ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization

Zheng Yu, *Ganxiang Yang, Xinyu Xing*

NORTHWESTERN UNIVERSITY
QUAECUMQUE SUNT VERA
1851

Memory Corruption Errors

- C/C++ lacks heap memory safety. (out-of-bounds, use-after-free).
- 2023 CWE top-most dangerous software weaknesses.
- Exploiting these vulnerabilities can lead to data corruption and privilege escalation.

1 Out-of-bounds Write
CWE-787 | CVEs in KEV: 70 | Rank Last Year: 1

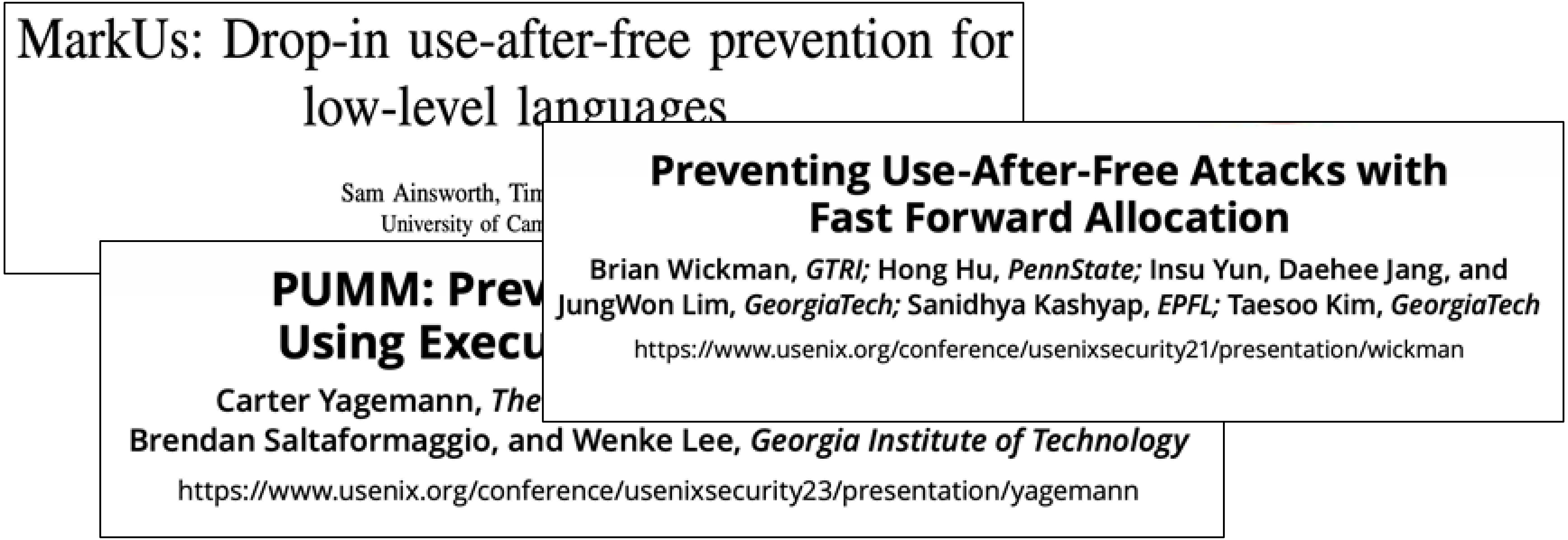
2 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-79 | CVEs in KEV: 4 | Rank Last Year: 2

3 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-89 | CVEs in KEV: 6 | Rank Last Year: 3

4 Use After Free
CWE-416 | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲

Temporal Memory Protection

- In the realm of temporal memory safety, several UAF defenses stand out for their remarkable performance. (<5%)



MarkUs: Drop-in use-after-free prevention for low-level languages

Sam Ainsworth, Tim
University of Cam

PUMM: Prev
Using Execu

Carter Yagemann, *The*
Brendan Saltaformaggio, and Wenke Lee, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/yagemann

Preventing Use-After-Free Attacks with Fast Forward Allocation

Brian Wickman, *GTRI;* Hong Hu, *PennState;* Insu Yun, Daehee Jang, and JungWon Lim, *GeorgiaTech;* Sanidhya Kashyap, *EPFL;* Taesoo Kim, *GeorgiaTech*

