KU LEUVEN GENT

Bounds Checking for the Masses

Towards Practical Exploit Mitigation in the Data-Only Attack Era

Adriaan Jacobs

Ericsson, 2025/02/25

FBI

Half a million widely trusted websites vulnerable to Heartbleed bug

A serious overrun vulnerability in the OpenSSL crypto... of SSL web servers which use certificat...

NEWS

The M... Worm

30 Yea...

The Guardian

Heartbleed: ... catastrophic

Code error means ... 'heartbeat' function ...

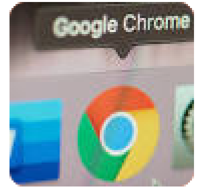
DAN GOODIN -

SC Magazine

Google patches new zero-day actively exploited in the Chrome browser

Google's most recent patch for Chrome is the fifth actively exploited zero-day targeted by threat actors this year in the popular browser.

3 days ago

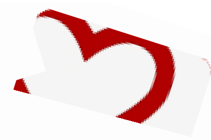


Washington P...

U.S. declares North Ko...

In a statement Tuesda... WannaCry ran...

19 De...



...ritical libwebp Vulnerability Under Active Exploitation - Gets Maximum CVSS Score

Google has assigned a new CVE identifier for a critical security flaw in the libwebp image library for rendering images in the WebP format...

5 days ago

...nerability

...-4034

SiliconANGLE

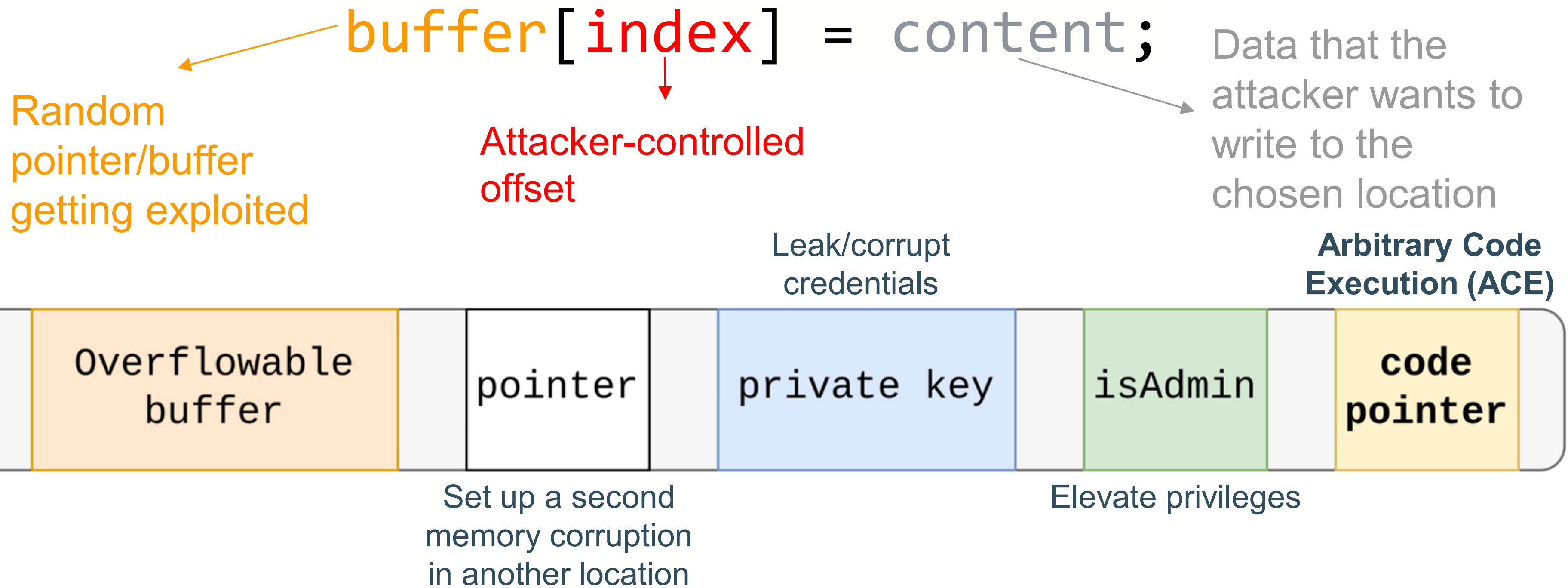
Chrome, Firefox and other br... WebP vulnerability

Google LLC, the Mozilla Foundation ... to fix a zero-day vulnerability affecting the...

3 weeks ago

Discovered in polkit

Memory Errors: Limitless Fun!



An Incomplete Timeline

The First doc. Overflow Attack
1972

Code Injection
1988
Smashing the stack
1996

Non Executable Stack
1997
NX-bit
2003

Ret2Libc
1997

Stack Canaries
1998

Heap Overflows
1998

Heap Mitigations
2000

Format String
2000

Format Guard
2001

ASLR
2001

Info. Leak
2002

Point Guard
2003
Instruction Set Random.
2003

Heap Spraying
2004
Heap Feng Shui
2007

CFI
2005

ROP
2007
JOP
2011
SROP
2016
BOP
2019

Shadow Stack
2007

Code Divers.
2008
Runtime Divers.: Shuffler
2016

JIT-ROP
2013

Code Pointer Integrity
2014
Cryptographic-CFI
2015
ARM PAC
2018

Non-Control Data Attacks
DOP
2008

XnR
2015
Heisenbyte
2015

COOP
2015

Vtable Protection
2016

DOP
2016

PAIRS
2019

Timeline

Hassan, Mohamed Tarek Ibn Ziad. "Hardware-Software Co-design for Practical Memory Safety." PhD thesis, Columbia University, 2022.

KU LEUVEN

An Incomplete Timeli

Hassan, Mohamed Tarek Ibn Ziad. "Hardware-Software Co-design for Practical Memory Safety." PhD thesis, Columbia University, 2022.

Herbert Bos. "Corruption of memory: Those who don't know history are doomed to repeat it" Keynote address, NDSS'24.

What's the Big Deal?

1. **Universal**: ***All computer systems*** run memory-unsafe code
2. **Guaranteed: *All memory-unsafe code*** has memory errors
3. **Replicated**: Everyone runs ***the same unsafe binaries***
4. **Impactful**: Memory errors lead to ***expressive exploits*** (e.g. code execution)

ALL MODERN DIGITAL INFRASTRUCTURE

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

2021 in Memory Unsafety - Apple's Operating Systems

"Memory unsafety continues to dominate the total percentage of security bugs on Apple's platforms."

CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY

AMERICA'S CYBER DEFENSE AGENCY

The Urgent Need for Memory Safety in Software Products

Chrome: 70% of all security bugs are memory safety issues

Google software engineers are looking into ways of eliminating memory management-related bugs from Chrome.

https://xkcd.com/2347/

KU LEUVEN

What’s the Big Deal?

1. **Un** me
2. **Gu** me
3. **Re** ***bi***
4. **Im** ***ex***

Initial Infection Vector

For the fifth year in a row, exploits were the most frequently observed initial infection vector in Mandiant incident response investigations. For intrusions in which an initial infection vector was identified, 33% began with exploitation of a vulnerability. This is a decline from 2023, during which exploits represented the initial intrusion vector for 38% of intrusions, but nearly identical to the share of exploits in 2022, 32%.

2347/

https://services.google.com/fh/files/misc/m-trends-2025-en.pdf

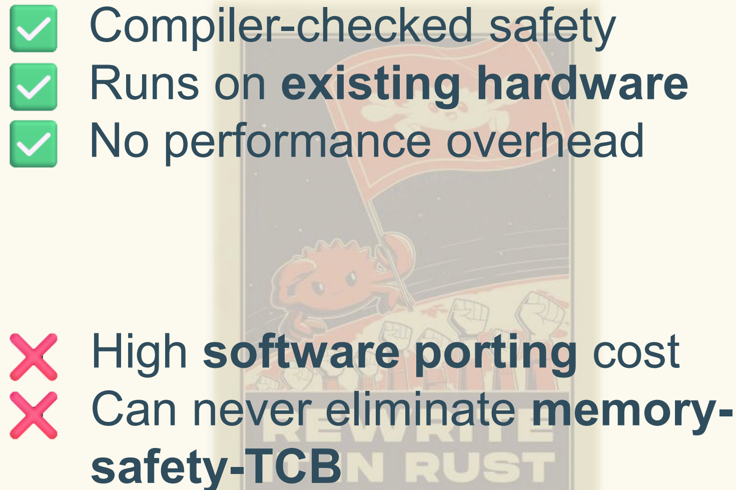
Didn’t We Solve This Problem?

Memory-Safe Software

Memory-Safe Hardware

Didn’t We Solve This Problem?

