Bounds Checking with Taint-Based Analysis[1]

Security Improvements through Runtime Bounds Checking

Group 15: Ammar Ahmed, Marwa Houalla, Daniel Liu, Wenzhao Qiu

2024-04-14

Table of contents

Control Flow Integrity

Google

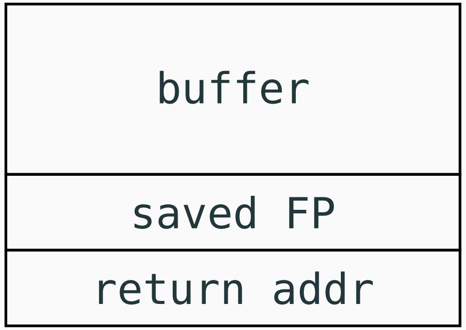
Google Security Engineering Technical Report
March 4, 2024

Secure by Design:
Google's Perspective on Memory Safety

"Memory safety bugs are responsible for the majority (~70%) of severe vulnerabilities in large C/C++ code bases"

Control Flow Hijacking

```
1139: sub    $0x18,%rsp
113d: mov    %rdi,%rsi
1140: mov    %rsp,%rdi
1143: call   1030 <strcpy@plt>
1148: add    $0x18,%rsp
114c: ret
```



Control Flow Hijacking

```
1139: sub    $0x18,%rsp
113d: mov    %rdi,%rsi
1140: mov    %rsp,%rdi
1143: call   1030 <strcpy@plt>
1148: add    $0x18,%rsp
114c: ret
```

marwa\0\0\0
saved FP
return addr

Control Flow Hijacking

```
1139: sub    $0x18,%rsp
113d: mov    %rdi,%rsi
1140: mov    %rsp,%rdi
1143: call   1030 <strcpy@plt>
1148: add    $0x18,%rsp
114c: retc
```

AAAAAAAA

FPFPFPFP

wherever i want

Figure 5: (credit 388)

Control Flow Integrity Techniques

- Data Execution Prevention
- Stack Canaries
- Address Space Layout Randomization

None of these actually address the real problem: **buffer overflows**

Bounds Checking

Bounds Checking at Compile Time

```rust
fn main() {
    let mut a: [i64; 3] = [1, 2, 3];
    a[4] = 5;
}
```

```
[daniel@tripledelete workspace]$ cargo build
   Compiling workspace v0.1.0 (/home/daniel/Desktop/workspace)
error: this operation will panic at runtime
 --> src/main.rs:3:5
  |
3 |     a[4] = 5;
  |     ^^^^ index out of bounds: the length is 3 but the index is 4
  |
  = note: `#[deny(unconditional_panic)]` on by default
```

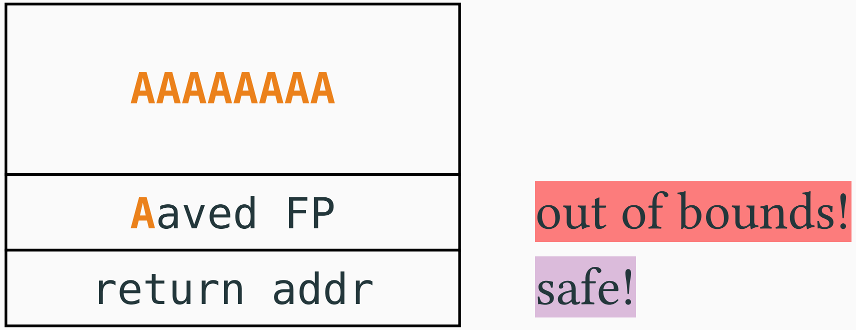
Bounds Checking at Run Time

```rust
#[allow(unconditional_panic)]
fn main() {
    let mut a: [i64; 3] = [1, 2, 3];
    a[4] = 5;
}
```

```
[daniel@tripledelete workspace]$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/workspace`
thread 'main' panicked at src/main.rs:4:5:
index out of bounds: the len is 3 but the index is 4
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Bounds Checking as a Defense

```
1139: sub    $0x18,%rsp
113d: mov    %rdi,%rsi
1140: mov    %rsp,%rdi
1143: call   1030 <strcpy@plt>
1148: add    $0x18,%rsp
114c: ret
```



```c
#include <string.h>

void vulnerable(char* p) {
    char buffer[16];
    char* d = buffer;
    while (*p) {
        *d++ = *p++;
    }
}

int main(int argc, char** argv) {
    vulnerable(argv[1]);
}
```

```
define dso_local void @vulnerable(ptr noundef %0) #0 {
  %2 = alloca ptr, align 8
  %3 = alloca [16 x i8], align 16
  %4 = alloca ptr, align 8
  ; setup...
  br label %6
  ; loop condition...
  br i1 %9, label %10, label %16
10:                                                ; preds = %6
  %11 = load ptr, ptr %2, align 8
  %12 = getelementptr inbounds i8, ptr %11, i32 1
  store ptr %12, ptr %2, align 8
  ; increment...
  br label %6, !llvm.loop !6
16:                                                ; preds = %6
  ret void
}
```

Efficient Bounds Checking

Representing Bounds information

1. Fat Pointers

- Store the lower and upper bounds of the object along with the pointer
- Fast, but modifies pointer format

```
struct FatPtr_T {
    T*        rp;    /* raw pointer */
    char*     base;  /* of allocated memory region */
    size_t    size;  /* of allocated memory region, in bytes */
};
```

(credit: William Klieber, CMU)

Representing Bounds information

2. Object Storage

- Store bounds information alongside the object that the pointer dereferences to
- Does not work if pointer is modified

Implementing Bounds Checking

1. Two Branch
    - Used in Greg McGary's bounds checker

