Detecting Unread Memory Using Dynamic Binary Translation

Jon Eyolfson and Patrick Lam
University of Waterloo

Abstract. Reading from uninitialized memory—that is, reading from memory before it has been written to—is a well-known memory usage error, and many static and dynamic tools verify that programs always write to memory before reading it. This work investigates the converse behaviour—writes that never get read, which we call “unread writes”. Such writes are redundant—at best, they do not perform any useful work; furthermore, work done to compute the values to be written could corrupt the program state or cause a crash. We present a novel dynamic analysis, implemented on top of the Pin dynamic binary translation framework, which detects instances of unread writes at runtime. We have implemented our analysis and present experimental data about the prevalence of unread writes in a set of benchmark applications.

1 Introduction

Modern languages and compilers detect memory usage errors caused by reads from uninitialized memory: in Java, it is an error to read variable x before writing a value to it, and gcc warns about uses of uninitialized variables. Programs also contain the converse phenomenon: writes to memory which are never read. Such writes are redundant; at best, they don’t perform any useful work. Computations that produce values used only in unread writes do not contribute to the goal of the program, gratuitously consume computational and memory resources, and may, in the worst case, crash the program—for example, the Ariane 5 crash was caused by an exception while computing an unused value.

Because compilers detect memory problems ahead of time, most compilers only report errors and warnings at an intraprocedural level, and only for local variables and private fields of classes. (gcc 4.6, for instance, reports warnings for unused but set variables.) Static approaches to memory error detection require detailed pointer information to detect memory errors on heap accesses: the compiler needs to know which heap references may and must alias, so that it can determine the access history of individual abstract memory locations. Must-alias analysis is critical for reducing the rate of false positives. However, implementations of whole-program must-alias analyses are rare.

Recently, Valgrind’s Memcheck tool has used dynamic binary translation to detect memory errors, including reads from uninitialized memory, at runtime.

1 Section 2.1, http://www.di.unito.it/~damiani/ariane5rep.html
Purify [2] detects a similar class of errors by inserting instrumentation code at compile time. In either case, runtime verification can ensure the absence of memory errors on an observed execution. Dynamic analyses need not reason about the heap, as a pointer comparison suffices to disambiguate heap addresses.

Our Tracerory tool implements a dynamic analysis to detect unread memory in realistic C and C++ applications. It supports multithreaded programs. We detect 1) *unread memory allocations* and 2) *unread writes* to the heap—writes with no corresponding read. When a developer runs their code under Tracerory, it reports instances of unread memory. Developers can use the report to manually inspect flagged program points and fix their code.

Figure 1 shows a high-level overview of our tool’s operation. Tracerory takes two inputs: an executable to be monitored, and specifications about which parts of the program to monitor. While Tracerory executes the program, its runtime monitor processes the stream of memory allocations, reads, and writes, reporting unread writes and memory allocations.

The contributions of this paper are:

– the identification of unread memory as a source-level phenomenon of interest;
– a novel dynamic analysis to detect unread memory in programs at runtime;
– an implementation of our dynamic analysis in the Pin dynamic binary translation framework; and
– qualitative and quantitative results outlining the prevalence of unread memory in a collection of open-source benchmarks.

## 2 Overview

This section presents, using an example, the two suspicious memory usage patterns that our Tracerory dynamic monitoring tool detects. Section 5 presents additional instances of unread writes drawn from real-world programs.