

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

Zheng Yu

09/25/2024





Agenda

- Background: Heap Memory Errors and Defenses
- Threat Model
- Design & Metadata Management
- Compiler Optimization
- Evaluation and Results



Agenda

- Background: Heap Memory Errors and Defenses
- Threat Model
- Design & Metadata Management
- Compiler Optimization
- Evaluation and Results

Heap Memory Errors

- C/C++ inherently lack heap memory safety mechanisms.
- 2023 CWE top-most dangerous software weaknesses.
- Exploiting these vulnerabilities even can lead to 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 Errors

- Occurs when a program accesses memory that has already been freed or is no longer valid. (**Use After Free, Double Free** etc)
- Many prior works have focused on developing defenses against temporal memory vulnerabilities, with several notable for their **low overhead**.

```
void main() {  
    char *buf = malloc(16);  
    free(buf);  
    buf[0] = 'y';  
}
```

Temporal Memory Errors

- Occurs when a program accesses memory that has already been freed or is no longer valid. (**Use After Free**, **Double Free** etc)
- Many prior works have focused on developing defenses against temporal memory vulnerabilities, with several notable for their **low overhead**.

Method	Conference	Overhead
MarkUs	S&P 20	10.01%
FFMalloc	Security 21	2.30%
PUMM	Security 23	2.04%

Spatial Memory Errors

- Spatial memory errors occur when a program accesses memory outside the allocated bounds.
- While there also many defenses against spatial memory errors have been proposed, most of them suffer from **performance, compatibility or security** issues.

```
void main() {  
    char *buf = malloc(16);  
    buf[32] = 'x';  
}
```



Spatial Memory Defense

Performance Issue

- **X** > 50%: *SoftBound* (PLDI 09) / *ASAN* (ATC 12) / *LowFat* (CCS 16) / *ESAN* (PLDI 18) / *SanRazor* (OSDI 21) / *ASAN--* (USENIX Security 22)
- **X** > 30%: *SGXBound* (EuroSys 17) / *DeltaPointer* (EuroSys 18) / *PACMem* (CCS 22) / *TailCheck* (OSDI 23)



Spatial Memory Defense

Compatibility Issue

- **✗ Compatibility with UAF defense:** Since SOTA UAF defenses introduce custom allocators, OOB defenses that also implement their own allocators may face challenges in integrating with UAF defenses. (*LowFat / ESAN / TailCheck*)
- **✗ Compatibility in Large-scale Program:** Some defenses reduce the available memory space to a very limited range (4GB), making them difficult to deploy them in large-scale programs. (*SGXBound / DeltaPointer*)






Spatial Memory Defense

Security Issue

- **✗ Bypassed by Non-linear Overflow:** Some tools are inherently designed for detection or debugging and are not suitable for spatial memory defense, as they can be bypassed by non-linear overflows. (*ASAN, SanRazor, ASAN-- , TailCheck*)
- **✗ Bypassed by Underflow:** Some defenses forego underflow checks to minimize performance overhead. (*TailCheck / DeltaPointer*)



ShadowBound

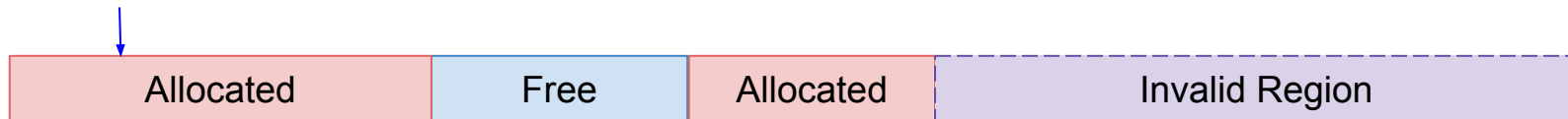
-  Low performance overhead, approximately 6%.
-  Works seamlessly with UAF defenses, scalable to large programs.
-  Provide robust spatial memory security.