```
start:
   flag=(ptr >= low)
   if(flag) then low_pass
   trap
low_pass:
   flag=(ptr < high)
   if(flag) then high_pass
   trap
high_pass:
```

2. One Branch

```
start:
  tmp=(unsigned)(ptr-low)
  flag=(tmp < size)
  if(flag) then ok
  trap
ok:
```

3. Bound Instruction

```
start:
  bound ptr, b_ptr
```

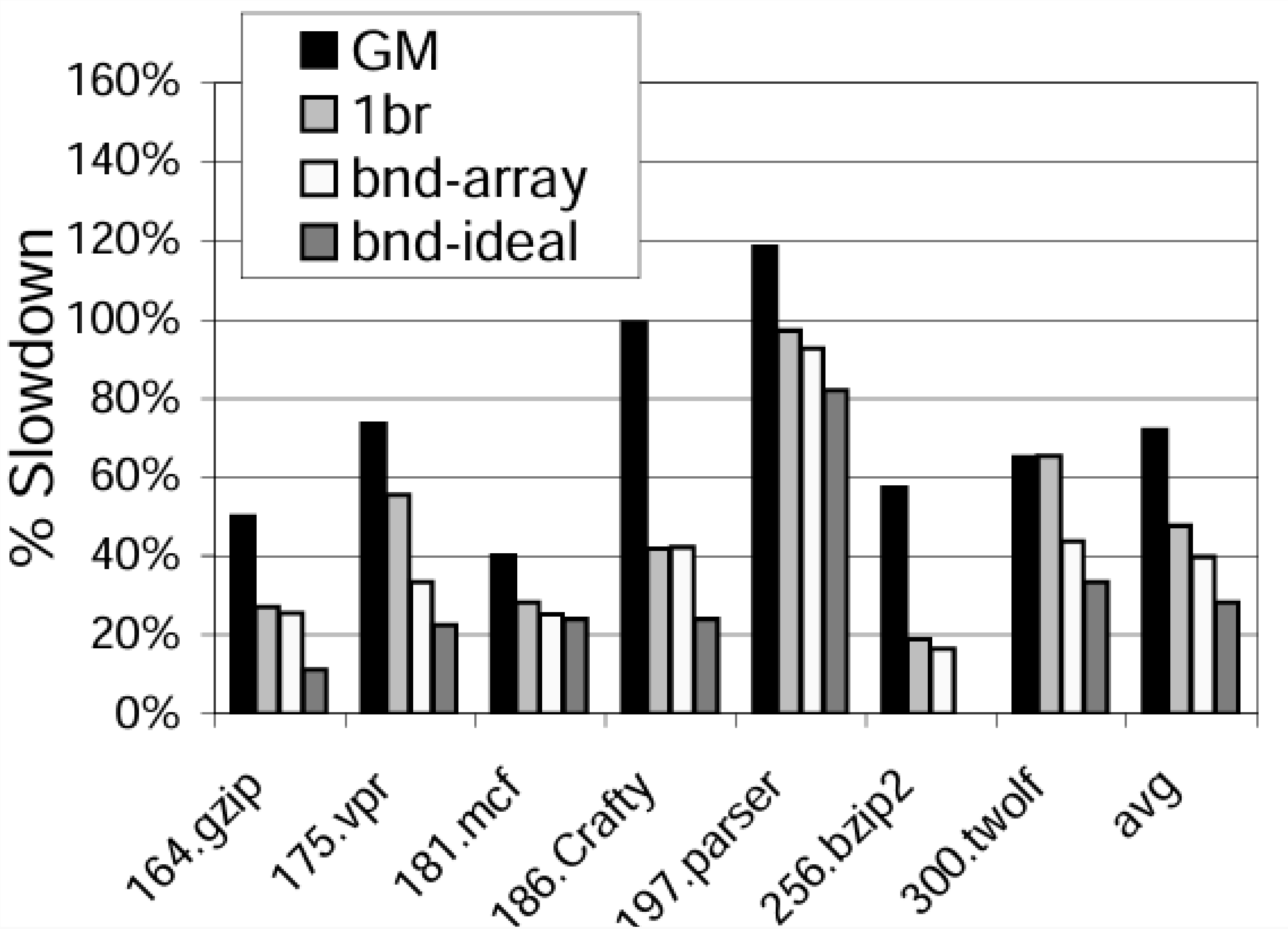
Bound Instruction for Arrays

- Bound instruction uses upper and lower bounds stored in fat pointers
- Global and stack arrays are not referenced with pointers
- Now what?

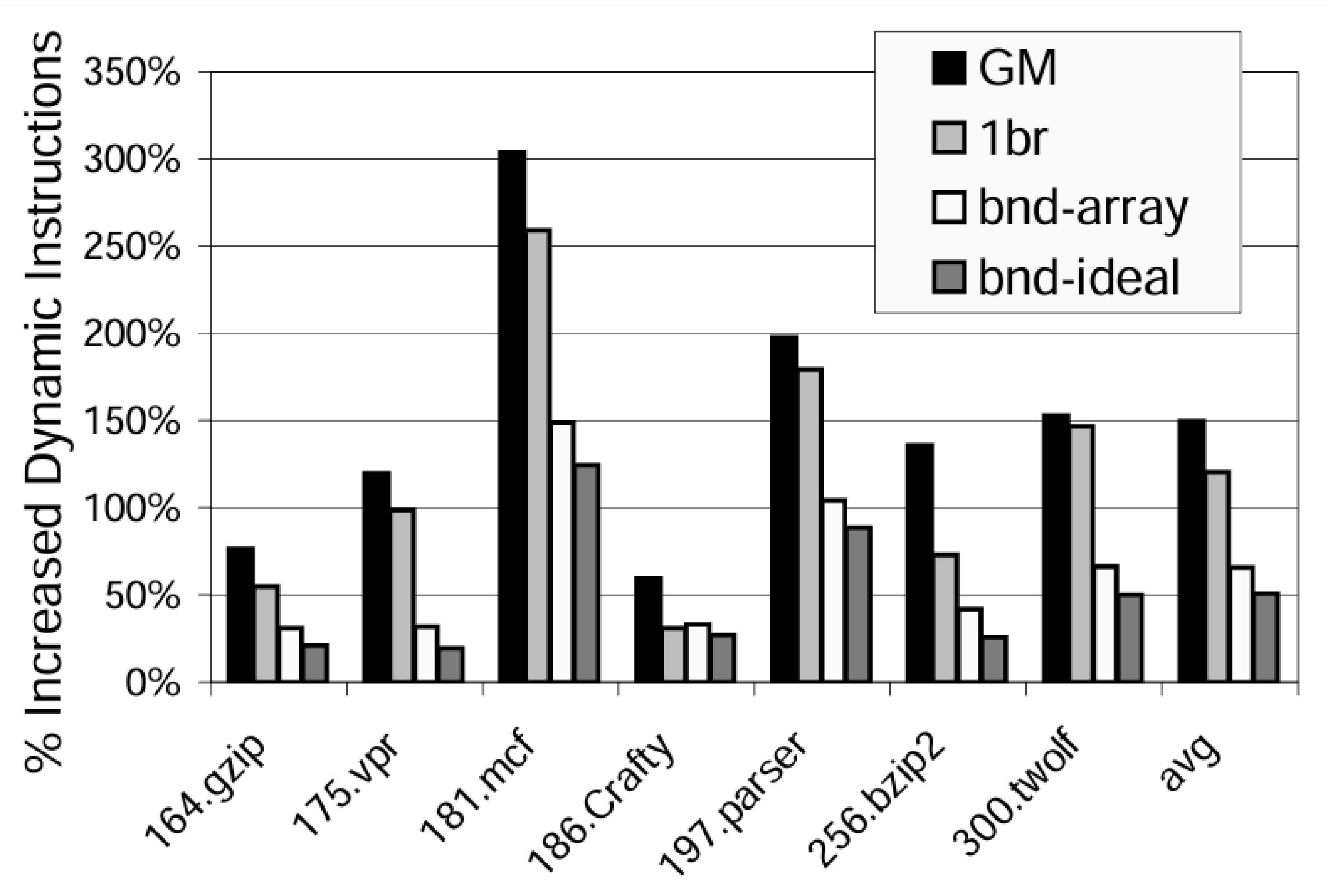
- Bound instruction uses upper and lower bounds stored in fat pointers
- Global and stack arrays are not referenced with pointers
- Now what?
- **Solution: allocate static memory to hold meta-data bounds information**

  + Use this memory location as the second operand in bound

Analyzing Performance



Analysing Overhead



Towards Optimal Bounds Checking

Towards Optimal Bounds Checking

So far:

- Efficient bounds checking in C

Pros:

- Complete protection
- Reasonably fast

Cons:

- Significant overhead:
  - McGary's compiler: Overhead of 71%
    + Single branch compare = 48%
    + x86 bound instruction = 40% (AMD Athlon Processor)

Can we do better?

- Taint-based bounds checking

So far:

- Efficient bounds checking in C

Towards Optimal Bounds Checking

So far:

- Efficient bounds checking in C

Pros:

- Complete protection
- Reasonably fast

Cons:

- Significant overhead:
  - McGary's compiler: Overhead of 71%

    + Single branch compare = 48%

    + x86 bound instruction = 40% (AMD Athlon Processor)

Can we do better?

- Taint-based bounds checking

Towards Optimal Bounds Checking

So far:

- Efficient bounds checking in C

Pros:

- Complete protection
