



# FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers

Yves Younan, Senior Research Engineer

Talos Security Intelligence and Research Group, Cisco

# Overview

---

- Introduction
- FreeSentry approach
- Evaluation
- Conclusion

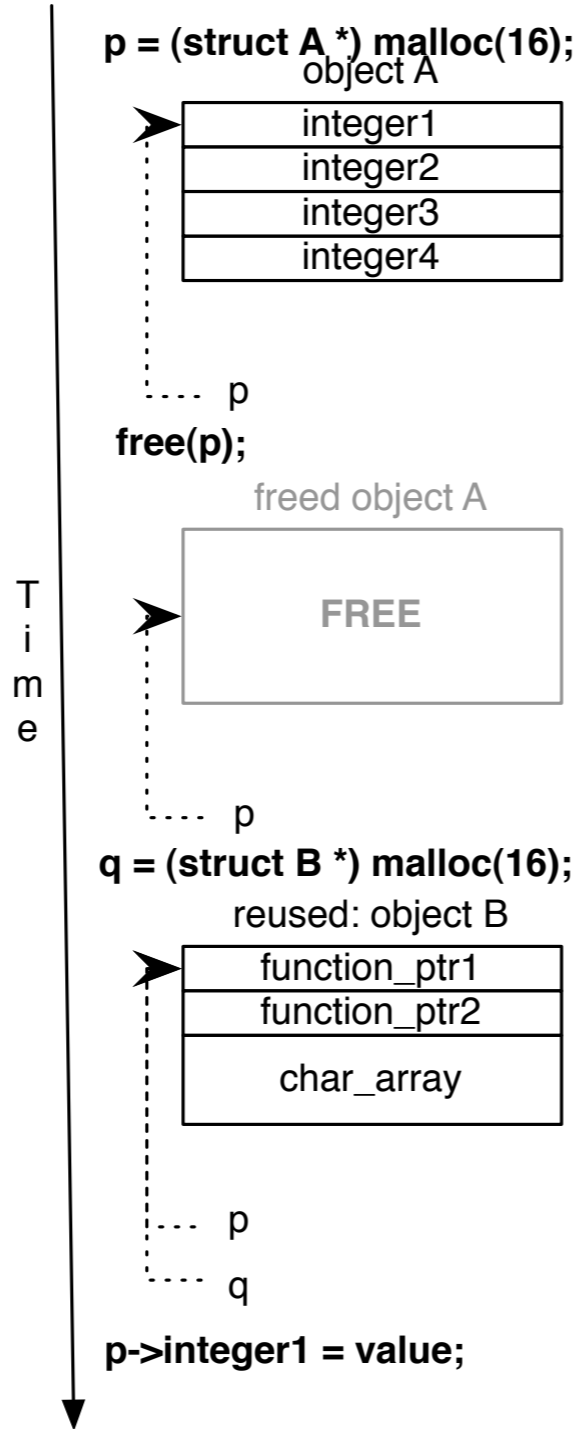


# Introduction

- Programs in C/C++: memory error vulnerabilities
- Mitigations make exploitation harder or sometimes impossible
- Use-after-free vulnerabilities are now some of the most important vulnerabilities
- Most exploited vulnerability for Windows Vista and 7.
- Most common memory error in Internet Explorer
- Use-after-free occurs when a pointer exists to memory that has been freed and is accessed



# Use after free



# Approach

- Link objects back to pointers
  - When the object is freed, invalidate pointers
- Pointer creation or modification
  - Register address of pointer as referring to object
  - Memory allocation
    - Assign labels to object, to reflect object bounds
- When object is freed
  - Check if pointer is still pointing to the object
    - If so, invalidate pointer
      - If the program dereferences the pointer, it crashes

