PICO: A Presburger In-bounds Check Optimization for Compiler-based Memory Safety Instrumentations

Tina Jung, Fabian Ritter, Sebastian Hack

Compiler Design Lab
Saarland University
https://compilers.cs.uni-saarland.de

UNIVERSITÄT DES SAARLANDES

SIC Saarland Informatics Campus

Memory Safety in C

What is memory safety?

Memory Safety in C

What is memory safety?

memory safe $\Leftrightarrow$ no spatial or temporal errors

Memory Safety in C

What is memory safety?

memory safe ⇔ no spatial or temporal errors

Spatial

```
int Ar[15];
Ar[20] = ...;
```

Memory Safety in C

What is memory safety?

memory safe ⇔ no spatial or temporal errors

Spatial

```
int Ar[15];
Ar[20] = ...;
```

Temporal

```
int *Ar = malloc(15 * sizeof(int));
free(Ar);
Ar[0] = ...;
```

Memory Safety: Still an issue?

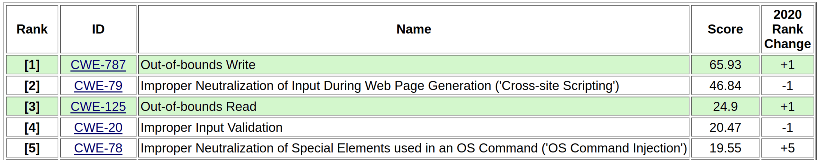
2021

© The MITRE Corporation

Memory Safety: Still an issue?

MOST DANGEROUS Top 25 CWE™ SOFTWARE WEAKNESSES

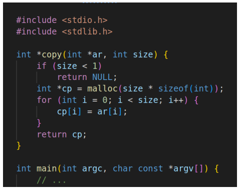
2021



Rank	ID	Name	Score	2020 Rank Change
[1]	CWE-787	Out-of-bounds Write	65.93	+1
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84	-1
[3]	CWE-125	Out-of-bounds Read	24.9	+1
[4]	CWE-20	Improper Input Validation	20.47	-1
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55	+5

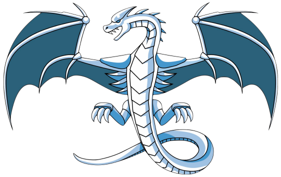
Compiler-based Memory Safety Instrumentations

C Program



```
#include <stdio.h>
#include <stdlib.h>

int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}

int main(int argc, char const *argv[]) {
    // ...
```

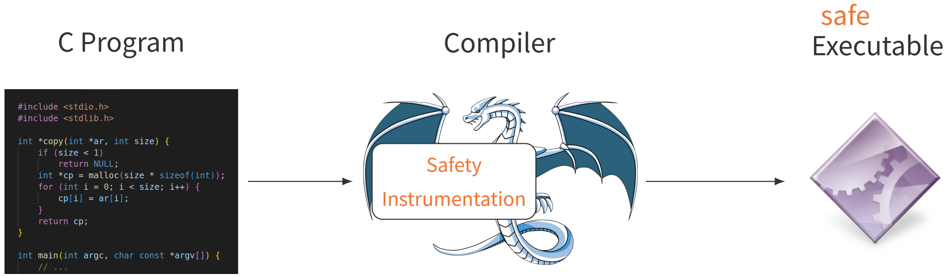
Compiler



© Apple Inc.

Executable

Compiler-based Memory Safety Instrumentations

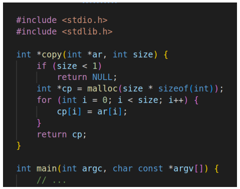


```c
#include <stdio.h>
#include <stdlib.h>

int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}

int main(int argc, char const *argv[]) {
    // ...
```

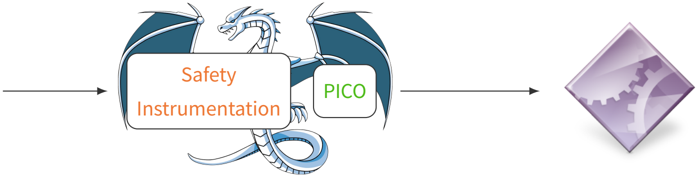
© Apple Inc.

Compiler-based Memory Safety Instrumentations

C Program



```
#include <stdio.h>
#include <stdlib.h>

int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}

int main(int argc, char const *argv[]) {
    // ...
```



© Apple Inc.

Compiler-based Memory Safety Instrumentations

Where does the overhead come from?