Agenda

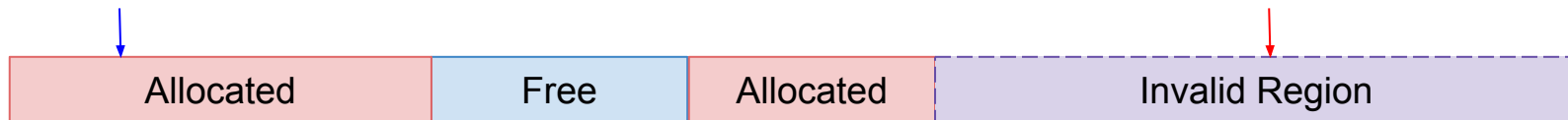
- Background: Heap Memory Errors and Defenses
- Threat Model
- Design & Metadata Management
- Compiler Optimization
- Evaluation and Results

Threat Model



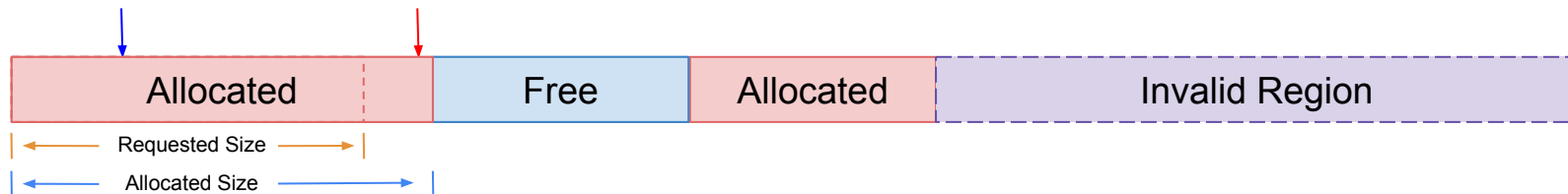
- **Inaccessible:** If the system has removed permission to access the associated address, the OOB will result in a crash.
- **Accessible & No Overlap:** The address remains accessible but still falls within the original heap chunk, without overlapping with other heap chunks or the freed regions.
- **Accessible & Overlap:** The memory remains accessible, and the associated address may overlap with another region.

Threat Model



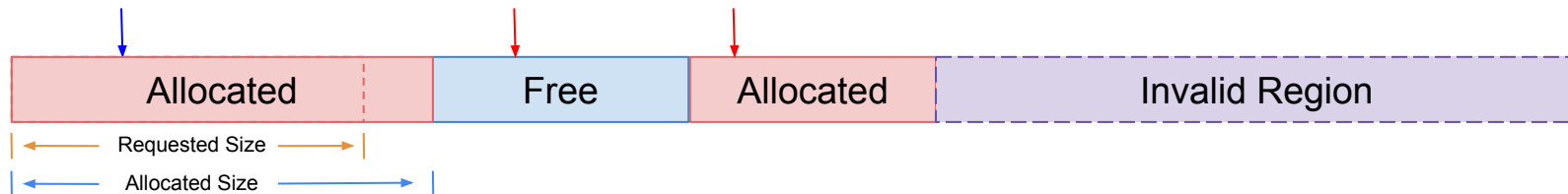
- **Inaccessible:** If the system has removed permission to access the associated address, the OOB will result in a crash.
- **Accessible & No Overlap:** The address remains accessible but still falls within the original heap chunk, without overlapping with other heap chunks or the freed regions.
- **Accessible & Overlap:** The memory remains accessible, and the associated address may overlap with another region.

Threat Model



- **Inaccessible:** If the system has removed permission to access the associated address, the OOB will result in a crash.
- **Accessible & No Overlap:** The address remains accessible but still falls within the original heap chunk, without overlapping with other heap chunks or the freed regions.
- **Accessible & Overlap:** The memory remains accessible, and the associated address may overlap with another region.

Threat Model



- **Inaccessible:** If the system has removed permission to access the associated address, the OOB will result in a crash.
- **Accessible & No Overlap:** The address remains accessible but still falls within the original heap chunk, without overlapping with other heap chunks or the freed regions.
- **Accessible & Overlap:** The memory remains accessible, and the associated address may overlap with another region.

Threat Model

ShadowBound can be deployed either in **conjunction with** other UAF defense mechanisms or **independently**. When deployed alongside other UAF defenses, it is presumed that the target program **contains one or more heap out-of-bounds and use-after-free** vulnerabilities. If *ShadowBound* is used independently, the assumption is **limited to the presence of heap out-of-bounds** vulnerabilities. In this threat model, an attacker can only attempt to exploit these vulnerabilities to potentially escalate privileges. Our goal is to **prevent these vulnerabilities to being exploitable**.



