KU LEUVEN

Hoisting and Deleting Redundant Instrumentation at the IR Level

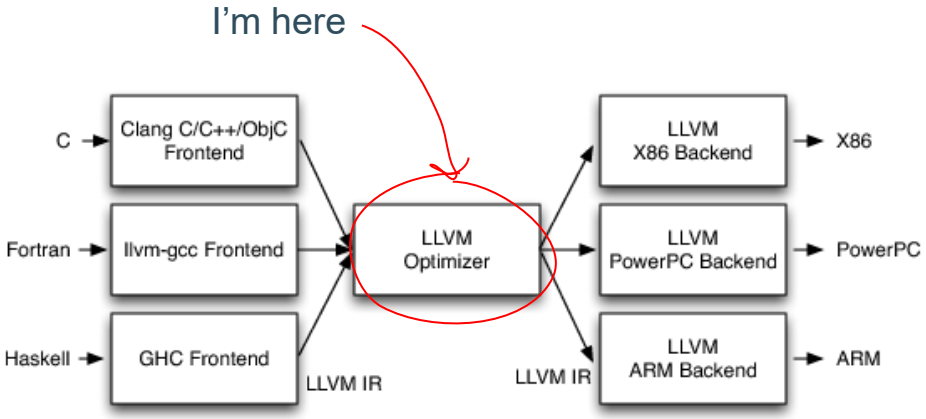
DistriNet

Adriaan Jacobs

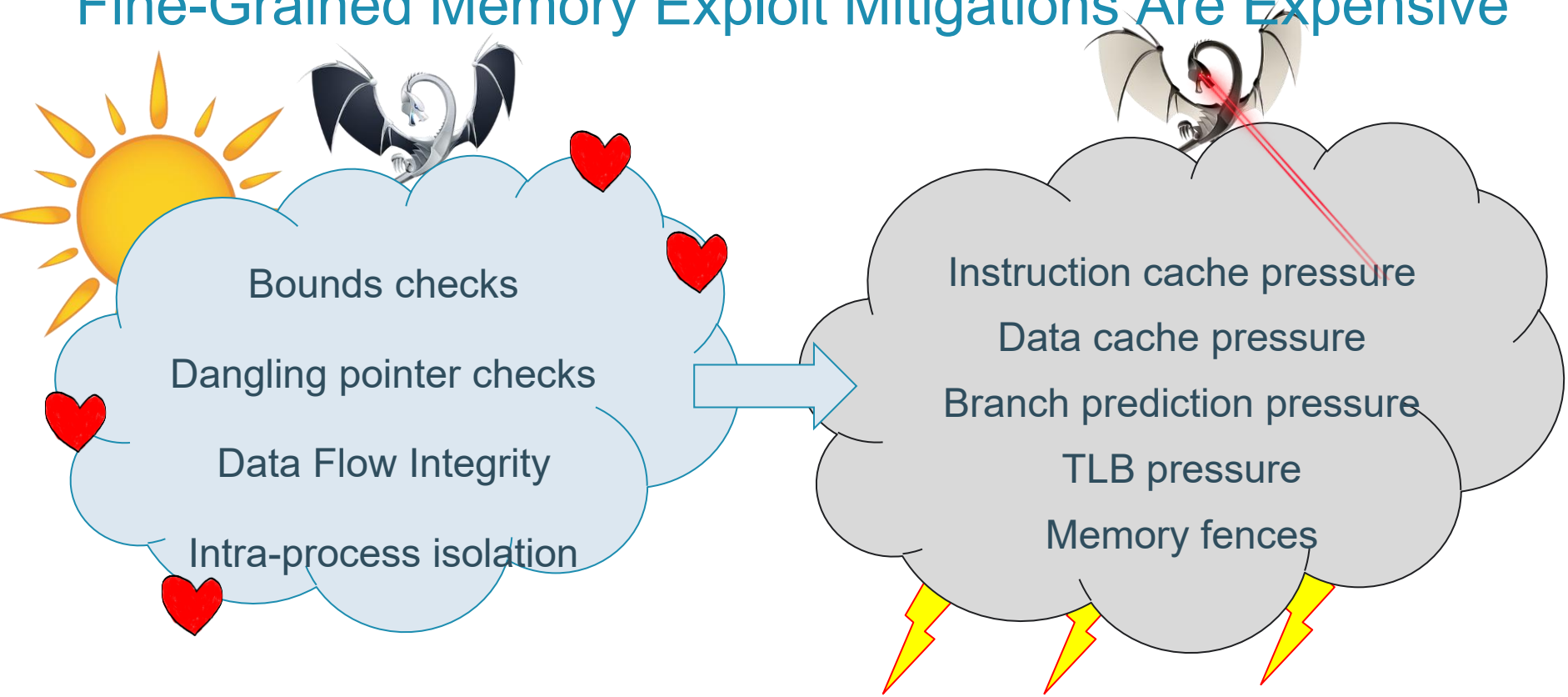
KU LEUVEN GENT

My Use of LLVM

- Memory exploit mitigations
- Entirely in the middle-end
  - Cross-ISA compatibility
  - Full access to optimizations
  - Easier to analyze
- Focus on reducing instrumentation overhead



Fine-Grained Memory Exploit Mitigations Are Expensive



Oleksenko, Oleksii, et al. "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack." Proceedings of the ACM on Measurement and Analysis of Computing Systems 2.2 (2018): 1-30.

Redundant Instrumentation

Statically Computable