https://www.usenix.org/conference/usenixsecurity21/presentation/wickman

Spatial Memory Protection

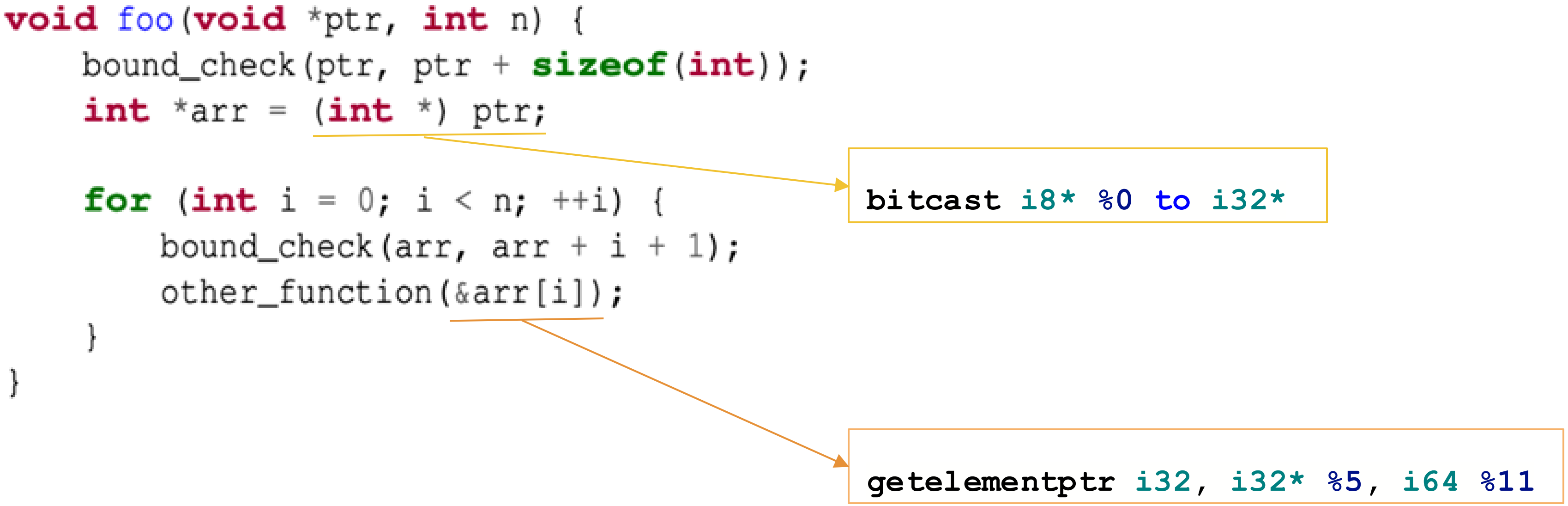
- Redzone Based Checker (ASAN/SANRazor/ASAN–)
  - ❌ High Performance Overhead (> 30%)
  - ❌ Can be bypassed through non-linear out-of-bounds.
- Bounds Tracking (LowFat/ESAN/SoftBound/SGXBound)
  - ❌ Hard to cooperate with SOTA uaf defense (Conflict Allocator).
  - ❌ High Performance Overhead (> 15%)
- State of the arts (DeltaPointer)
  - ✅ Well Performance Overhead (~10%)
  - ❌ Restrict Memory Space to 4GB.

ShadowBound

- ✅ Low Performance Overhead (~6%)
- ✅ Provide Robust Spatial Security.
- ✅ Can work with various UAF defense.

Checking Position

Insert Boundary Checking at Pointer Arithmetic



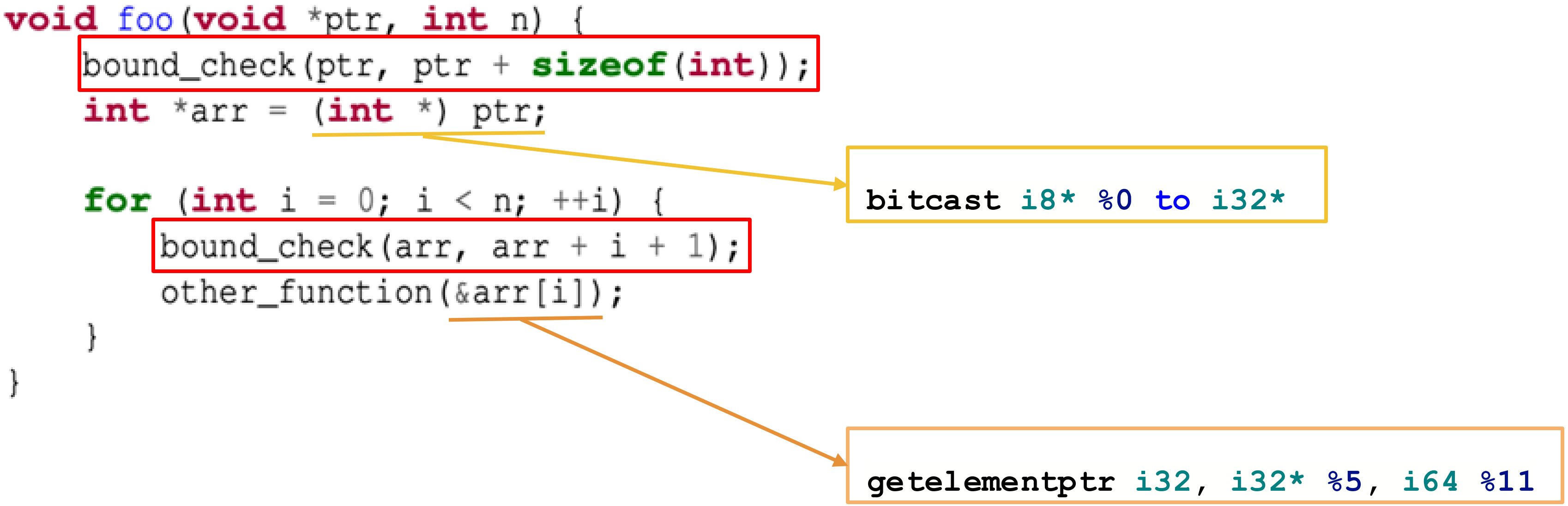
```
void foo(void *ptr, int n) {
    bound_check(ptr, ptr + sizeof(int));
    int *arr = (int *) ptr;

    for (int i = 0; i < n; ++i) {
        bound_check(arr, arr + i + 1);
        other_function(&arr[i]);
    }
}
```

```
bitcast i8* %0 to i32*
```

```
getelementptr i32, i32* %5, i64 %11
```

Checking Position

Insert Boundary Checking at Pointer Arithmetic



```
void foo(void *ptr, int n) {
    bound_check(ptr, ptr + sizeof(int));
    int *arr = (int *) ptr;

    for (int i = 0; i < n; ++i) {
        bound_check(arr, arr + i + 1);
        other_function(&arr[i]);
    }
}
```

