate, the heavyweight mode discovers all buggy, unspecified, or improperly specified memory accesses.

4.6.4 Thread Permissions. In Permchecker's dynamic instrumentor, a per-thread permission tracker is implemented as two extensible bit-vectors representing read and write permissions each with one bit per tpestate. When a bit in the vector is set, the associated thread has that kind of (read or write) permission to the tpestate associated with the offset of the bit into the bit-vector. This mechanism directly relates to how the lightweight and heavyweight modes differ.

The lightweight mode can be thought of as an opt-in checking model, where all tpestates implicitly have their read and write bits set, except an explicitly named set of them. The one exception to this occurs when a thread's permissions are annotated as `pck_NONE(<ts>)`, which does cause the given `<ts>` to be checked. In contrast the heavyweight mode can be thought of as an opt-out checking model, where all tpestates implicitly have their read and write bits unset. As a result every single memory access is checked by default, and the developer must tell Permchecker to explicitly allow a memory access by annotating it.

Table 1. Each feature Permchecker touches, what system component it involves, and the feature’s purpose with respect to typestate.

Feature	Component	Typestate Purpose
FFI	VM modification	Traffics typestates from non-native VM code with prerogative over typestate.
Annotations	Clang extension	Distributes a typestate permission over an entire VM component: functions and classes.
Layout Spec	Preprocessing tool	Allows for developer-defined relationships among typestates as an allocation hierarchy.
Codegen	Preprocessing tool	Supports the expression of common permission idioms and typestate transitions.
Macros	VM boilerplate	Manages error-prone calculations for offsets among typestates.
Proxy objects	C++ overloading	Integrates typestate operations with the VM’s host language for ease-of-use.
Tracking API	pck namespace	Flexibility to manage typestates at non-function or class boundaries.
Assembly instr. Checker	JIT compiler Dynamic Instrumentor	Typestate checking of runtime generated code. Applies a simple and rational checking model across every single memory access.
Shadow memory	Library	Maintains a snapshot of the typestate map.

5 DEBUGGING TOOLCHAIN ARCHITECTURE

Different components of a VM have access to widely varying mechanisms to support typestate tracking. A simple numeric typestate tracking system is a modest endeavor in and of itself, short of supporting fully featured contracts and logic. In achieving the former, Permchecker takes advantage of the language features listed in Table 1 in order to manage and track typestates.

5.1 Artifacts & Mechanisms

In Figure 7 we show how typestate information flows through the toolchain at different levels of abstraction. For instance, the dynamic binary instrumentor (DBI) intercepts function calls at run time to the typestate tracking interface. The DBI then uses Pin’s API for instrumenting memory accesses, in order to check typestate permissions. The idea is that a shared universe of typestate identifiers applies across all abstraction levels. This is necessary because each level accesses and mutates the exact same pieces of memory according to specially crafted policies. These policies all require unsafe low-level access, unlike any other domain to our knowledge, in order to extract every ounce of performance from components including the JIT compiler, VM host-language, allocator, garbage collector, and heap mutator.

Many of the components listed in Figure 7 center around compilation of a C++ or C-like implementation language for the memory manager. The Permchecker toolchain most seamlessly allows for the instrumentation, compilation, and dynamic interception of typestate-related runtime events of a memory manager implemented in C++. We now discuss, in Sections 5.2 and 5.3, our experience using Permchecker components in two memory managers other than Hotspot.