Memory-Safe Software



- ✅ Compiler-checked safety
- ✅ Runs on **existing hardware**
- ✅ No performance overhead

- ❌ High **software porting** cost
- ❌ Can never eliminate **memory-safety-TCB**

Memory-Safe Hardware

- ✅ Hardware-enforced safety
- ✅ Low software porting cost
- ✅ **Memory-safety-TCB** eliminated

- ❌ High **hardware replacement** cost
- ❌ Non-zero **performance overhead**

Didn’t We Solve This Problem?

Memory-Safe Software

- ✅ Compiler-checked safety
- ✅ Runs on **existing hardware**
- ✅ No performance overhead

- ❌ High **software porting** cost
- ❌ Can never eliminate **memory-safety-TCB**

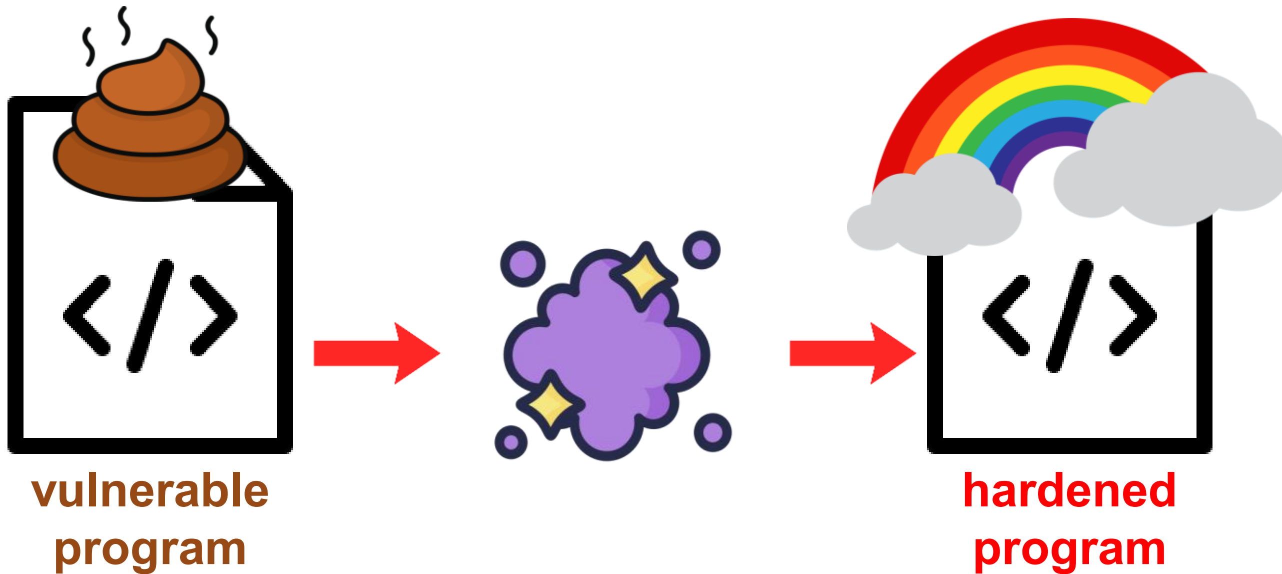
Memory-Safe Hardware

- ✅ Hardware-enforced safety
- ✅ Low software porting cost
- ✅ **Memory-safety-TCB** eliminated

- ❌ High **hardware replacement** cost
- ❌ Non-zero **performance overhead**

EXPLOIT MITIGATIONS

Rapid Adoption: Exploit Mitigations



Exploit Mitigations: What Are Our Options?

More Practical

Formal Verification

Bounds Checking

Isolation of Data

Integrity of Use

Intrusion Detection

Pointer Use

Pointer Invalidation

Arbitrary Memory Access Primitive

Access Non-Control Data

Corrupt Code Pointer

Data-Only Attack

Control-Flow Hijack

Damage the Victim

RIIR, VeriFast

Memory Safety Enforcement

Code Pointer Integrity (CPI)

Control Flow Integrity (CFI)

More Secure

Attackers Are Only Getting More Creative

Control Flow Hijacking

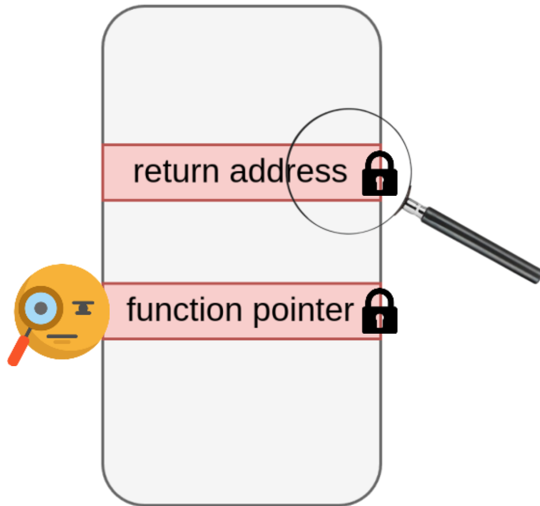
```
while (connect_limit--) {
  pkt = read_from_network(buf);
  srv->log(pkt);
  if (pkt->type == NONE)
    return;
  if (pkt->type == STREAM)
    send(srv->fd, srv->stream);
  else {
    srv->typ = *type;
    srv->total += *size;
  }
}
```

Data-Only Attacks

```
while (connect_limit--) {
  pkt = read_from_network(buf);
  srv->log(pkt);
  if (pkt->type == NONE)
    return;
  if (pkt->type == STREAM)
    send(srv->fd, srv->stream);
  else {
    srv->typ = *type;
    srv->total += *size;
  }
}
```

Attackers Are Only Getting More Creative

Control Flow Hijacking



Data-Only Attacks

KU LEUVEN

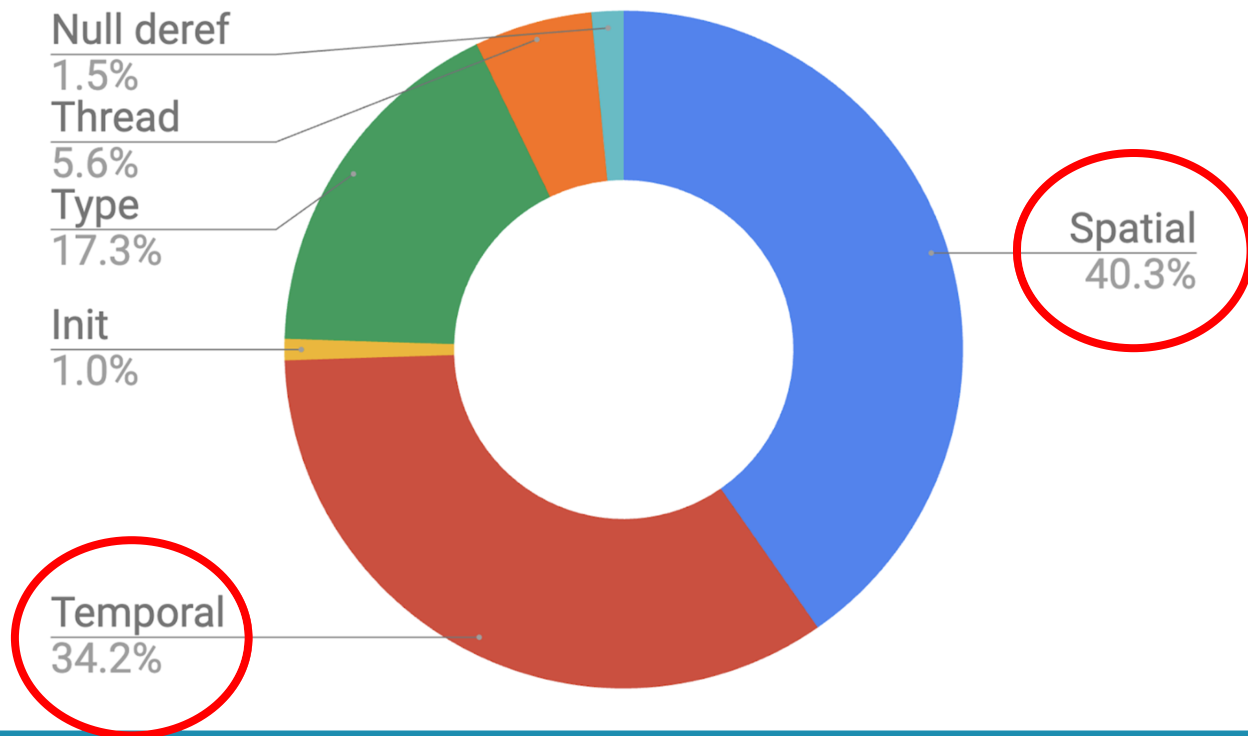
Data-Only Attack Mitigations: What Are Our Options?