- Reasonably fast

Cons:

- Significant overhead:

McGary's compiler: Overhead of 71%

+ Single branch compare = 48%

+ x86 bound instruction = 40% (AMD Athlon Processor)

Can we do better?

- Taint-based bounds checking

So far:

- Efficient bounds checking in C

Pros:

- Complete protection
- Reasonably fast

Cons:

- Significant overhead:
  - McGary's compiler: Overhead of 71%

    + Single branch compare = 48%

    + x86 bound instruction = 40% (AMD Athlon Processor)

Can we do better?

- Taint-based bounds checking

Towards Optimal Bounds Checking

So far:

- Efficient bounds checking in C

Pros:

- Complete protection
- Reasonably fast

Cons:

- Significant overhead:
  - McGary's compiler: Overhead of 71%

    + Single branch compare = 48%

    + x86 bound instruction = 40% (AMD Athlon Processor)

Can we do better?

- Taint-based bounds checking

Towards Optimal Bounds Checking

So far:

- Efficient bounds checking in C

Pros:

- Complete protection
- Reasonably fast

Cons:

- Significant overhead:
  - McGary's compiler: Overhead of 71%

    + Single branch compare = 48%

    + x86 bound instruction = 40% (AMD Athlon Processor)

Can we do better?

- Taint-based bounds checking

Taint-Based Bounds Checking

Defn - Security technique used to prevent buffer overflow attacks by tracking the flow of untrusted, or “tainted,” data through a program

Aka bounds checking with shortcuts

Taint-Based Bounds Checking

Defn - Security technique used to prevent buffer overflow attacks by tracking the flow of untrusted, or “tainted,” data through a program

A.k.a. bounds checking with shortcuts

Taint-Based Bounds Checking

Two shortcuts:

1. Interface Analysis
2. Memory Writer

Interface Analysis

Interface Analysis

