Permchecker: A Toolchain for Debugging Memory Managers with Typestate

ANONYMOUS AUTHOR(S)

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.

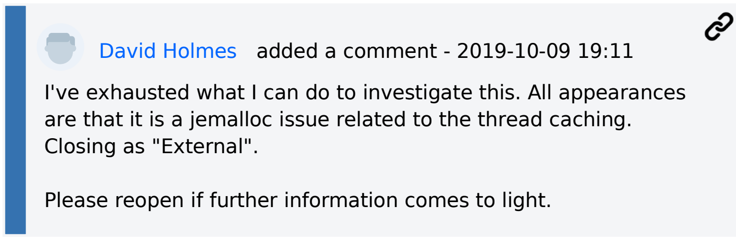
The implementation of Permchecker uses a combination of compile-time annotation and instrumentation, and dynamic binary instrumention. Because the overhead of DBI is fairly high, Permchecker is suitable for a testing and debugging setting and not for deployment. It works on a wide variety of existing systems, including explict 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 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:

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/


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 arithmetic, 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 specifically designed to aid in the debugging of memory managers, including both explicit allocators and garbage collectors. Permchecker's design is based on the observation that all memory managers perform essentially 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 address range reserved for nursery objects. Similarly, the argument to a manual free() implementation is an untyped pointer, and the implementation of 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.

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

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

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

1.1 Contribution

In this work, we develop a notion of *typestate* applicable to how a memory manager partitions memory and dynamically reuses it according to some algorithm. The goal of Permchecker is to provide a way for the developer of a runtime system to specify and check the lifecycles of memory typestates across all abstraction levels of the system. A relatively simple tracking API allows the developer to associate a typestate with a chunk of memory. A permissions API and annotations can then be used to declare access permissions. This formulation of dynamic typestate is a natural extension of the static typestate program analysis techniques developed in the 1980's [Strom and Yemini 1986].

The contributions presented in this paper are:

- A technique for specifying the hierarchical layout and decomposition of memory employed by the memory manager, and a tool that uses this specification to generate an API that the code can use to track structures of memory using explicit typestates.
- Compile-time support, implemented in the Clang C/C++ compiler, for automatically translating common typestate transitions from function annotations to API calls.
- A Pin-based DBI tool that tracks the typestate 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.

Typestate Permissions. It is crucial for the correct operation of a memory manager for runtime system code to only ever access the types of memory it is supposed to. For example, a free-list allocator should not find itself managing memory originally controlled 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 typestate since the type can change over the course of execution. After assignment, code which accesses the location must have permission to access the assigned typestate. An error is thus reported as soon as a memory access occurs over a typestate 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*."*

Incrementality. A primary goal of this work is to provide practical debugging tools and techniques that can help diagnose bugs in existing systems. To this end, Permchecker is built as a lightweight C/C++ API and accompanying heavyweight DBI tool runnable on virtually any memory manager. In this paper, we present our use of this tool to debug OpenJDK's Hotspot VM, Oracle's open-source version of their industrial-strength JVM. Not discussed further in this paper is our use of Permchecker to debug and verify memory layouts found in various manual and automatic memory managers alike. These include dlmalloc [Lea 1991], Doug Lea's implementation of the standard malloc/free interface, and the Jikes Research Virtual Machine [IBM 2005], a classic Java-in-Java implementation of a JVM. Without the trial-and-error process of studying this multitude of diverse systems, we would not have landed upon an *incremental* usage of Permchecker where the programmer starts out with a simple model of their heap, adding more and more typestate information over time.

3 BACKGROUND & RELATED WORK

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

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

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

Static Solutions. Typestate finds numerous applications in both static analysis [Strom and Yemini 1986] and static programming language features. For instance typestate in Rust [Weiss et al. 2019] derives from the language's support for linear and affine types at compile-time, by enforcing memory movement semantics statically in the context of a single variable. In contrast typestates supported by Permchecker derive from the toolchain's tracking of shadow memory at runtime, and enforcing memory access permissions dynamically in the context of an individual memory location. The key difference here is that a single variable cannot be aliased by constructing it from dynamic calculations, whereas a memory location *must* be aliasable with pointer arithmetic in the implementation of a memory manager.