More Practical

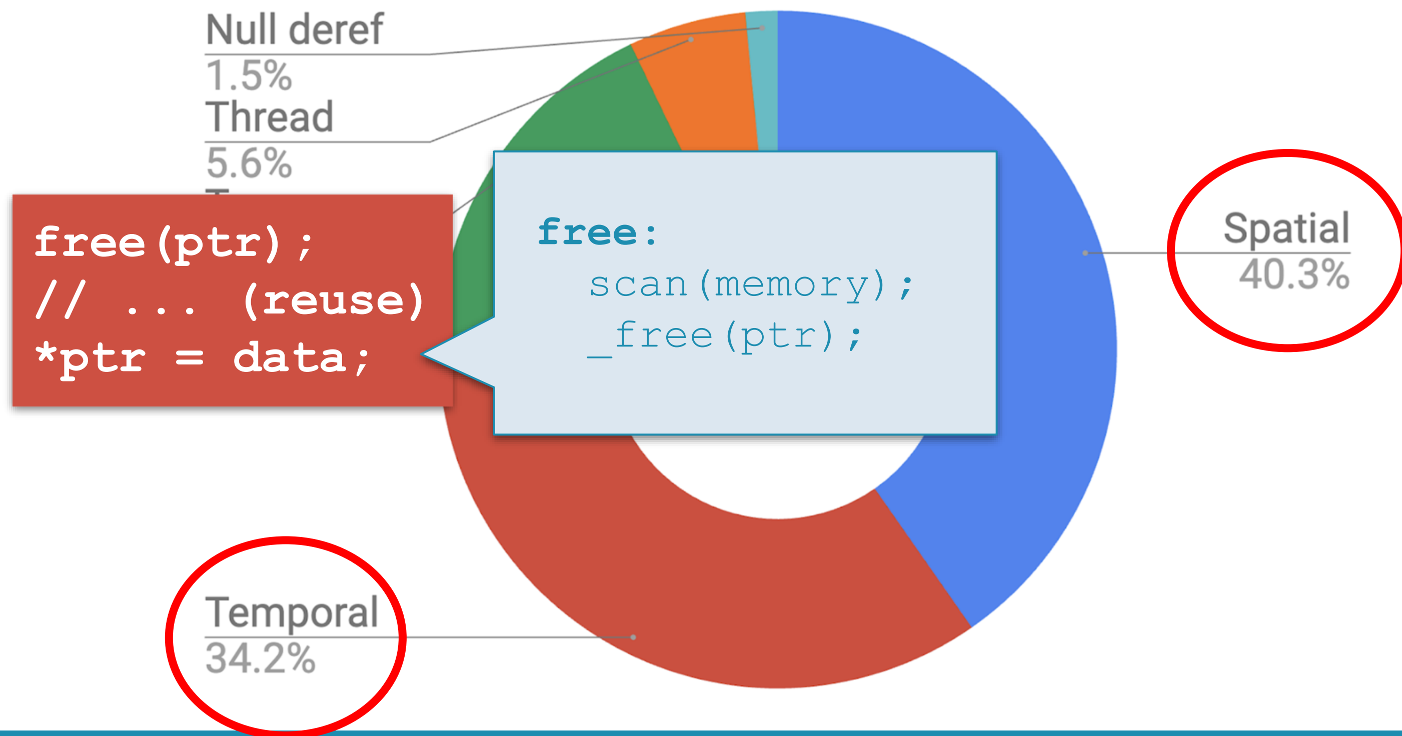
More Secure

KU LEUVEN

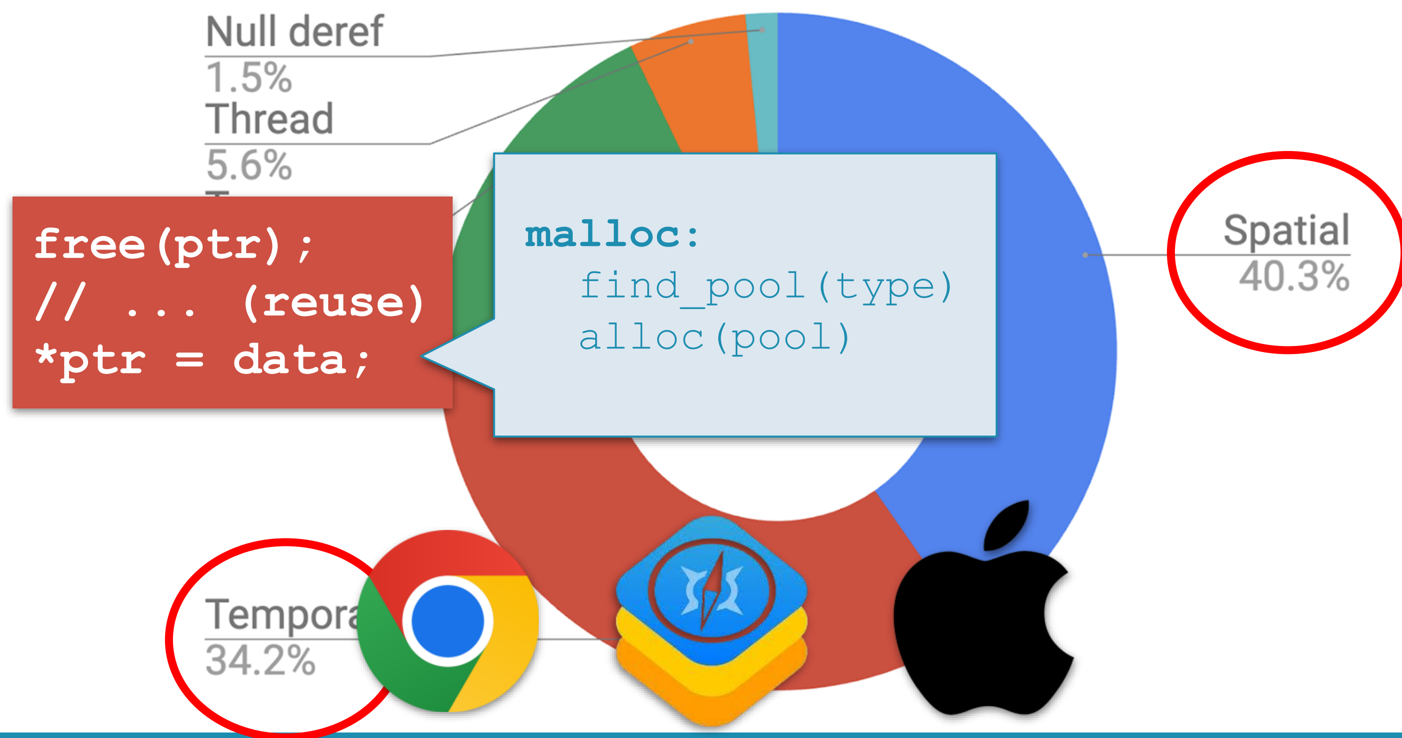
Memory Safety 0-days Exploited in the Wild