- Result of instrumentation is clear at compile-time
- Elide instrumentation/reject program

```
char data[16];
assert_in_bounds(data, 15);
data[15] = '\0';
```

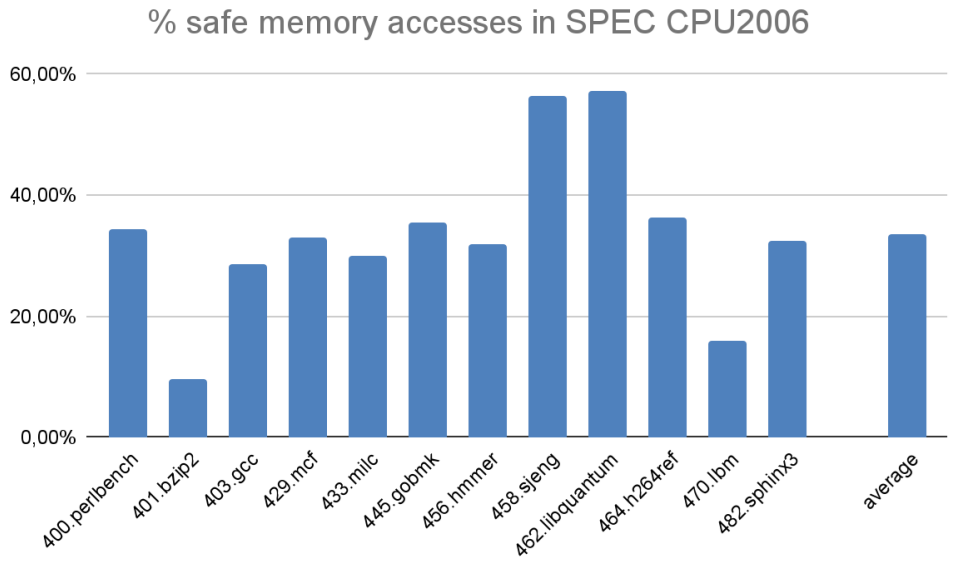
Previously Computed

- Result of instrumentation can be derived from earlier computation
  - Different instrumentation
  - Self-repetition (loop)
- Elide or move instrumentation

```
char data[16];
assert_in_bounds(data, i);
data[i] = ...;
assert_in_bounds(data, i);
data[i] = ...;
```

Safe Access Analysis

- Backwards Data Flow Analysis
- Mostly top-level
- Inter-procedural
- SCEV for pointer offsets
- Positive/negative offsets
- ~33% of memory accesses
- Pad allocations?