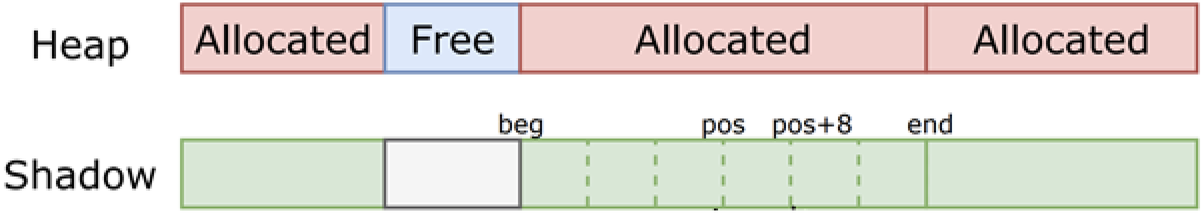
```
bitcast i8* %0 to i32*
```

```
getelementptr i32, i32* %5, i64 %11
```

Ensure the base pointer and result pointer belong to same object

Metadata Design

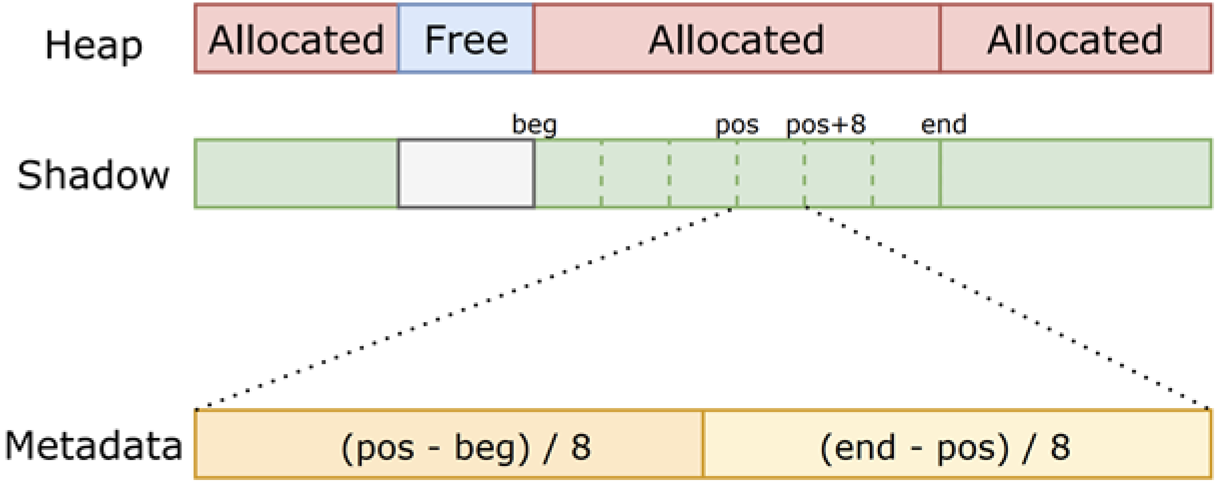
How we store each pointer's boundary?



1. Heap memory size are equal to shadow memory size.
2. Each aligned 8 bytes heap memory are mapped into 8 bytes shadow memory.

Metadata Design

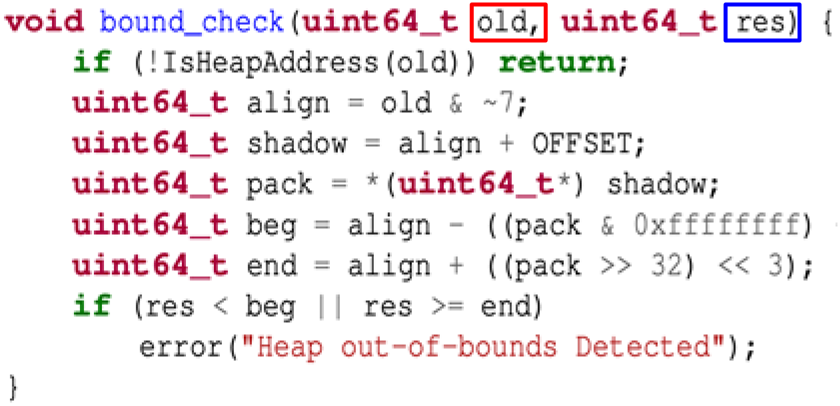
How we store each pointer's boundary?



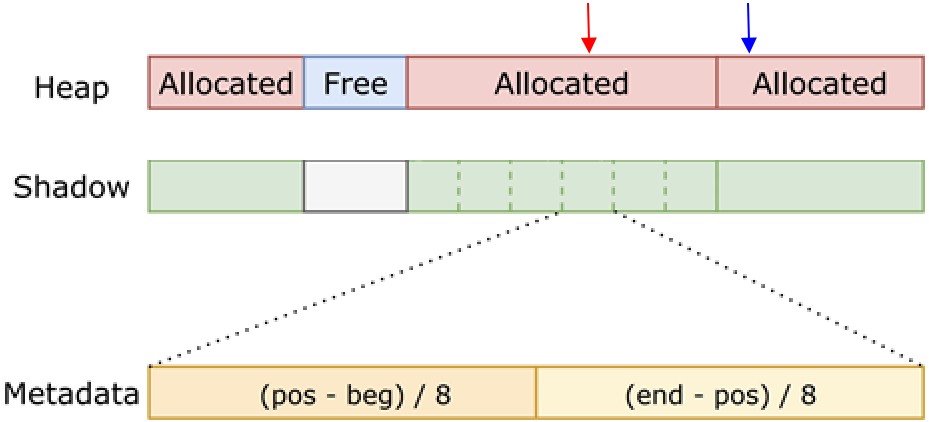
Why 64 bits is enough to save two size_t variables?

1. All mainstream allocators default to 8-byte or 16-byte aligned allocations.
2. The maximum single-time allocation size is limited to 8 GB (2^33 bits).