Smart Pointers. Much like linear and affine types for enforcing usage constraints at compile-time on variables, a smart pointer can check or enforce certain usage properties at runtime *on a variable* as well. One canonical example is a reference-counted smart pointer for distinguishing live allocated objects from garbage. Such a reference counting scheme tracks the number of known pointers stored in memory in places where they can, in the future, be loaded into variables (registers, at runtime) and referenced as such with a load or store instruction. This ability to access memory indicates the memory remains allocated. A memory manager however not only manages allocated

```
<id>     := [A-Z][_a-zA-Z0-9]*
<ts>     := <id>        // A typestate is a unique identifier
<prim> := bytes | words
<stmt> := <ts> -> <ts>
   | <ts> -> seq       { <exp> , ... }         // One or more ','-separated exps
   | <ts> -> union { <exp> | ... }             // One or more '|'-separated exps
   | # <exp>     // Prefix notation for "zero-or-more" repetitions
<exp>   := <ts> | <stmt> | [0-9]+ <prim>
<spec> := <stmt>+
```

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

memory: it also manages and requires access to any and all freed memory. Thus, without reachability as a proxy for whether or not a memory location should be accessible, reference-counted smart pointers alone cannot sufficiently check memory management code for safety.

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

Bug Injection Methodologies. The process of collecting or generating a corpora of bugs is often motivated by a specific kind of mistake a programmer can make: off-by-one errors, operator selection defects [Rice et al. 2017], bounds-checking mistakes [Dolan-Gavitt et al. 2016], and more. The mechanics of these mistakes generally lend themselves to being constructed by source code mutation [Roy et al. 2018]. What this bug creation technique does not generally take into account, to our knowledge, is the modeling of undesirable algorithmic operations of the system having bugs injected into it.

In a memory manager, we know of a relatively fixed and small set of algorithmic operations that are bad: use-after-free, overlapping allocations, among others. In contrast, the categories of linguistic (syntactic and semantic) mistakes a programmer can make are limited only by which symbols the programmer decides to type and that the compiler will accept. The key difference then is that injecting invalid algorithmic operations into a system realistically stresses a broad category of resulting downstream behaviors of the system, while injecting linguistic bugs does not. Injecting linguistic bugs requires both a far larger enumeration of sources of linguistic mistakes and increasingly clever tactics for eliminating uninteresting bugs, such as ones which virtually always crash the program.

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 typestates in a memory manager. In doing so we illuminate the role of the compiler, VM annotations, and DBI engine in tracking and checking the typestates. 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 typestates over a process' address space. To start, the following are some important high-level definitions clarifying the scope of typestate in this work:

Typestate. This is a uniquely identifiable state of a memory location, coupled with the valid value types that can reside in that location. A mutable typestate 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 typestate restricts an assembly instruction to accessing a specific set of memory locations.

Set of typestates. This is a collection of typestates, 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 typestate. Unlike polymorphic types where the programmer typically defines new functionality to extend existing code, our variant of polymorphic typestate allows the programmer to model existing functionality in the presence of new typestates.

4.1 Allocation Hierarchies

In Figure 1 we define a minimal syntax for defining typestates of memory. The idea is that typestates 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:

```
Pages  -> # union { Region | ... }
Region -> seq { # Object, # words }
Object -> seq { Header, # Field }                                    (L1)
Header -> union { 2 words | Array -> 3 words }
Field    -> # words
```

This hierarchy defines 6 typestates: 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 typestates at some point during runtime (or implicitly in an UNMAPPED typestate). For instance, an allocated heap region with a single 10-word object allocated into it involves three typestates. The first 2 words of the region are in the Header typestate, 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.