Compiler-based Memory Safety Instrumentations

Where does the overhead come from?

```
int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}
```

Compiler-based Memory Safety Instrumentations

Where does the overhead come from?

```
int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}
```

Instrumentation

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    intptr_t cp_bs = get_base(cp);
    intptr_t cp_bnd = get_bound(cp);
    for (int i = 0; i < size; i++) {
        check_ib(ar+i, sizeof(int), ar_bs, ar_bnd);
        check_ib(cp+i, sizeof(int), cp_bs, cp_bnd);
        cp[i] = ar[i];
    }
    return cp;
}
```

Compiler-based Memory Safety Instrumentations

Where does the overhead come from?

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    intptr_t cp_bs = get_base(cp);
    intptr_t cp_bnd = get_bound(cp);
    for (int i = 0; i < size; i++) {
        check_ib(ar+i, sizeof(int), ar_bs, ar_bnd);
        check_ib(cp+i, sizeof(int), cp_bs, cp_bnd);
        cp[i] = ar[i];
    }
    return cp;
}
```

Compiler-based Memory Safety Instrumentations

Where does the overhead come from?

```c
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    intptr_t cp_bs = get_base(cp);
    intptr_t cp_bnd = get_bound(cp);
    for (int i = 0; i < size; i++) {
        check_ib(ar+i, sizeof(int), ar_bs, ar_bnd);
        check_ib(cp+i, sizeof(int), cp_bs, cp_bnd);
        cp[i] = ar[i];
    }
    return cp;
}
```

`max(0, 2·size)` check_ib calls

→ Incurs high runtime overhead

Compiler-based Memory Safety Instrumentations

What can PICO do about it?

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    intptr_t cp_bs = get_base(cp);
    intptr_t cp_bnd = get_bound(cp);
    for (int i = 0; i < size; i++) {
        check_ib(ar+i, sizeof(int), ar_bs, ar_bnd);
        check_ib(cp+i, sizeof(int), cp_bs, cp_bnd);
        cp[i] = ar[i];
    }
    return cp;
}
```

Compiler-based Memory Safety Instrumentations

What can PICO do about it?

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    intptr_t cp_bs = get_base(cp);
    intptr_t cp_bnd = get_bound(cp);
    for (int i = 0; i < size; i++) {
        check_ib(ar+i, sizeof(int), ar_bs, ar_bnd);
        check_ib(cp+i, sizeof(int), cp_bs, cp_bnd);
        cp[i] = ar[i];
    }
    return cp;
}
```

$\xrightarrow{\text{PICO}}$

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    abortifn((ar_bs - ar) <= 0 &&
             (ar_bnd - ar) >= sizeof(int) * size);
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}
```

Compiler-based Memory Safety Instrumentations

What can PICO do about it?

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    intptr_t cp_bs = get_base(cp);
    intptr_t cp_bnd = get_bound(cp);
    for (int i = 0; i < size; i++) {
        check_ib(ar+i, sizeof(int), ar_bs, ar_bnd);
        check_ib(cp+i, sizeof(int), cp_bs, cp_bnd);
        cp[i] = ar[i];
    }
    return cp;
}
```

$\xrightarrow{\text{PICO}}$

```
int *copy(int *ar, int size) {
    intptr_t ar_bs = get_base(ar);
    intptr_t ar_bnd = get_bound(ar);
    if (size < 1)
        return NULL;
    abortifn((ar_bs - ar) <= 0 &&
             (ar_bnd - ar) >= sizeof(int) * size);
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}
```

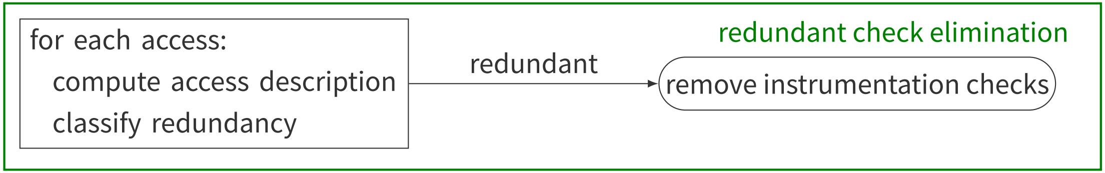
→ 0 or 1 check(s), loop trip count independent