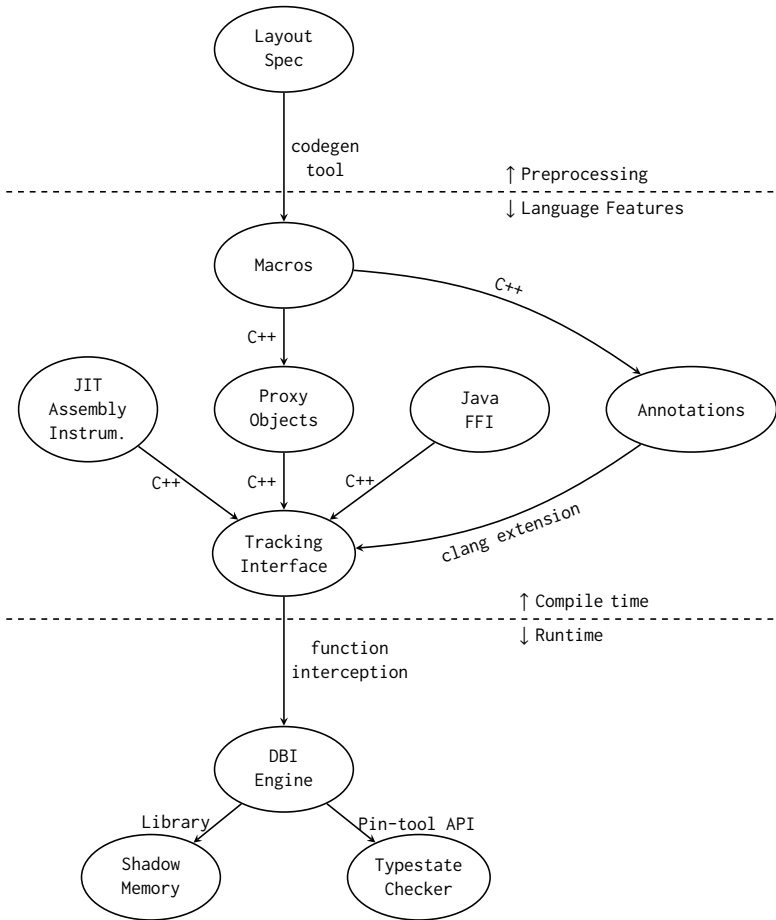


Fig. 7. Architecture of the Permchecker toolchain, indicating how typestate tracking and permissions are implemented. Nodes are artifacts in the system which involve typestate, and arrows are mechanisms for enacting their purpose.

5.2 Jikes RVM

In the Jikes RVM, Permchecker aided in the detection and diagnosis of a latent memory layout bug in an uncommon configuration of the runtime system. This bug manifests when two header fields of an object are defined to be different sizes, resulting in a memory error. This error is caused by an incorrect size variable being used to compute the offset of one of the two object header fields. The bug however is benign in the default, common, system configuration because the incorrect size variable happened to be the correct value. Notably, the offset calculation and resulting errant memory access is implemented in Java.

Detecting this Java bug with the Permchecker components listed in Table 1 involves three conditions. First, a mature FFI to C++ or assembly is necessary to communicate typestate information to members of the pck namespace as compiled with Clang or GCC². We consider this burden minimal, as a FFI is an already integral part of memory manager implementations. This was the case

²Clang only required for function and class annotations.

883 with Jikes RVM, for which a readily extensible FFI to C++ allowed us to instrument the VM with
884 tpestate transitions and code permissions.

885 The second condition is that the layout specification language’s codegen and accompanying
886 tpestate annotations do not apply. Instead, custom VM-specific annotations for defining code
887 permissions were implemented into the staged compilation process of Jikes RVM. Functionally
888 these annotations are equivalent to the Clang extension for permissions. However, the presence of
889 a separate bootstrapping Java compiler in the Jikes RVM build process complicates the detection
890 of early memory corruption with annotations. For example, in order to achieve 100% memory
891 coverage with tpestates a mechanism for tracking a memory location across bootstrapping stages
892 is necessary. We leave this to future work.

893 The final condition arises from the fact that debugging information, such as filenames and line
894 numbers, are not necessarily readily available to the permission-checking Pin tool in the form of
895 DWARF debugging information. To address this, the Pin tool includes the ability to receive on
896 the command line a mapping from instruction pointers to filenames and line numbers. While this
897 ability is not utilized in our study of Hotspot, our experimentation with the Jikes RVM requires it
898 to report helpful diagnostic information. As a part of Jikes RVM’s compilation process a debugging
899 map is produced, thereby reducing the burden of enabling Permchecker’s diagnostic capabilities
900 to mechanically translating the information into a reportable form.

901 Finally, with respect to Jikes RVM, the latent memory layout bug discussed earlier closely mir-
902 rors the Hotspot header layout bug causing the error reported in Figure 6. In the Jikes RVM bug,
903 Permchecker reports a memory access to one object header field when the VM only had permission
904 to access another (adjacent) field. Upon inspecting the recorded tpestates for surrounding mem-
905 ory locations it becomes apparent that the pointer access is in conflict with memory as-allocated,
906 leading directly to the source of the bug.

907

908

909

910

5.3 dmalloc

911 This section discusses the applicability of Permchecker’s dynamic tpestate checking capabilities
912 to the implementation of a manual memory allocator.

913

914 *Instrumentation Burden.* In the dmalloc we fully annotate and instrument tpestate transitions
915 for memory related to a program’s heap as allocated through the standard malloc and free func-
916 tions. The source code for unmodified dmalloc version 2.8.6, ignoring mspace variants of the code,
917 amounts to around 5,500 lines of code and documentation. After annotation, the source code con-
918 tains an additional 600 lines of code including code permission annotations, tpestate transitions,
919 and manual tpestate assertions.

920

921 Another 400 lines of boilerplate code consists of glue code for Permchecker, tpestate identifier
922 definitions, and tpestate transition macros. Chronologically, we studied dmalloc prior to devel-
923 oping any of the components listed in Table 1, except the last two rows comprising the dynamic
924 checker and shadow memory. Therefore, the 400 lines of code added to dmalloc correspond to
925 code automated in our study of Hotspot. Of the 600 lines of code annotations and instrumentation
926 listed, about 100 correspond to explicit calls to protect and unprotect in dmalloc. This effort is
reduced by half by the Clang-based function annotations, as applied to Hotspot.