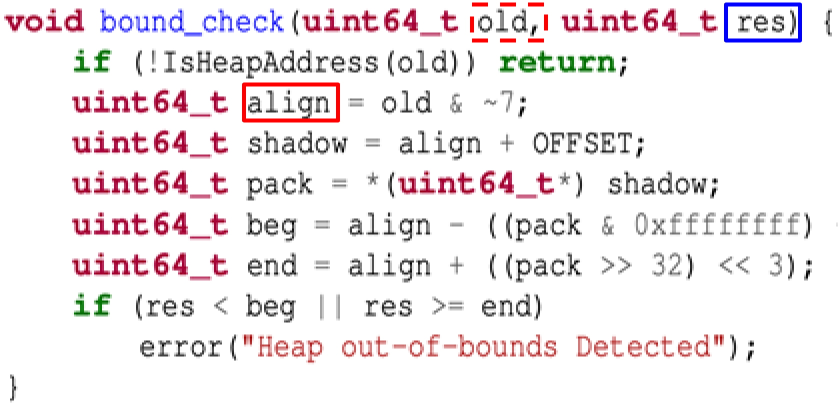
Boundary Checking



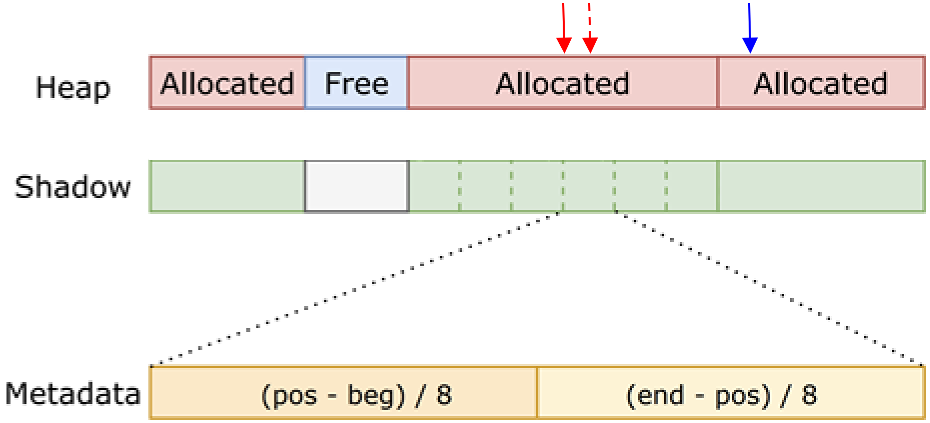
```
void bound_check(uint64_t old, uint64_t res) {
    if (!IsHeapAddress(old)) return;
    uint64_t align = old & ~7;
    uint64_t shadow = align + OFFSET;
    uint64_t pack = *(uint64_t*) shadow;
    uint64_t beg = align - ((pack & 0xffffffff)
    uint64_t end = align + ((pack >> 32) << 3);
    if (res < beg || res >= end)
        error("Heap out-of-bounds Detected");
}
```



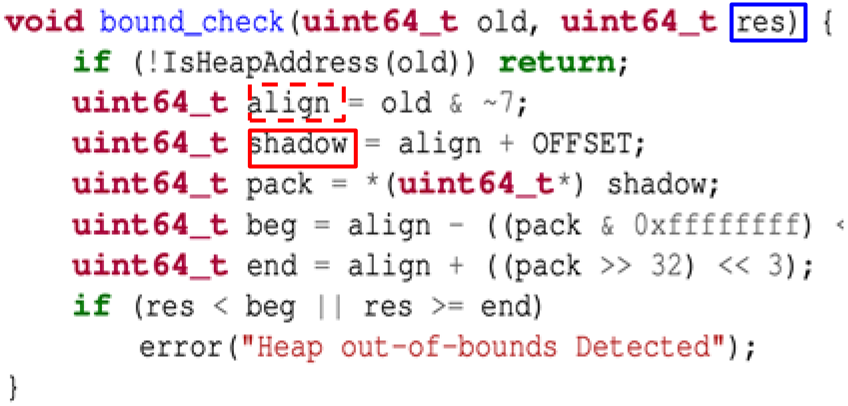
Boundary Checking



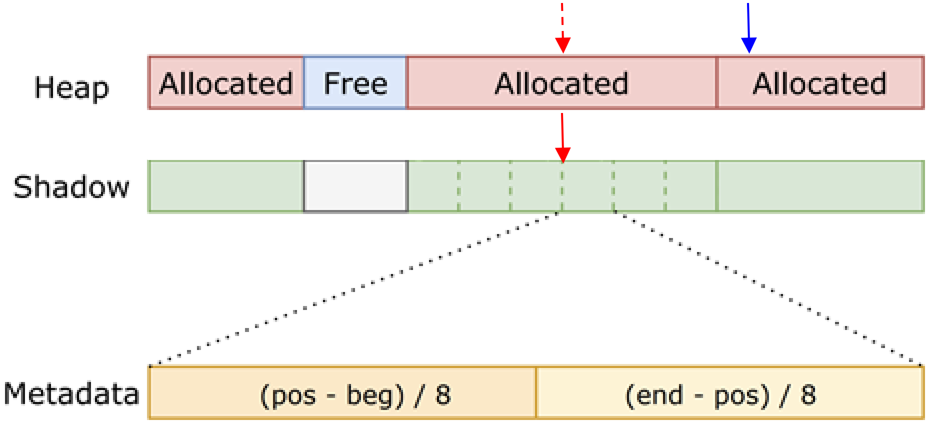
```
void bound_check(uint64_t old, uint64_t res) {
    if (!IsHeapAddress(old)) return;
    uint64_t align = old & ~7;
    uint64_t shadow = align + OFFSET;
    uint64_t pack = *(uint64_t*) shadow;
    uint64_t beg = align - ((pack & 0xffffffff)
    uint64_t end = align + ((pack >> 32) << 3);
    if (res < beg || res >= end)
        error("Heap out-of-bounds Detected");
}
```