Agenda

- Background: Heap Memory Errors and Defenses
- Threat Model
- Design & Metadata Management
- Compiler Optimization
- Evaluation and Results



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

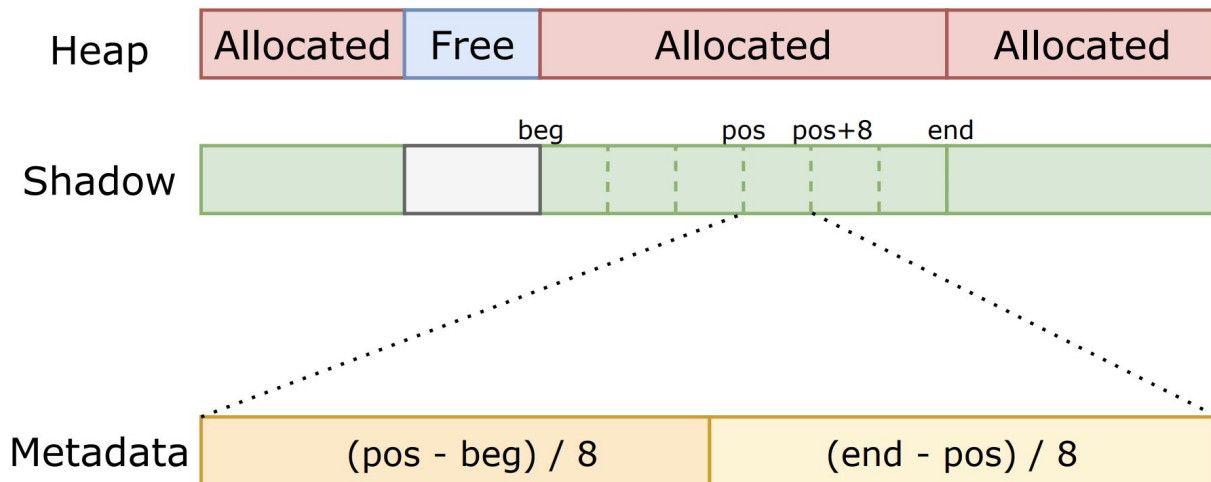
How we store each pointer's boundary?



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?

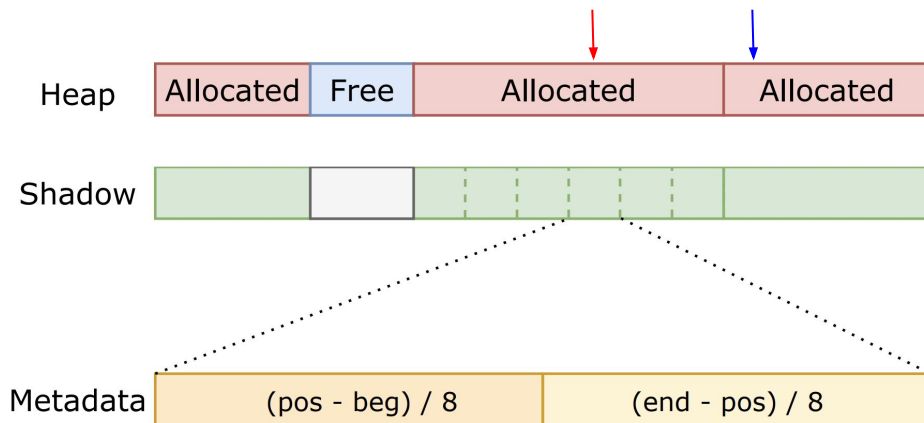


Why 64 bits is enough to save two `size_t` variables?

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} bytes).

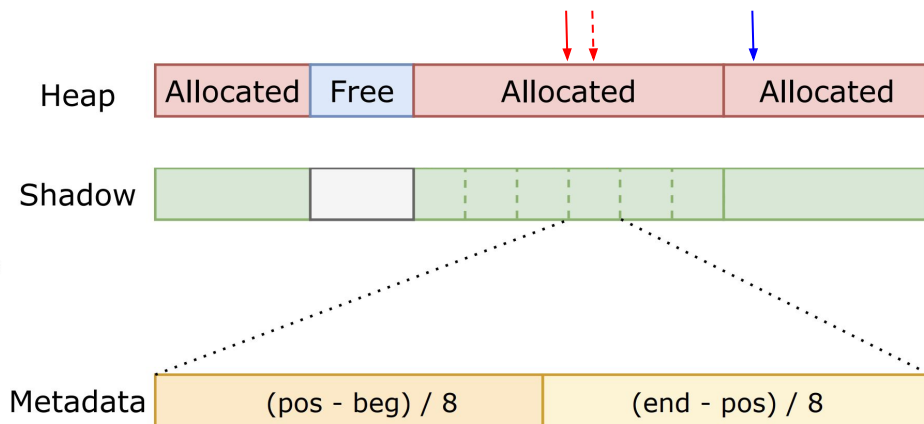
Boundary Checking

```
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) << 3);  
    uint64_t end = align + ((pack >> 32) << 3);  
    if (res < beg || res >= end)  
        error("Heap out-of-bounds Detected");  
}
```