927

928 *Lessons Learned.* **Annotating a legacy memory manager with tpestate permissions is a**
929 **practice in pattern matching** as one spelunks through the code. For example, dmalloc exten-
930 sively uses C structs to “declare a *view* into memory allowing access to necessary fields at known

931

931

offsets from a given base.” [Lea 1991]³Annotating access to such fields is a systematic refactoring-like process of annotating each field (de)reference operation.

It is ideal to test the annotation of a memory manager with tpestates gradually. While the appropriate tpestate for a function is typically readily apparent from textual context in the program, annotations are no less susceptible to typos, categorical mistakes, and missing annotations by the programmer. We discovered this in our study of `dlmalloc` by testing the system after scanning for instances of a code pattern (such as accessor functions), and annotating them all appropriately.

The most common instrumentation failure mode we observed involved failing to adequately generalize an implementation pattern, resulting in missed annotations. Such failures become immediately apparent when tested by `Permchecker` in conjunction with inspection of code and memory coverage statistics. It is reassuring to recognize that these simple procedures can effectively rule out memory corruption – procedures including manual refactoring, iterative testing, and code and memory coverage inspection.

In first studying `dlmalloc` then switching to `Hotspot`, we pinpointed the language tools needed to instrument a memory manager (1) systematically, (2) gradually, and (3) progressively towards ruling out memory stomping. We then built and studied these tools, each listed in Table 1, in the context of `Hotspot`. Ongoing future work now involves developing more expressive tooling for post-hoc tpestate inspection, live (GDB-like) breakpoints, historical program tracing, and program slicing in terms of tpestates.

6 BUG INJECTIONS

In this section we present a methodology, and application thereof, for injecting bugs into a memory manager. The idea is that each injected bug is an instantiation of a template designed based on real bugs found in memory managers. Each template encapsulates the memory safety or correctness effects of the original bugs, with consistent reproducibility of many unique injectable bugs at modest scale. Having such unique bugs at scale then allows us to stress the system for a broad array of behaviors and failure modes.

6.1 Object Cloning Bug Injection

The core functionality of one bug template used to inject bugs into `Hotspot` is shown in Figure 8. The code from this figure is placed in the compaction phase of the Shenandoah garbage collector. More generally, the idea is that this particular template can be injected anywhere in the code where an object pointer (oop type) is available. The template works by delaying bug injection until some number of executions have happened. Once this happens the code injects a single non-recurring bug that will not simply crash the system outright.

To run this template, we must first decide on an initial value for `__CORRUPTION_COUNTDOWN`. This value tracks how many objects have been forwarded during concurrent compaction, and therefore changing it varies which exact objects are affected by the injected bug. By varying it over a large number of sufficiently long-running program runs, we now have numerous injected bugs each with differing effects on the downstream behavior of the system.

It should be noted that this bug template is not entirely divorced from tpestate annotations. In fact, the functions `is_forwarded`, `klass`, and `forwardee` *are* annotated with the appropriate access permissions, allowing ostensibly buggy code to access memory. However, it is evident these functions are used here in an appropriate, non-buggy, manner. They are all used to access header words of valid heap objects for which the tpestates match the usage of the values being read.

³Emphasis added, quotes removed.

```

981 1 oop p = ...;
982 2 if (DoInjectCorruption && InjectID == 1 && !__CORRUPTION) {
983 3   if (__CORRUPTION_SIZE > 0 && p->klass() == first_kls) { // 3rd case
984 4     HeapWord *src = (HeapWord*) p;
985 5     HeapWord *dst = first_forwardee;
986 6     Copy::aligned_conjoint_words(src, dst, size);
987 7     oop(dst)->init_mark_raw();
988 8     log_info(permchecker)("Corruption #1");
989 9     __CORRUPTION = 1;
990 10  } else if (__CORRUPTION_SIZE == 0 && // 2nd case
991 11    __CORRUPTION_COUNTDOWN == 0 && p->is_forwarded()) {
992 12    __CORRUPTION_SIZE = (size_t)p->size();
993 13    first = (HeapWord*) p;
994 14    first_kls = p->klass();
995 15    first_forwardee = (HeapWord*)p->forwardee();
996 16  } else if (__CORRUPTION_COUNTDOWN > 0) { // 1st case
997 17    __CORRUPTION_COUNTDOWN--; } }
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029

```

Fig. 8. The core functionality of a template for injecting object cloning bugs into Hotspot. In the third conditional branch, we count down some number of executions of this code fragment. In the second branch, we then record an object located at pointer “p”. Finally in the first branch, we discover a second object of the same Klass type and (*incorrectly*) forward its contents to the destination of the first object. The variables `first`, `first_kls`, `first_forwardee`, `__CORRUPTION_COUNTDOWN`, `__CORRUPTION_SIZE`, `DoInjectCorruption`, `InjectID`, and `__CORRUPTION` are static class members.