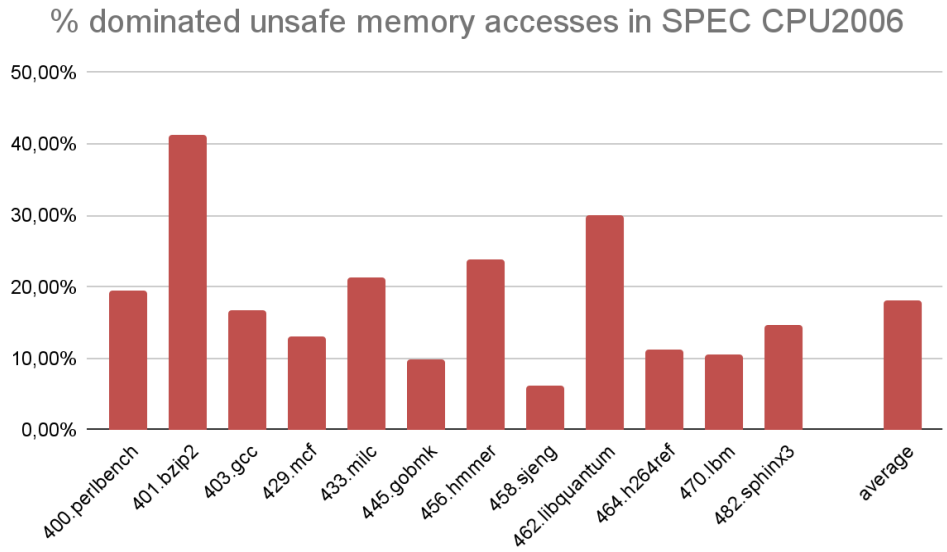


KU LEUVEN

Domination Pruning

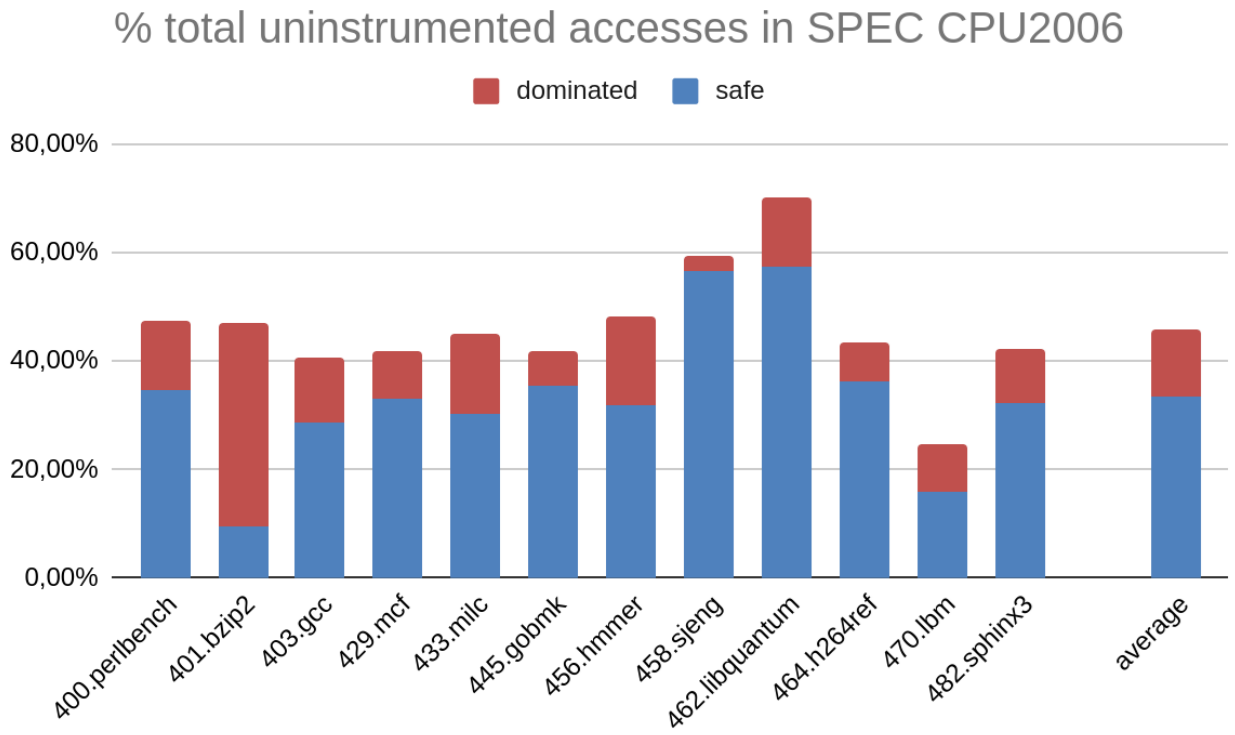
- Guaranteed to be checked already
- ~20% of unsafe accesses

```
char data[16];
assert_in_bounds(data, i);
if (data[i] == 'a') {
   assert_in_bounds(data, i);
   data[i] = 'b';
}
```