Memory Safety 0-days Exploited in the Wild

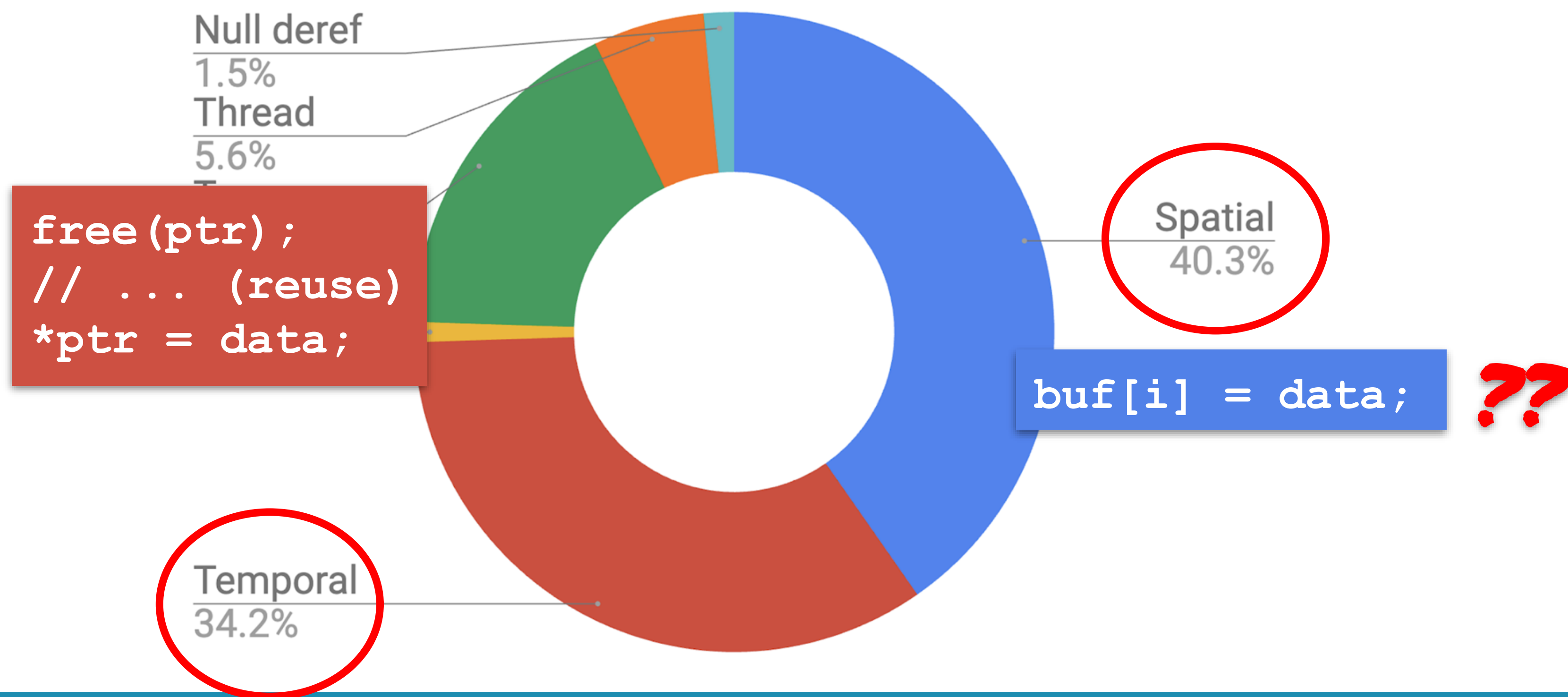


Memory Safety 0-days Exploited in the Wild

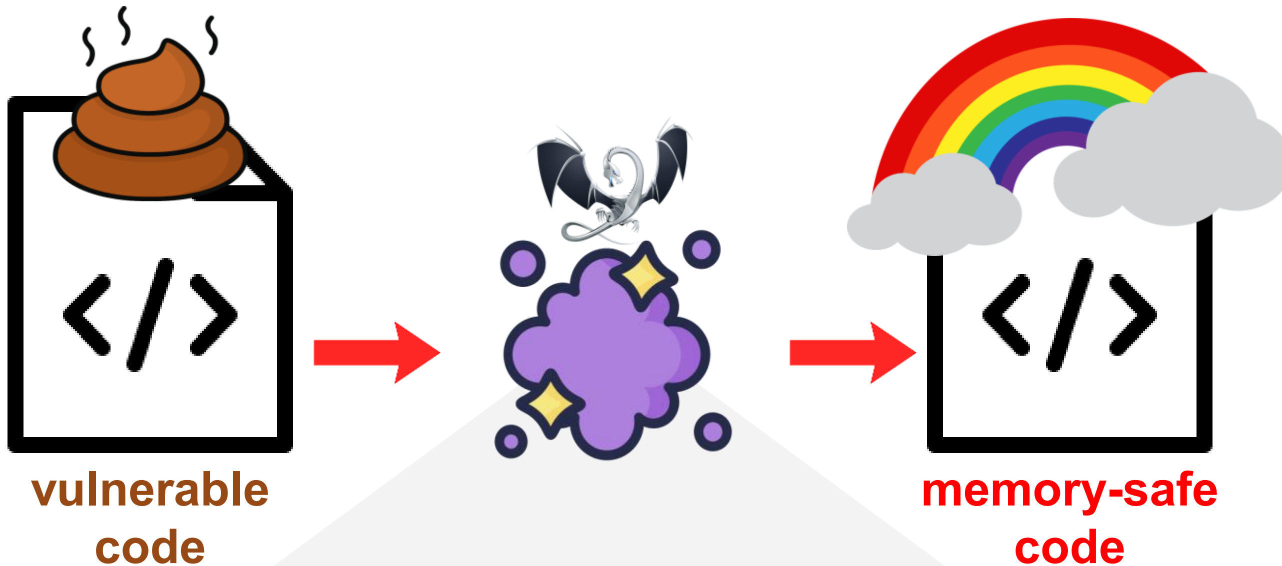


https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html

Memory Safety 0-days Exploited in the Wild



How Bounds Checkers Work ...




```
if (i < 0 || i >= size)
      trap();
buf[i] = data;
```

How Bounds Checkers Work ... They Don't

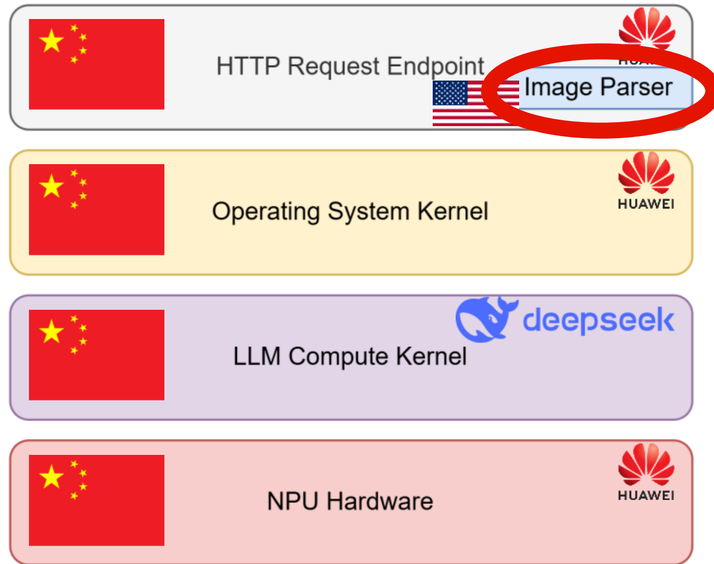
NOT ADOPTED

Approach	Type	Hardware Assistance	Pointer Instrumentation				Target Memory			Compatibility					Run-Time Overhead
			Deref.	Arith.	Escape	Copy	Stack	Heap	Global	Ext. code Interop	Benign OoB ptrs	Maximal Address Space (2^x)	Max. #Live Objects	Max. Alloc Size (GB)	
Fat Pointers [15–18]	P	-	✓	-	-	-	✓	✓	✓	○	●	▲	▲	▲	red ●
SoftBound [19]	P	-	✓	-	✓	✓	✓	✓	✓	◐	●	▲	▲	▲	red ●
Intel MPX [39, 122]	P	Intel MPX	✓	-	✓	✓	✓	✓	✓	◐	●	▲	▲	▲	red ●
MESH [41]	P	-	✓	✗	✓	[illegible]	✗	✓	✗	◐	●	48	2^{16}	▲	red ●
PACMem [20]	P	ARM PA	✓	✓	✓	-	✓	✓	✓	◐	●	39	2^{24}	4 GB	red ●
CUP [27]/Midfat [40]	P	-	✓	✓	✓	-	✓	✓	✓	◐	●	▲	2^{31}	4 GB	orange ●
CGuard [30]	P	-	✓	✓	✓	-	✓	✓	✓	○	◐	48	▲	▲	orange ●
Delta Pointers [28]	P	-	✓	✓	✓	-	✓	✓	✓	○	◐	32	▲	2 GB	orange ●
J&K [14]	O	-	-	✓	-	-	✓	✓	✓	◐	○†	▲	▲	▲	red ●
CRED [24]	O	-	✓	✓	-	-	✓	✓	✓	●	◐	▲	▲	▲	red ●
PariCheck [34]	O	-	-	✓	-	-	✓	✓	✓	◐	◐	38	▲	▲	red ●
Baggy Bounds$^{64\text{-bit}}$ [22]	O	-	-	✓	-	-	✓	✓	✓	●	◐	38	▲	▲	red ●
LowFat [23, 38, 93]	O	-	-	✓	-	-	✓	✓	✓	●	○	▲	▲	32 GB	orange ●
CAMP [25]	O	-	-	✓	-	-	✗	✓	✗	●	○†	▲	▲	▲	red ●
ShadowBound [56]	O	-	-	✓	-	-	✗	✓	✗	●	○	▲	▲	8 GB	orange ●
TailCheck [31]	R	-	✓	✗	✓	-	✗	✓	✓	◐	●	48	▲	▲	green ●
StickyTags [78]	R	ARM MTE	✓	✗	-	-	✓	✓	✗	●	●	▲	▲	▲	green ●
ZOMETag [121]	P+O+R	ARM MTE	✓	✓	-	-	✓	✓	✓	●	●‡	48	$16 \times 2^{48-32} = 2^{20}$	4 GB	orange ●
FRAMER [21, 70]	P+O	-	✓	✓	✓	-	✓	✓	✓	◐	◐‡	▲	▲	▲	red ●