It is not until the Copy operation that it becomes clear that a memory error is present. During this operation, we read the contents of a valid object from `src`. But as it turns out, we write these contents to a location which *already contains an object as well*. This fact about the tpestate of the destination is already tracked by Permchecker when the destination first received a valid object, and therefore the operation is in error.

When the developer is told about this error, it might be tempting to think there is a specification error and that the developer should give the copy operation permission to overwrite allocated tpestates. This option however makes no sense after minimal scrutiny, because transporting an object from one location to another should never be allowed to write to memory already in an allocated state. Instead, the developer must do one of two things. One, he can check that the destination is in the appropriate free tpestate, transition it to the allocated tpestates, and then perform a copy operation with permission to write to the allocated tpestates. Or two, he can specify the copy operation has permission to write to the free tpestate, and then transition the memory to the allocated tpestates after copying.

The latter technique is preferable because it is less error-prone: the developer cannot forget to insert a tpestate check, because the check for free memory is implied by a new annotation on the copy operation. The key is that with or without adding either the permission annotation or tpestate transition, Permchecker reports an error. Based on this reasoning, it is generally better practice when instrumenting a memory manager to rely on automatic checks by the dynamic instrumentor than to use a technique that relies on the developer to check a tpestate before changing it. Discovering this sort of best practice is one overarching goal of this work: to not only build correct systems, but to build error-resistant debugging and instrumentation techniques.

Table 2. Breakdown of bug templates injected into Hotspot, as detected with and without Permchecker. The second column group, Injections & Pck, indicate the total number of executions of the VM and how many of those were detected by Permchecker, respectively. The third column group indicates how the VM reacts in the absence of Permchecker.

Template	Injections	Pck	Ad-hoc	OS Error	Output	OOM	Benign
Object cloning	13, 812	13, 812	4, 820	0	8, 957	0	35
Use-after-free (R)	836	832	0	379	452	1	4
Use-after-free (W)	799	799	0	346	432	0	21
Overflow (padding)	8, 658	8, 658	313	0	0	0	8, 345
Overflow (no pad)	10, 026	10, 026	9364	662	0	0	0
Wasted Memory	1, 000	0	0	0	0	0	1, 000
Move Failure	976	649	562	31	2	5	376

6.2 Bug Template Results

In Table 2 we see how well Permchecker managed to detect the injection and execution of a series of bug templates, including the one just discussed in Section 6.1. A bug template when instantiated and tested with specific program inputs is an *error benchmark*. A key strategy we developed to create good error benchmarks was to confine the effects of a bug to a single execution of some operation by the memory manager. This strategy provides the following benefits:

- There is an increased chance of the bug evading traditional error detection mechanisms, mimicking the rarity of incidence exhibited by similar real bugs.
- No static analysis is necessary in order to synthesize or construct conditional statements which help rarify the bug’s execution.
- The number of injections scales because the number of dynamic operations performed by the garbage collector is unbounded, unlike the number of static injection sites.

Based on this strategy, we created the bug templates listed below. For each template, the number of injections reported in Table 2 reflects the number of times we were able to get the template’s preconditions to be satisfied. For example, the use-after-free templates require finding an object near the end of its containing region, and therefore required more compute power to accumulate the same number of injections as some of the other templates.

Object cloning - this bug template is the code fragment from Figure 8, which discovers two object pointers of the same Klass type, and overwrites the memory of one with the other. This template generally works in any VM function with access to at least one heap object pointer per execution.

Use-after-free (read) - this bug template discovers a single object pointer near the end of a region, waits for its contents to be evacuated, and then redirects the application’s next field access to use the freed memory as a base pointer in its load instruction.

Use-after-free (write) - like above, but the field access is a write instruction causing garbage to be “corrupted” and the intended object field to not be updated.

Overflow (padding) - this bug template duplicates the Klass metadata for a single object instance when it gets allocated, modifying the duplicate Klass to indicate this one object has some multiple of an object’s alignment less of memory than it requires based on the fields in the object.

Overflow (no padding) - like above, but the injection only happens when the last field of the object ends on an object alignment boundary.

Wasted Memory - this bug template operates similarly to the overflow templates, but the Klass is modified to indicate the object consumes *more* memory than it requires.

1079 **Move Failure** - this bug template discovers an allocated object residing at the end of a memory
1080 region and skips compacting it, possibly causing other object(s) to point to a valid-looking object
1081 that the memory manager believes to be free memory that can be allocated into.