PICO Overview

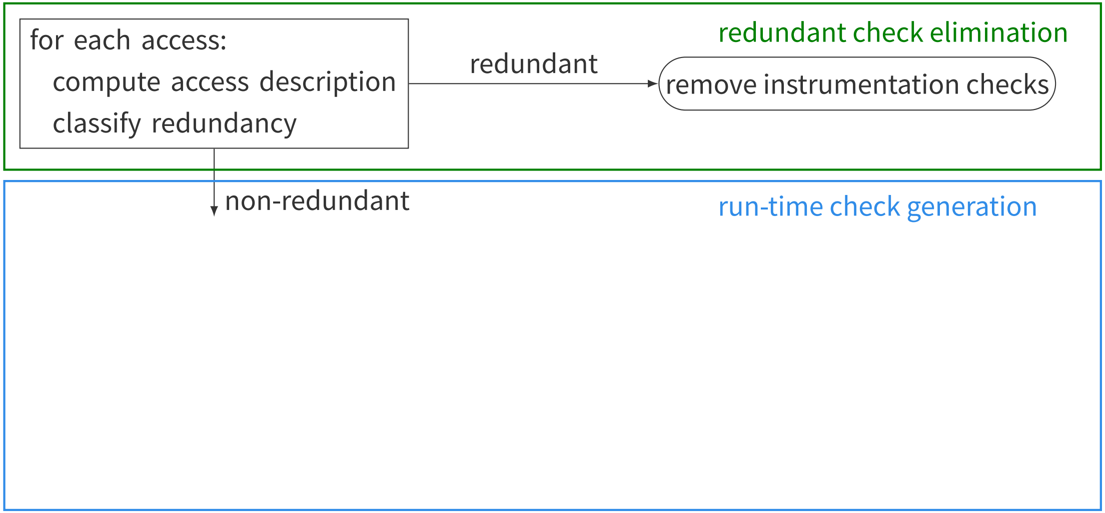
```
for each access:
  compute access description
  classify redundancy
```

redundant check elimination

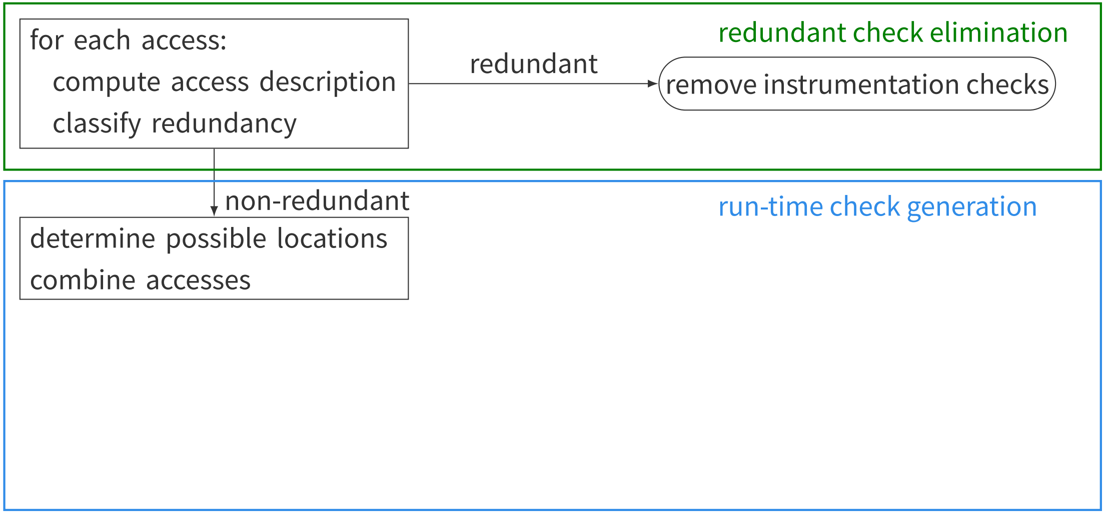
PICO Overview


redundant check elimination
for each access:
compute access description
classify redundancy
redundant
remove instrumentation checks