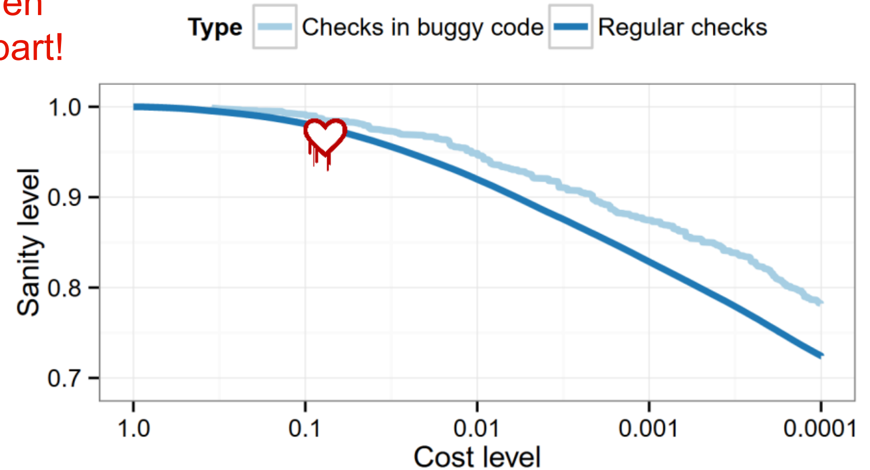
Legend: **P** Pointer-based, **O** Object-based, **R** Red-zone-based (aka Tripwire), ✓ Has this feature, ✗ Lacks this feature, ✓ Has this feature (but is fundamentally flawed), ✗ Lacks this feature (but no fundamental issue in the design), - Feature is not required, ● Full Compatibility, ◐ Partial Compatibility, ○ No Compatibility, ● <30% (average) overhead, ● <50% (average) overhead, ● >50% (average) overhead, ▲ The maximum possible by the underlying OS/architecture, † Only off by One, ‡ Within limited range. If the defense includes more than just spatial checks, we model their spatial component alone.

A. Jacobs et al. "**N00B: Bounds Checking for the Masses.**" *Under Submission*.

Is Performance Really The (Only) Problem?



Harden this part!



Wagner et al. "High System-Code Security with Low Overhead" S&P'15.

KU LEUVEN

Existing Bounds Checkers Are Stuck

Propagate bounds information with all pointers throughout the program.

POINTER-BASED

```
struct fat_ptr {
    void* ptr;
    void* upper_bound;
    void* lower_bound;
};
```

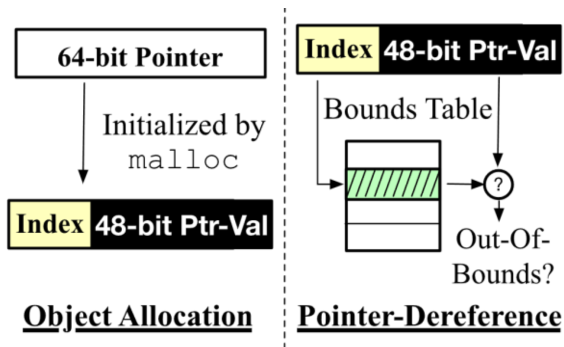

KU LEUVEN

Existing Bounds Checkers Are Stuck

How to recover intended referent during dereference?

Propagate bounds information with all pointers throughout the program.

POINTER-BASED



 Saileshwar, Gururaj, et al. "Heapcheck: Low-cost hardware support for memory safety." TACO'22.

KU LEUVEN

Existing Bounds Checkers Are Stuck

How to recover intended referent during dereference?

Propagate bounds information with all pointers throughout the program.

POINTER-BASED

Legend: ignored | canonical | effective address

	Bits			
ARM TBI	63–55: 8-bit IGNORED	55–47: 8-bit 0...0	47–0: 48-bit addr	
AMD UAI	63–56: 7-bit IGNORED	56–46: 10-bit 0...0	46–0: 47-bit addr	
Intel LAM U57	63: 0	62–56: 6-bit IGNORED	56–46: 10-bit 0...0	46–0: 47-bit addr
Intel LAM U48	63: 0	62–47: 15-bit IGNORED	47: 0	46–0: 47-bit addr

Saileshwar, Gururaj, et al. "Heapcheck: Low-cost hardware support for memory safety." TACO’22.

Existing Bounds Checkers Are Stuck

How to recover intended referent during dereference?

Propagate bounds information with all pointers throughout the program.

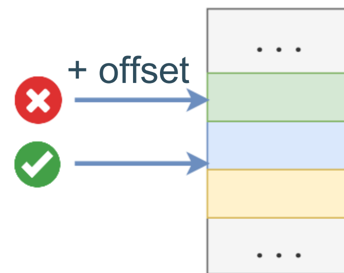
POINTER-BASED

ignored | canonical | effective address

	63	55	47	0
ARM TBI	8-bit IGNORED	8-bit 0...0	48-bit addr	
	63	56	46	0
AMD UAI	7-bit IGNORED	10-bit 0...0	47-bit addr	
	63 62	56	46	0
Intel LAM U57	0 6-bit IGNORED	10-bit 0...0	47-bit addr	
	63 62	47 46		0
Intel LAM U48	0 15-bit IGNORED 0		47-bit addr	

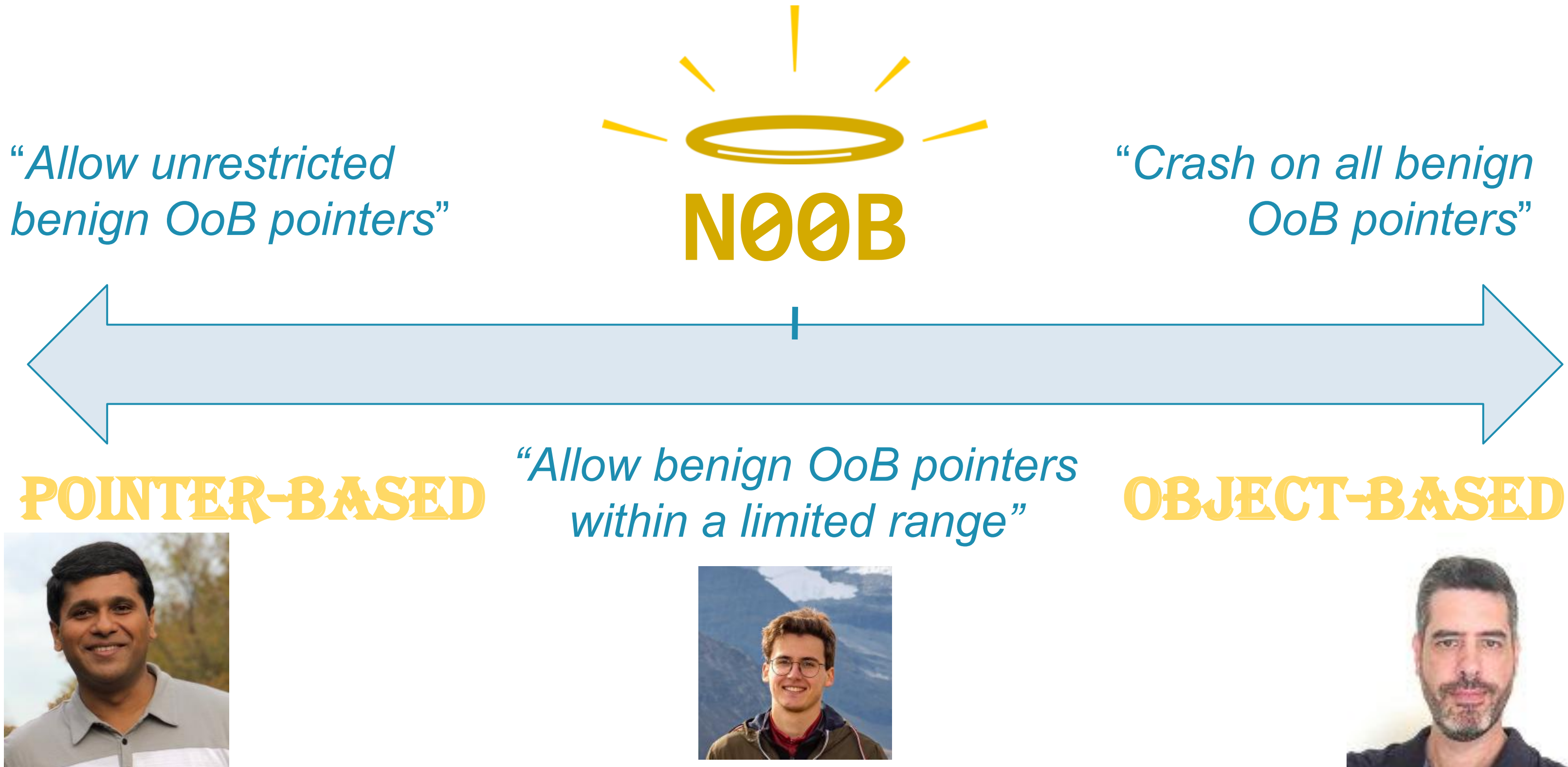
Never let pointers go out of bounds in the first place.