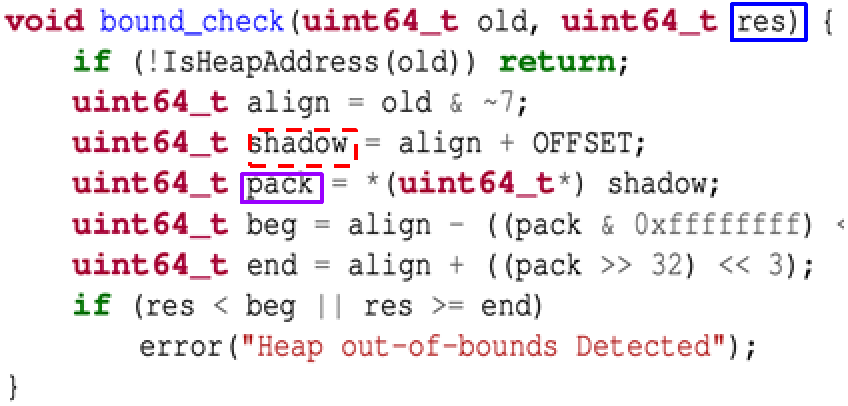
Boundary Checking



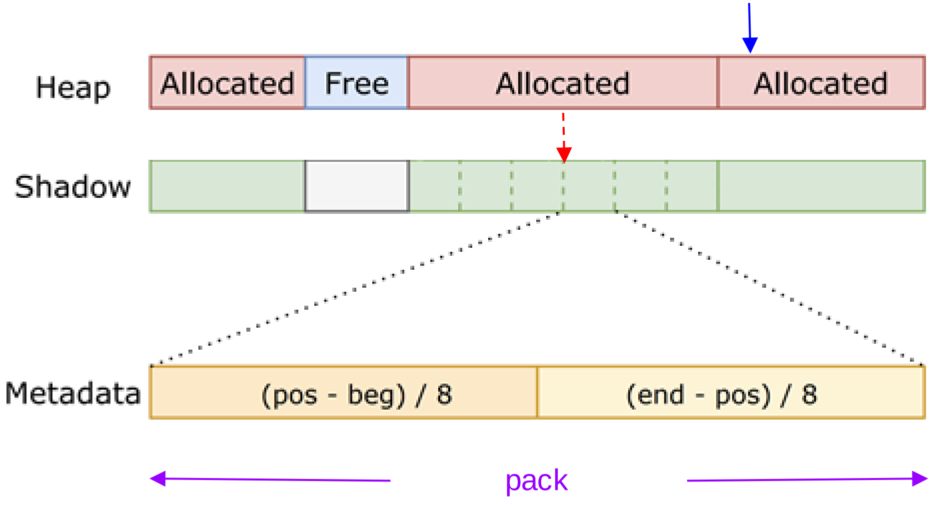
```
void bound_check(uint64_t old, uint64_t res) {
    if (!IsHeapAddress(old)) return;
    uint64_t align = old & ~7;
    uint64_t shadow = align + OFFSET;
    uint64_t pack = *(uint64_t*) shadow;
    uint64_t beg = align - ((pack & 0xffffffff) <
    uint64_t end = align + ((pack >> 32) << 3);
    if (res < beg || res >= end)
        error("Heap out-of-bounds Detected");
}
```



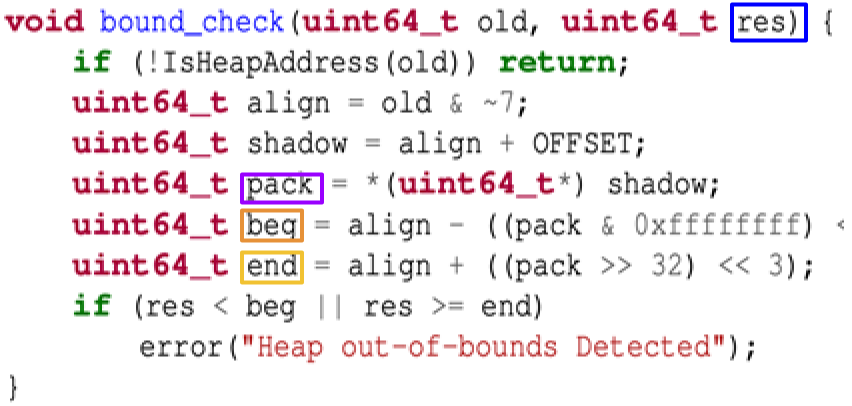
Boundary Checking



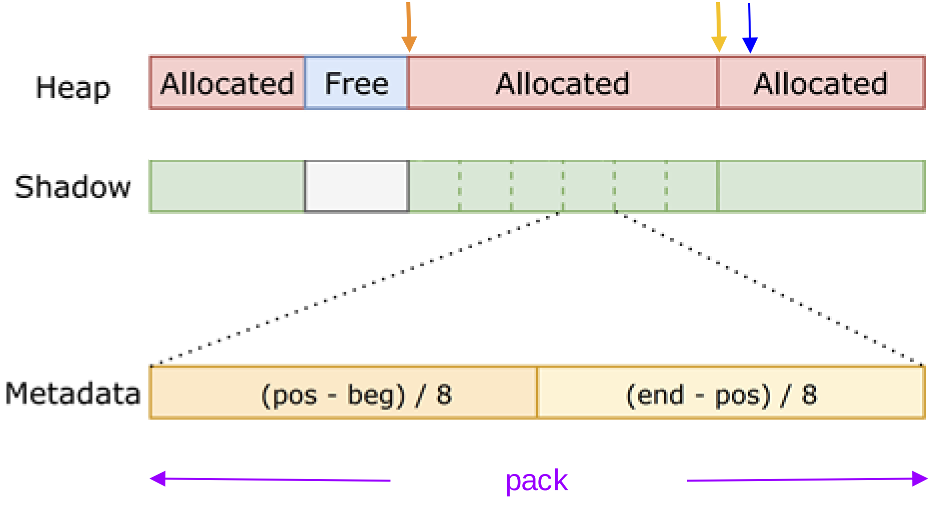
```
void bound_check(uint64_t old, uint64_t res) {
    if (!IsHeapAddress(old)) return;
    uint64_t align = old & ~7;
    uint64_t shadow = align + OFFSET;
    uint64_t pack = *(uint64_t*) shadow;
    uint64_t beg = align - ((pack & 0xffffffff) <
    uint64_t end = align + ((pack >> 32) << 3);
    if (res < beg || res >= end)
        error("Heap out-of-bounds Detected");
}
```



