Divergent Representations: When Compiler Optimizations Enable Exploitation

Andreas D. Kellas*, Alan Cao†, Peter Goodman†, Junfeng Yang*
*Columbia University, †Trail of Bits

*{andreas.kellas, junfeng}@cs.columbia.edu, †{alan.cao, peter}@trailofbits.com

Divergent Representations: Key Takeaways

Can compiler optimizations enable exploitation of existing vulnerabilities?

Divergent Representations: Key Takeaways

Can compiler optimizations enable exploitation of existing vulnerabilities?

We discovered “divergent representations”.

Divergent Representations: Key Takeaways

Can compiler optimizations enable exploitation of existing vulnerabilities?

We discovered “divergent representations”.

Enable exploitation of real software: e.g., SQLite.

Divergent Representations: Key Takeaways

Can compiler optimizations enable exploitation of existing vulnerabilities?

We discovered “divergent representations”.

Enable exploitation of real software: e.g., SQLite.

Common: 45% of scanned projects.

Divergent Representation: Definition

A source code variable compiled so that some of its uses have different semantic representations.

Divergent Representation: Definition

A source code variable compiled so that some of its uses have different semantic representations.

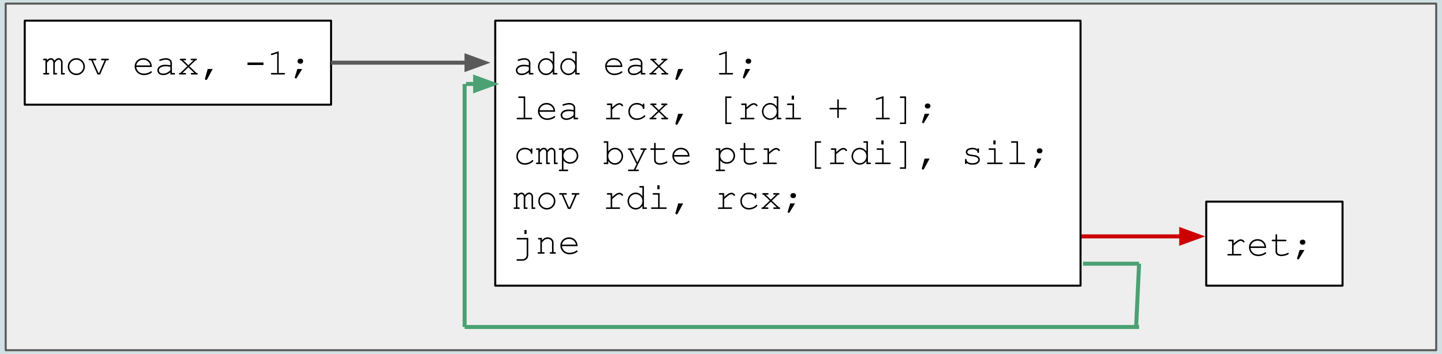
```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

Divergent Representation: Definition

A source code variable compiled so that some of its uses have different semantic representations.

```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

Clang14 -O1



```
mov eax, -1;
```

```
add eax, 1;
lea rcx, [rdi + 1];
cmp byte ptr [rdi], sil;
mov rdi, rcx;
jne
```