OBJECT-BASED



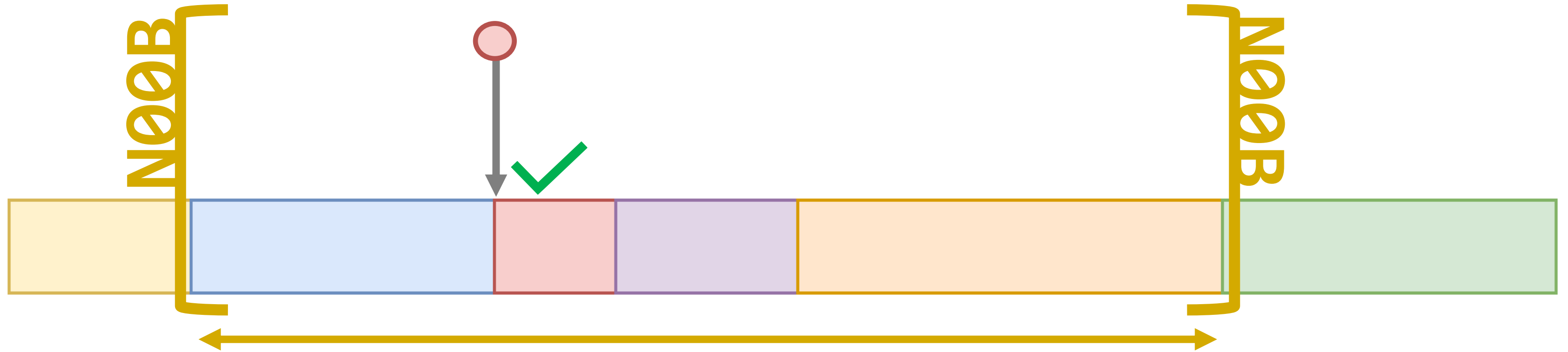
 Saileshwar, Gururaj, et al. "Heapcheck: Low-cost hardware support for memory safety." TACO’22.

N00B: Hybrid Bounds Checking

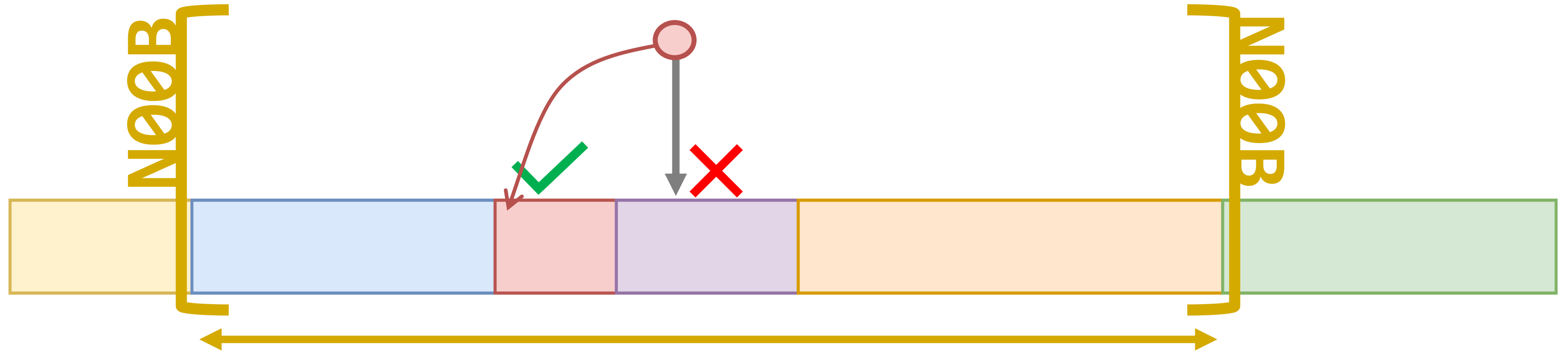


KU LEUVEN

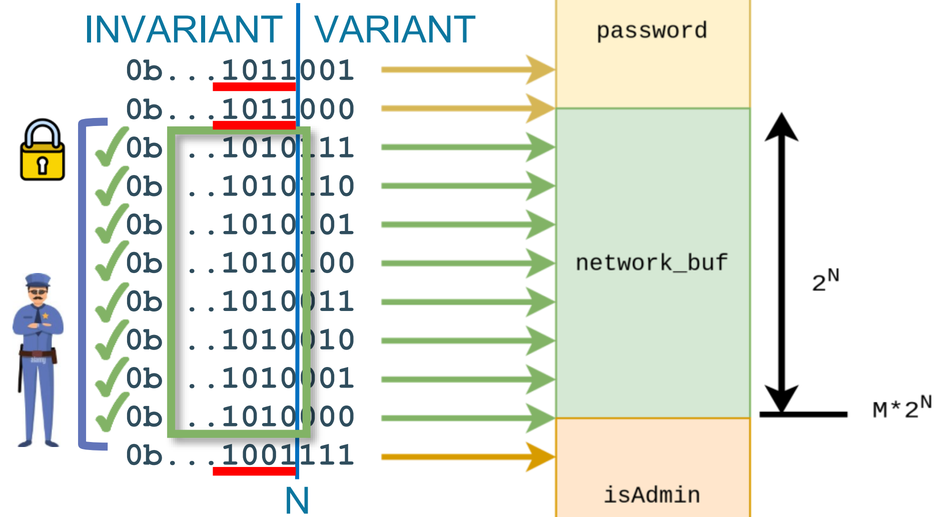
The N0OB Idea

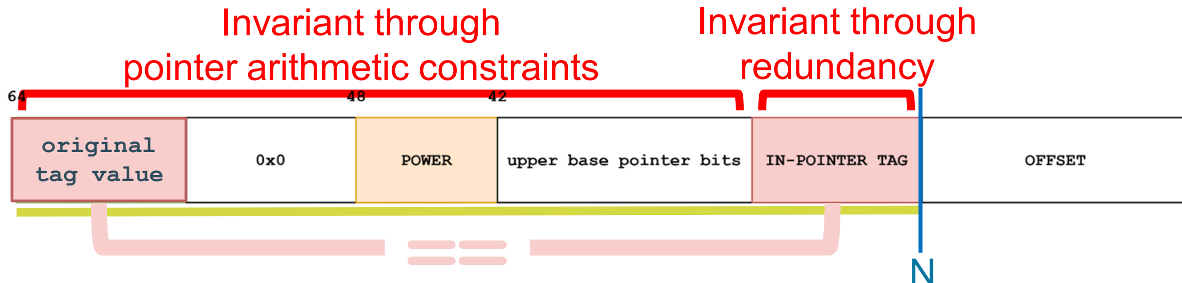


The N00B Idea



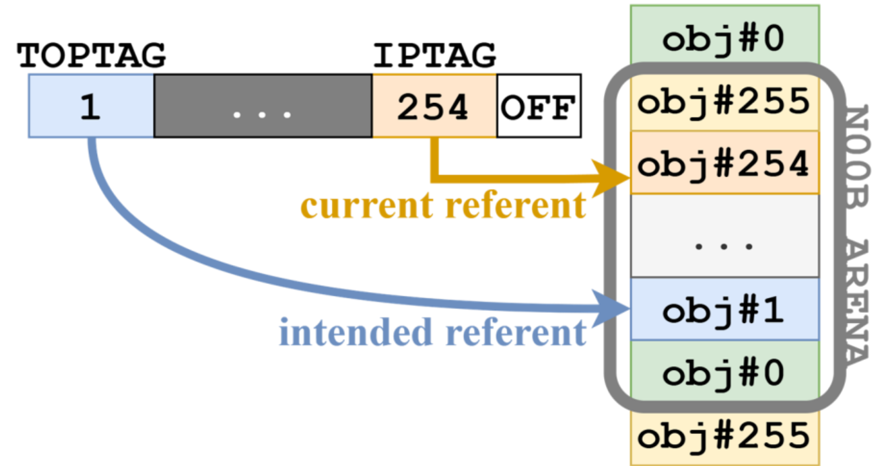
The N00B Idea

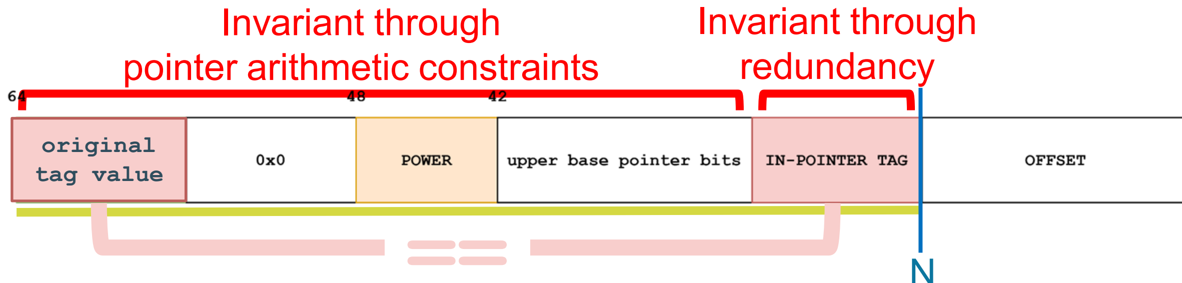




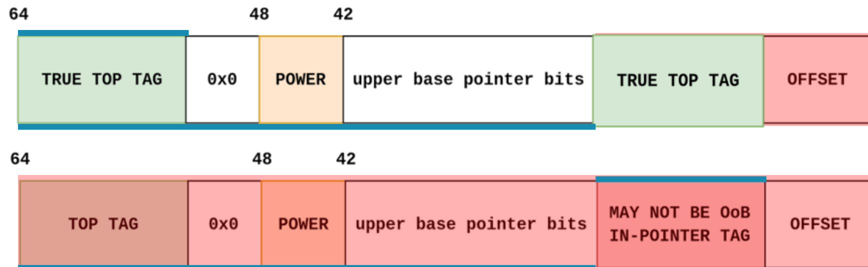
The N00B Idea

- **“Far” Overflows**: coarse-grained arithmetic checks (~ Low-Fat Pointers)
- **“Near” Overflows**: fine-grained tag checks (~ *Implicit* Memory Tagging)





Twice the Checks, Twice the Overhead?