```python
def monitor_ps5_pro_release():
    url = "https://www.daniel-r-us.com/ps5-pro"

    while True:
        try:
            response = requests.get(url)
            soup = BeautifulSoup(response.text, 'html.parser')

            if "PS5 Pro" in soup.text:
                print(f"PS5 Pro is available on Daniel R Us!")
                break

            time.sleep(60)  # Check every minute
        except Exception as e:
            pass
```

Interface Analysis

Evil hacker Halderman

Interface Analysis

```
def monitor_ps5_pro_release():
    soup = BeautifulSoup(response.text, 'html.parser')
```

Evil hacker Halderman

b'AAAAAAA....A’

Interface Analysis

Observations:

- Buffer overflow attacks can occur when malicious input is provided via *external* interfaces of an application

-External interfaces:

- return values from library calls (gets)
- argv

Solution:

- Just bounds check objects that get their values from external input
- We call such objects *TAINTED*

Observations:

- Buffer overflow attacks can occur when malicious input is provided via *external* interfaces of an application
- External interfaces:
  - return values from library calls (gets)
  - argv

Solution:

- Just bounds check objects that get their values from external input
- We call such objects *TAINTED*

Observations:

- Buffer overflow attacks can occur when malicious input is provided via *external* interfaces of an application
- External interfaces:
  - return values from library calls (gets)
  - argv

Solution:

- Just bounds check objects that get their values from external input
- We call such objects *TAINTED*

Interface Analysis

```python
def monitor_ps5_pro_release():
    url = "https://www.daniel-r-us.com/ps5-pro"

    while True:
        try:
            response = requests.get(url)
            soup = BeautifulSoup(response.text, 'html.parser')

            if "PS5 Pro" in soup.text:
                print(f"PS5 Pro is available on Daniel R Us!")
                break

            time.sleep(60)
        except Exception as e:
            pass
```

Interface Analysis

```python
def monitor_ps5_pro_release():
    url = "https://www.daniel-r-us.com/ps5-pro"

    while True:
        try:
            response = requests.get(url) # TAINTED
            soup = BeautifulSoup(response.text, 'html.parser')

            if "PS5 Pro" in soup.text:
                print(f"PS5 Pro is available on Daniel R Us!")
                break

            time.sleep(60)
        except Exception as e:
            pass
```

```python
def monitor_ps5_pro_release():
    url = "https://www.daniel-r-us.com/ps5-pro"

    while True:
        try:
            response = requests.get(url) # TAINTED
            perform_bounds_checking(response)
            soup = BeautifulSoup(response.text, 'html.parser')

            if "PS5 Pro" in soup.text:
                print(f"PS5 Pro is available on Daniel R Us!")
                break

            time.sleep(60)
        except Exception as e:
            pass
```

What if an object references a *TAINTED* object?

```
buffer = get_external_input()  # buffer is TAINTED
alias = buffer  # alias now also TAINTED
```

Then we must bounds check the alias:

- Implement using propagation of TAINTED pointers
- Data-flow analysis

What if an object references a *TAINTED* object?

```
buffer = get_external_input()  # buffer is TAINTED
alias = buffer  # alias now also TAINTED
```

Then we must bounds check the alias:

- Implement using propagation of TAINTED pointers
- Data-flow analysis

Memory Writer

Memory Writer

When do buffer overflow attacks occur?

- When an attacker can write more data to a buffer than it can hold.

When do buffer overflow attacks occur?

- When an attacker can **write** more data to a buffer than it can hold.

Writes to arrays and pointer dereferences (left-hand side of assignments) are marked for bounds-checking:

```
char x[100];
int c, i = 0;
char *y = x, *z;
while ((c = getchar()) != EOF) {
    x[i++] = c;  // Write operation that needs bounds checking
}
z = y;  // TAINTED flow via scalar assignment and aliasing
```

Writes to arrays and pointer dereferences (left-hand side of assignments) are marked for bounds-checking:

```
char x[100];
int c, i = 0;
char *y = x, *z;
while ((c = getchar()) != EOF) {
    x[i++] = c;  // FAT pointer
}
z = y;  // TAINTED flow via scalar assignment and aliasing
```

Memory Writer

Memory Writer Algorithm:

1. Find all *FAT* pointers

Propagate FAT status via data-flow graph Profit??

Memory Writer

Memory Writer Algorithm:

1. Find all *FAT* pointers
2. Propagate FAT status via data-flow graph

Profit??

Memory Writer

Memory Writer Algorithm:

1. Find all *FAT* pointers
2. Propagate FAT status via data-flow graph
3. Profit??

Memory Writer

Memory Writer Algorithm:

1. Find all *FAT* pointers
2. Propagate FAT status via data-flow graph
3. Profit?? **No.**

Memory Writer

Memory Writer AND Interface Analysis:

- Combination for *optimal* bounds-checking

Algorithm:

1. Interface analysis *first*, identify external inputs, and mark related buffers as TAINTED
2. Apply Memory Writer to TAINTED buffers
3. Profit.

Memory Writer

Memory Writer AND Interface Analysis:

- Combination for *optimal* bounds-checking

Algorithm:

1. Interface analysis *first*, identify external inputs, and mark related buffers as TAINTED
2. Apply Memory Writer to TAINTED buffers
3. Profit.