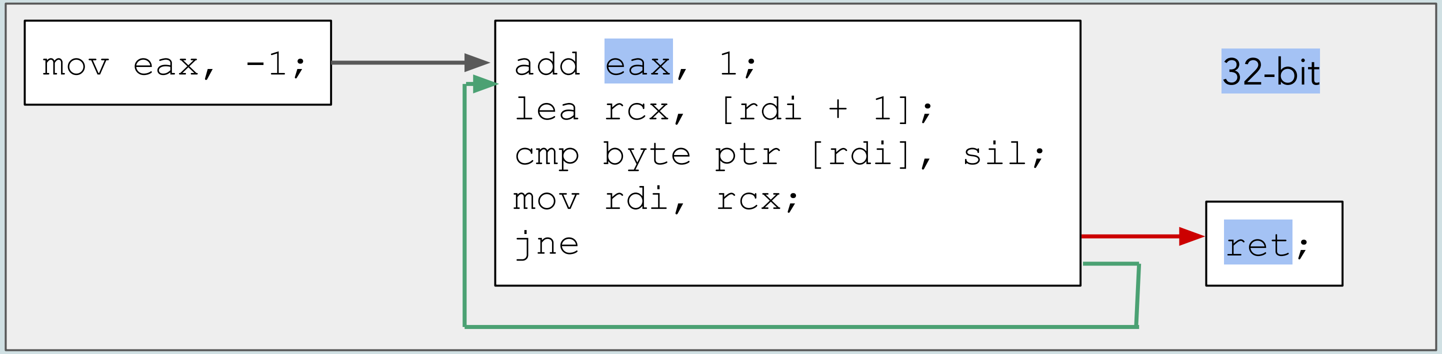
```
ret;
```

Divergent Representation: Definition

A source code variable compiled so that some of its uses have different semantic representations.

```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

Clang14 -O1



```
mov eax, -1;
```

```
add eax, 1;
lea rcx, [rdi + 1];
cmp byte ptr [rdi], sil;
mov rdi, rcx;
jne
```

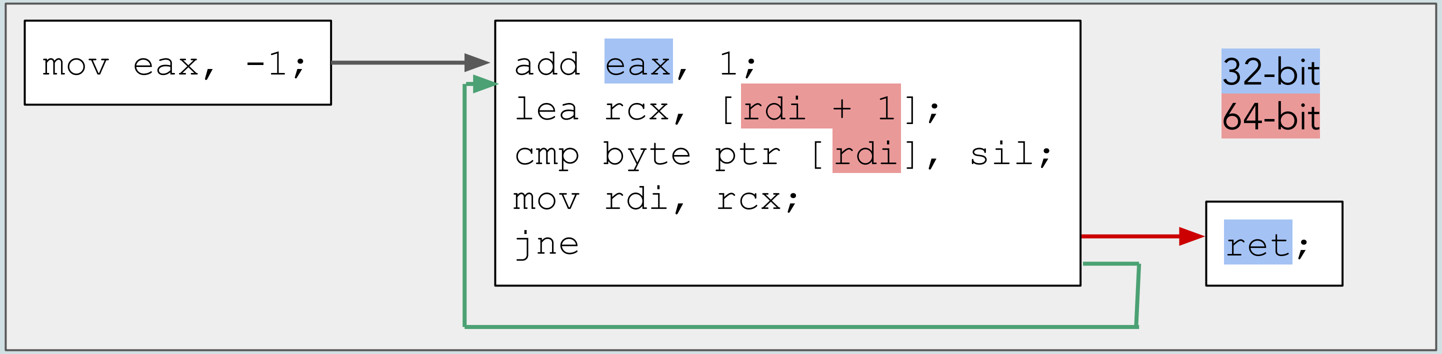
```
ret;
```

Divergent Representation: Definition

A source code variable compiled so that some of its uses have different semantic representations.

```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

Clang14 -O1


```
mov eax, -1;
```

```
add eax, 1;
lea rcx, [rdi + 1];
cmp byte ptr [rdi], sil;
mov rdi, rcx;
jne
```

```
ret;
```