KU LEUVEN

Twice the Checks, Twice the Overhead?

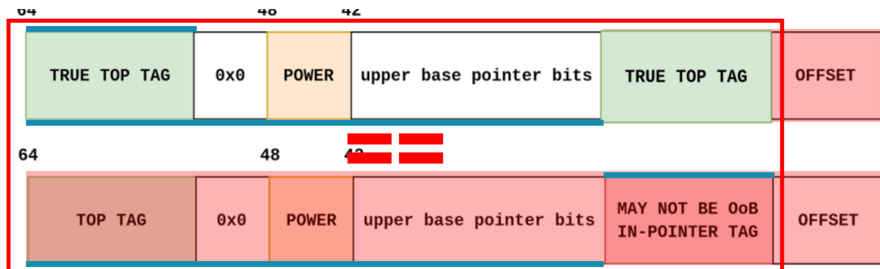
Table 4: Run-time overhead comparison of **N0OB** versus the fastest current software-based bounds checkers.

Bounds Checker	SPEC CPU2006	SPECspeed 2017
CAMP [26]	76.1% [76]	48.6% [76]
ShadowBound [53]	20.1% [76]	5.72%
Low-Fat [24]	26–43%	27–42%
Delta Pointers [29]	35% [61]	–
CGuard [31]	–	42.1%
N0OB	**29–54%**	**29–54%**

N0OBalloc LowFat N0OB

Run-Time Ratio

400.perlbench 401.bzip2 403.gcc 429.mcf 445.gobmk 456.hmmer 458.sjeng 462.libquantum 464.h264ref 471.omnetpp 473.astar 483.xalancbmk 433.milc 444.namd 447.dealII 450.soplex 453.povray 470.lbm 482.sphinx3 **2006 geomean** 600.perlbench_s 602.gcc_s 605.mcf_s 620.omnetpp_s 623.xalancbmk_s 625.x264_s 631.deepsjeng_s 641.leela_s 657.xz_s 619.lbm_s 638.imagick_s 644.nab_s **2017 geomean** **overall geomean**




```
1 N = (base >> 42) & 0x7f;
2 referent = (base >> N) & ~TAG_MAX; /*arena base*/
3 toptag = base >> (64 - TW);        /*toptag*/
4 referent |= toptag;                /*orig obj*/
5 if (referent != (ptr >> N)) trap();
```


KU LEUVEN

Twice the Checks, Twice the Overhead?

Table 4: Run-time overhead comparison of **NOOB** versus the fastest current software-based bounds checkers.

Bounds Checker	SPEC CPU2006	SPECspeed 2017
CAMP [26]	76.1% [76]	48.6% [76]
ShadowBound [53]	20.1% [76]	5.72%
Low-Fat [24]	26–43%	27–42%
Delta Pointers [29]	35% [61]	–
CGuard [31]	–	42.1%
NOOB	**29–54%**	**29–54%**

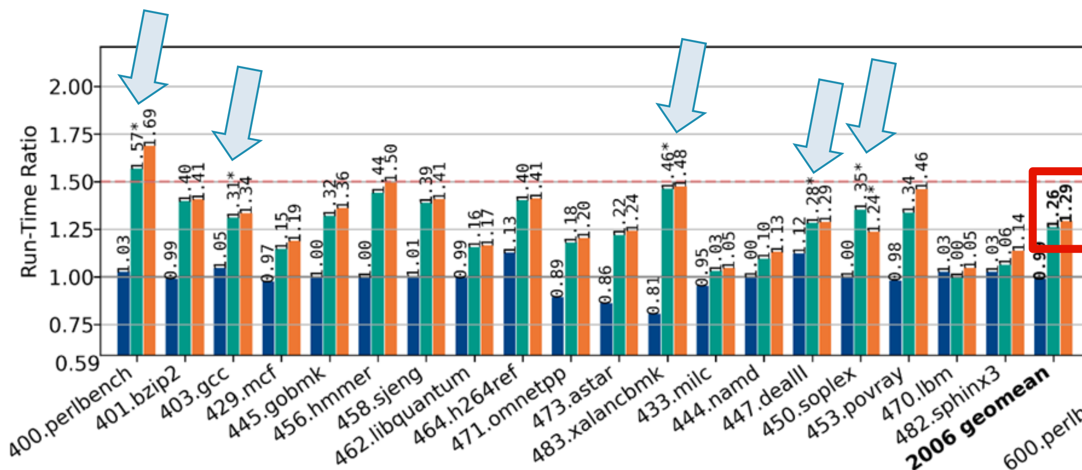
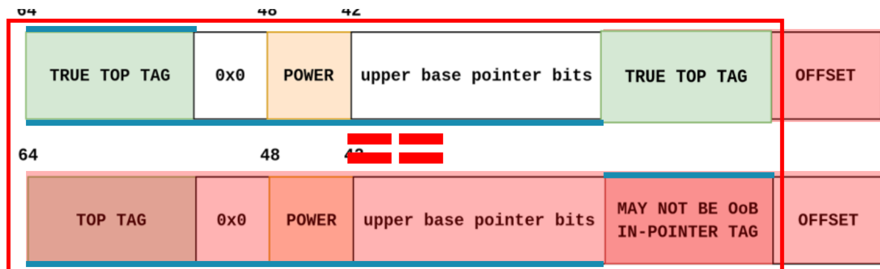


Table 3: Unique benign out-of-bounds cases in SPEC programs that Low-Fat Pointers crashes on.

Benchmark	Unique occurrences
400.perlbench	8
403.gcc	45
483.xalancbmk	49
447.dealII	11
600.perlbench_s	807
623.xalancbmk_s	1
641.leela_s	7




```
1 N = (base >> 42) & 0x7f;
2 referent = (base >> N) & ~TAG_MAX; /*arena base*/
3 toptag = base >> (64 - TW);        /*toptag*/
4 referent |= toptag;                /*orig obj*/
5 if (referent != (ptr >> N)) trap();
```

N00B Is Not Perfect...

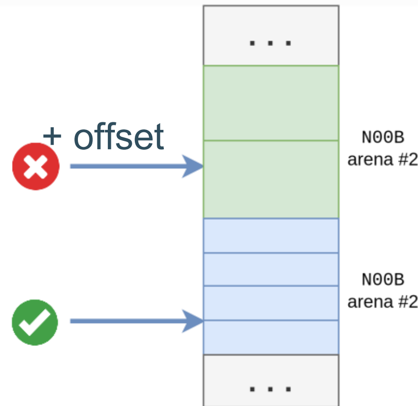
Table 2. Applied N00B compatibility patches.