KU LEUVEN

Deleted Instrumentation



Post-Domination Pruning

- Guaranteed to be checked in the future
- Asan--
- TOCTOU problem
- Limit impact?

```
char data[16];
while (cond) {
   assert_in_bounds(data, i);
   data[i]++;
}
// ...
assert_in_bounds(data, i);
data[i] = 0;
```

?

Zhang, Yuchen, et al. "Debloating Address Sanitizer." 31st USENIX Security Symposium (USENIX Security 22). 2022.

Structured Access Pruning

- Both upper & lower bound have been checked already
- Hard to prove, TOCTOU
- Memory Access Skeleton [Spindle]

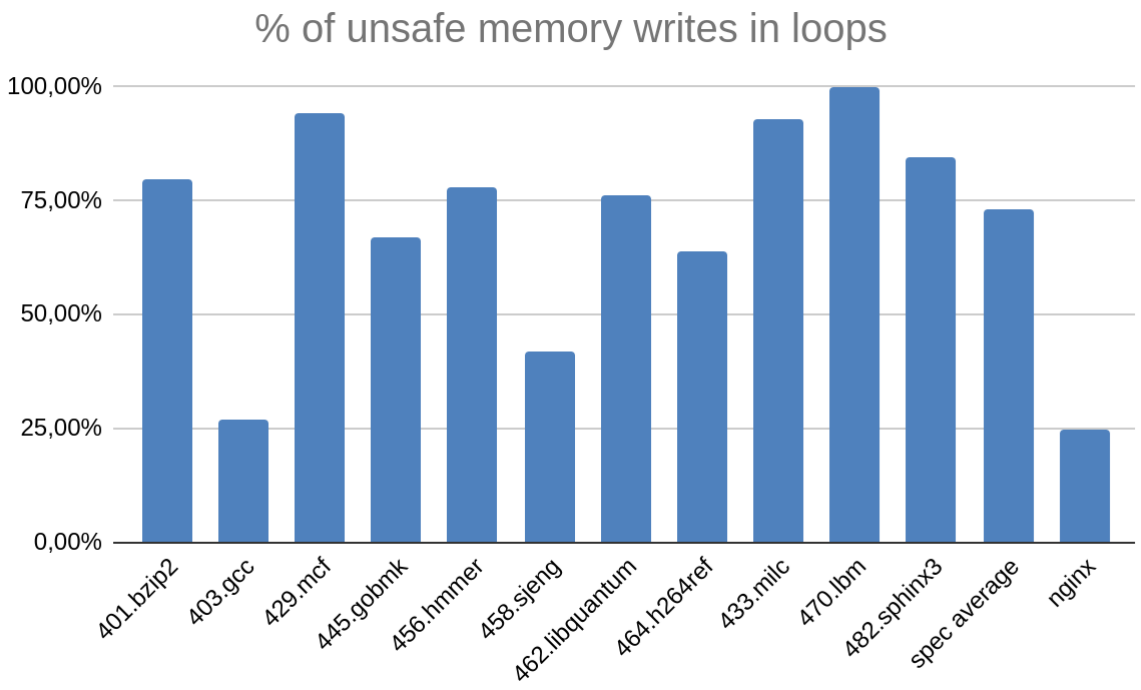
```
char data[16];
assert_in_bounds(data, i-1);
data[i - 1] = 'K';
assert_in_bounds(data, i+1);
data[i + 1] = 'A';
assert_in_bounds(data, i);
data[i] = 'A';
```