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-arithmetic 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:

```
Object -> union {
      Free    -> # words
   | Alloc -> seq { Header, # words } }
Header -> union {                                                   (L2)
      Arr -> seq { MarkWord, KlassPtr, Len, Gap } // 3 words
   | Cls -> seq { MarkWord, KlassPtr } }               // 2 words
```

In this Code L2, we define the typestates of an Object and its Header. An object is either in the Free typestate, or it is in the Alloc typestate 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, depending on whether or not the object is an array with an array length (Len) field. The subfields MarkWord, KlassPtr, Len, and Gap can then be given concrete sizes, assuming a 64-bit architecture, like follows:

```
MarkWord -> 1 words
KlassPtr -> 1 words                                                 (L3)
Len        -> 4 bytes
Gap        -> 4 bytes
```

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-leve pointer update causes the object to no longer be accessible by the Java heap. While more precise, this definition would clearly require more complex code to track[1] 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 runtime. This boilerplate includes a consistent centralized naming scheme for managing varied hierarchies of allocable typestates.

To start, a preprocessing tool in our toolchain process a layout description, generating a series of named typestates, among other code snippets to reduce boilerplate during annotation of a memory management algorithm. From Code L2, we get out a series of typestates defined like follows:

[1]Such as auxiliary reference counting.

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

In this series of typestates in Code C1, TypeState is a class for unique identifiers, and any top-level defined <stmt> gets associated with it a nested hierarchy of addressable typestates. For instance, an addressable 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_Header_* *are* generated. The key is that one can choose which typestat contexts are useful to track based on how memory is managed.

One interesting distinction is, for example, the distinction between the typestates PCK_Object_Alloc_Header_Arr_MarkWord and PCK_MarkWord. Clearly, two different memory locations in either typestate could contain values that *look* identical, and in fact are semantically identical for two values of the same type. The difference then is in how the two different memory locations were allocated. The latter PCK_MarkWord is applicable to a top-level variable in the C++ implementation of the Hotspot VM, not located in the Java heap. In contrast the former is a fully-qualified mark word of an allocated array object header. This typestate is applicable to a piece of 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 useful distinction. The idea is that by default internal memory allocation context is all-but lost once an initialization is complete. With the two typestates however, we can now retain the context for validation or diagnosis purposes if and when a corruption occurs.

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

```
#define PCK_ALL_MarkWord PCK_MarkWord PCK_Object_Alloc_Header_Cls_MarkWord \
   PCK_Object_Alloc_Header_Arr_MarkWord PCK_Header_Arr_MarkWord \
   PCK_Header_Cls_MarkWord                                          (C2)
```

This macro is boilerplate for describing a fairly common idiom in memory managers. That is, some accessor function intended to access a mark word typically by default does not care which variant it accesses. By annotating code with this PCK_ALL_MarkWord typestate collection, we get the benefit of a coarse polymorphic typestate 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 typestate observed at runtime.

```
namespace pck {
class Map;
class TypeState; // Includes an identifying member field of u64_t
Map  create();
void destroy(Map m);
TypeState Map::get(Address a);
void         Map::put(Address a, TypeState ts); }
```

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

4.4 Typestate Tracking

The VM's interface to the typestate tracker is shown in Figure 2. Apart from its connection to the DBI engine, 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 typestates. The special UNMAPPED typestate is the default value for unaddressable memory, and UNMAPPED can be explicitly referenced as a typestate. The interface further uses array-access ove loading to provide the following map update and query syntax:

```
Address a; TypeState ts;
map[a] = ts;                                    (C3)
pck::assert(map[a] == ts)
```

In this Code C3, we have defined some address a and some typestate 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 typestate at address a and check that it matches ts. Note that the equality and map-access operations return proxy object types tracking what address was accessed, and the arguments to any comparisons. This proxy information allows pck::assert to report a useful error message in terms of the address, observed typestate, and expected typestate of the comparison, rather than simply reporting that the assertion failed.

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

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

```
namespace pck {
enum Perms { RW, R, W, NONE };
Map::protect(TypeState ts, Perms ps);
Map::unprotect(); }
```

Fig. 3. The VM-level interface to set memory access protections by typestate, 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 typestate. These previous protections are restored by a subsequent matching call to unprotect.