Program	Revision	Year	Message
403.gcc	r62672	2003	"Don't use offset pointers."
403.gcc	r89543	2004	"Avoid undefined pointer arithmetic on qty_table"
403.gcc	r79945	2004	"Don't play queer offsetting games with reg_known_value"

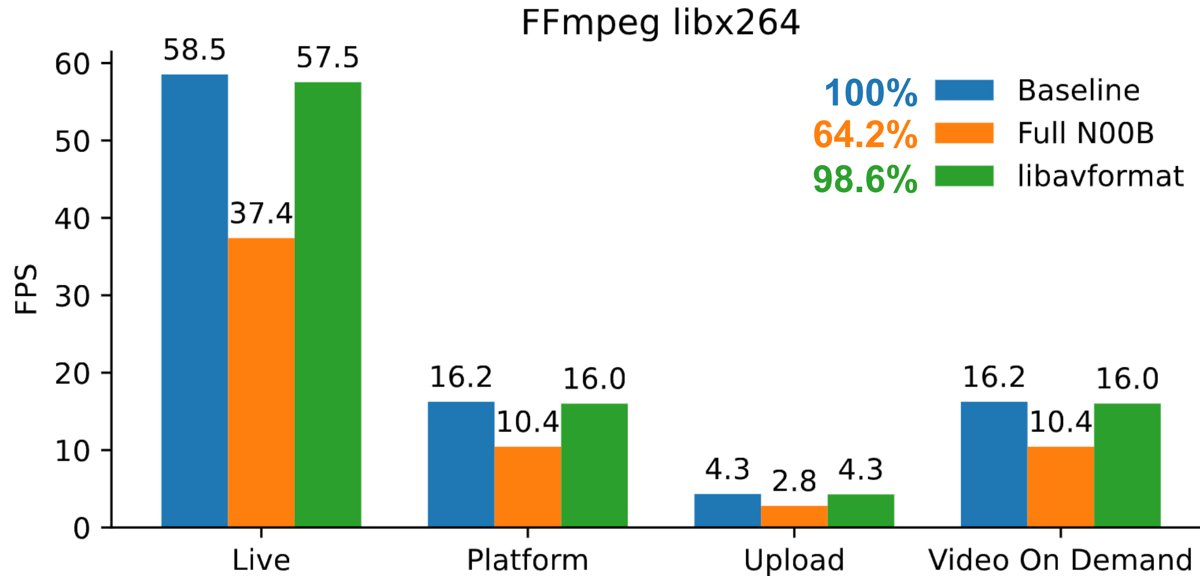
KU LEUVEN

N00B Is Not Perfect...

```
1 offset = realloc(array, grow) - array;
2 for (i = 0; i < size; i++)
3     array[i].ptr += offset;
```



N00B Is Not Perfect... But It *Works*



KU LEUVEN

N00B:
Bounds Checking for the Masses

Adriaan Jacobs
DistriNet, KU Leuven

Carlo Ramponi
University of Trento

Jonas Roels
DistriNet, KU Leuven

Bruno Crispo
University of Trento

Silviu Vlasceanu
Huawei Research

Mahmoud Ammar
Huawei Research

Stijn Volckaert
DistriNet, KU Leuven

Under submission to ACM CCS'26

So What's Industry Doing?

So What's Industry Doing?

```
int x = 0;
_Ptr<int> x = &x;
// no arithmetic on x allowed

int a[5] = { 0, 1, 2, 3, 4};
_Array_ptr<int> p : count(5) = a;
// p points to 5 elements.
```

C++ Safe Buffers

1. STL **hardening**
   "terminate the program if you go out of bounds."
2. **Reject** raw pointer arithmetic

-fbounds-safety
Enforcing bounds safety for production C code

Most code **already contains bounds** information

- Specify through **annotations**

```
void foo(int *__counted_by(N) ptr, size_t N);
```

https://www.checkedc.org/

https://orodu.net/resources/2022-12-31-why-subspace/cpp_safe_buffers_2022-11-30.pdf

https://clang.llvm.org/docs/BoundsSafety.html

So What's Industry Doing?

Micro…

CHECKED C

C++ Safe Buffers

Enforcing bounds safety for production C code

1. Focus on **gradual adoption**: opt-in & interoperable

2. Willing to make **manual source code changes**

3. Maximize **static information** & use **type system**

```
int x = 0;
_Ptr<int> x = &x;
// no arithmetic on x allowed

int a[5] = { 0, 1, 2,
_Array_ptr<int> p : co
// p points to 5 eleme
```

out of bounds."

e already
bounds
information

rough
ons

```
void foo(int *__counted_by(N) ptr, size_t N);
```


https://www.checkedc.org/

https://orodu.net/resources/2022-12-31-why-subspace/cpp_safe_buffers_2022-11-30.pdf

https://clang.llvm.org/docs/BoundsSafety.html

Bounds Checking: Academics versus Industry

Industry

Slow to **adopt**

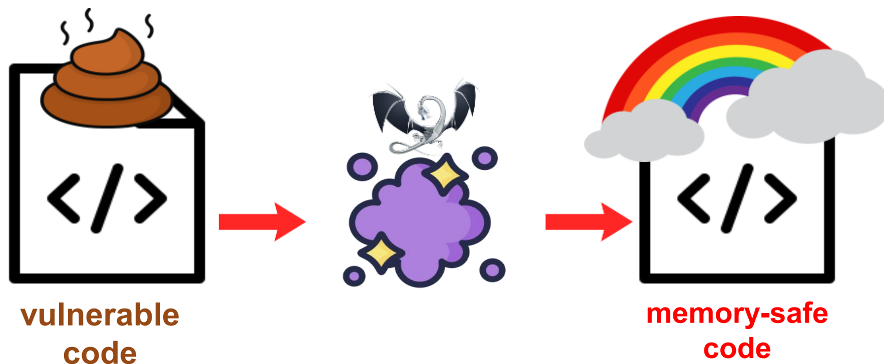
Fast to **run**

Academia

Fast to **adopt**

Slow to **run**

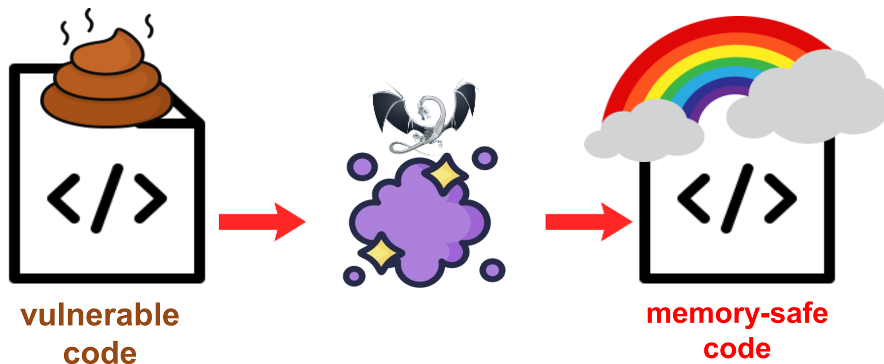
Type Information Is Key




```
struct node {
    int data;
    struct node* prev, next;
};

while ((node = node->next)) {
    if (node->data > 5)
        return node;
}
```

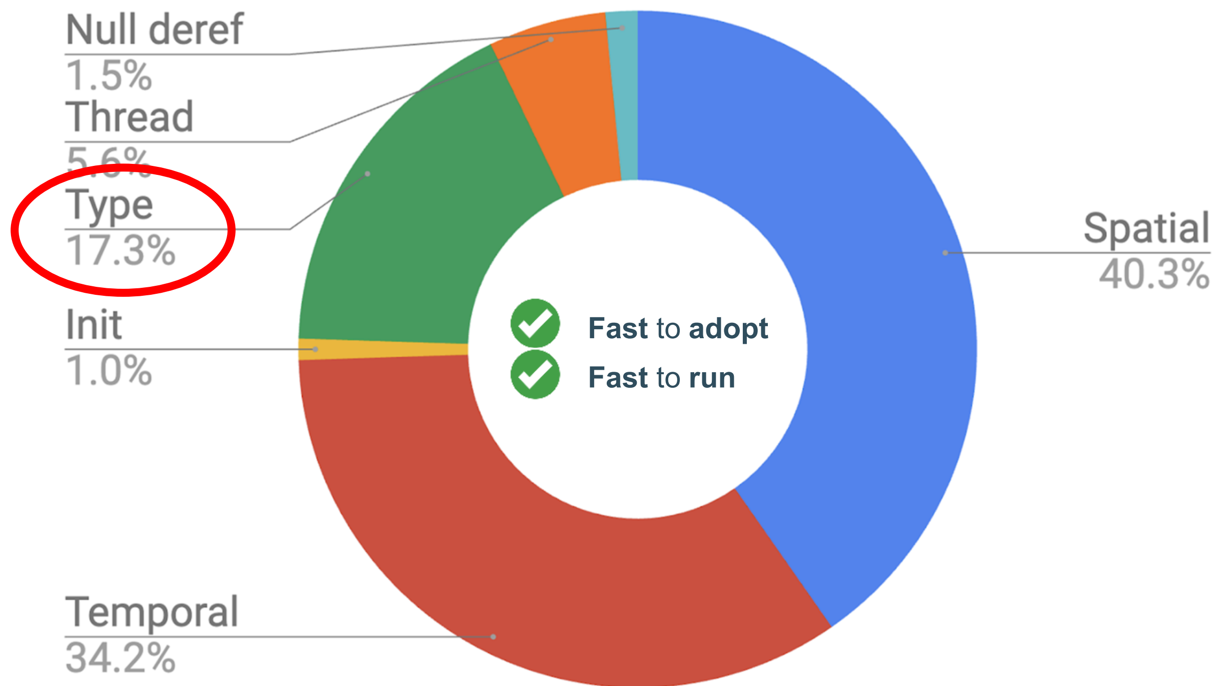
Type Information Is Key




```
struct node {
    int data;
    struct node* __single prev, next;
};

while ((node = node->next)) {
    if (node->data > 5)
        return node;
}
```

Type Information is Key



https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html

Questions?