Chapter 7

Summary of Memory-Safety Enforcer

7.1 Performance and Coverage

Figure 7.1 gives the runtime overhead of MSE using EPT, EPT plus redundant checks analysis, and EPT plus both redundant checks analysis and pointer-range analysis. The average slowdown with all analyses is 170% when checking reads and writes (a 17% improvement over EPT), and 43.7% when checking writes only (a 19% improvement over EPT). Note that Redundant Checks Analysis and Pointer-Range Analysis are orthogonal, and often give complementary improvements (for example, in comparing Figures 7.1 and 6.9, bh shows a big improvement from pointer-range analysis but not redundant checks analysis, while ammp shows a big improvement from redundant checks analysis but not pointer-range analysis).

The improvements due to redundant checks analysis and pointer-range analysis are significant but not overwhelming, and must be weighed against the analysis complexity. Figure 7.2 gives the analysis times of EPT, redundant checks analysis, and pointer-range analysis, as a multiple of the compilation time of the uninstrumented program. EPT is based on a fast flow-insensitive points-to analysis that scales well to the larger programs, while the two flow-sensitive analyses are noticeably slower for the large benchmarks. The analysis of mesa is slow because the program has many large structures (with over a thousand fields) which our implementation currently does not handle efficiently.

Another metric that may be of interest is tracked coverage, or the percentage of static locations (variables, malloc objects, and strlit objects) that were classified as tracked. With the naive classification, the tracked coverage is 100%, because all user locations are
Figure 7.1 Pointer-Range Analysis: Runtime Overhead
tracked. With EPT and Range Analysis, the tracked coverage is given in Figure 7.3: when checking reads and writes, the average coverage is 16.4% with EPT, and 13.6% with EPT and Range Analysis, and when checking writes only, the average is 9.7% with EPT and 7.6% with EPT and Range Analysis. The tracked coverage for tile is high in read-write checking mode because the program declares a large number of string literals, most of which are classified as tracked. Improvements in tracked coverage from range analysis arise if the analysis can guarantee that all accesses to an array (including via pointers) are in-bounds, and classify them as untracked. Observe that the coverage for write-only checking is lower than for read-write checking, because certain locations are only read but not written via checked dereferences. Thus, checking only writes actually increases the likelihood of detecting an invalid access, though it would fail to detect any invalid reads.

In the context of a security tool, a better measure of coverage is how many “sensitive” locations are tracked. Of the control-sensitive locations listed in Chapter 2, only function pointers, longjmp buffers, and exec/system call arguments are part of user memory: if any of these are classified as tracked, then an invalid write into these locations would not be

---

1Recall that redundant checks analysis does not reduce the size of tracked-locs(P), so there is no change in tracked coverage when using redundant checks analysis.
Figure 7.3 Static Tracked Coverage
detected. For all other sensitive locations, any attempt to overwrite them via a checked dereference would be detected.

Of the 151 function pointers in the benchmarks, only two are classified as tracked: these two (both in vortex) are tracked because they are assigned a value via a call to the library function `memcpy`, which includes a write via an unsafe dereference. Of the 11 `longjmp` buffers in the benchmarks, 9 are classified as tracked. Six of these (in li) are classified as tracked due to imprecise handling of structures by the points-to analysis, two (in perl) are arrays of `longjmp` buffers that are accessed via unsafe dereferences, and one (in gcc) was assigned via a call to `bcopy`, which includes a write via an unsafe dereference. Of the seven arguments to `exec` and `system` calls (in perl, m88ksim, gcc, and gap), five were classified as tracked: these were stored in arrays and manipulated by unsafe dereferences in string or array operations.

### 7.2 Comparison with Other Tools

Figure 7.4 gives a comparison of the execution-time slowdown of the MSE to two related tools, CCured (version 1.2.3) and Cyclone (version 0.6), for the Cyclone benchmarks. All were compiled with -O3 optimization, and executed on the same machine on the same inputs.