Taint-Based Bounds Checking

Putting it all together, we have a bounds-checker that:

- Analyzes external memory interactions,
- Monitors writes to memory, and
- Tracks their references

This is **Taint-based** bounds checking.

Results and Evaluation

Benchmarking

Results

Benchmarking

Small web server: `ATPhttpd-0.4b`

Results

Benchmarking

Small web server: `ATPhttpd-0.4b`

- Using Mem-Write with Interface Optimization:
  - Caught vulnerability
  - 6% slowdown

Benchmarking

Small web server: `ATPhttpd-0.4b`

- Using Mem-Write with Interface Optimization:
  - Caught vulnerability
  - 6% slowdown

SPEC 2000 Integer programs

Benchmarking

Small web server: `ATPhttpd-0.4b`

- Using Mem-Write with Interface Optimization:
  - Caught vulnerability
  - 6% slowdown

SPEC 2000 Integer programs

- *Problem:* No network traffic
-

Benchmarking

Small web server: `ATPhttpd-0.4b`

- Using Mem-Write with Interface Optimization:
  - Caught vulnerability
  - 6% slowdown

SPEC 2000 Integer programs

- *Problem:* No network traffic
- *Solution:* Applied bounds checking to all system call interfaces to the program

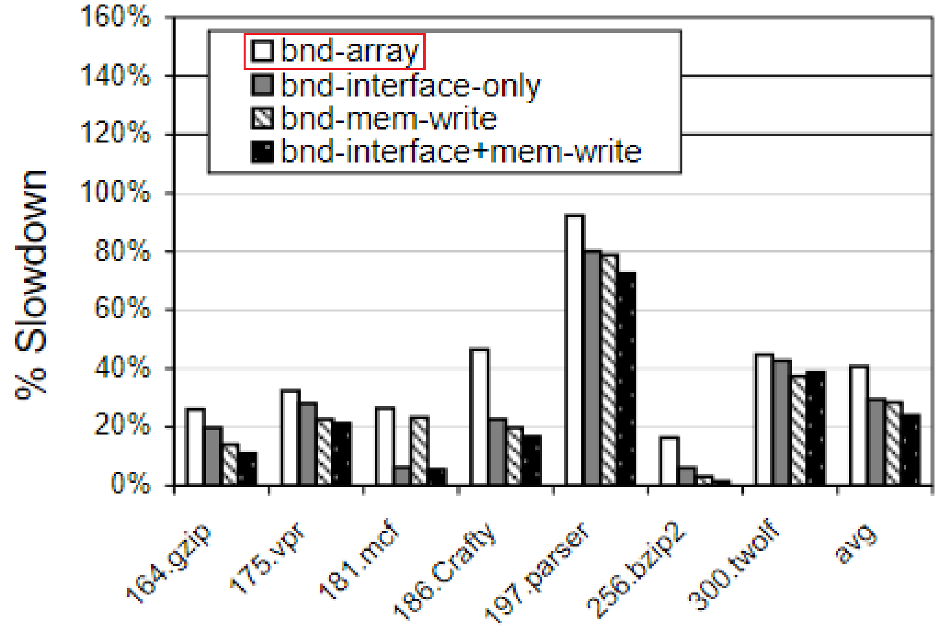


Fig. 8. Performance advantage of interface and memory-writer optimizations.