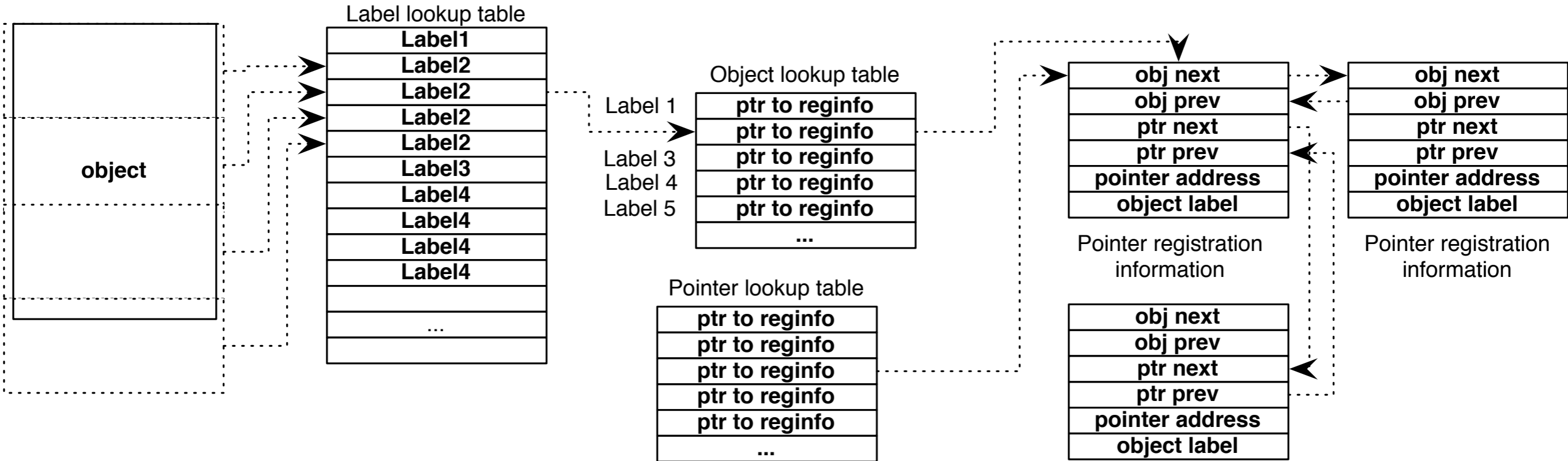


# Approach

- Does not change pointer representation
- This means it works with unprotected code
- Allows developers to opt-out of functions:
  - malloced/freed memory will still be tracked/invalidated
  - pointers in functions that have opted-out are not tracked
  - allows developers to improve performance by opting out if a function is guaranteed to be safe



# Memory layout



# Approach: basic example

Original:

```
int main() {
    char *a, *b; int i;
    a = malloc(20);
    b = a+5;
    free(a);
    b[2] = 'c';

    a = malloc(20);
    for (i = 0; i<5; i++)
        b = a+i;
    free(a);
    *b = 'c';
}
```

Freesentried

```
int main() {
    char *a, *b; int i;
    a = malloc(20);
    regptr(&a);
    b = a+5;
    regptr(&b);
    free(a);
    b[2] = 'c';

    a = malloc(20);
    regptr(&a);
    for (i = 0; i<5; i++)
        b = a+i;
    regptr(&b);
    free(a);
    *b = 'c';
}
```





# Freeing memory

- When memory is freed
  - Lookup the label of the object, use that label to index the object lookup table, retrieve the ptr reg info
  - Contains pointer address and the stored object label
  - If the stored label == label of the object being freed,
    - Check if the pointer still points to the object
    - If so, invalidate the pointer, continue with next pointer



# Reallocating memory

- Realloc can return different memory from the memory being modified
- Any call to realloc should invalidate all pointers
  - This will break many programs
    - Turns out most programs are not well written
  - Instead, we only invalidate when the memory has moved
  - If used as a testing tool, behavior should be set to always invalidate

# Pointer arithmetic

- Simply increasing or decreasing is ignored as we assume objects remain in bounds
  - If a pointer points out-of-bounds, it will not be invalidated:
    - Unprotected code could have modified the pointer to point to an adjacent object
    - Invalidating would break the program

# Pointer arithmetic

```
char main() {  
    char *a, *b;  
    int difference;  
    a = malloc(100);  
    b = a + 8;  
    free(a);  
    difference = b - a;  
}
```

- Not entirely clear if this is valid C code:
  - C standard: It is valid to subtract 2 pointers if they both refer to the same object
  - However, memory has been freed and pointers are not dereferenced, do they still refer to the same object?