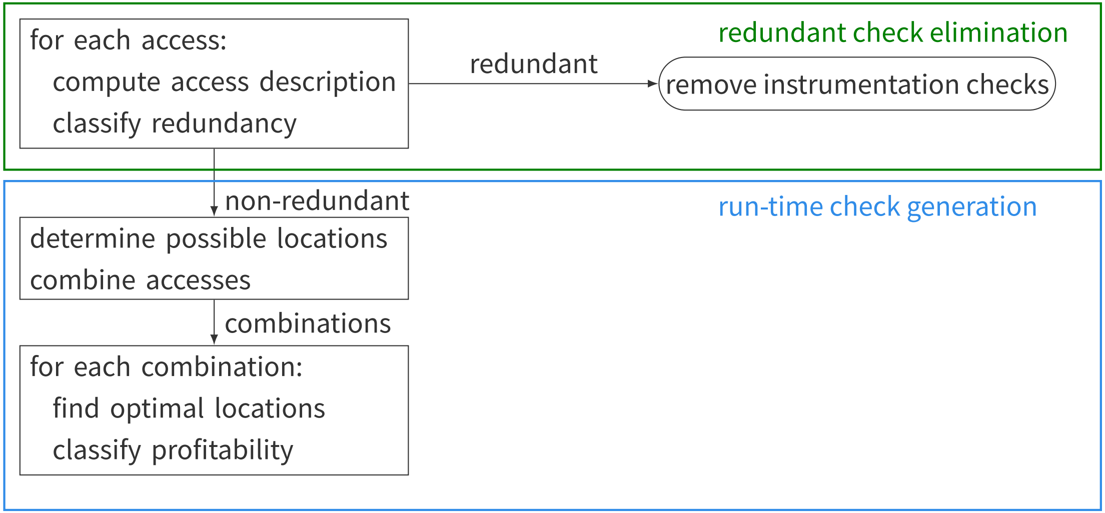
PICO Overview


redundant check elimination
for each access:
compute access description
classify redundancy
redundant
remove instrumentation checks
non-redundant
run-time check generation

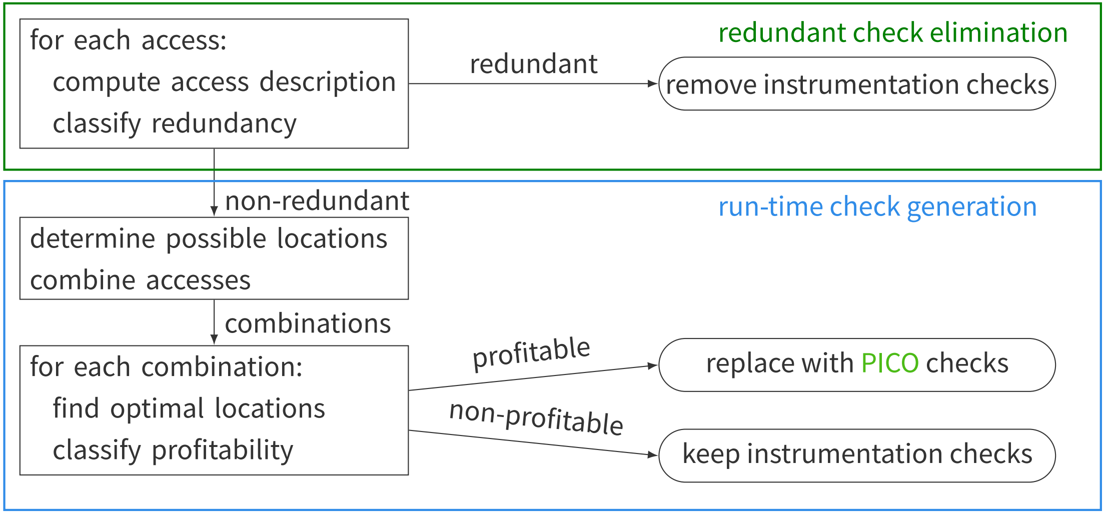
PICO Overview


redundant check elimination
for each access:
compute access description
classify redundancy
redundant
remove instrumentation checks
non-redundant
run-time check generation
determine possible locations
combine accesses

PICO Overview



PICO Overview



Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

```
...
int *cp = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    cp[i] = ...;
}
...
```

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

$$\{Mem \mid \overbrace{cp \leq Mem < cp + size}^{\text{accessible}}\}$$

```
...
int *cp = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    cp[i] = ...;
}
...
```

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

```
...
int *cp = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    cp[i] = ...;
}
...
```

$$\{Mem \mid \overbrace{cp \leq Mem < cp + size}^{\text{accessible}}\}$$

$$\{Mem \mid Mem = \underbrace{cp + i}_{\text{accessed}} \wedge \underbrace{0 \leq i < size}_{\text{derived from loop}}\}$$

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

$$\{Mem \mid \overbrace{cp \leq Mem < cp + size}^{\text{accessible}}\}$$

```
...
int *cp = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    cp[i] = ...;
}
...
```

$$\{Mem \mid Mem = \underbrace{cp + i}_{\text{accessed}} \wedge \underbrace{0 \leq i < size}_{\text{derived from loop}}\}$$

$\rightarrow$ canonical way to determine in-bounds conditions from such descriptions

Possible Check Locations

Considered: Dominating Locations

Possible Check Locations

Considered: Dominating Locations

```
int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}
```

Possible Check Locations

Considered: Dominating Locations

```
int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = get_mem(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}
```

Possible Check Locations

Considered: Dominating Locations

```
  int *copy(int *ar, int size) {
      if (size < 1)
          return NULL;
      int *cp = get_mem(size * sizeof(int));
→     for (int i = 0; i < size; i++) {
→         cp[i] = ar[i];
      }
      return cp;
  }
```