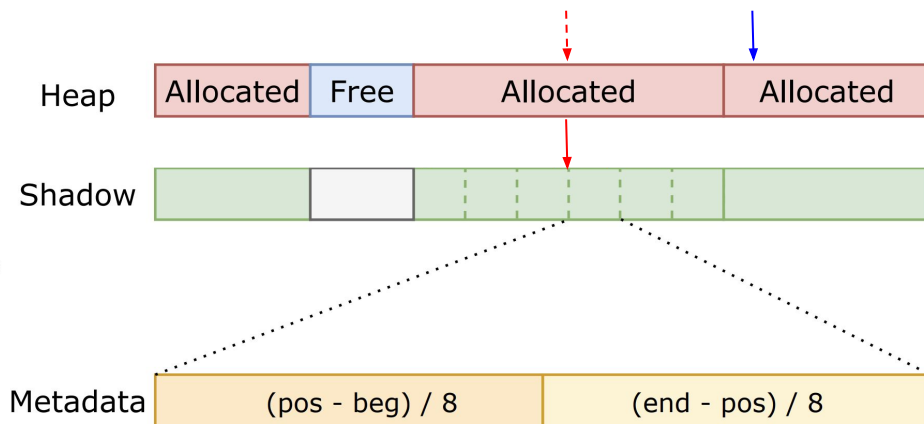
Boundary Checking

```
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) << 3);  
    uint64_t end = align + ((pack >> 32) << 3);  
    if (res < beg || res >= end)  
        error("Heap out-of-bounds Detected");  
}
```



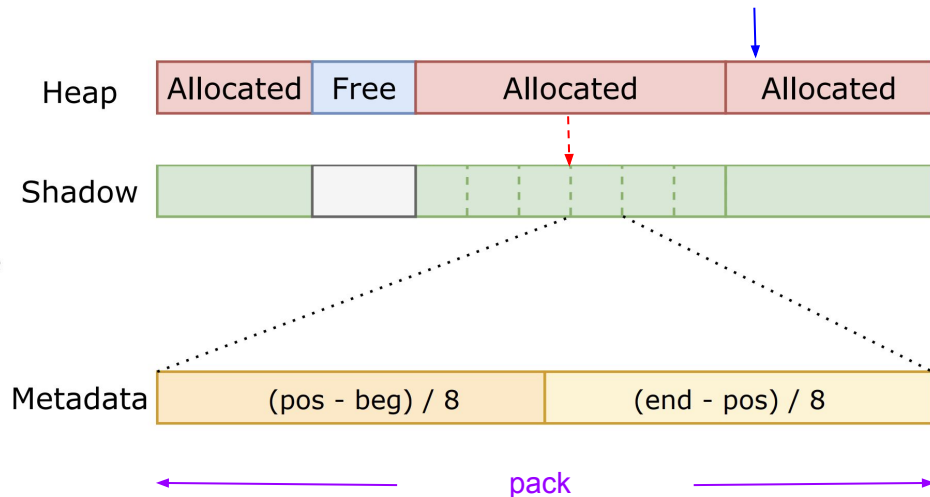
Boundary Checking

```
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) << 3);  
    uint64_t end = align + ((pack >> 32) << 3);  
    if (res < beg || res >= end)  
        error("Heap out-of-bounds Detected");  
}
```