4.4.2 Code Generation. For some of the more verbose typestates, Permchecker's preprocessor generates helper macros. One such macro manages all the appropriate typestate updates for when a typestate with numerous nested components is to be allocated. For example, for tracking the nested typestates associated with an array header the preprocessor generates the following macro based on Code L2:

```
#define transition_Object_Alloc_Header_Arr (a) { \
  map[words(a, 1)] = PCK_Object_Alloc_Header_Arr_MarkWord; \            (C4)
  map[words(a + words(1), 1)] = PCK_Object_Alloc_Header_Arr_KlassPtr; \
  map[bytes(a + words(2), 4)] = PCK_Object_Alloc_Header_Arr_Len; \
  map[bytes(a + words(2) + bytes(4), 4)] = PCK_Object_Alloc_Header_Arr_Gap; }
```

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

4.5 Permission Tracking

Next, we need to be able to assign permissions to a piece of code in terms of the typestates the code is allowed to access. The primary mechanism the toolchain supports for this purpose is permission attributes on functions and classes in C++. The exact low-level mechanism to track this information involves intercepting specific function calls by Permchecker's DBI tool. In view of this mechanism, we conclude this section by discussing what Permchecker's typestate permissions can and cannot detect.

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-typestate 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 the specified permission to access any memory

```
__attribute__((pck_R (TypeState...)))
__attribute__((pck_W (TypeState...)))
__attribute__((pck_RW(TypeState...)))
__attribute__((pck_NONE(TypeState...)))
```

	Total	Mark & Klass
Lines added	1830	49
Map updates	144	25
Annotations	64	24
Typestates	60	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 typestates. 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: Breakdown of VM modifications.

address in the given typestate. A subsequent matching call to unprotect returns the permissions to what they were before. 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 two nested call frames for which the associated functions both have permission to access the same typestate in the same way. Such behavior is desirable when in one calling context the leaf call frame for a function will inherit access permission to a typestate, but in another calling context the same function's call frame will *not* inherit the necessary typestate. With a stateful unprotect call, the calling frame *with* permission over the typestate would incorrectly lose that permission if the leaf frame were to remove permission upon completion.

4.5.1 Function & Class Annotations. In Figure 4, on the left, we have a series of typestate permission annotations for functions and classes. On a function, an annotation compiles to calls to protect and unprotect in the prologue and epilogues of the function. On a class, the annotation gets distributed across functions in that class to the same effect. Based on this model, an annotated function calling some other function conservatively grants the callee the same permissions it has.

Also in Figure 4, on the right, is a breakdown of modifications made to the Hotspot VM to achieve permission checking of object header words. The modifications reported are the ones attributable to the memory management developer, with the Total column including modifications necessary to support tracking of the Java heap across the entire allocation hierarchy. These numbers represent a reasonable upper bound on the annotation burden of dealing with a small number of the most complex typestates in a memory manager.

The overall number of lines of code added involved modifications to the JIT compiler, assembler, argument parsing, imports, error reporting, foreign function interface, and write barrier. Apart from these tasks, 144 and 64 lines were respectively added to track typestates and to apply permissions to code. Of those typestate management lines of code, 49 dealt with just a variety of 16 typestates pertaining to the mark or klass word of an object. The remaining 44 of 60 typestates were added to provide useful allocation context for when the mark or klass word is accessed erroneously. Such typestates include allocated object fields, free regions, and reserved pages of memory, among others.

4.5.2 Low-Level Tracking Mechanism. In order to track typestate permissions, Permchecker's DBI tool relies on intercepting calls to the protect and unprotect functions. As a result, calls to these functions show up in the compiled version of the memory manager. This mechanism, however, has its own tradeoffs and possible sources of errors that we must consider. The key is that we want robust error detection in the presence of *any* incorrect code, be it in the bootstrapping compiler (Clang), the memory manager, or some other VM component with purview over memory layout. In the next few subsections we go on to discuss how Permchecker reports useful typestate mismatches in the presence of either a specification or an implementation error.