Wang, Haojie, et al. "Spindle: Informed memory access monitoring." 2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018.

Self-Imposed Redundancy: Loop Optimizations

```
const int i = ...;
while (cond()) {
   assert_in_bounds(data, i);
   data[i] = ...;
}
```

Self-Imposed Redundancy: Loop Optimizations



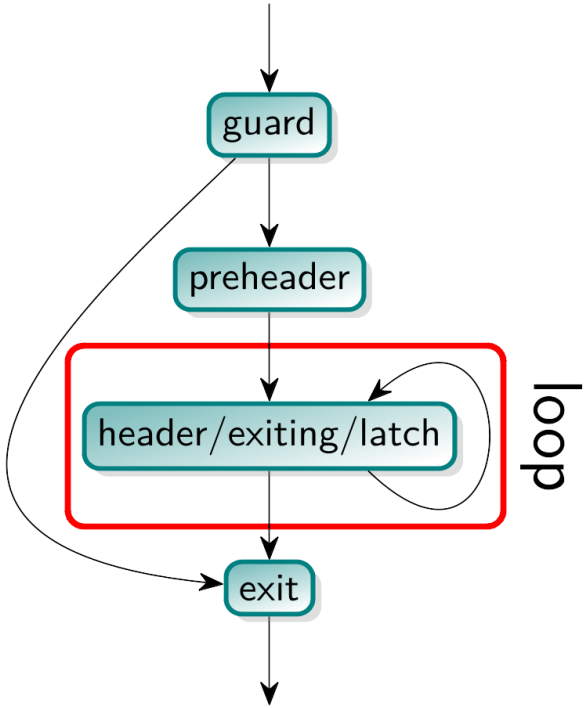
Self-Imposed Redundancy: Loop Optimizations

```
const int i = ...;
while (cond()) {
   assert_in_bounds(data, i);
   data[i] = ...;
}
```

Self-Imposed Redundancy: Loop Optimizations

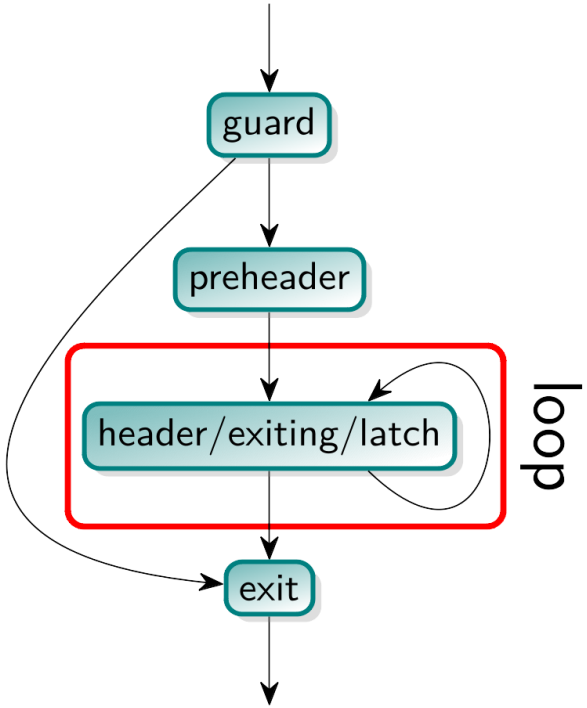
```
const int i = ...;
assert_in_bounds(data, i); ?
while (cond()) {
   data[i] = ...;
}
```