![Figure 7.4 Runtime Overhead Comparison](image)

CCured [Nec+02, Con+03] and Cyclone [Jim+02] are two variations of the C language that enforce memory safety (for both reads and writes) at runtime. They both make use of
fat pointers to detect spatial access errors; to prevent temporal access errors, they restrict the memory management to the use of a garbage collector (for CCured) or a region-based memory manager (for Cyclone). CCured, like the MSE, uses static analysis to classify pointers into categories of safety, instrumenting only pointers for which memory safety cannot be statically verified; additionally, user-supplied annotations can be used to improve the static analysis results. Cyclone also has different types of pointers for which different amounts of runtime checks are instrumented, but the type of each pointer is specified by the programmer. In both cases, the use of fat pointers results in a more restrictive memory-safety model; e.g., dereferencing a pointer that has been cast from an integer is not allowed.

The average slowdown for our tool on these benchmarks is 192.5% when checking reads and writes, and 52.2% when checking only writes (note that our worst-performing benchmark for checking only writes, grobner, is included in this test set). For CCured, the average slowdown (363.8%) is much higher than the average slowdowns reported for other test cases in their papers [Nec+02, Con+03]; the CCured team verified the poor performance for cacm and matxmult on the unmodified source files, but were able to significantly improve the performance of cacm with some (potentially-unsound) programmer-added annotations [Nec04]. For Cyclone (average slowdown 56.6%), the source code had been translated (manually) to Cyclone, so human intervention played a part in identifying pointers that should be made fat pointers.

Although these test cases are far from comprehensive, they demonstrate that our approach can be competitive with or faster than tools like CCured and Cyclone while maintaining the low-level control of C and without requiring programmer changes to source code.

7.3 Effectiveness of the MSE

7.3.1 Fault-Injection Study

One potential weakness of the MSE compared to a fat-pointer approach (like CCured) is that the MSE is not guaranteed to detect all memory-safety errors. If an out-of-bounds
dereference happens to access another tracked location, the dereference would not be recognized by the MSE as an invalid access. To get an idea of the extent to which this may be a problem in practice, we conducted a simple fault-injection study.

For each Cyclone benchmark, we identified all array declarations and malloc calls. Our plan was to create, for each array declaration or malloc call, a variant of the program in which the size of the array or malloc’d block was decreased by one; e.g., for the declaration `int a[SIZE]`, a variant would be created in which the declaration of `a` was replaced by `int a[SIZE-1]`, and for the call `malloc(e)`, a variant would be created in which the call was replaced by `malloc(e-1)`. The idea behind each of these changes was to trigger an off-by-one error. We would then instrument each variant with CCured to determine the number of cases in which a memory-access error was triggered (at runtime, on our test inputs). Since the fat-pointer approach is guaranteed to detect spatial access errors, this number could be used as a baseline against which the number of error cases detected by the MSE could be compared. However, it turned out that CCured’s dynamic memory allocator increases the size of each allocated block to the next word boundary — a practice that enables efficient bounds checking — so we altered the fault-injection methodology slightly to account for this, by decreasing the size of each mutated malloc argument by one word rather than one byte.

Figure 7.5 summarizes the test cases: column (a) lists the number of variants created and tested for each program, while column (b) shows the number of variants for which there was an invalid access (as detected by CCured).

When instrumented with the MSE, all of the error cases in column (b) were detected. This was true in both read-write checking mode and write-only checking mode, and when using either the naive classification scheme, the extended points-to analysis, or the full suite of static analyses (extended points-to, redundant checks, pointer range). This experiment

---

2For aes, CCured reported two additional errors which were not memory-safety violations. In both cases, a pointer to an array of \( N-1 \)-element arrays is passed to a function that expected a pointer to an array of \( N \)-element arrays; this was disallowed by CCured, even though the misaligned array declarations did not result in a subsequent invalid access.
Figure 7.5 Fault Injection Error Cases

would tend to suggest that, in practice, the likelihood of an out-of-bounds dereference accessing another tracked location is quite low, and that, in practice, our approach is as effective as the fat-pointer approaches in detecting memory-safety errors.