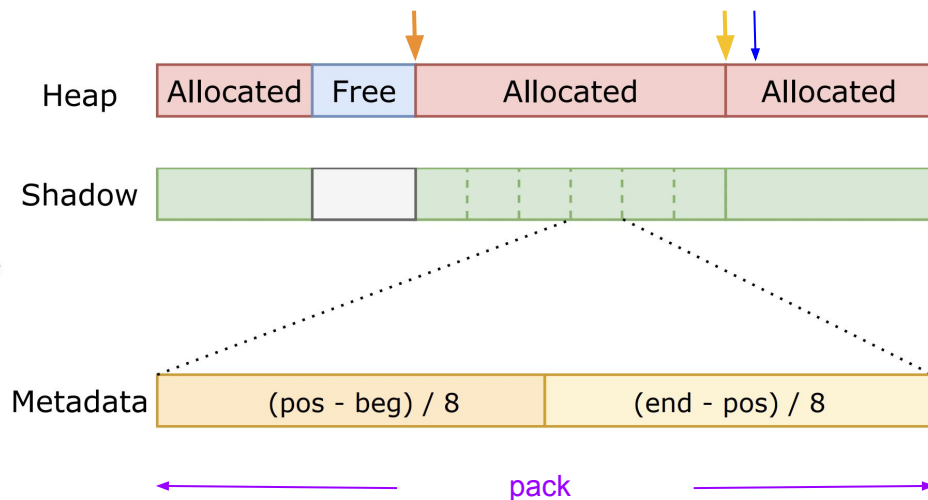
Boundary Checking

```
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) << 3);
    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) << 3);
    uint64_t end = align + ((pack >> 32) << 3);
    if (res < beg || res >= end)
        error("Heap out-of-bounds Detected");
}
```





Agenda

- Background: Heap Memory Errors and Defenses
- Threat Model
- Design & Metadata Management
- Compiler Optimization
- Evaluation and Results



Compiler Optimization

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



Compiler Optimization

- Runtime-Driven Checking Elimination
- Directional Boundary Checking
- Merge Metadata Extraction
- Security Pattern Identification - *DataGuard* (NDSS 22)
- Redundant Checking Elimination - *PACMem* (USENIX Security 23)



Compiler Optimization

- **Runtime-Driven Checking Elimination**
- Directional Boundary Checking
- Merge Metadata Extraction



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. However, 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);  
}
```



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) {  
    bound_check(c, &c[0]);  
    c[0] = 'x';  
    bound_check(c, &c[1]);  
    c[1] = 'y';  
    bound_check(c, &c[2]);  
    c[2] = 'z';  
    bound_check(c, c + 1);  
    escape(c + 1);  
}
```



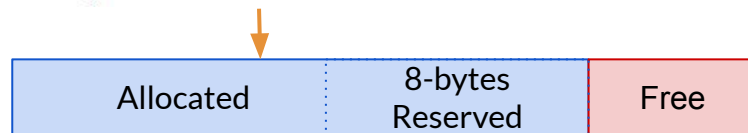
Compiler Optimization

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) {  
    bound_check(c, &c[0]);  
    c[0] = 'x';  
    bound_check(c, &c[1]);  
    c[1] = 'y';  
    bound_check(c, &c[2]);  
    c[2] = 'z';  
    bound_check(c, c + 1);  
    escape(c + 1);  
}
```



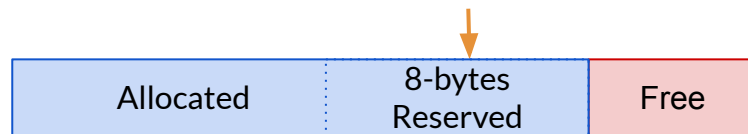
Compiler Optimization

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';  
    bound_check(c, c + 1);  
    escape(c + 1);  
}
```



Compiler Optimization

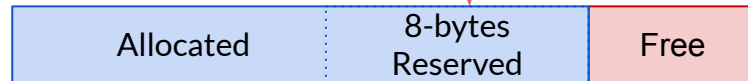
Runtime-Driven Checking Elimination

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';  
    bound_check(c, c + 1);  
    escape(c + 1);  
}
```

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





Compiler Optimization

- Runtime-Driven Checking Elimination
- **Directional Boundary Checking**
- Merge Metadata Extraction



Compiler Optimization

Directional Boundary Checking

- Boundary checking consists of two parts: an underflow check and an overflow check.
- By determining the sign (positive or negative) of the offset between the base pointer and the result pointer, we can optimize the process by inserting only the necessary check for one side.

```
void bar(struct obj *o) {  
    for (int i = 0; i < o->len; ++i) {  
        uint64_t lbound = ...;  
        uint64_t rbound = ...;  
        if (o->a + i < lbound || rbound <= o->a + i + 1)  
            error("Heap out-of-bound detected");  
        other_function(&o->a[i])  
    }  
}
```



Compiler Optimization

Directional Boundary Checking

- Boundary checking consists of two parts: an underflow check and an overflow check.
- By determining the sign (positive or negative) of the offset between the base pointer and the result pointer, we can optimize the process by inserting only the necessary check for one side.