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

```
// Begin perms: stack (RW)
mov $0x21,%esi
mov $0x2,%edx
// Begin perms: stack (RW) & mark word (W)
call protect // Mangled name: _ZN3pck3Map7protectEyNS_5PermsE      (A1)
mov  -0x28(%rbp),%rax
movl  $0x1,(%rax) // Write to a mark word
call unprotect // Mangled name: _ZN3pck3Map9unprotectEv
// Ending perms: stack (RW)
```

Code A1 executes as follows. We store immediate values for a mark word's typestate (0x21) into register esi and for write access permission (0x2) into edx. On line 3 we call protect to register this permission for the current thread. Next on line 4, we load an address to an object off the stack and into register rax. Finally on line 5, we set four bytes of the object header to 0x1 to indicate it is a regular unlocked object. After this, a call to unprotect reverts the executing thread's permissions to what they were at lines 1 and 2. The idea is that if line 5 writes to 4 bytes in a valid mark word typestate then the program did not have a memory error, resulting in a true negative check by Permchecker.

4.5.4 True Positives. Now consider a bug where the mark word annotation is in the wrong place, or where the C++ compiler itself erroneously reorders the memory access to the object's mark word. Therefore the memory access occurs outside the desired scope to our protect call, like follows:

```
// Begin perms: stack (RW)
mov $0x21,%esi
mov $0x2,%edx
call protect // Begin perms: stack (RW) & mark words (W)      (A2)
call unprotect // Begin perms: stack (RW)
mov -0x28(%rbp),%rax
movl $0x1,(%rax) // Violation!
```

In this Code A2 Permchecker will report an access violation because line 6 here did *not* have permission to write to the mark-word even though line 6 is supposed to be able to do so. The key is that Permchecker does not make a claim of truth as to which permission was correct: the code or the memory. By doing so, Permchecker can usefully report localized memory errors without needing to assume *any* code is necessarily correct. Thus, not only is a bug in the implementation reported as a typestate mismatch, but a specification (or compiler) bug is reported as such too.

4.5.5 Benign Checks. In the two snippets of assembly code above, we in fact accessed two different kinds of memory: a mark word, and a pointer to that mark word found on the stack. For the true-negative, both memory accesses occur within the scope of our call to protect, but we only needed the mark word access to be checked. In checking both, we made a tradeoff in how Permchecker was

```
// lo addr --> |-----------------|                 // |-----------------| <-- lo addr
//                 . fields...            .        // . fields...                .
// [ k words ] | (prev obj)               |        // | (prev obj)             | [ k words ]
// raw ptr --> |-----------------|                 // |-----------------| <-- raw & obj
//     [ 1 word ] | fwding pointer        |        // | mark/fwd word          | [ 1 word ]
// obj ptr --> |-----------------|                 // |-----------------|
//     [ 1 word ] | mark word             |        // | klass ptr              | [ 1 word ]
//                 |-----------------|             // |-----------------|
//     [ 1 word ] | klass ptr             |        // | fields...              | [ n words ]
//                 |-----------------|             // | (variable size) |
// [ n words ] | fields...                |        // .                          .
//                 | (variable size) |             // .                          .
//                 .                      .        // |-----------------| <-- obj end
//                 .                      .        // | Unallocated due | [ 1 word ]
// hi addr --> |-----------------|                 // | to elimination         | <-- hi addr
```

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

designed. The idea is that it is often sufficient to check a memory access against a set of possibly unrelated typestates when doing so simplifies our permission specification.

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

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

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

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 instruction[2] 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