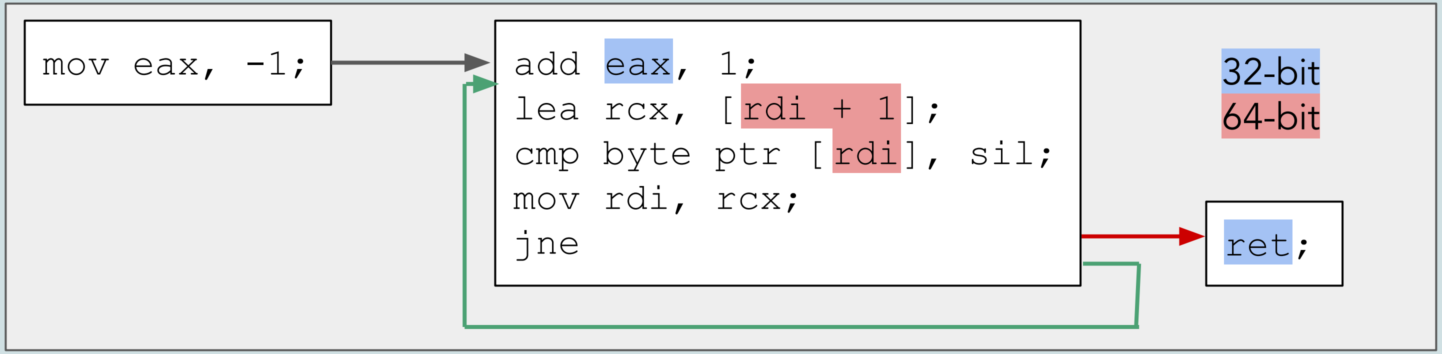
Divergent Representation: Definition

A source code variable compiled so that some of its uses have different semantic representations.

```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

If `i` overflows: divergent values

Clang14 -O1



```
mov eax, -1;
```

```
add eax, 1;
lea rcx, [rdi + 1];
cmp byte ptr [rdi], sil;
mov rdi, rcx;
jne
```

```
ret;
```

Previous work [1,2] showed:

compiler optimizations + undefined behavior = unexpected vulnerabilities

[1] Wang et al., “Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior”, SOSP, 2013.
[2] D’Silva et al., “The Correctness-Security Gap in Compiler Optimization”, IEEE S&P, 2015.

Previous work [1,2] showed:

compiler optimizations + undefined behavior = unexpected vulnerabilities

```
if (buf + i < buf)
    return;
buf[i] = '\0';
```

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
    return POLLERR;
```

[1] Wang et al., “Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior”, SOSP, 2013.
[2] D’Silva et al., “The Correctness-Security Gap in Compiler Optimization”, IEEE S&P, 2015.

Previous work [1,2] showed:

compiler optimizations + undefined behavior = unexpected vulnerabilities

```
if (buf + i < buf)
     return;
buf[i] = '\0';
```

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
    return POLLERR;
```

True iff undefined behavior occurs.

[1] Wang et al., “Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior”, SOSP, 2013.
[2] D’Silva et al., “The Correctness-Security Gap in Compiler Optimization”, IEEE S&P, 2015.

Previous work [1,2] showed:

compiler optimizations + undefined behavior = unexpected vulnerabilities

```
~~if (buf + i < buf)~~
     ~~return;~~
buf[i] = '\0';
```

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
~~if (!tun)~~
    ~~return POLLERR;~~
```

True iff undefined behavior occurs.

[1] Wang et al., "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior", SOSP, 2013.
[2] D'Silva et al., "The Correctness-Security Gap in Compiler Optimization", IEEE S&P, 2015.

Previous work [1,2] showed:

compiler optimizations + undefined behavior = unexpected vulnerabilities

```
~~if (buf + i < buf)~~
    ~~return;~~
buf[i] = '\0';
```

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
~~if (!tun)~~
    ~~return POLLERR;~~
```

This work: benign patterns in compiled code to exploit existing vulnerabilities.

Previous work [1,2] showed:

compiler optimizations + undefined behavior = unexpected vulnerabilities

```
~~if (buf + i < buf)~~
    ~~return;~~
buf[i] = ‘\0’;
```

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
~~if (!tun)~~
    ~~return POLLERR;~~
```

This work: benign patterns in compiled code to exploit existing vulnerabilities.

Similar to ROP gadgets.

Case Study: SQLite CVE-2022-35737

Vulnerability (7.5 CVSS): signed integer overflow creates stack buffer overflow.