1082

1083 6.3 Choice of User-Level Application

1084 One kind of memory bug we wish to study is one that does not crash the program at all. Instead,
1085 it affects the correctness of the user application such as program output. The incidence of output-
1086 based errors relies heavily on the domain and specifics of the user application. Characteristics
1087 affecting whether or not an output error will arise include object allocation patterns, drag time
1088 between an object's last use and when it becomes unreachable, and more. For the results in Table 2,
1089 we modified a version of GCbench [Ellis et al. 2014] to maximize fragility by minimizing drag.

1090 Fragility, here, means how likely an application is to compute the wrong result if one of its
1091 objects gets corrupted. The original unmodified version of GCbench is a prime example of a non-
1092 fragile benchmark. GCbench by default iteratively constructs a number of binary trees of certain
1093 depths, in order to stress the *performance* characteristics of a garbage collector. The benchmark is
1094 not actually intended to compute anything of algorithmic value. Therefore, so long as the applica-
1095 tion terminates at the end of main, it "computed" the correct value.

1096 In contrast our modified version of GCbench maximizes fragility. It does this by computing a
1097 hash over the contents of each node in a binary tree shortly before expecting the tree to become
1098 garbage. This hash iteratively accumulates over each node allocated. The hash is printed upon pro-
1099 gram completion. This strategy aggressively links correctness of application output to the presence
1100 of heap corruption. This strategy is further ripe for automation over any existing application in
1101 domains other than just performance analysis.

1102

1103 6.4 Analysis & Effectiveness

1104 With the use-after-free (write) template, both pointer updates and primitive field updates were af-
1105 fected throughout the various error benchmarks. As a result a small number of the benchmarks ter-
1106 minated significantly early (with incorrect program output), presumably because a failed pointer
1107 initialization lopped-off a significant portion of a recursive data structure before it was processed
1108 by the application. The same behavior appears to have happened with the use-after-free (read)
1109 template, where some of the incorrect-output benchmarks terminated early because a load of a
1110 pointer to a large recursive sub-structure was redirected to a smaller one.

1111 Of the 4 benign results with the use-after-free (read) template, all of them went undetected by
1112 Permchecker. This means that the freed memory was necessarily quickly reallocated by the mem-
1113 ory manager, bringing it to a typestate validly accessible by the application mutator. Subsequently
1114 the application accessed a field at the wrong address, but happened to read the correct value so-as
1115 to be benign. We believe these 4 injections have to do with leaves of a binary tree being imple-
1116 mented as the NULL value. As a result a memory access to the left or right child of a node was
1117 corrupted, but would have received the NULL value it was supposed to read anyways.

1118 This behavior exposes a limit to Permchecker's checking capability. Permchecker cannot detect
1119 a dangling pointer error when the memory pointed to by the dangling pointer is reallocated to
1120 a similarly allocated object of the same typestate and size. Valgrind's Memcheck tool is similarly
1121 unable to detect this subclass of dangling pointers, for a fundamentally identical reason. Neither
1122 tool checks how a pointer value is obtained, just that an observed access to the referenced memory
1123 is permissible in each tool's checking model.

1124

1125 *6.4.1 Observable Behaviors.* In Table 2 we break down the frequency with which each bug tem-
1126 plate causes a variety of observable behaviors in the absence of Permchecker. These behaviors are a

1127

1128 holistic collection of failure modes that a developer reasons about while debugging. The following
 1129 list defines each failure mode:

- 1130 • **Ad-hoc:** a VM assertion error, usually from a suspicious looking pointer or value.
- 1131 • **OS Error:** a segmentation fault, bus error, or other OS error even if the VM intercepts it.
- 1132 • **OOM:** an out-of-memory error reported by the VM.
- 1133 • **Output:** the program successfully terminates with incorrect user-level textual output.
- 1134 • **Benign:** the program successfully terminates with correct user-level textual output.
- 1135 • **Deadlock:** corruption causing non-termination. This was not observed in testing.

1136
 1137 We consider “benign” program behavior to be *bad* when a corruption is known to have occurred.
 1138 In fact, all the benign bugs from Table 2 can rightly be considered failure modes – in each case
 1139 the system does not exhibit signs of known corruption. The “Wasted Memory” benchmark, too,
 1140 corrupts the layout of memory. This corruption occurs benignly in a way that the Permchecker
 1141 annotations were unable to detect because no read or write instructions access the “extra” memory.
 1142

1143 **6.4.2 Utility of Error Reports.** It is desirable for the contents of an error reported by Permchecker
 1144 to be diagnostically “effective.” Subjectively, an effective error report is one which identifies infor-
 1145 mation pertinent to the source of the error. With tpestate checking, a simple first-order metric
 1146 is whether or not either the permissible or the observed tpestates, of a Permchecker error report,
 1147 indicates the memory (or layout) which was actually corrupted.