[2]Requires debugging symbols to get source locations.

```
[error][pck] /jdk13/src/.../oop.inline.hpp:123
  Permchecker Violation in Thread #2:
    Expected typestates: PCK_MarkWord, ...
    Observed typestate: PCK_Object_Header_Alloc_KlassPtr
    Address = 0x00007ffb1c43e008
    Address is preceded by 1 words of typestate PCK_Object_Header_Alloc_MarkWord
    Address starts 1 words of typestate PCK_Object_Header_Alloc_KlassPtr
    This is followed by 6 words of typestate PCK_Object_Alloc
    Dumping typestates to /memdbg/permchecker.3952.log
    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 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 want a way to check the validity of each memory access. The goal is to ensure that for each address in some typestate, the only load and store instructions operating over the address occur when the code has permission to access it. In this section we present how we achieve this goal by discussing 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 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 typestate are checked for access permissions. In this mode, a program with no updates to the typestate map will never produce a violation because all memory locations remain in the UNMAPPED typestate. 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 typestate. Memory accesses to locations with unknown typestates are ignored.

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.	JIT compiler	Typestate checking of runtime generated code.
Checker	DBI (Pin tool)	Applies a simple and rational checking model across every single memory access.
Shadow memory	DBI library	Maintains a snapshot of the typestate map.

Fig. 7. Each feature Permchecker touches, what system component it involves, and the feature's purpose with respect to typestate.

4.6.3 Heavyweight Debugging. In the heavyweight mode, Permchecker reports an impermissible memory access for all typestates, including UNMAPPED, as an error. In this mode, a program with n updates to the typestate 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 typestates have been annotated i the memory manager, thus increasing system-wide sensitivity. While the lightweight mode allows the developer to build the lifecycle of an individual typestate, the heavyweight mode discovers all buggy, unspecified, or improperly specified memory accesses.

4.6.4 Thread Permissions. In Permchecker's DBI engine, a per-thread permission tracker is implemented as two extensible bit-vectors representing read and write permissions each with one bit per typestate. When a bit in the vector is set, the associated thread has that kind of (read or write) permission to the typestate 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 typestates 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 typestates 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.

5 DEBUGGING TOOLCHAIN ARCHITECTURE

Different components of a VM have access to widely varying and fundamentally different mechanisms to support typestate tracking. A simple numeric typestate tracking system is a modest

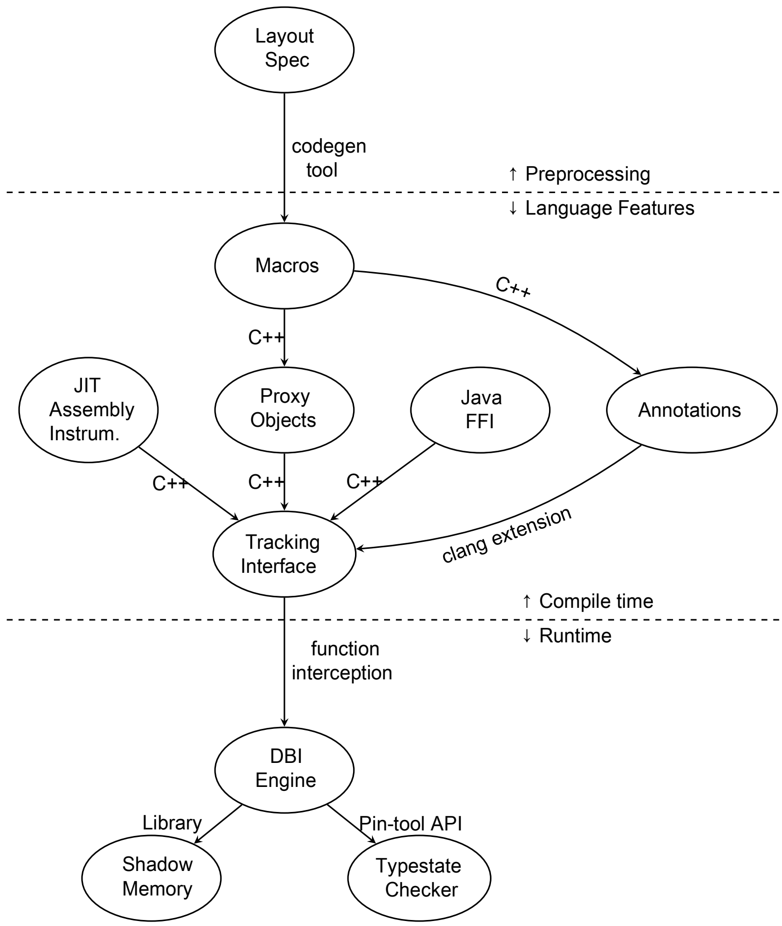


Fig. 8. Architecture of the Permchecker toolchain, indicating how typestate tracking and permissions are implemented. Nodes are artifacts in the system which involve typestate, and arrows are mechanisms for enacting their purpose.

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 Figure 7 in order to manage and track typestates.

Artifacts & Mechanisms. In Figure 8 we show how typestate information flows through the toolchain at different levels of abstraction. For instance, the DBI engine intercepts function calls at runtime to the typestate tracking interface. The DBI engine 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.