Possible Check Locations

```
 int *copy(int *ar, int size) {
     if (size < 1)
         return NULL;
     int *cp = get_mem(size * sizeof(int));
→    for (int i = 0; i < size; i++) {
→        cp[i] = ar[i];
     }
     return cp;
 }
```

Considered: Dominating Locations

Limitations:

- Value from call/memory
- Possible no-return call
- No loop bounds available
- Non-affine control-flow, e.g. `if (a*b > x)...`

Combined Checks

Which accesses to check together?

```
void init(int *ar, int size) {
    for (int i = 0; i < size; i += 2) {
        if (i+1 < size) {
            ar[i+1] = 1;
        }
        ar[i] = 0;
    }
}
```

Combined Checks

Which accesses to check together?

- Same array

```
void init(int *ar, int size) {
    for (int i = 0; i < size; i += 2) {
        if (i+1 < size) {
            ar[i+1] = 1;
        }
        ar[i] = 0;
    }
}
```

Combined Checks

Which accesses to check together?

- Same array

```
void init(int *ar, int size) {
    for (int i = 0; i < size; i += 2) {
        if (i+1 < size) {
→           ar[i+1] = 1;
        }
→       ar[i] = 0;
    }
}
```

Combined Checks

Which accesses to check together?

```
void init(int *ar, int size) {
    for (int i = 0; i < size; i += 2) {
        if (i+1 < size) {
            ar[i+1] = 1;
        }
        ar[i] = 0;
    }
}
```

- Same array
- Common possible check location

Combined Checks

Which accesses to check together?

```
void init(int *ar, int size) {
    for (int i = 0; i < size; i += 2) {
        if (i+1 < size) {
            ar[i+1] = 1;
        }
        ar[i] = 0;
    }
}
```

- Same array
- Common possible check location
- Optimal: Exponential → greedy, combine as many as possible

Cost Function

Cost to place a **C**heck at a **L**ocation:

$$cost(L, C) := execFrequency(L) \cdot complexity(L, C)$$

Cost Function

Cost to place a **C**heck at a **L**ocation:

$$cost(L, C) := execFrequency(L) \cdot complexity(L, C)$$

Relative to other locations

Cost Function

Cost to place a **C**heck at a **L**ocation:

$$cost(L, C) := \underbrace{execFrequency(L)}_{\text{Relative to other locations}} \cdot \underbrace{complexity(L, C)}_{\text{Number of operations in the check}}$$

Cost Function

Cost to place a **C**heck at a **L**ocation:

$$cost(L, C) := execFrequency(L) \cdot complexity(L, C)$$

Relative to other locations

Number of operations in the check

```
if (size < 1)
  return NULL;
abortifn(access_is_ib);
```

Cost Function

Cost to place a **C**heck at a **L**ocation:

$$cost(L, C) := \underset{\substack{\uparrow \\ \text{Relative to other locations}}}{execFrequency(L)} \cdot \underset{\substack{\uparrow \\ \text{Number of operations in the check}}}{complexity(L, C)}$$

```
abortifn(size < 1 || access_is_ib);
if (size < 1)
  return NULL;
abortifn(access_is_ib);
```

Check Placement: Location and Profitability

Which location and check to use?

Check Placement: Location and Profitability

Which location and check to use?

- Use Minimum Cut to choose a location L_p

Check Placement: Location and Profitability

Which location and check to use?

- Use Minimum Cut to choose a location L_p
- Instrumentation check always at the access location L_a

Check Placement: Location and Profitability

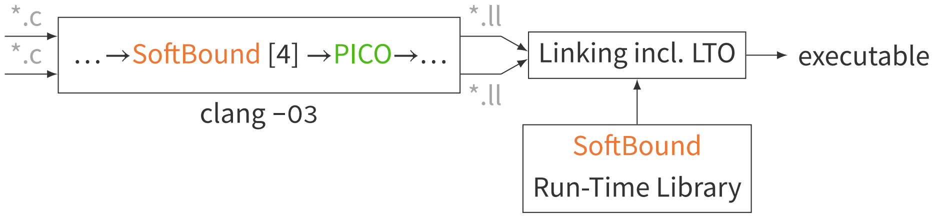
Which location and check to use?