Case Study: SQLite CVE-2022-35737

Vulnerability (7.5 CVSS): signed integer overflow creates stack buffer overflow.

Exploit: overwrite saved return address and return.

Case Study: SQLite CVE-2022-35737

Vulnerability (7.5 CVSS): signed integer overflow creates stack buffer overflow.

Exploit: overwrite saved return address and return.

- Requires precise data conditions.

Case Study: SQLite CVE-2022-35737

Vulnerability (7.5 CVSS): signed integer overflow creates stack buffer overflow.

Exploit: overwrite saved return address and return.

- Requires precise data conditions.

Conditions only satisfiable because of a divergent representation.

Case Study: SQLite divrep helps exploit buffer overflow

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != ‘\0’; len++) {
     if (input[len] == quote) nquotes++;
     while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
     memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

1. Scan input string: count quotes and total number of bytes.

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
     if (input[len] == quote) nquotes++;
     while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
     memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

1. Scan input string: count quotes and total number of bytes.

2. Check: output string fits in stack-allocated buffer.

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

1. Scan input string: count quotes and total number of bytes.

2. Check: output string fits in stack-allocated buffer.
   - CAN OVERFLOW

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes  <= BUF_SIZE)
    memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

1. Scan input string: count quotes and total number of bytes.

2. Check: output string fits in stack-allocated buffer.
   - CAN OVERFLOW

3. Copy: input string to stack-allocated buffer, add escape characters.

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

1. Scan input string: count quotes and total number of bytes.

2. Check: output string fits in stack-allocated buffer.
   - CAN OVERFLOW

3. Copy: input string to stack-allocated buffer, add escape characters.
   - STACK BUFFER OVERFLOW

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

Exploit conditions:

`len + nquotes` must overflow
`len` must be small during memcpy

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input,  len);
```

Case Study: SQLite divrep helps exploit buffer overflow

Exploit conditions:

`len + nquotes` must overflow

`len` must be small during memcpy

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input, len);
```

Case Study: SQLite divrep helps exploit buffer overflow

Exploit conditions:

`len + nquotes` must overflow

`len` must be small during memcpy

Problem: `len < nquotes` →
`len` must overflow →
negative memory index.

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input,  len);
```

Case Study: SQLite divrep helps exploit buffer overflow

Exploit conditions:

`len + nquotes` must overflow

`len` must be small during memcpy

Problem: `len < nquotes` →

`len` must overflow →

negative memory index.

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[ len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input,  len);
```

🟦 32-bit

🟥 64-bit

Case Study: SQLite divrep helps exploit buffer overflow

Exploit conditions:

`len + nquotes` must overflow

`len` must be small during memcpy

Problem: `len < nquotes` →

`len` must overflow →

negative memory index.

Key insight: increment `len` with different semantics to meet conditions.

```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[ len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input,  len);
```

32-bit

64-bit

Case Study: SQLite divrep helps exploit buffer overflow

Exploit conditions:
`len + nquotes` must overflow
`len` must be small during memcpy

Problem: `len < nquotes` →
`len` must overflow →
negative memory index.

Key insight: increment `len` with different semantics to meet conditions.

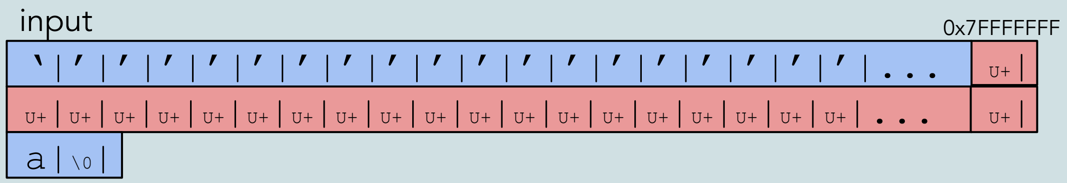
E.g., avoid negative memory offsets by using unicode characters to increment `len` with 64-bit semantics whenever a 32-bit value is undesirable.