Loop Rotated Form




```
const int i = ...;
if (cond()) { // guard
    // preheader
    do {
        assert_in_bounds(data, i);
        data[i] = ...;
    } while (cond());
}
```

Loop Rotated Form




```
const int i = ...;
if (cond()) { // guard
    // preheader
    assert_in_bounds(data, i);
    do {
        data[i] = ...;
    } while (cond());
}
```

Loop-Invariant Hoisting

```
const int i = ...;
do {
    if (othercond) {
        assert_in_bounds(data, i);
        data[i] = ...;
    }
} while (cond());
```

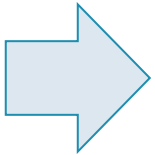
Loop-Invariant Hoisting

```
const int i = ...;
do {
    if (othercond) {
        assert_in_bounds(data, i);
        data[i] = ...;
    }
} while (cond());
```

Loop-variant?
Loop-invariant?

Loop-Invariant Hoisting/Sinking

```
const int i = ...;
do {
 if (othercond) {
  assert_in_bounds(data, i);
  data[i] = ...;
 }
} while (cond());
```

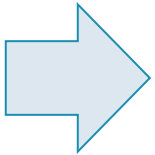


```
const int i = ...;
bool hit = false;
do {
 if (othercond) {
  hit = true;
  data[i] = ...;
 }
} while (cond());
if (hit)
 assert_in_bounds(data, i);
```


Zhang, Yuchen, et al. "Debloating Address Sanitizer." 31st USENIX Security Symposium (USENIX Security 22). 2022.

Loop-Invariant Hoisting/Sinking

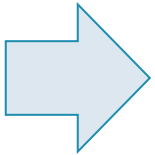
```
const int i = ...;
do {
 if (othercond) {
  assert_in_bounds(data, i);
  data[i] = ...;
 }
} while (cond());
```



```
const int i = ...;
bool hit = false;
do {
 if (othercond) {
  if (!hit)
   assert_in_bounds(data, i);
  hit = true;
  data[i] = ...;
 }
} while (cond());
```

Loop-Invariant Hoisting/Sinking

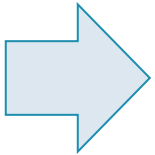
```
const int i = ...;
do {
 if (othercond) {
  assert_in_bounds(data, i);
  data[i] = ...;
 }
} while (cond());
```



```
const int i = ...;
bool safe = inbounds(data, i);
do {
 if (othercond) {
  assert(safe);
  data[i] = ...;
 }
} while (cond());
```

Loop-Invariant Hoisting/Sinking

```
const int i = ...;
do {
 if (othercond) {
  ENTER_TRUSTED;
  data[i] = ...;
  EXIT_TRUSTED;
 }
} while (cond());
```



```
const int i = ...;
ENTER_TRUSTED;
do {
 if (othercond) {
  data[i] = ...;
 }
} while (cond());
EXIT_TRUSTED;
```

Induction Variable-Dependent Instrumentation

- LLVM's ScalarEvolution
- Trip count, recurrence

```
int i = 0;
do {
 assert_in_bounds(data, i);
 data[i] = ...;
 i++;
} while (i < size);
```

Induction Variable-Dependent Instrumentation

- LLVM's ScalarEvolution
- Trip count, recurrence

```
int i = 0;
assert_in_bounds(data, 0);
assert_in_bounds(data, size - 1);
do {
 data[i] = ...;
 i++;
} while (i < size);
```

Induction Variable-Dependent Instrumentation

- LLVM's ScalarEvolution
- Trip count, recurrence

```
int i = 0;
assert_in_bounds(data, 0);
assert_in_bounds(data, size - 1);
do {
 data[i] = ...;
 i++;
} while (i < size);
```

Opaque Loop-Bound Pointer Behavior

```
struct S {
 UChar*   zbits;
 Int32    numZ;
 // ...
};

do {
 s->zbits[s->numZ] = (s->bsBuff >> 24);
 s->numZ++;
 s->bsBuff <<= 8;
 s->bsLive -= 8;
} while (s->bsLive >= 8);
```