- Use Minimum Cut to choose a location L_p
- Instrumentation check always at the access location L_a

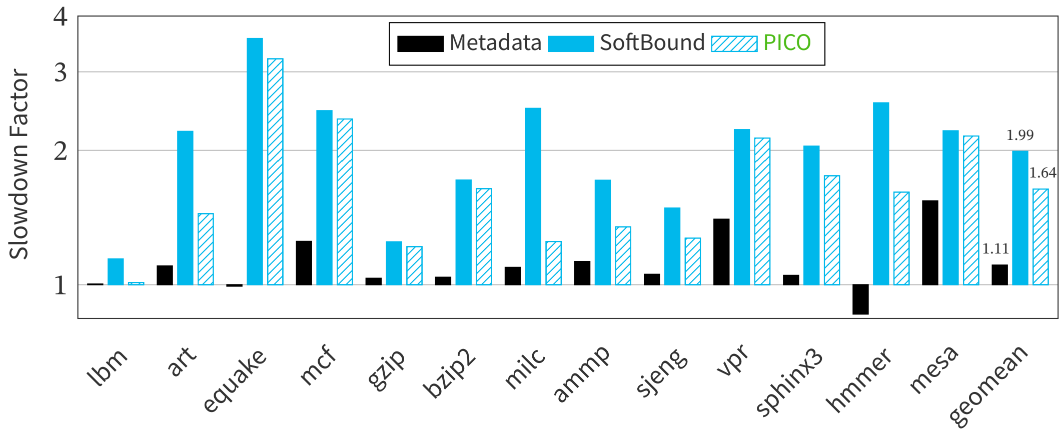
→ Profitable if:

$$cost(L_p, PICO\ check) < cost(L_a, Instrumentation\ check)$$

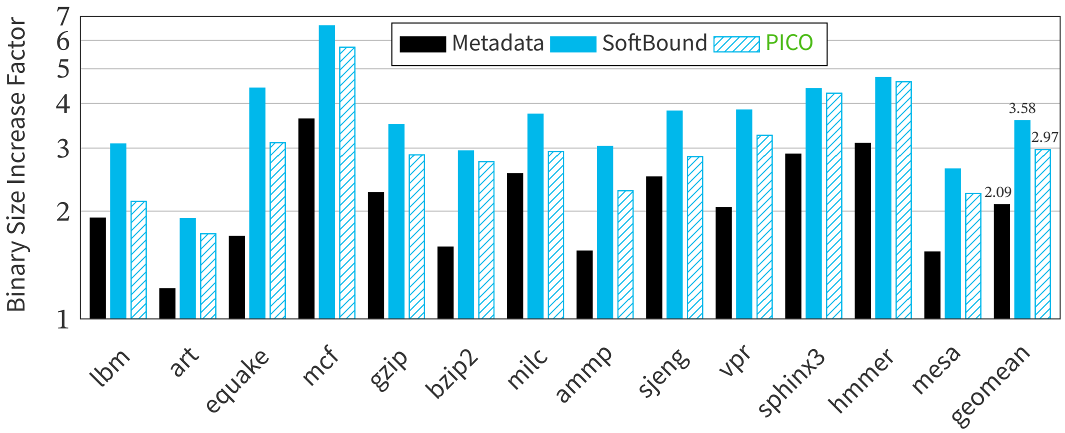
Evaluation: Setup



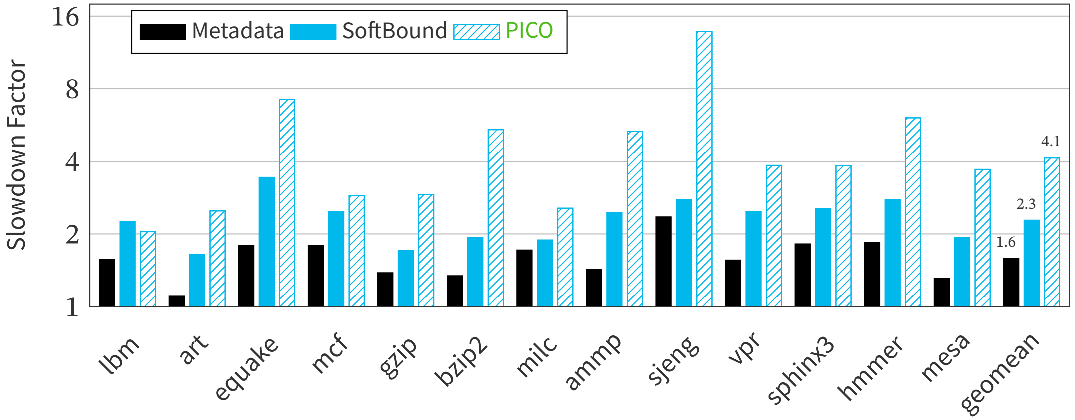
Evaluation: Runtime



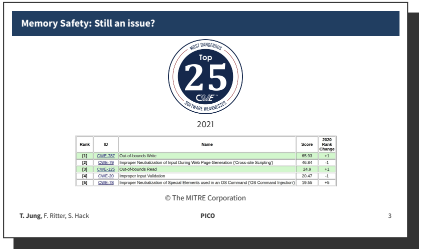
Evaluation: Binary Size



Evaluation: Compile-Time



Summary



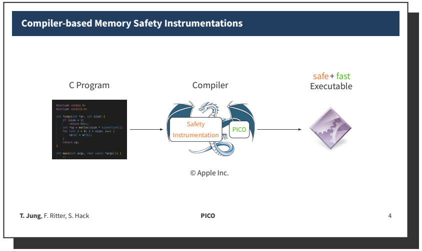
Memory Safety: Still an issue?