32-bit
64-bit

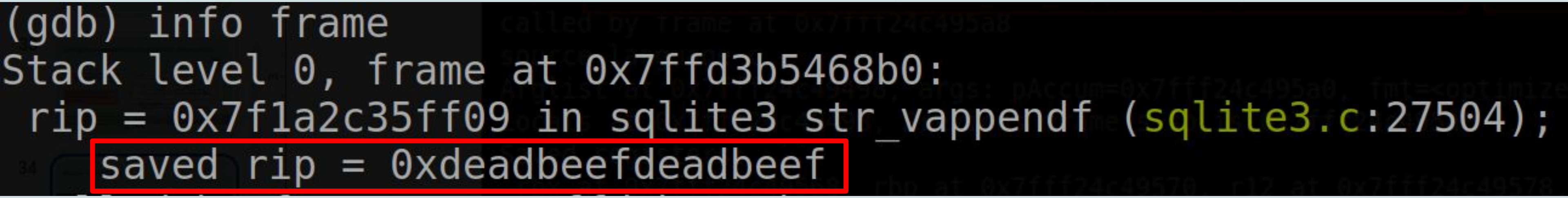
```
int len, nspecial;
char output[BUF_SIZE];

for (i=0; input[len] != '\0'; len++) {
    if (input[len] == quote) nquotes++;
    while (unicode_prefix(input[ len])) len++;
}

if (len + nquotes <= BUF_SIZE)
    memcpy_and_escape(output, input, len);
```



Case Study: SQLite divrep helps exploit buffer overflow



```
(gdb) info frame
Stack level 0, frame at 0x7ffd3b5468b0:
 rip = 0x7f1a2c35ff09 in sqlite3 str_vappendf (sqlite3.c:27504);
   saved rip = 0xdeadbeefdeadbeef
```

*Canaries not considered.

Divergent Representations: How common are they?

Divergent Representations: How common are they?

Source code search - C/C++

CodeQL queries for patterns that may be optimized with integer widening.

Divergent Representations: How common are they?

Source code search - C/C++

CodeQL queries for patterns that may be optimized with integer widening.

Binary code search

Binary Ninja plugins to identify instances of different register sizes and semantics for same variable.

Divergent Representations: How common are they?