7.3.2 Finding Bugs

While the MSE was designed primarily as a security tool, it can also be used for debugging. When instrumented to check both reads and writes, a number of invalid read accesses were discovered in several of the SPEC benchmarks (compress, gcc, go, ijpeg, parser, vortex).\(^3\) The gcc bug reads via a dangling pointer dereference (pointing to a local variable in an expired scope); the rest of the errors are out-of-bounds array accesses. Though these bugs do not appear to be vulnerable to malicious attacks, this shows that our approach can also be used to detect bugs, in a spirit similar to Purify [Has+92], but with a lower runtime overhead.

7.3.3 Detecting Attacks

To demonstrate the efficacy of our tool, we instrumented two Linux (RedHat 6.2) programs that have known vulnerabilities and exploits. In both cases, our instrumented program detected the invalid write during a run where we attempted to perform the exploit, and halted

\(^3\)The bugs in compress, go, ijpeg were also reported in the first CCured paper [Nec+02].
the program before the exploit was able to gain control. Exploits were obtained from the Packet Storm website [Pac].

The first vulnerable program we tested is **traceroute**, the vulnerability and exploit for which were described in Section 2.3.

The second program, **cfingerd**, is a configurable **finger** daemon that allows each user to turn on or off the ability for others to look up information about them. A user can supply a generic message in a file `.nofinger` in their home directory that will be displayed by **cfingerd**. Version 1.4.2 of **cfingerd** has a buffer-overflow bug in the function that processes the data in the `.nofinger` file: the data is read into an 80-byte buffer with no bounds checking. An attacker can thus put a string longer than 80 characters in their `.nofinger` file to overwrite the function’s return address. The exploit we tested put the shellcode within the first 80 bytes, then padded the string with a suitable number of bytes beyond 80, followed by the address of the shellcode at a position determined (by code inspection and experimentation) to coincide with the return address in the activation record. Since by default the **cfingerd** daemon is executed as root, this exploit gives the attacker root privileges on the system.

We also instrumented the 20 test cases developed by Wilander and Kamkar to evaluate dynamic buffer-overflow prevention tools [Wil+03]. These test cases simulate the range of possible attacks that

1. overwrite one of the following attack targets: a return address on the activation record, an “old base pointer” on the activation record, a function pointer, or a **longjmp** buffer;

2. either use a contiguous buffer overflow to write to the attack target, or use the overflow to write a pointer value to allow a re-directed write directly to the attack target;

3. overflow either a buffer on the stack or a buffer in the heap/BSS/data segment.

With the unoptimized MSE, all the attacks except those that write directly to a function pointer or a **longjmp** buffer were detected — this is as expected, since the unoptimized MSE
classifies all user locations (including function pointers and longjmp buffers) as tracked. With the EPT analysis, all 20 attacks were detected.

7.4 Conclusion

We have described the Memory-Safety Enforcer, which instruments programs to detect invalid pointer dereferences at runtime. The tagged-memory approach, which tags each byte of memory with one bit to indicate whether the byte is a valid target of a checked dereference, allows runtime checking to be efficient without restricting the flexibility of the C language or reporting false positives. While the approach is not guaranteed to prevent all invalid accesses, it will detect a large class of known security attacks, including stack smashing and the more subtle multiple-free exploit. Further, the likelihood of detecting an invalid access can be improved with better static analysis.

Three static analyses, based on Points-To Analysis, Redundant Checks Analysis, and Pointer-Range Analysis, have been described, and have been demonstrated to be effective at improving both runtime overhead and the likelihood of detecting an error. The runtime overhead of 43.7% for checking writes only is better than the performance of tools with similar goals, and should be low enough for the approach to be used in deployed software.