```
oop p = ...;
if (DoInjectCorruption && InjectID == 1 && !__CORRUPTION) {
  if (__CORRUPTION_SIZE > 0 && p->klass() == first_kls) { // 3rd case
    HeapWord *src = (HeapWord*) p;
    HeapWord *dst = first_forwardee;
    Copy::aligned_conjoint_words(src, dst, size);
    oop(dst)->init_mark_raw();
    log_info(permchecker)("Corruption #1");
    __CORRUPTION = 1;
  } else if (__CORRUPTION_SIZE == 0 && // 2nd case
      __CORRUPTION_COUNTDOWN == 0 && p->is_forwarded()) {
    __CORRUPTION_SIZE = (size_t)p->size();
    first = (HeapWord*) p;
    first_kls = p->klass();
    first_forwardee = (HeapWord*)p->forwardee();
  } else if (__CORRUPTION_COUNTDOWN > 0) { // 1st case
    __CORRUPTION_COUNTDOWN--; } }
```

Fig. 9. 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.

6 BUG INJECTION BENCHMARKS

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 9. 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. 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 typestate 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

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
Premature Free	976	649	562	31	2	5	376

Fig. 10. 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.

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 typestates match the usage of the values being read.

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 typestate 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 typestates. 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 typestate, transition it to the allocated typestates, and then perform a copy operation with permission to write to the allocated typestates. Or two, he can specify the copy operation has permission to write to the free typestate, and then transition the memory to the allocated typestates after copying.

The latter technique is preferable because it is less error-prone: the developer cannot forget to insert a typestate 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 typestate 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 DBI tool than to use a technique that relies on the developer to check a typestate 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.

6.2 Results

In Figure 10 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. We consider a bug template when tested in a specific system configuration, to represent 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.
- We get a large number of possible injections because the number of dynamic operations performed by the garbage collector is effectively 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 Figure 10 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 9, 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.

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

6.2.1 Choice of User-Level Application. One particular class of memory management bugs we wish to study are ones which do not crash the program at all, but instead affect the correctness of the user-level application such as its output. The incidence of output-based errors relies heavily on the chosen domain and implementation of the user-level application. Application characteristics affecting whether or not an output error will arise include ones like object allocation and death patterns, mean amount of drag time between last use of an object and when the object becomes unreachable, overall fragility, and more. For the results in Figure 10, we modified a version of GCBench [Ellis et al. 2014] so as to maximize fragility by minimizing drag.

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

In contrast our modified version of GCBench maximizes fragility. It does this by computing a hash over the contents of each node in a binary tree shortly before we expect the tree to become garbage. This hash value iteratively accumulates through every node allocated by the application, and then prints the hash before the application terminates. This strategy aggressively links the correctness of the application's output to the presence of heap corruption. We further believe this strategy is ripe for automation over any existing application in other domains than just performance analysis.

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

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

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

6.2.3 Observable Behaviors. In Figure 10 we also break down the frequencies with which each bug template caused a variety of observable behaviors in the absence of Permchecker. In debugging practice, these behaviors are a holistic collection of failure modes that a developer would have to reason about when confronted with them. The following list gives descriptions for how we defined each of these failure modes:

- Ad-hoc: an error detected and reported by the VM itself, usually from a suspicious looking pointer or other unexpected value.
- OS Error: a segmentation fault, bus error, or other OS-level error even if the VM intercepts it to provide more information.
- OOM: an out-of-memory error reported by the VM. This could be caused by a corrupted allocation loop that never terminates.
- Output: the program successfully terminates with textually incorrect output from the user-level application.