1148 In all of the Permchecker error reports pertaining to Table 2, each report included the pertinent
 1149 (corrupted) tpestate. For instance in the “Overflow (padding)” template, Permchecker always re-
 1150 ports an error where the program had permission to access an object `Field`, but in fact accesses a
 1151 `Padding` tpestate. Similarly, with the “Overflow (no pad)” template, Permchecker always reports
 1152 an error involving access to a `MarkWord` but observes a `Field` access, or vice-versa.

1153 The takeaway is that tpestate checking improves sensitivity over the existing ad-hoc error
 1154 checks. Checking also exhibits increased diagnostic value by reporting an observed type, in-lieu
 1155 of just an expected type implied by the stack trace of an ad-hoc check. In essence, Permchecker is
 1156 capable of meaningfully diagnosing a generalized class of *initialization* bugs.

1157 An initialization bug is a bug in which a resulting program error is traceable to a root cause based
 1158 on the most recent component of the system which (1) initialized the memory to its observed state,
 1159 or (2) should have initialized it to its expected state. Part (1) indicates Permchecker’s bug detection
 1160 capabilities are similar to that of a taint analysis. Part (2) indicates Permchecker’s ability to detect
 1161 some control-flow bugs, e.g. a missing deallocation call.

1162 Notably missing from these two kinds of bugs are *value* and *logic* bugs. For example, a garbage
 1163 collection (GC) algorithm failing to track all GC roots leads to a premature free. Permchecker
 1164 then detects this situation as a memory access by the application to purportedly free memory.
 1165 This safety manifestation of a decidedly logic-based GC bug illuminates the difference between
 1166 diagnostic value and pure detection – Permchecker reports less effective information for value
 1167 and logic bugs than it does for initialization bugs.

1168
 1169 **6.4.3 Latency of Error Reports.** Prior to writing bug templates and testing their impact on the be-
 1170 havior of the VM, we had little basis for any kind of insight into how quickly tpestate annotations
 1171 would be able to detect a memory safety error compared to existing ad-hoc checks in the VM. In
 1172 one formulation, we believed existing and often value-based sanity checks in the VM might excel
 1173 at proactively preventing bad values from being operated over. In contrast, Permchecker’s tpestate
 1174 annotations might excel at detecting bad operations once they happen, after a bad value has
 1175 already been created computationally rather than loaded from memory.

1176

1177 For the bug templates we chose, this formulation was entirely not the case. In all the injections
1178 from Table 2, **Permchecker detects an error accounted for in the third column prior to**
1179 **any corresponding error accounted for in the fourth (ad-hoc) column.** Methodologically
1180 this ordering is determined by running the VM with *both* forms of checks enabled, but where the
1181 program simply logs the error reports rather than terminating immediately. This methodology
1182 has the benefit of obviating any need to create strictly reproducible thread schedules for the VM.
1183 Furthermore it is valid to compare program behaviors this way because program checkers have
1184 ostensibly no side-effects or impact on values in the VM. The tradeoff is that this methodology can-
1185 not compare one-to-one results of Permchecker with any similarly purposed, but non-composable
1186 with Permchecker, dynamic instrumentation tools.

1187

1188 7 MOTIVATING OBJECT MONITOR BUG RESULTS

1189 We now discuss the results of debugging and diagnosing the deadlock bug, as motivated at the
1190 beginning of Section 1. This discussion necessarily benefits from hindsight, as the bug was in fact
1191 diagnosed and fixed without the aid of Permchecker through manual instrumentation of Hotspot’s
1192 code which manages object monitors. An object monitor is a locking mechanism which prevents
1193 two or more threads from executing Java methods of the same object instance at the same time.

1194

1195 7.1 Annotation Considerations

1196 By instrumenting object monitor code with ad-hoc log statements, the original debugging devel-
1197 oper ultimately observed an erroneous lock operation. This erroneous operation is unique because
1198 it is memory corruption. Memory corruption typically involves an access to memory containing
1199 data of the wrong type. In this object monitor bug, however, this was not the case. The root cause
1200 occurs in a function checking the boundedness of a memory location on the Java stack. This func-
1201 tion incorrectly reports malloc’d memory adjacent to the stack as being part of the stack. Thus, a
1202 pointer to memory of the *correct* type (object monitor) is corrupted by code *allowed* to access it.

1203 In applying Permchecker annotations to Hotspot code related to object monitors, we observe
1204 some benefits as well as drawbacks to tpestate annotations. First, a permission annotation on a
1205 function is applicable to *all* callers of the function. As a result, a (calling) context sensitive tpestate
1206 property is in tension with code reuse – tpestate annotations must invariably be placed higher
1207 and higher in the call stack, leading to more and more annotations at individual call-sites.