2021

Rank	ID	Name	Score	2020 Rank Change
[1]	CWE-787	Out-of-bounds Write	65.93	+1
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84	-1
[3]	CWE-125	Out-of-bounds Read	24.9	+1
[4]	CWE-20	Improper Input Validation	20.47	-1
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55	+5

© The MITRE Corporation

T. Jung, F. Ritter, S. Hack **PICO** 3



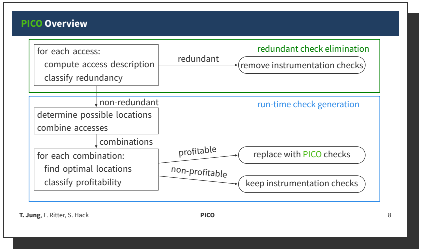
Compiler-based Memory Safety Instrumentations

C Program

```
#include <stdio.h>
#include <stdlib.h>

int *copy(int *ar, int size) {
    if (size < 1)
        return NULL;
    int *cp = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        cp[i] = ar[i];
    }
    return cp;
}

int main(int argc, char const *argv[]) {
    // ...
```

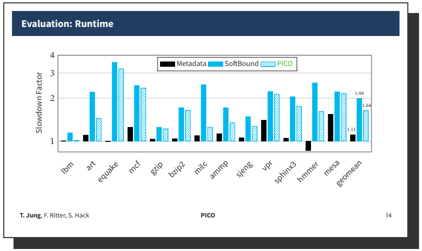
© Apple Inc.

T. Jung, F. Ritter, S. Hack **PICO** 4



PICO Overview

T. Jung, F. Ritter, S. Hack **PICO** 8



Evaluation: Runtime

T. Jung, F. Ritter, S. Hack **PICO** 14

References i

LLVM Wyvern: © Apple Inc. `https://llvm.org/Logo.html`

CWE Logo: © The MITRE Corporation `https://cwe.mitre.org/about/termsofuse.html`

[1] *Polly - Performing polyhedral optimizations on a low-level intermediate representation* by Tobias Grosser, Armin Groesslinger, Christian Lengauer in Parallel Processing Letters 2012

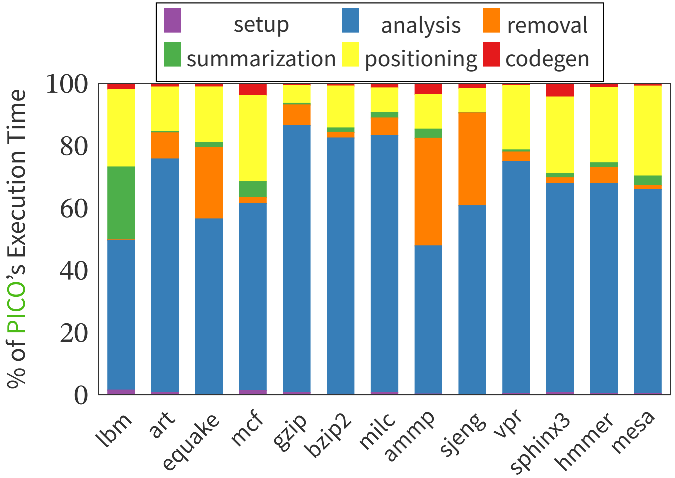
Polly Website: `https://polly.llvm.org/`

[2] SCEV: `https://llvm.org/doxygen/ScalarEvolution_8cpp_source.html`

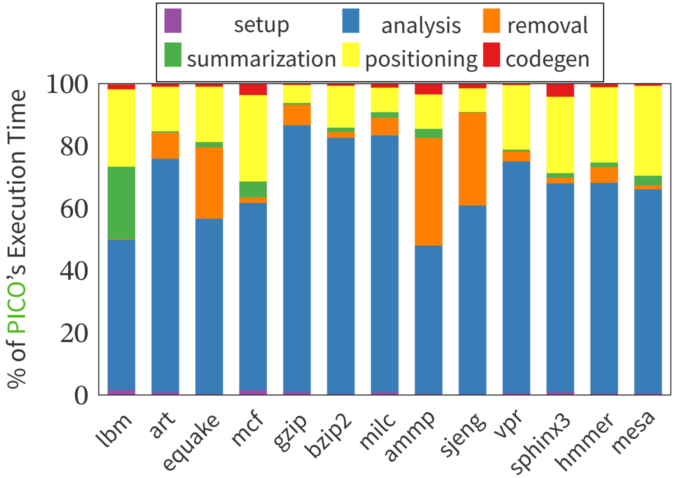
[3] *isl: An Integer Set Library for the Polyhedral Model* by Sven Verdoolaege in ICMS 2010