```
void bar(struct obj *o) {  
    for (int i = 0; i < o->len; ++i) {  
        uint64_t lbound = ...;  
        uint64_t rbound = ...;  
        if (rbound <= o->a + i + 1)  
            error("Heap out-of-bound detected");  
        other_function(&o->a[i])  
    }  
}
```


Compiler Optimization

Directional Boundary Checking

- Boundary checking consists of two parts: an underflow check and an overflow check.
- By determining the sign (positive or negative) of the offset between the base pointer and the result pointer, we can optimize the process by inserting only the necessary check for one side.

```
void bar(struct obj *o) {  
    for (int i = 0; i < o->len; ++i) {  
        uint64_t lbound = ...;  
        uint64_t rbound = ...;  
        if (rbound <= o->a + i + 1)  
            error("Heap out-of-bound detected");  
        other_function(&o->a[i])  
    }  
}
```

Hard to get the properties of the pointer from within the callee function.

Compiler can consistently utilize the computing process information



Compiler Optimization

- Runtime-Driven Checking Elimination
- Directional Boundary Checking
- **Merge Metadata Extraction**



Compiler Optimization

Merge Metadata Extraction

- Boundary checking contain two stage:
 - The extraction stage to obtain the **base pointer's** boundary.
 - Followed by a subsequent checking stage to validate the **result pointer**.
- The extraction stage is only determined by the base pointer, which provide the opportunity to merge them for different result pointer.

```
void foo(char *p) {  
    char *a = p + 1;  
    char *b = a + 2;  
    char *c, *d;  
    if (random() > 0.5)  
        c = a + 3;  
    else  
        c = b + 4;  
  