1208 With respect to memory layout oriented tpestates, however, there exist actionable and sys-
1209 temizable mitigations to the code smell of widespread annotations. In the case of Hotspot’s object
1210 monitor code, we mitigate this by specializing multiple instances of a `ObjectMonitor` C++ class
1211 for each allocation context (or hierarchy). The idea here is that the specialization of the existing
1212 (object monitor) functionality maintains functional equivalence and therefore the modification
1213 cannot by itself introduce (non-Heisenbug) program errors.

1214

1215 7.2 Object Monitor Refactoring and Debugging Process

1216 The allocation contexts for `ObjectMonitor` include certain C++ runtime system code managed by
1217 the C++ compiler, and compiled application code compiled by the Java compiler. An object monitor
1218 allocated for the former purpose is allocated into the malloc heap. In contrast, an object monitor
1219 allocated for the latter purpose is allocated onto the Java stack. By first separating out call-sites of
1220 accessors based on these two allocation contexts, Permchecker is then able to correctly validate
1221 layout safety in each context. The idea here is that this refactoring process is (1) minimally invasive
1222 with respect to functionality, (2) adds otherwise unknown information to the implementation /
1223 artifact, and (3) systematically applicable to all allocation hierarchies. The application of object
1224 monitor tpestates to Hotspot went as follows:

1225

- We identify allocation sites of object monitors, and annotate them with typestate transitions. These sites include ones in C++ as well as generated assembly from the Java compiler. A grep for the `ObjectMonitor` class was a good first step here.
- We refactor the `ObjectMonitor` class to specialize two versions of it with annotations.
- We propagate refactoring up to call sites for each of the allocation contexts: stack & malloc.
- Finally, we validate the memory layout by running preliminary test applications which exercise the object monitor related runtime system code.

In the last step above, Permchecker reports as a violation any (1) access patterns we missed or applied annotations to incorrectly, and (2) true memory layout errors. A missed access pattern may appear to Permchecker as an invalid memory access. A true memory layout error for the object monitor deadlock bug appears as a call site to an object monitor accessor function intending to lock a monitor on the Java stack, but is observed by Permchecker to be on the malloc heap.

7.3 Boundedness and Diagnosability

The most applicable class of properties also validated by existing memory checkers is *boundedness*. Boundedness is a property which helps define whether or not a pointer references memory within the bounds of a specified chunk, or class of chunks, of memory. The technique of modeling object ownership [Weiss et al. 2019] places annotations governing boundedness into code on a per-variable (per memory chunk) basis. This high level of per-chunk specificity achieves local integrity at the cost of global safety checking (in a memory manager).

The Memcheck [Nethercote and Seward 2007] tool validates boundedness by partitioning addresses into unallocated and allocated⁴ classes (of chunks of memory). This per-class level of specificity achieves global integrity, such as an absence of certain use-after-free errors. However, low per-chunk specificity rules out the detection of unspecified boundedness properties.

8 SCOPE, LIMITATIONS, & FUTURE WORK

Threats to Validity. Permchecker targets the debugging and diagnosis of the most difficult to diagnose bugs in a memory manager. While we discussed limitations to Permchecker’s ability to diagnose value and logic bugs, those bugs remain detectable when they manifest as corruption. Bugs which do not do so, or cannot be reproduced in the first place, go undetected.

Permchecker’s dynamic instrumentation, in particular, can perturb a runtime system such that JIT compilation performance statistics, thread-scheduling, and the timing of garbage collections prevent a rare bug from manifesting. As a result, there exist memory corruption bugs which Permchecker will *never* be able to observe. This is especially problematic because the domain of memory manager implementations fall squarely into the class of programs which are highly sensitive to timing considerations. In this regard, typestate checking of memory managers will greatly benefit from the development of zero-overhead dedicated hardware to support live debugging.

Overhead. Permchecker’s dynamic instrumentor incurs a 10x to 50x slowdown in the tests of bug templates listed in Section 6.2. This observed slowdown primarily owes itself to the instrumentation of each and every memory access by the VM: stack, Java heap, and C++ metadata alike. At each memory access, the most common fast path of the shadow-memory implementation performs two or three additional memory accesses. This behavior leaves substantial room for improvement, either in smarter data structures or, optimally, hardware support similar in nature to page tables.

Thread Safety. Thread safety in the Java Native Interface (JNI / FFI), JIT compiler, and mutator slow paths of Hotspot are entirely handled by existing VM code. Thus the only place in the toolchain

⁴And initialization status.

1275 where thread-safety became a non-trivial concern was in the implementation of shadow memory.
 1276 As such we implemented shadow memory as a lock-free tree-like map data structure in which
 1277 map updates and map allocations are atomic. In the presence of a data race involving an update
 1278 and a read access to the tpestate of the same memory location, the shadow memory provides no
 1279 guarantee as to which tpestate the read access observes: the old one or the new one.