# Pointer arithmetic

- Problem:
  - If pointers are invalidated by setting them to NULL or 3GB +unique, then subtraction will fail
- Solution:
  - Invalidate pointers by setting first 2 bits: causes pointers to be above 3GB (thus invalid)
  - However the arithmetic will still work
  - This causes a small incompatibility in Linux: a pointer above and below 0x40000000 or 0x80000000
  - Unlikely to occur, requires: memory allocated across boundary, 1 pointer above and 1 below, a free of that memory and then a subtraction of those 2 pointers

# Pointers copied as different types

- Limitation
  - If an object contains a pointer that is not copied as a pointer, we can't track the pointer
  - For example:
    - memcpy: copies memory as void type from one location to the other, results in pointer being missed
- Solution
  - Allow programmer to manually register a pointer if desired

# Optimizations

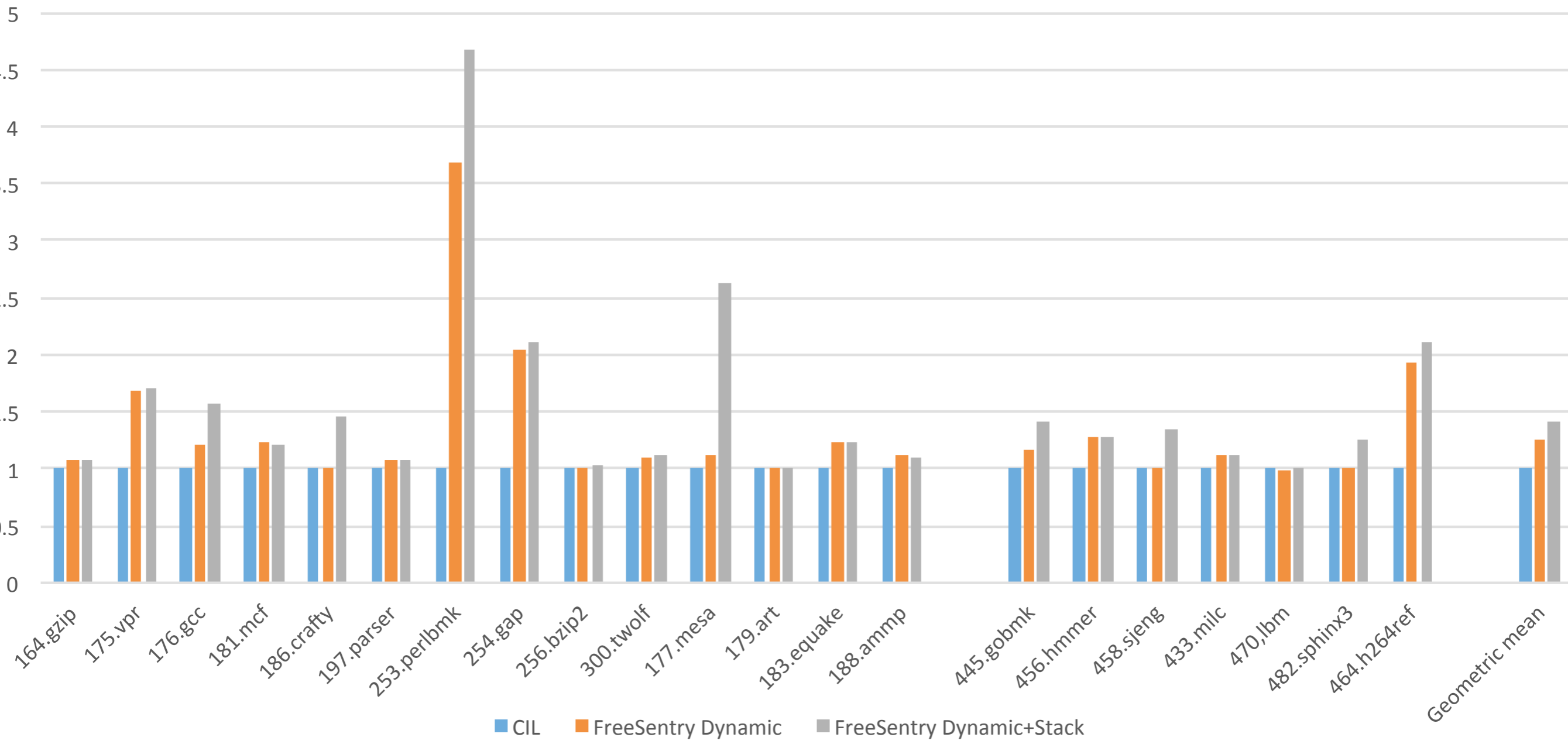
- Call graph analysis
  - Examine functions called by functions
    - Until we hit a leaf function or a library call
    - Provide models for often used libraries: libc, libm, openssl: indicate if library function calls free or not)
  - First optimization: remove tracking for local variable, if no calls to free and no address of variable is taken