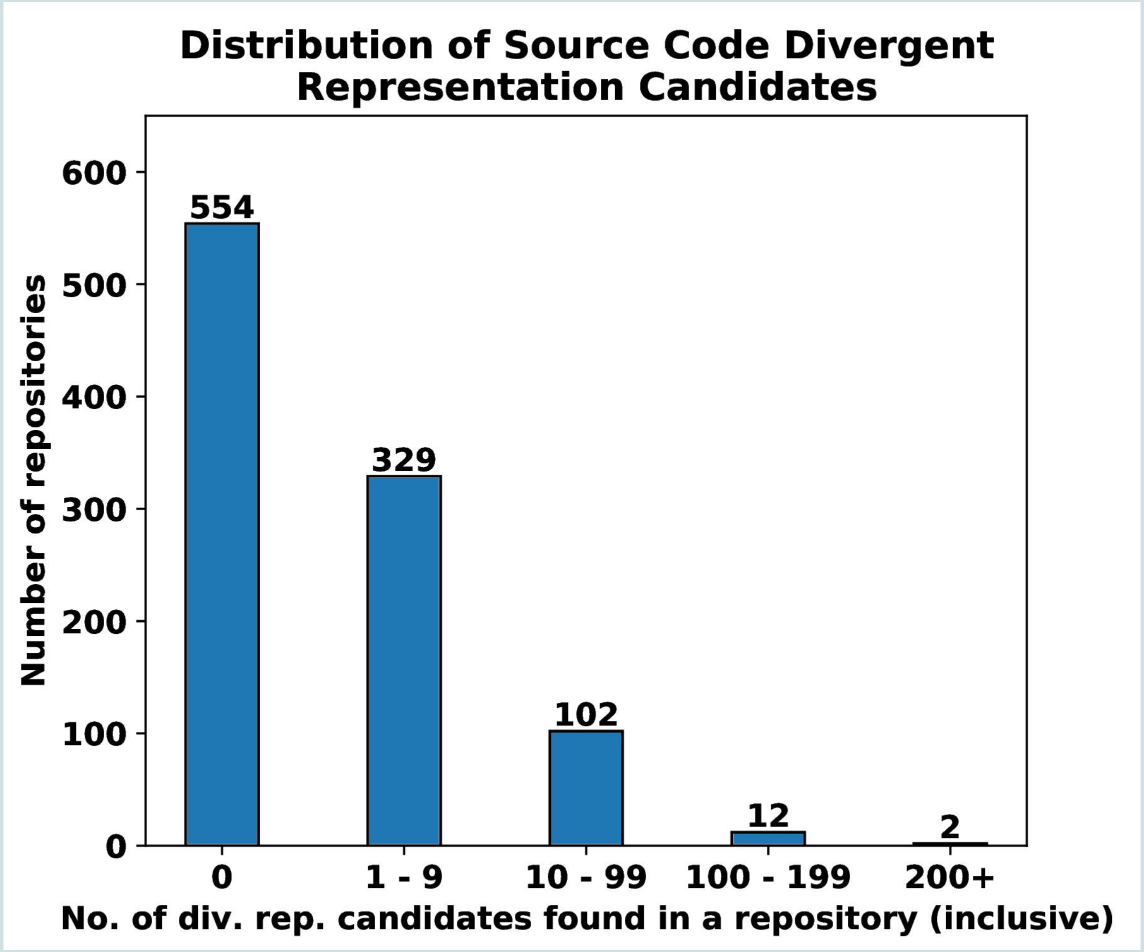
Source code search - C/C++

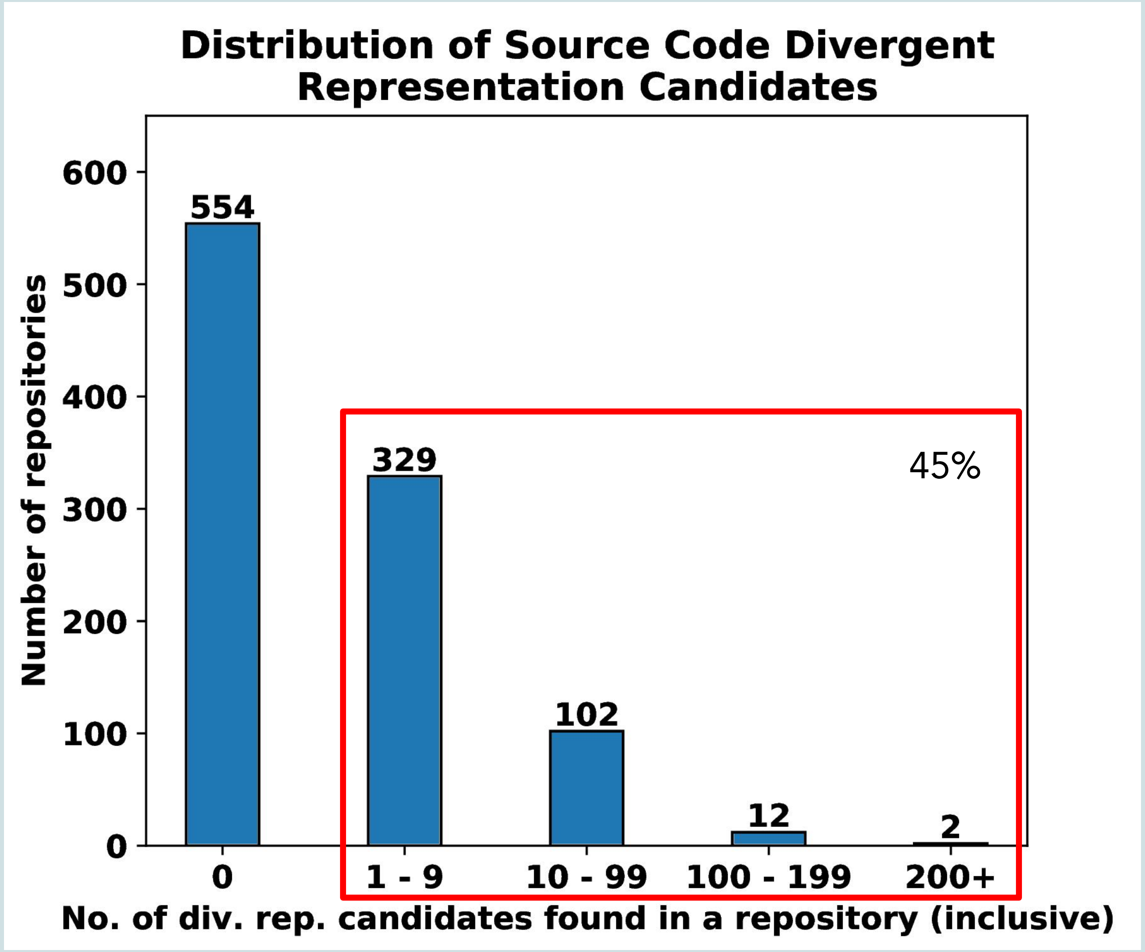
CodeQL queries for patterns that may be optimized with integer widening.