1280 *First-Order Permissions.* Permchecker is limited in scope to checking access permissions over a
 1281 fixed set of uniquely identifiable tpestates. This scope limits our ability to check the validity of
 1282 broader algorithmic contracts or predicates. For example, this predicate: “*the second write to some*
 1283 *Field at address α should be preceded by memory of tpestate MarkWord containing the value 0b00*”.
 1284 Such a check would require both a mechanism by which to communicate the desirable property
 1285 to the dynamic instrumentor, as well as efficient algorithms and data structures to track necessary
 1286 information and verify validity.

1287 Other debugging tools, notably GDB, supports features like conditional watchpoints and step-
 1288 wise debugging. These features allow a developer to know when the program accesses a particular
 1289 memory address, and to inspect subsequent instructions. Similarly, there is value in knowing when
 1290 the memory manager accesses a particular tpestate of memory. In this work we focused on build-
 1291 ing a toolchain to diagnose reproducible errors, but a natural future extension could involve more
 1292 traditional debugging features capable of inspecting tpestate at run time.
 1293

1294 ACKNOWLEDGMENTS

1295 This work was funded by NSF Grant #1717373.
 1296

1297 REFERENCES

- 1298 Karl Cronburg and Samuel Z. Guyer. 2019. Floorplan: Spatial Layout in Memory Management Systems. In *Proceedings of*
 1299 *the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (Athens, Greece)*
 1300 *(GPCE 2019)*. Association for Computing Machinery, New York, NY, USA, 81–93. [https://doi.org/10.1145/3357765.](https://doi.org/10.1145/3357765.3359519)
 1301 3359519
- 1302 B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale
 1303 Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, 110–121.
 1304 <https://doi.org/10.1109/SP.2016.15>
- 1305 John Ellis, Pete Kovac, and Hans Boehm. 2014. https://hboehm.info/gc/gc_bench/ Accessed: 2021-04-13.
- 1306 Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD.
- 1307 IBM. 2005. Jikes RVM. <http://www.jikesrvm.org/> Accessed: 2018-09-28.
- 1308 Doug Lea. 1991. A Memory Allocator. <http://g.oswego.edu/dl/html/malloc.html> Accessed: 2021-04-13.
- 1309 Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.
 1310 In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego,
 1311 California, USA) (*PLDI '07*). ACM, New York, NY, USA, 89–100. <https://doi.org/10.1145/1250734.1250746>
- 1312 Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record
 1313 And Replay For Deployability: Extended Technical Report. *CoRR* abs/1705.05937 (2017), 20 pages. [arXiv:1705.05937](http://arxiv.org/abs/1705.05937)
 1314 <http://arxiv.org/abs/1705.05937>
- 1315 Oracle. 2006. OpenJDK Hotspot Division. <http://openjdk.java.net/groups/hotspot/> Accessed: 2021-04-13.
- 1316 Andrew Rice, Edward Aftandilian, Ciera Jaspán, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. De-
 1317 tecting Argument Selection Defects. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 104 (Oct. 2017), 22 pages. <https://doi.org/10.1145/3133928>
- 1318 Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools
 1319 with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and*
 1320 *Symposium on the Foundations of Software Engineering (Lake Buena Vista, FL, USA) (ESEC/FSE 2018)*. Association for
 1321 Computing Machinery, New York, NY, USA, 224–234. <https://doi.org/10.1145/3236024.3236084>
- 1322 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Ad-
 1323 dress Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA,
 309–318. <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>

- 1324 R. E. Strom and S. Yemini. 1986. Tpestate: A programming language concept for enhancing software reliability. *IEEE*
1325 *Transactions on Software Engineering* SE-12, 1 (1986), 157–171.
- 1326 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A
1327 Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Program-*
1328 *ming Language Design and Implementation* (Phoenix, AZ, USA) (*PLDI 2019*). Association for Computing Machinery, New
1329 York, NY, USA, 48–62. <https://doi.org/10.1145/3314221.3314631>
- 1330 Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and A. Ahmed. 2019. Oxide: The Essence of Rust. *ArXiv*
1331 [abs/1903.00982](https://arxiv.org/abs/1903.00982) (2019).
- 1332
- 1333
- 1334
- 1335
- 1336
- 1337
- 1338
- 1339
- 1340
- 1341
- 1342
- 1343
- 1344
- 1345
- 1346
- 1347
- 1348
- 1349
- 1350
- 1351
- 1352
- 1353
- 1354
- 1355
- 1356
- 1357
- 1358
- 1359
- 1360
- 1361
- 1362
- 1363
- 1364
- 1365
- 1366
- 1367
- 1368
- 1369
- 1370
- 1371
- 1372