[4] *SoftBound: Highly Compatible and Complete Spatial Memory Safety for C* by Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic in PLDI ’09

Compile-time Attribution

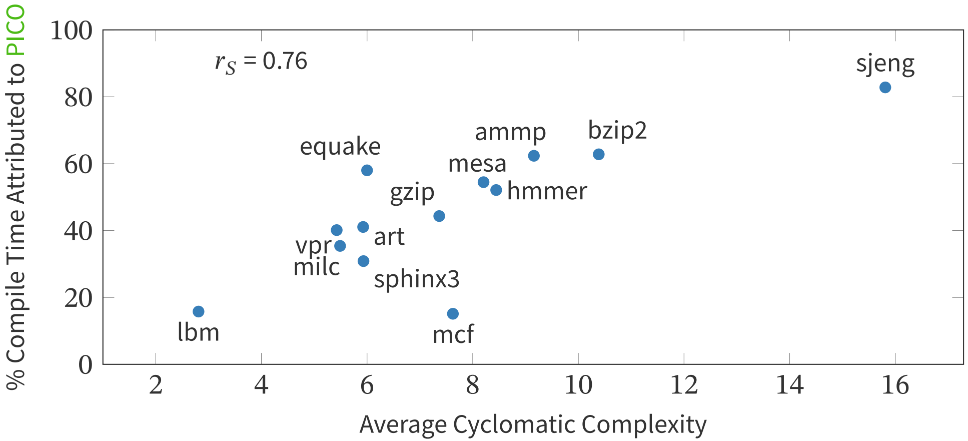


Compile-time Attribution



86% spent in ISL

Correlation to Cyclomatic Complexity



Checks Proven Safe and Replaced

bench.	#l/s	IB	repl.
lbm	401	8%	87%
art	583	65%	12%
equake	1,012	48%	30%
mcf	630	9%	45%
gzip	1,607	48%	22%
bzip2	3,585	6%	21%
milc	3,707	52%	35%
ammp	5,130	16%	54%
sjeng	5,136	60%	26%
vpr	4,668	43%	14%
sphinx3	5,711	14%	35%
hmmer	11,752	14%	32%
mesa	22,185	10%	43%

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

```
...
int *cp = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    cp[i] = ar[i];
}
...
```

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

`int *cp = malloc(size * sizeof(int))`:

$$\{Mem \mid cp \leq Mem < cp + size\}$$

`cp[i]`:

$$\{Mem \mid Mem = cp + l_i \wedge 0 \leq l_i < size\}$$
$$\rightarrow \{Mem \mid cp \leq Mem < cp + size\}$$

```
...
int *cp = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) {
    cp[i] = ar[i];
}
...
```

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

```
int *cp = malloc(size * sizeof(int)):
```

$$\{Mem \mid cp \leq Mem < cp + size\}$$

```
cp[i]:
```

$$\{Mem \mid Mem = cp + l_i \land 0 \leq l_i < size\}$$
$$\rightarrow \{Mem \mid cp \leq Mem < cp + size\}$$

$$OOBMem := Access \cap \overline{Bounds}$$

$$OOB := \pi_{Mem}(OOBMem)$$
$$IB := \overline{OOB}$$

Presburger Access Description

- Flow-sensitive program analysis
- Based on Scalar Evolution[1] and Polly[2], using ISL[3]

```
int *cp = malloc(size * sizeof(int)):
```

$$\{Mem \mid cp \leq Mem < cp + size\}$$

```
cp[i]:
```

$$\{Mem \mid Mem = cp + l_i \land 0 \leq l_i < size\}$$
$$\rightarrow \{Mem \mid cp \leq Mem < cp + size\}$$

$$OOBMem := Access \cap \overline{Bounds}$$

$$\{Mem \mid cp \leq Mem < cp + size\}$$
$$\cap \quad \overline{\{Mem \mid cp \leq Mem < cp + size\}} = \{\}$$

$$OOB := \pi_{Mem}(OOBMem)$$

$$false$$

$$IB := \overline{OOB}$$

$$true$$