- *bnd-array*
  - 40% average overhead/slowdown

Results

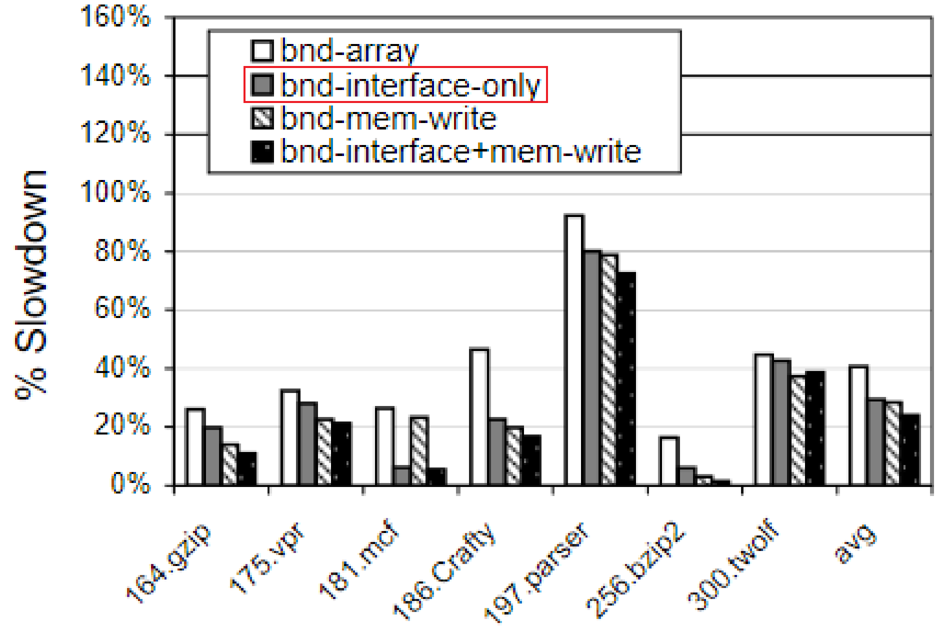


Fig. 8. Performance advantage of interface and memory-writer optimizations.

- *bnd-array*
  - 40% average overhead/slowdown
- *bnd-interface-only:* 29%

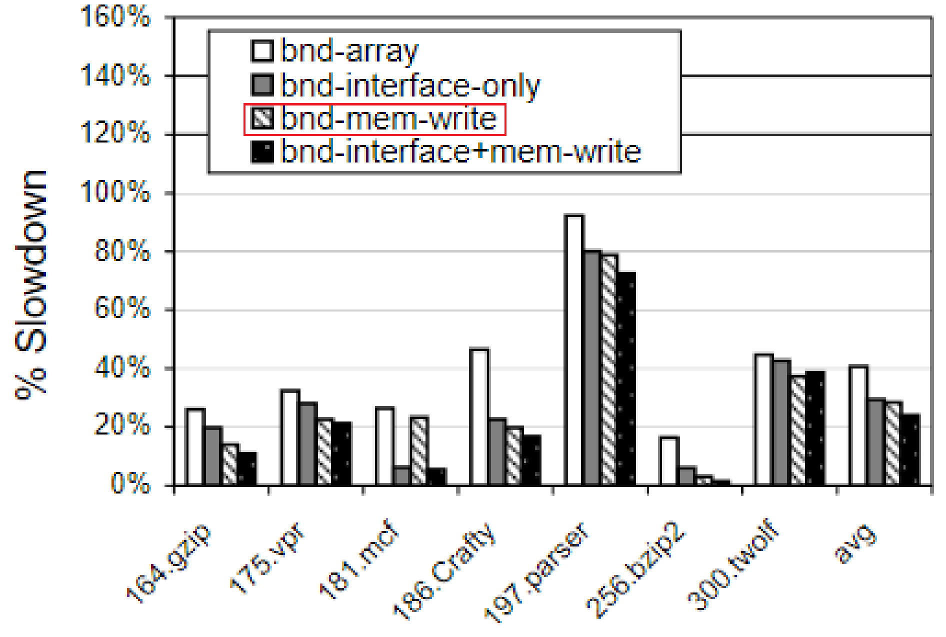


Fig. 8. Performance advantage of interface and memory-writer optimizations.

- *bnd-array*
  - 40% average overhead/slowdown
- *bnd-interface-only:* 29%
- *bnd-mem-write:* 28%

Results

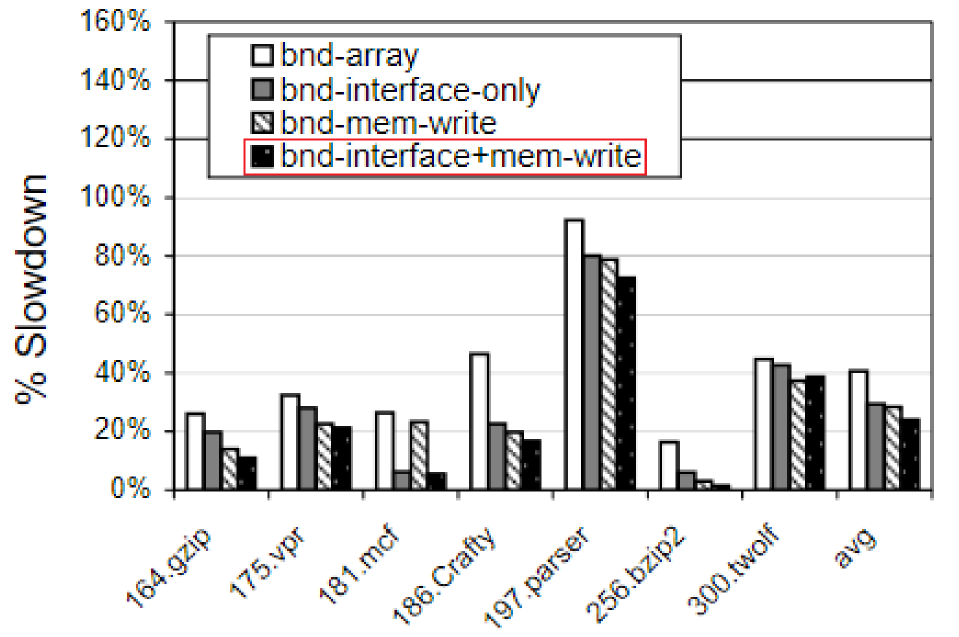


Fig. 8. Performance advantage of interface and memory-writer optimizations.

- *bnd-array*
  - 40% average overhead/slowdown
- *bnd-interface-only*: 29%
- *bnd-mem-write*: 28%
- *bnd-interface+mem-write*: 24%

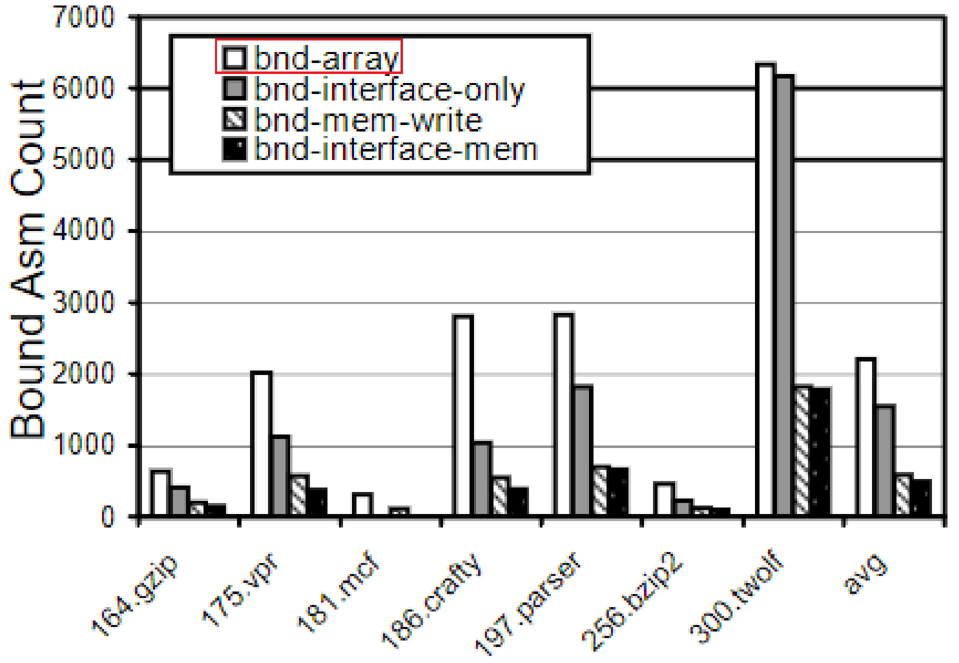


Fig. 9. Reduction in the number of static bounds instruction in the binary.

- *bnd-array*
  - 2203 bounds instructions on average

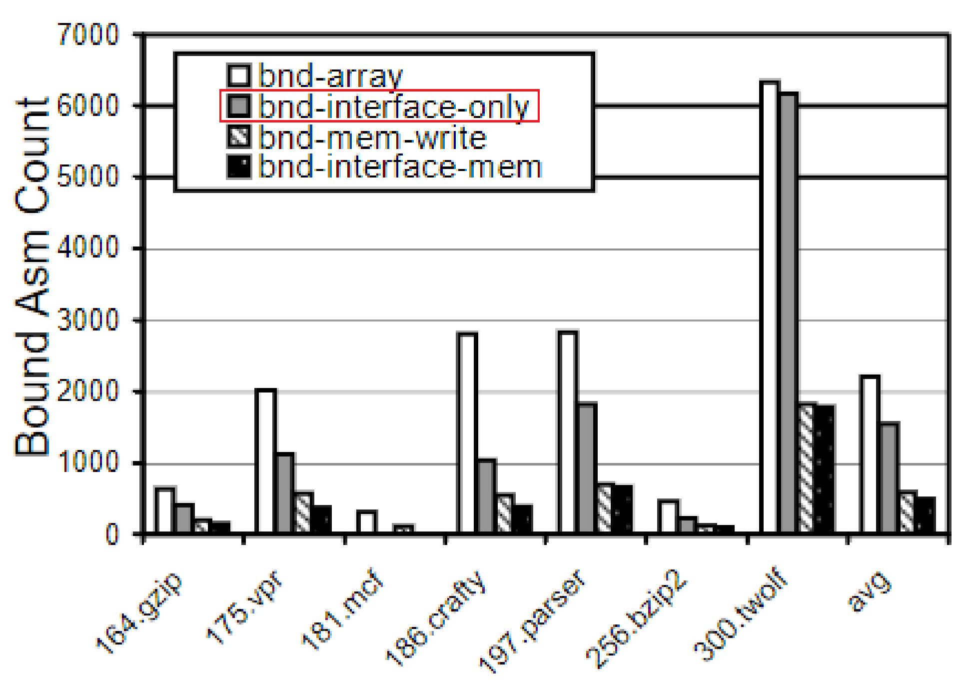


Fig. 9. Reduction in the number of static bounds instruction in the binary.

- *bnd-array*
  - 2203 bounds instructions on average
- *bnd-interface-only:* 1573

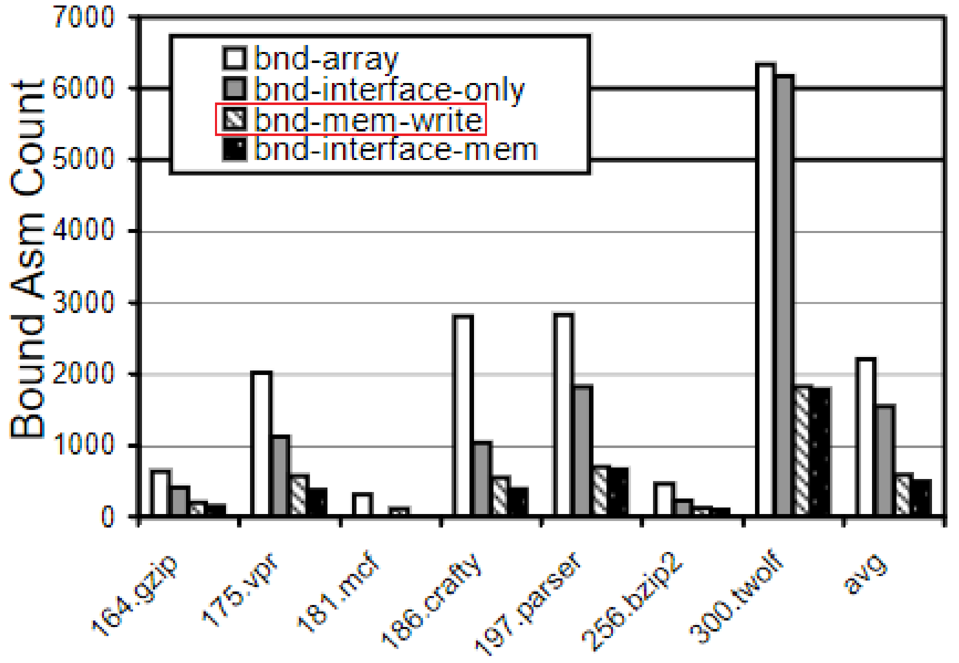