Boundary Checking



```
void bound_check(uint64_t old, uint64_t res) {
    if (!IsHeapAddress(old)) return;
    uint64_t align = old & ~7;
    uint64_t shadow = align + OFFSET;
    uint64_t pack = *(uint64_t*) shadow;
    uint64_t beg = align - ((pack & 0xffffffff) <
    uint64_t end = align + ((pack >> 32) << 3);
    if (res < beg || res >= end)
        error("Heap out-of-bounds Detected");
}
```



Compiler Optimization

- **Runtime-Driven Checking Elimination**
- Directional Boundary Checking
- Security Pattern Identification
- Merge Metadata Extraction
- Redundant Checking Elimination

Compiler Optimization Runtime-Driven Checking Elimination

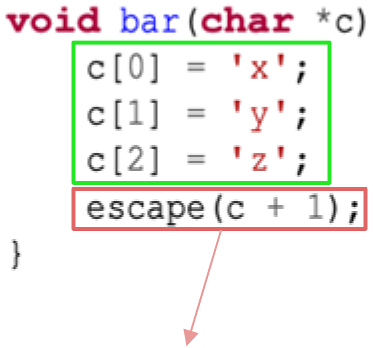
- If each heap chunk has **infinite space**, out-of-bounds access becomes impossible, rendering all boundary checks redundant and eliminable.
- It's **impractical** to allocate infinite or even very large spaces for every chunk due to the potential for high memory overhead.
- ShadowBound chooses an improved approach to **balance time overhead and memory overhead**. Specifically, ShadowBound **reserves a fixed n bytes** for every heap chunk, denoted as reserved space. Then, ShadowBound will try to find all eliminable boundary checks using the reserved space provided by the runtime.

Compiler Optimization

Runtime-Driven Checking Elimination

ShadowBound can remove the boundary checking if

- The offset between the result pointer and base pointer can be confirmed to be **less than n bytes at compile time.**
- The result pointer will **never be used as a base pointer** in another boundary checking.

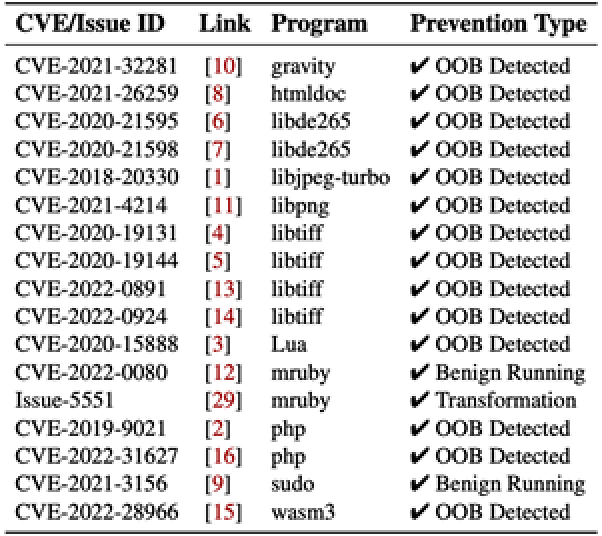


```
void bar(char *c) {
    c[0] = 'x';
    c[1] = 'y';
    c[2] = 'z';
    escape(c + 1);
}
```

The pointer c + 1 is passed to another function, indicating that it may potentially be used as a base pointer for boundary checking

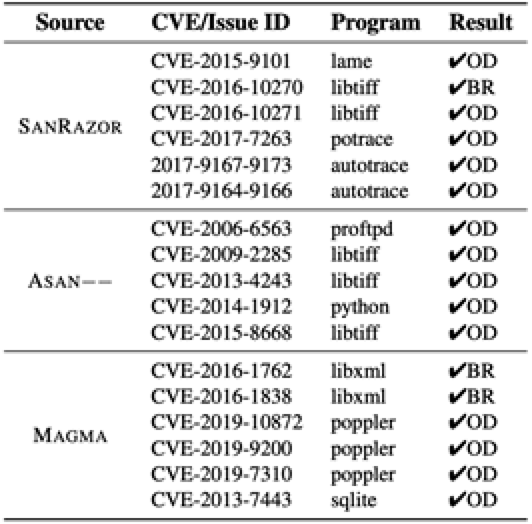
Security Evaluation Real World Vulnerabilities

- Safeguard 19 programs against 34 exploitable out-of-bound bugs.