# Optimizations

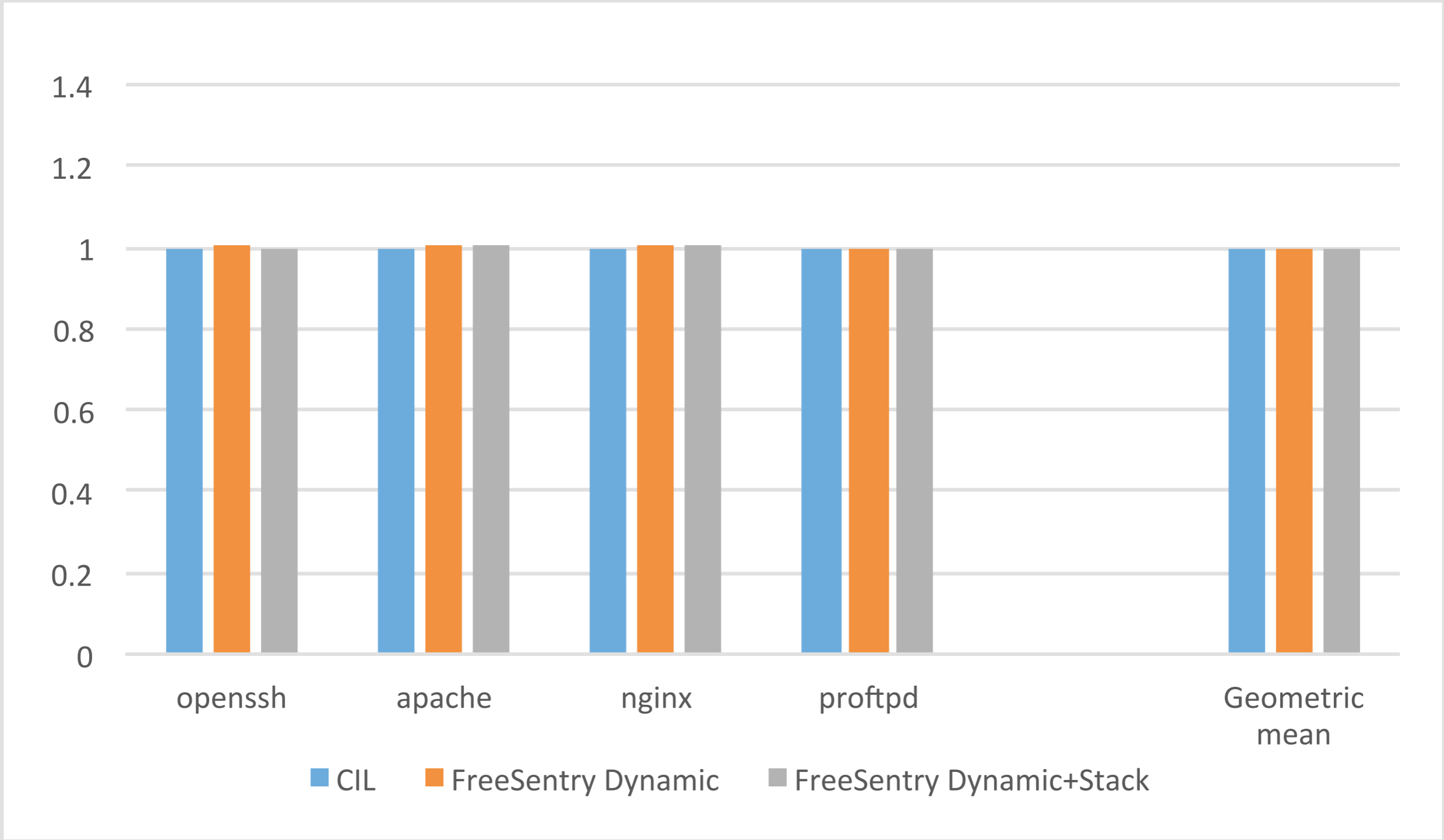
- Second optimization: Loop optimization
  - If no calls to free or unexpected exits from the loop (i.e. returns)
  - Registration for simple pointer assignments (no arithmetic or dereferencing on the left-hand side) can be moved out of loop
  - Pointers are still tracked, but since they are overwritten every loop iteration, the registration only happens after the loop is done



