Minemu: The World’s Fastest Taint Tracker

Erik Bosman, Asia Slowinska, and Herbert Bos
Vrije Universiteit Amsterdam

Abstract. Dynamic taint analysis is a powerful technique to detect memory corruption attacks. However, with typical overheads of an order of magnitude, current implementations are not suitable for most production systems. The research question we address in this paper is whether the slow-down is a fundamental speed barrier, or an artifact of bolting information flow tracking on emulators really not designed for it. In other words, we designed a new type of emulator from scratch with the goal of removing superfluous instructions to propagate taint. The results are very promising. The emulator, known as Minemu, incurs a slowdown of 1.5x-3x for real and complex applications and 2.4x for SPEC INT2006, while tracking taint at byte level granularity. Minemu’s performance is significantly better than that of existing systems, despite the fact that we have not applied some of their optimizations yet. We believe that the new design may be suitable for certain classes of applications in production systems.

Keywords: dynamic taint tracking, JIT compilation, intrusion detection.

1 Introduction

Fifteen years after Aleph One’s introduction to memory corruption [17], and despite a plethora of counter-measures (like ASLR [3], PaX/DEP [18], and canaries [7]), buffer overflows alone rank third in the CWE SANS top 25 most dangerous software errors

Dynamic taint analysis (DTA) [16,6] is very effective at stopping most memory corruption attacks that divert a program’s control flow. Moreover, the wealth of information it collects about untrusted data makes it well-suited for forensics and signature generation [26]. Unfortunately, software DTA is so slow that in practice its use is limited to non-production machines like honeypots or malware analysis engines.

In this paper, we describe Minemu, a new emulator architecture that speeds up dynamic taint analysis by an order of magnitude compared to well-known taint systems like taint-check [16], Vigilante [6], and Argos [20]. Specifically, Minemu brings down the slowdown due to taint analysis to 1.5x-3x for real applications. Unless your application really starves for performance, a slowdown of, say, 2x to be safe from most memory corruption attacks might be a reasonable price for many security-sensitive systems.

Current counter measures do not stop memory corruption. Typical memory corruption attacks overwrite a critical value in memory to divert a program’s flow of control to code injected or selected by the attacker. We argue that current protection mechanisms (like PAX/DEP, ASLR, and canaries) are insufficient. Consider for instance, the buffer

underrun vulnerability in Figure 1. The example is from a Web server request parsing procedure in nginx-0.6.32 [1]—in terms of market share across the million busiest sites, the third largest Web server in the world [2], hosting about 23 million domains worldwide at the time of writing. The buffer underrun allows attackers to execute arbitrary programs on the system. They do not trample over canaries. They do not execute code in the data segment. Since they call into libc, they are not stopped by ASLR either.

In reality, the situation is worse. All defense mechanisms used in practice, including the three above, have weaknesses that allow attackers to circumvent them, and/or situations in which they cannot be applied (e.g., JIT code requires data pages to be executable). Moreover, a recent report indicates that many programs either do not use features like DEP or ASLR at all, or use them incorrectly [25]. Finally, legacy binaries often cannot even be protected using such measures.

**Dynamic Taint Analysis.** (DTA) is one of the few techniques that protect legacy binaries against all memory corruption attacks on control data. Because of its accuracy, the technique is very popular in the systems and security community—witness a string of publications in the last few years in tier-1 venues, including SOSP [6], CCS [30], NDSS [16], ISCA [9], MICRO [8], EUROSY [20], ASPLOS [28], USENIX [5, 12], USENIX Security [29], Security & Privacy [24], and OSDI [13]—it is clearly well liked.

Frustratingly though, DTA is too slow to be used in production systems. In practice, its use is limited to non-production machines like honeypots or malware analysis engines. With slow-downs that often exceed an order of magnitude, few are keen to apply taint analysis to, say, their webserver or browser.

**Contributions.** The research question we address in this paper is whether the slowdown is a fundamental performance barrier, or an artifact of bolting information flow tracking on emulators not designed for it? To answer this question, we designed a new emulator architecture for the x86 architecture from scratch—with the sole purpose of minimizing the instructions needed to propagate taint. The emulator, Minemu, reduces the slowdown of DTA in most real applications to a factor of 1.5 to 3. It is significantly faster than existing solutions, even though we have not applied some of their most significant optimizations yet. We believe that the new design may be suitable for certain classes of applications in production systems.

Specifically, what we did not do is rely on static analysis. In principle, it is possible to improve performance by means of statically analyzing the program to determine which instructions need taint tracking and which do not. Unfortunately, static analysis and even static disassembly of stripped binaries is still an unsolved problem. Therefore, the authors of the best-known work in this category [23], assume the presence of at least some symbolic information (like the entry points of functions). In practice, this is typically not available. In fact, we do not even check at (dynamic) translation time whether the data is tainted (whether we could follow a fast path) as proposed by the authors of LIFT [22]. In LIFT terminology, Minemu always takes the slow path. As a result, Minemu’s performance is independent of the amount of taint in the inputs.

---