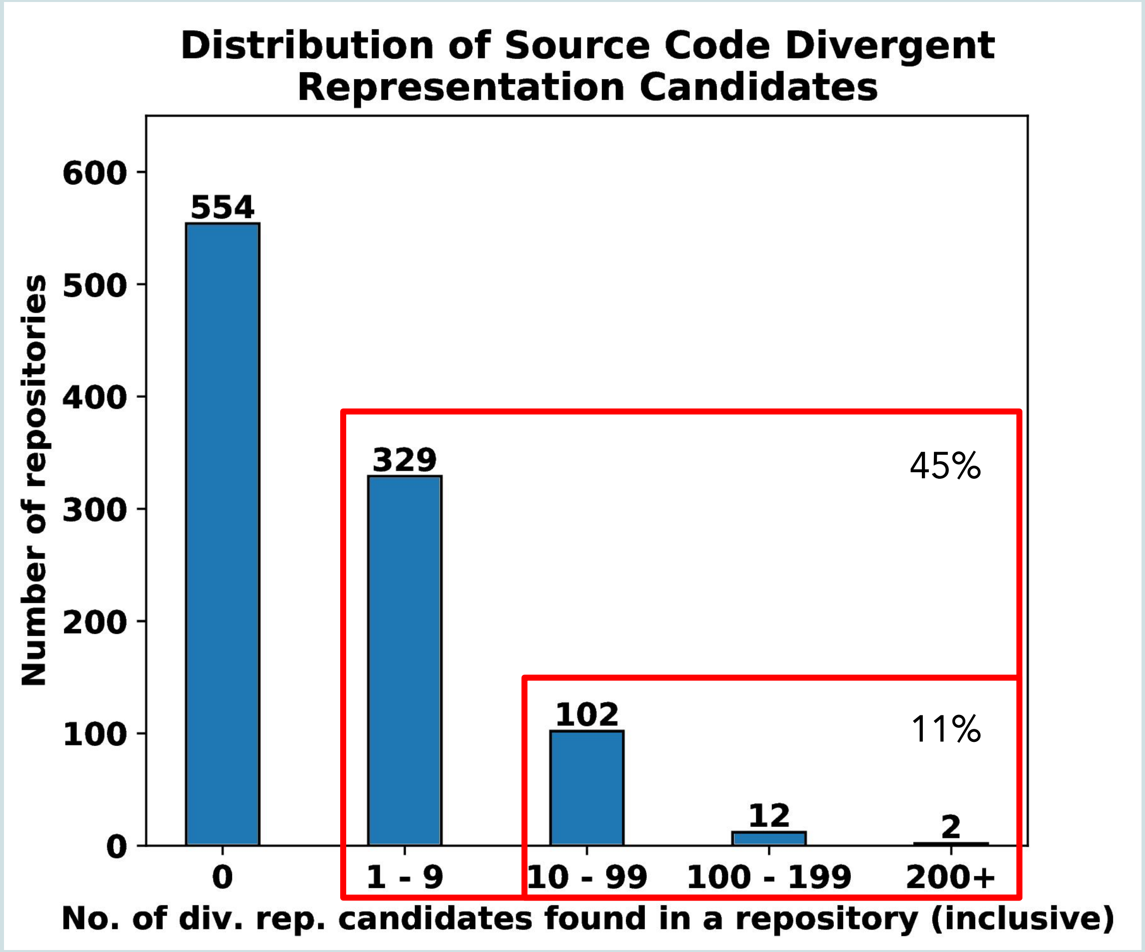
Binary code search

Binary Ninja plugins to identify instances of different register sizes and semantics for same variable.

Counts are under-approximations: other forms of divergent representations may exist.


Distribution of Source Code Divergent Representation Candidates
Number of repositories
600
500
400
300
200
100
0
554
329
102
12
2
0
1 - 9
10 - 99
100 - 199
200+
No. of div. rep. candidates found in a repository (inclusive)


Distribution of Source Code Divergent Representation Candidates
Number of repositories
600
500
400
300
200
100
0
554
329
102
12
2
0
1 - 9
10 - 99
100 - 199
200+
No. of div. rep. candidates found in a repository (inclusive)
45%


Distribution of Source Code Divergent Representation Candidates
Number of repositories
600
500
400
300
200
100
0
554
329
102
12
2
45%
11%
0
1 - 9
10 - 99
100 - 199
200+
No. of div. rep. candidates found in a repository (inclusive)

libsqlite3.so: # of divergent representations in compiled program

Optimization Level	Clang	GCC
-O0	0	0
-O1	23	33
-O2	26	37
-O3	30	53

Divergent Representations: Can they be prevented?

Divergent Representations: Can they be prevented?

Disable optimizations?

Divergent Representations: Can they be prevented?

Disable optimizations?

- Unsatisfying: we want performant programs.

Divergent Representations: Can they be prevented?

Disable optimizations?

- Unsatisfying: we want performant programs.

Optimizations should be correct, performant

Divergent Representations: Can they be prevented?

Disable optimizations?

- Unsatisfying: we want performant programs.

Optimizations should be correct, performant, and not aid attackers.

Divergent Representations: Can they be prevented?

Disable optimizations?

- Unsatisfying: we want performant programs.

Optimizations should be correct, performant, and not aid attackers

```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

```
size_t i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

Divergent Representations: Can they be prevented?

Disable optimizations?

- Unsatisfying: we want performant programs.

Optimizations should be correct, performant, and not aid attackers

```
int i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

```
size_t i;
for (i=0; buf[i] != ch; i++) {}
return i;
```

Our tools should reason about divergent representations:

- Source code: linters
- Binary: decompilers

Divergent Representations: Summary

A source code variable compiled so that some of its uses have different semantic representations.

Divergent Representations: Summary

A source code variable compiled so that some of its uses have different semantic representations.

- Enable exploits: e.g., SQLite
- Common: 45% of C/C++ projects

Divergent Representations: Summary

A source code variable compiled so that some of its uses have different semantic representations.

- Enable exploits: e.g., SQLite
- Common: 45% of C/C++ projects

Benign in isolation, but dangerous with a vulnerability.

Divergent Representations: Summary

A source code variable compiled so that some of its uses have different semantic representations.

- Enable exploits: e.g., SQLite
- Common: 45% of C/C++ projects

Benign in isolation, but dangerous with a vulnerability.

- Must understand causes and risks.
- Ought to prevent when acceptable.