# Performance overhead



# Performance overhead servers



# Performance study

- Gap performs computations in groups
- It contains 71k lines of code
- Profiling it shows 2 most heavily used functions are NewBag and ProdInt
  - NewBag is an allocation function, calls malloc/free: 150 lines
  - ProdInt is 174 lines and returns the product of 2 integers

# Performance study

- Default run of gap has an overhead of 104.28%
- Opting out of these 2 functions (304 lines of code) drops overhead to 33.42%
- Opting out of NewBag means the pointers in NewBag aren't tracked, however functions that call NewBag will have the memory that NewBag allocates labelled and invalidated and will have their pointers tracked
- Number of calls to our tracking function drops from 2.3 billion to 1.3 billion

# Security evaluation

- C programs with public exploits
- When run, crashes when trying to access freed memory

CVE ID	Description	Result
CVE-2003-0015	Double free vulnerability in CVS <= 1.1.4	Protected
CVE-2004-0416	Double free vulnerability in CVS 1.12.x-1.12.8 and 1.11.x-1.11.16	Protected
CVE-2007-1521	Double free vulnerability in PHP < 4.4.7 and < 5.2.2	Protected
CVE-2007-1522	Double free vulnerability in session extension in PHP 5.2.0 and 5.2.1	Protected
CVE-2007-1711	Double free vulnerability in PHP 4.4.5 and 4.4.6	Protected

# Security evaluation

- Found previously unknown bug in benchmarks: perlbnk:
  - use-after-free due to reallocated memory being moved to a new location and a stale pointer being used
- Bug privately reported in ClamAV 0.98.4 by Damien Millescamps with sample which triggers a double free (CVE-2014-9050).
  - Compiling with FreeSentry prevents the double free
  - Fixed in version 0.98.5

# Conclusion

- Use-after-frees are an important vulnerability
- Few mitigations exist that protect against it
- FreeSentry provides protection at a moderate cost with easy deployment
- Provides flexibility for programmer to improve performance
- For programs with high I/O, overhead is low
- Can also be used as a testing tool