CVE/Issue ID	Link	Program	Prevention Type
CVE-2021-32281	[10]	gravity	✔ OOB Detected
CVE-2021-26259	[8]	htmldoc	✔ OOB Detected
CVE-2020-21595	[6]	libde265	✔ OOB Detected
CVE-2020-21598	[7]	libde265	✔ OOB Detected
CVE-2018-20330	[1]	libjpeg-turbo	✔ OOB Detected
CVE-2021-4214	[11]	libpng	✔ OOB Detected
CVE-2020-19131	[4]	libtiff	✔ OOB Detected
CVE-2020-19144	[5]	libtiff	✔ OOB Detected
CVE-2022-0891	[13]	libtiff	✔ OOB Detected
CVE-2022-0924	[14]	libtiff	✔ OOB Detected
CVE-2020-15888	[3]	Lua	✔ OOB Detected
CVE-2022-0080	[12]	mruby	✔ Benign Running
Issue-5551	[29]	mruby	✔ Transformation
CVE-2019-9021	[2]	php	✔ OOB Detected
CVE-2022-31627	[16]	php	✔ OOB Detected
CVE-2021-3156	[9]	sudo	✔ Benign Running
CVE-2022-28966	[15]	wasm3	✔ OOB Detected

Table 2: Heap out-of-bounds Prevention Results for ShadowBound on Real-World Vulnerabilities.



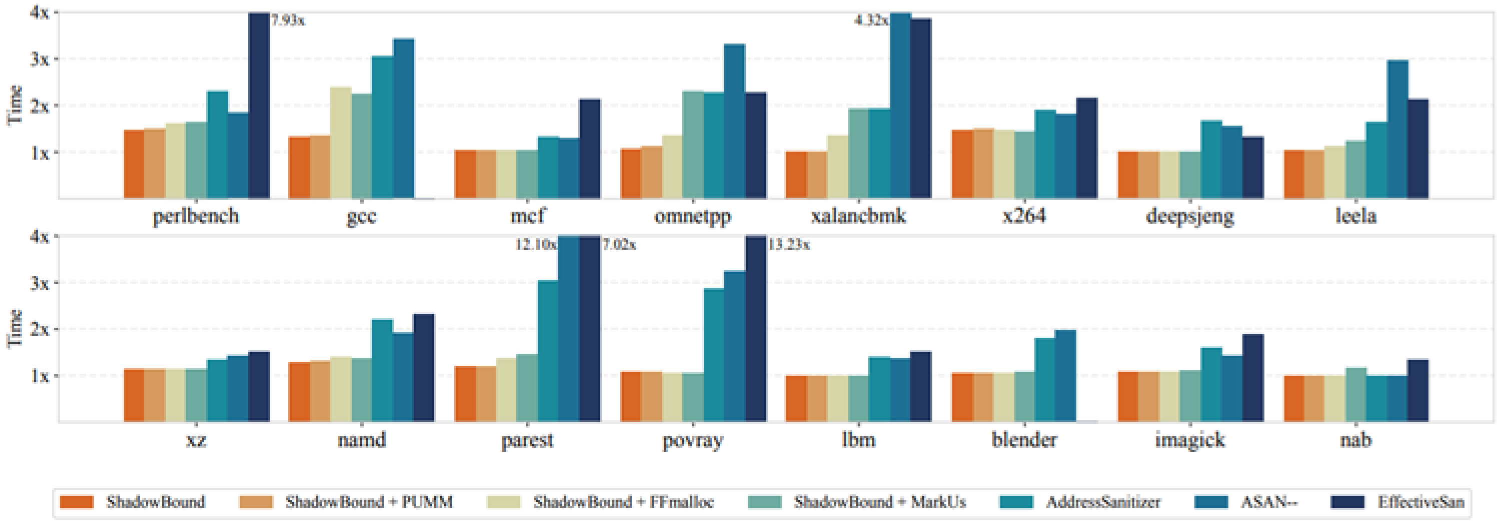
Source	CVE/Issue ID	Program	Result
SanRazor	CVE-2015-9101	lame	✔OD
	CVE-2016-10270	libtiff	✔BR
	CVE-2016-10271	libtiff	✔OD
	CVE-2017-7263	potrace	✔OD
	2017-9167-9173	autotrace	✔OD
	2017-9164-9166	autotrace	✔OD
ASan--	CVE-2006-6563	proftpd	✔OD
	CVE-2009-2285	libtiff	✔OD
	CVE-2013-4243	libtiff	✔OD
	CVE-2014-1912	python	✔OD
	CVE-2015-8668	libtiff	✔OD
Magma	CVE-2016-1762	libxml	✔BR
	CVE-2016-1838	libxml	✔BR
	CVE-2019-10872	poppler	✔OD
	CVE-2019-9200	poppler	✔OD
	CVE-2019-7310	poppler	✔OD
	CVE-2013-7443	sqlite	✔OD

Table 7: Security evaluation for ShadowBound on vulnabilities from prior works.

Performance Evaluation

SPEC CPU 2017

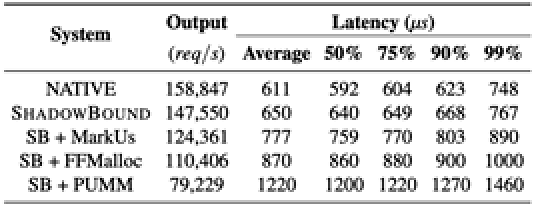
- On SPEC CPU 2017, the geomean time overhead of each system is **5.72%**, 6.60%, 9.95%, 16.20%, 62.03%, 79.85% and 138.76%.



Performance Evaluation

Real World Application

- We assessed using Nginx, Chakra, and Chromium. It introduces negligible overhead to the tested real-world programs.



System	Output	Latency (μs)				
	(req/s)	**Average**	**50%**	**75%**	**90%**	**99%**
NATIVE	158,847	611	592	604	623	748
SHADOWBOUND	147,550	650	640	649	668	767
SB + MarkUs	124,361	777	759	770	803	890
SB + FFMalloc	110,406	870	860	880	900	1000
SB + PUMM	79,229	1220	1200	1220	1270	1460

Table 4: Evaluation Results of Native, SHADOWBOUND and its variants: Output and Latency Analysis on Nginx. In the Latency column, Average denotes the average latency of the requested connections, while the remaining values depict latency distribution.