    for (d = c; d < p + 100; d++)  
        *d = 'x';  
}
```



Agenda

- Background: Heap Memory Errors and Defenses
- Threat Model
- Design & Metadata Management
- Compiler Optimization
- Evaluation and Results

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 vulnerabilities from prior works.

Security Evaluation

Synthesis Vulnerabilities

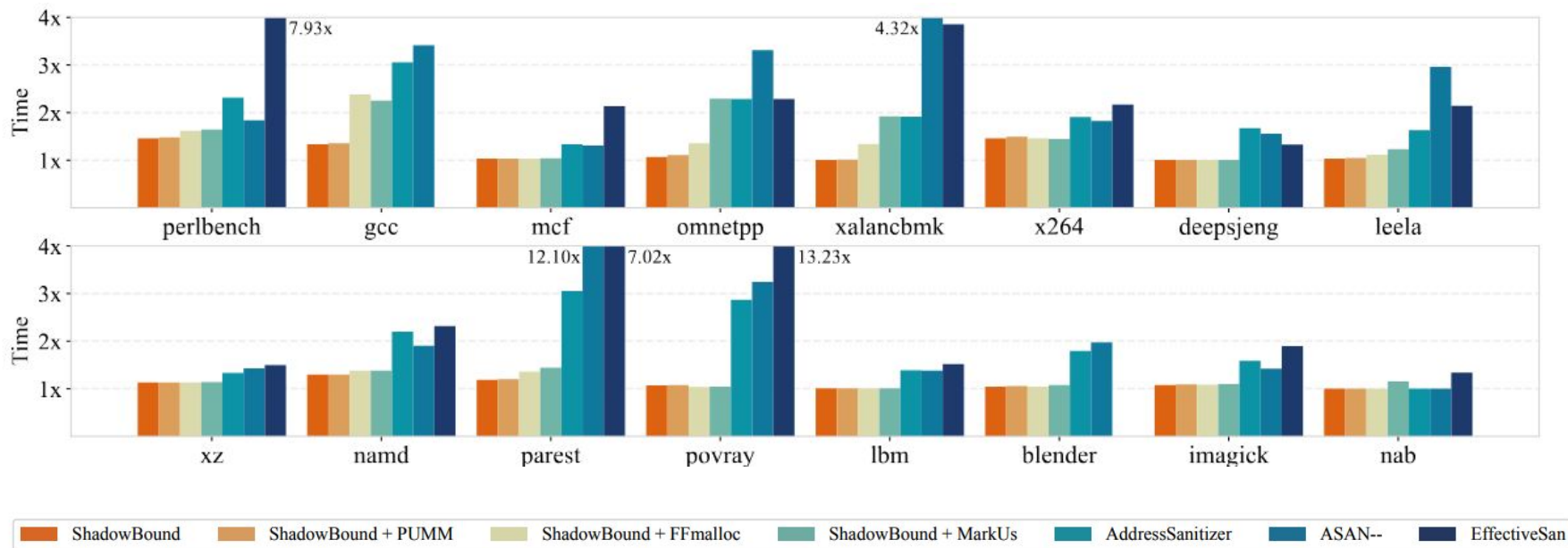
- we undertook synthesis vulnerability testing, generating 244 inputs to trigger out-of-bounds bugs in various ways.

Program	#Input	#OD	#BR	#TF
cxxflit	1	1	0	0
libpcap	4	2	2	0
libxml2_reader	127	127	0	0
libxml2_xml	61	46	15	0
proj4	3	0	3	0
zstd	48	45	3	0
Total	244	221	23	0

Table 3: Heap out-of-bounds Prevention Results for SHAD-OWBOUND on Synthesis Vulnerabilities.

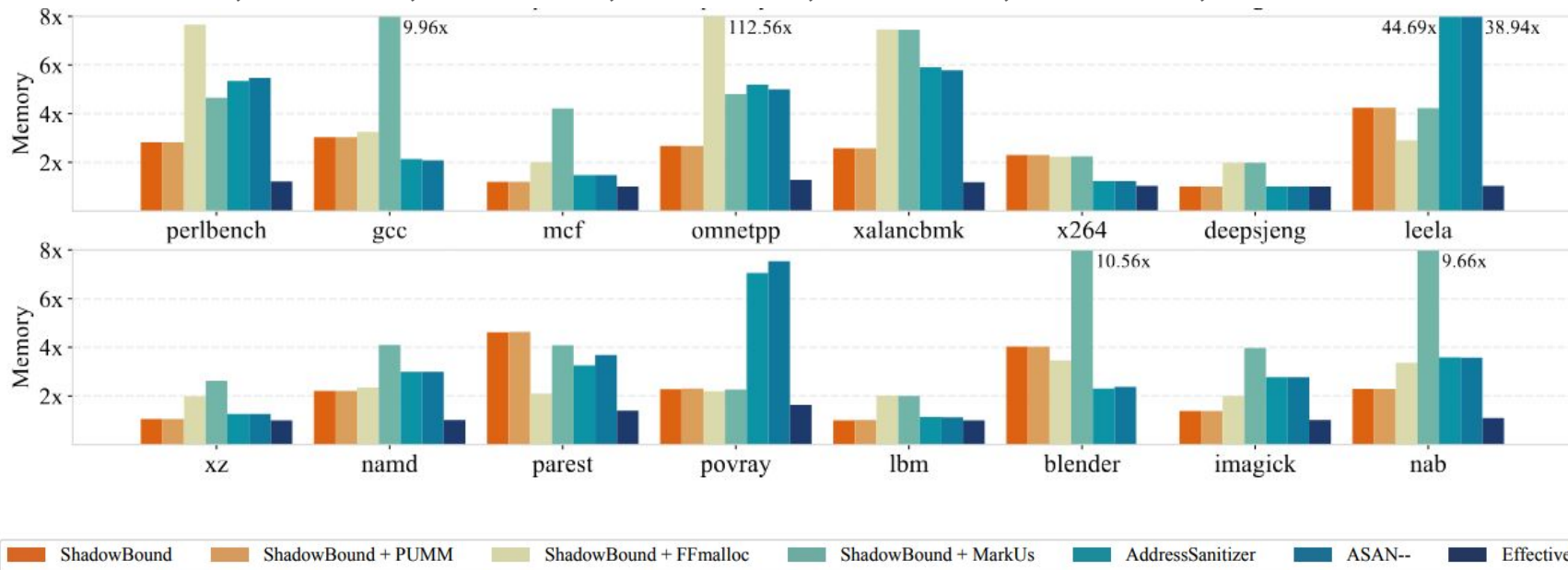
Performance Evaluation SPEC CPU 2017

- 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 SPEC CPU 2017

- On SPEC CPU 2017, the geomean memory overhead of each system is 54.59%, 55.29%, 218.20%, 302.51%, 116.55%, 112.33%, 2.70%



Performance Evaluation Real World Application

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