Opaque Loop-Bound Pointer Behavior

```
struct S {
 UChar*   zbits;
 Int32    numZ;
 // ...
};

do {
 s->zbits[s->numZ] = (s->bsBuff >> 24);
 s->numZ++;
 s->bsBuff <<= 8;
 s->bsLive -= 8;
} while (s->bsLive >= 8);
```

Opaque Loop-Bound Pointer Behavior

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:
— [a list of relatively sane things]
— a character type.

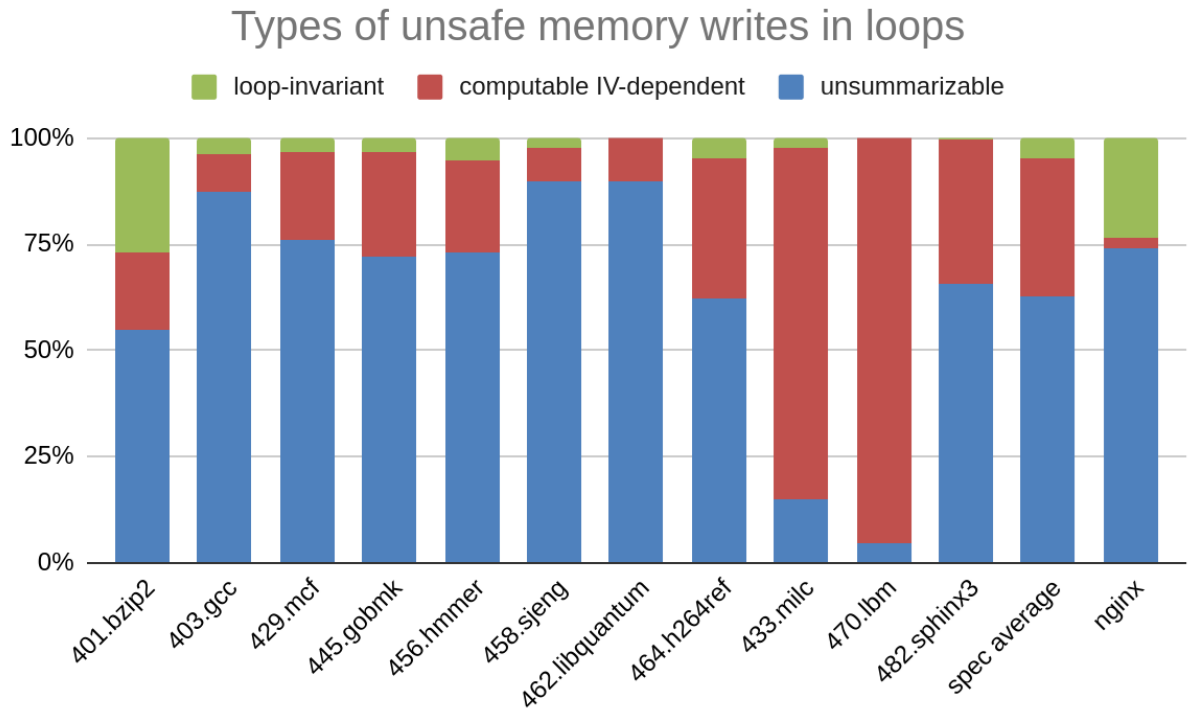
ISO. *ISO/IEC 9899:2011 Information technology --- Programming languages --- C*. pub-ISO, 2011, p. 683 (est.), http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

Opaque Loop-Bound Pointer Behavior

```
struct S {
 UChar*   zbits;
 Int32    numZ;
 // ...
};

do {
 s->zbits[s->numZ] = (s->bsBuff >> 24);
 s->numZ++;
 s->bsBuff <<= 8;
 s->bsLive -= 8;
} while (s->bsLive >= 8);
```

Applicability of Loop Optimizations



Questions?