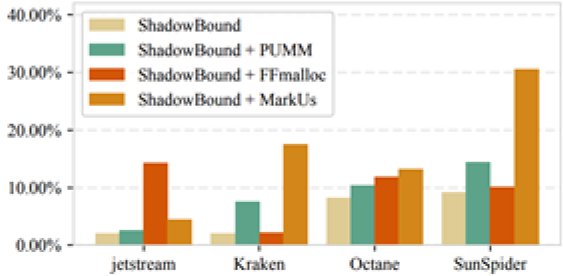
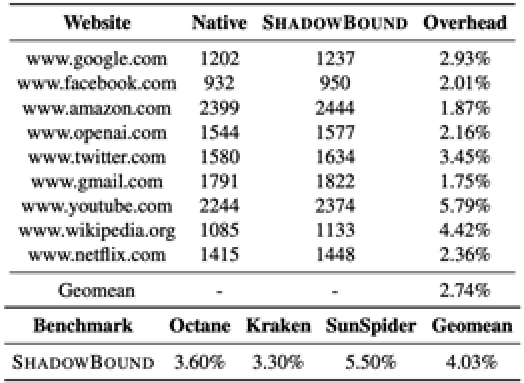


Figure 4: Runtime overhead comparison of SHADOWBOUND and its variants on the Chakra engine: The geometric mean overhead for each system is 4.17%, 7.28%, 7.86%, 13.28%.



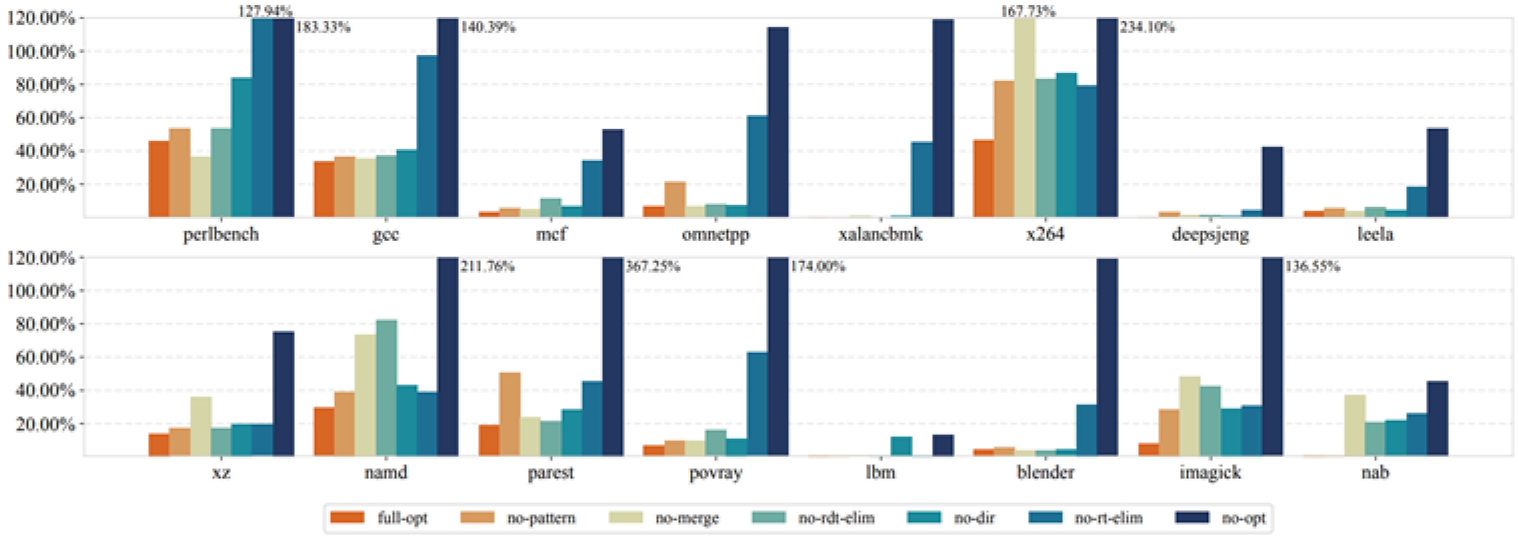
Website	Native	SHADOWBOUND	Overhead
www.google.com	1202	1237	2.93%
www.facebook.com	932	950	2.01%
www.amazon.com	2399	2444	1.87%
www.openai.com	1544	1577	2.16%
www.twitter.com	1580	1634	3.45%
www.gmail.com	1791	1822	1.75%
www.youtube.com	2244	2374	5.79%
www.wikipedia.org	1085	1133	4.42%
www.netflix.com	1415	1448	2.36%
Geomean	-	-	2.74%

Benchmark	Octane	Kraken	SunSpider	Geomean
SHADOWBOUND	3.60%	3.30%	5.50%	4.03%

Table 5: Runtime overhead on Chromium: website loading times and JavaScript benchmarks.

Ablation Study

- The ablation study is used to to understand the performance of each compiler optimization.



Conclusion

- **Efficient Protection**: ShadowBound uses a novel metadata design to quickly fetch pointer boundaries, ensuring compatibility with various Use-After-Free defenses and providing minimal overhead.

- **Optimized Performance**: ShadowBound implements custom optimization techniques for boundary checking, significantly reducing time overhead.

- **Proven Effectiveness**: Evaluations show ShadowBound consistently provides robust memory protection with minimal overhead in benchmarks and real-world applications.

Thank You

- Zheng Yu
- zheng.yu@northwestern.edu
- Twitter: @dataisland99
- **Seeking intern, visiting and collaboration opportunities.**