- Benign: the program successfully terminates with the correct user-level textual output, regardless of how long it took to run.
- Live/deadlock: corruption that leads to a non-terminating program. This was not observed in our testing.

Notably, we consider a "benign" behavior a failure mode in the known presence of a (injected) bug. This is because a program run observed to be benign does not necessarily preclude memory corruption from having occurred. It only indicates that the bug was benign for the chosen execution trace. In fact, all the benign bugs from Figure 10 can rightly be considered failure modes of the system to exhibit more obvious signs of corruption or typestate mismatch. The "Wasted Memory" benchmark, too, corrupts the layout of memory, albeit only ever benignly in a way that our Permchecker annotations were unable to detect because no read or write instructions access "extra" memory.

6.2.4 Utility of Error Reports. In addition to a tool with high sensitivity and specificity in reporting the presence or lack of an error, it is also desirable for the contents of the error report to be useful. Subjectively, a useful error report is one which identifies information pertinent to the source of the error. With typestate checking, a simple first-order metric is whether or not either the permissible or the observed typestates, of a Permchecker error report, indicates the memory (or layout) which was actually corrupted.

In all of the Permchecker error reports pertaining to Figure 10, the error report included the pertinent typestate for each of the typestates. For instance in the "Overflow (padding)" template, Permchecker always reported an error where the program had permission to access an object Field, but in fact accessed some kind of Padding typestate. Similarly with the "Overflow (no pad)" template, Permchecker always reported an error involving permission to access a MarkWord but observed a Field access, or vice-versa. The takeaway is that typestate checking did not only improve sensitivity over the existing ad-hoc error checks. Checking also exhibits improved diagnostic usefulness by reporting an observed type, in-lieu of just an expected type implied by the stack trace of an ad-hoc check.

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

For the bug templates we chose, this formulation was entirely not the case. In all the injections from Figure 10, Permchecker detects an error accounted for in the third column prior to any corresponding error accounted for in the fourth (ad-hoc) column. Methodologically this ordering is determined by running the VM with *both* forms of checks enabled, but where the program simply logs the error reports rather than terminating immediately. This methodology has the benefit of obviating any need to create strictly reproducible thread schedules for the VM. Furthermore it is valid to compare program behaviors this way because program checkers have ostensibly no side-effects or impact on values in the VM. The tradeoff is that this methodology cannot compare one-to-one results of Permchecker with any similarly purposed, but non-composable with Permchecker, DBI tools.

7 SCOPE, LIMITATIONS, & FUTURE WORK

Overhead. Permchecker's DBI tool incurred a 10x to 50x slowdown in the tests reported 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 where thread-safety became a non-trivial concern was in the implementation of shadow memory. As such we implemented shadow memory as a lock-free tree-like map data structure in which map updates and map allocations are atomic. In the presence of a data race involving an update and a read access to the typestate of the same memory location, the shadow memory provides no guarantee as to which typestate the read access observes: the old one or the new one.

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

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

REFERENCES

Karl Cronburg and Samuel Z. Guyer. 2019. Floorplan: Spatial Layout in Memory Management Systems. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Athens, Greece) *(GPCE 2019)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3357765.3359519

B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Jose, CA, 110–121. https://doi.org/10.1109/SP.2016.15

John Ellis, Pete Kovac, and Hans Boehm. 2014. https://hboehm.info/gc/gc_bench/ Accessed: 2021-04-13.

Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD.

IBM. 2005. Jikes RVM. http://www.jikesrvm.org/ Accessed: 2018-09-28.

Doug Lea. 1991. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html Accessed: 2021-04-13.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

Oracle. 2006. OpenJDK Hotspot Division. http://openjdk.java.net/groups/hotspot/ Accessed: 2021-04-13.

Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting Argument Selection Defects. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 104 (Oct. 2017), 22 pages. https://doi.org/10.1145/3133928

Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/3236024.3236084

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address 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

R. E. Strom and S. Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171.

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and A. Ahmed. 2019. Oxide: The Essence of Rust. *ArXiv* abs/1903.00982 (2019).