Fig. 9. Reduction in the number of static bounds instruction in the binary.

- *bnd-array*
  - 2203 bounds instructions on average
- *bnd-interface-only:* 1573
- *bnd-mem-write:* 581

Fig. 9. Reduction in the number of static bounds instruction in the binary.

- *bnd-array*
  - 2203 bounds instructions on average
- *bnd-interface-only:* 1573
- *bnd-mem-write:* 581
- *bnd-interface+mem-write:* 495

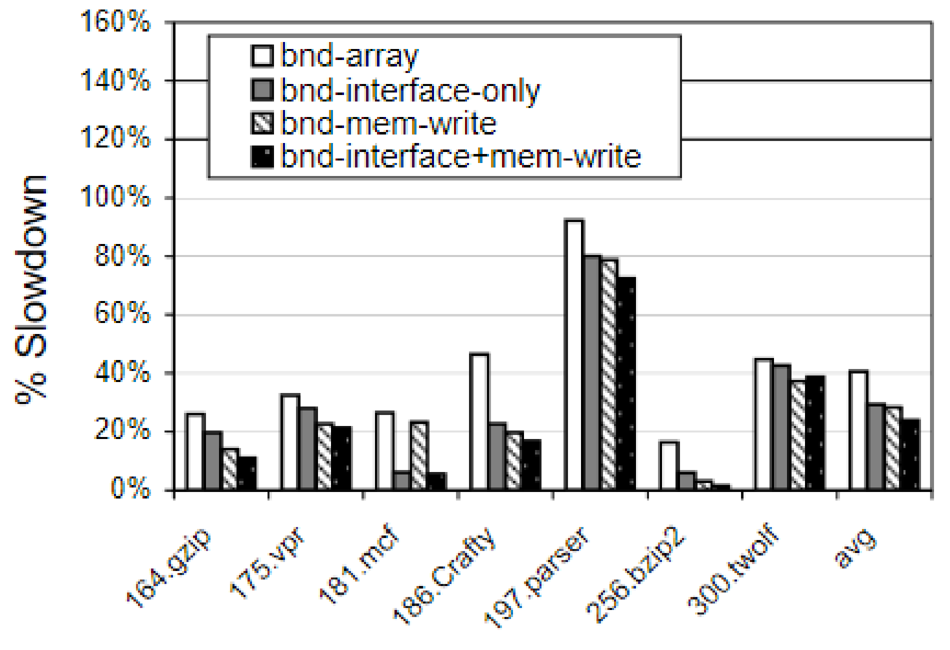


Fig. 8. Performance advantage of interface and memory-writer optimizations.

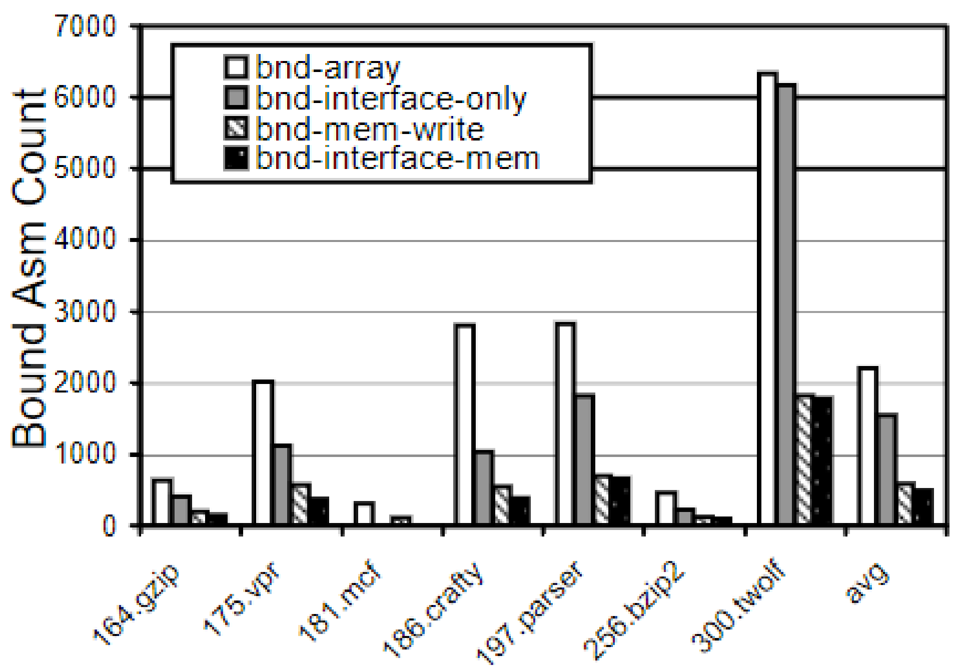


Fig. 9. Reduction in the number of static bounds instruction in the binary.

Conclusion

Limitations

- Path and context insensitive
  - Simplifies the analysis, but may result in retention of some unnecessary bounds checks
-

Limitations

- Path and context insensitive
  - Simplifies the analysis, but may result in retention of some unnecessary bounds checks
- Dependent on x86 bounds instruction

Bibliography

[1] W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala, “Bounds checking with taint-based analysis,” in *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, in HiPEAC'07. Ghent, Belgium: Springer-Verlag, 2007, pp. 71–86.