System	Output (req/s)	Latency (μ 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.

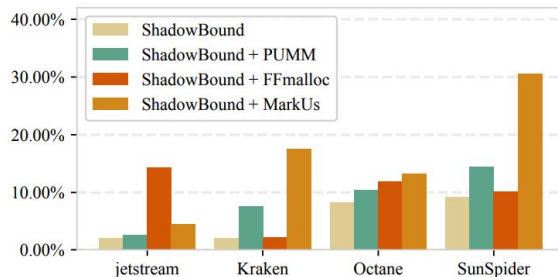


Figure 4: Runtime overhead comparison of SHADOWBOUND and its variants on the Chakra engine: The geometric mean overhead for each system is 4.17%, 7.28%, 7.86%, 13.28%.

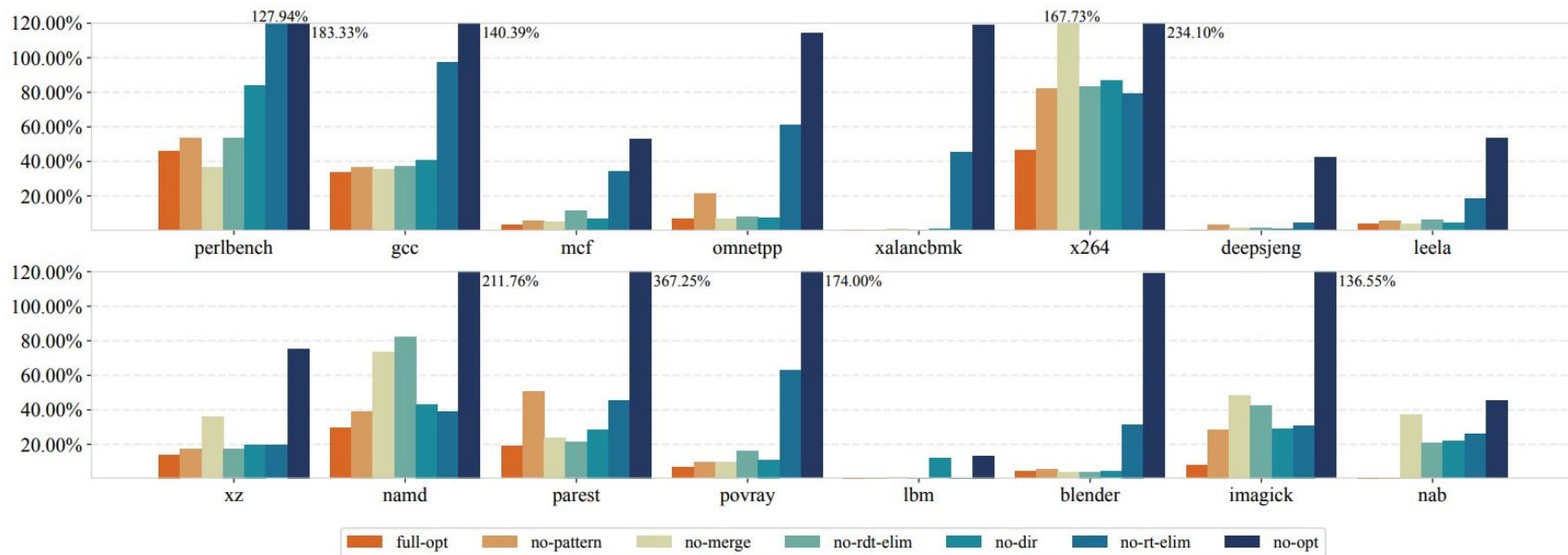
Website	Native	SHADOWBOUND	Overhead
www.google.com	1202	1237	2.93%
www.facebook.com	932	950	2.01%
www.amazon.com	2399	2444	1.87%
www.openai.com	1544	1577	2.16%
www.twitter.com	1580	1634	3.45%
www.gmail.com	1791	1822	1.75%
www.youtube.com	2244	2374	5.79%
www.wikipedia.org	1085	1133	4.42%
www.netflix.com	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 bars show the time overhead of ShadowBound with full optimization, ShadowBound with each optimization disabled, and ShadowBound without optimization. The geomean value is 5.72%, 9.51%, 11.56%, 11.76%, 12.86%, 29.28% and 99.69%





Takeaways

- **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.

Zheng Yu ([@dataisland99](https://twitter.com/dataisland99))

<https://dataisland.org>

zheng.yu@northwestern.edu