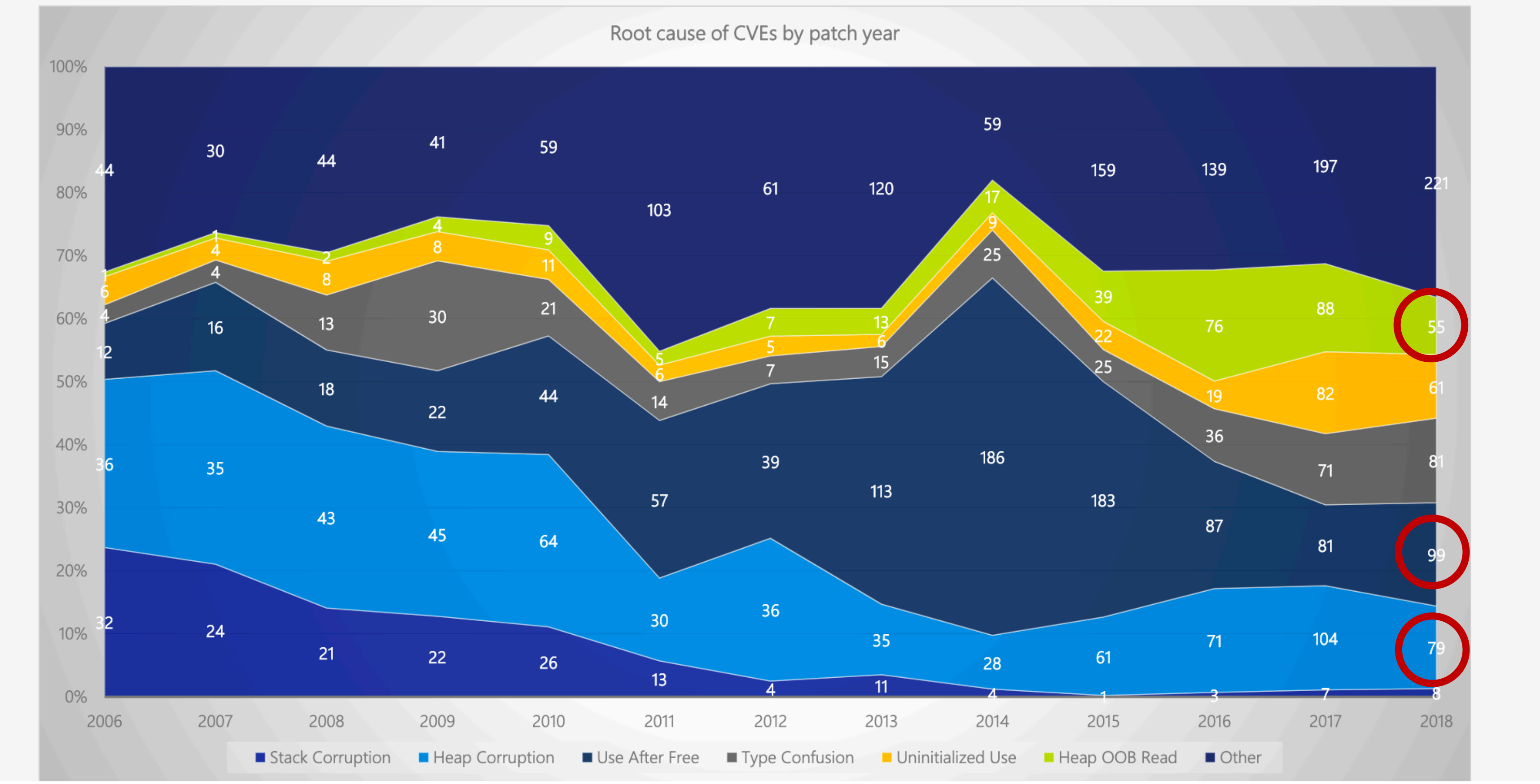
Automatic Techniques to Systematically Discover New Heap Exploitation Primitives

Insu Yun, Dhaval Kapil, and Taesoo Kim

Georgia Institute of Technology

Heap vulnerabilities are the most common, yet serious security issues.



$$\%\ of\ heap\ vulnerabilities = \frac{233}{604} = 39\%$$

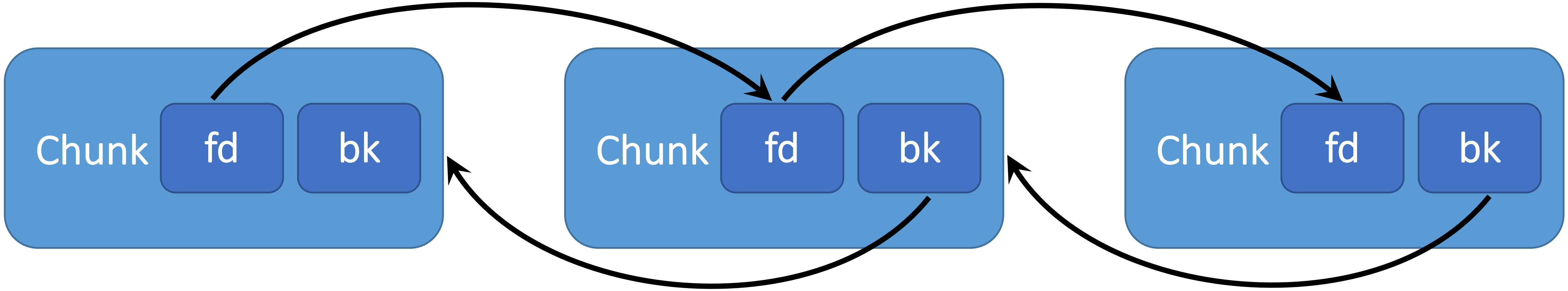
From “Killing Uninitialized Memory: Protecting the OS Without Destroying Performance”, Joe Bialek and Shayne Hiet-Block, CppCon 2019

Heap exploitation techniques (HETs) are preferable methods to exploit heap vulnerabilities

- Abuse underlying allocator to achieve more powerful primitives (e.g., arbitrary write) for control hijacking
  - Application-agnostic: rely on only underlying allocators
  - Powerful: e.g., off-by-one null byte overflow → arbitrary code execution

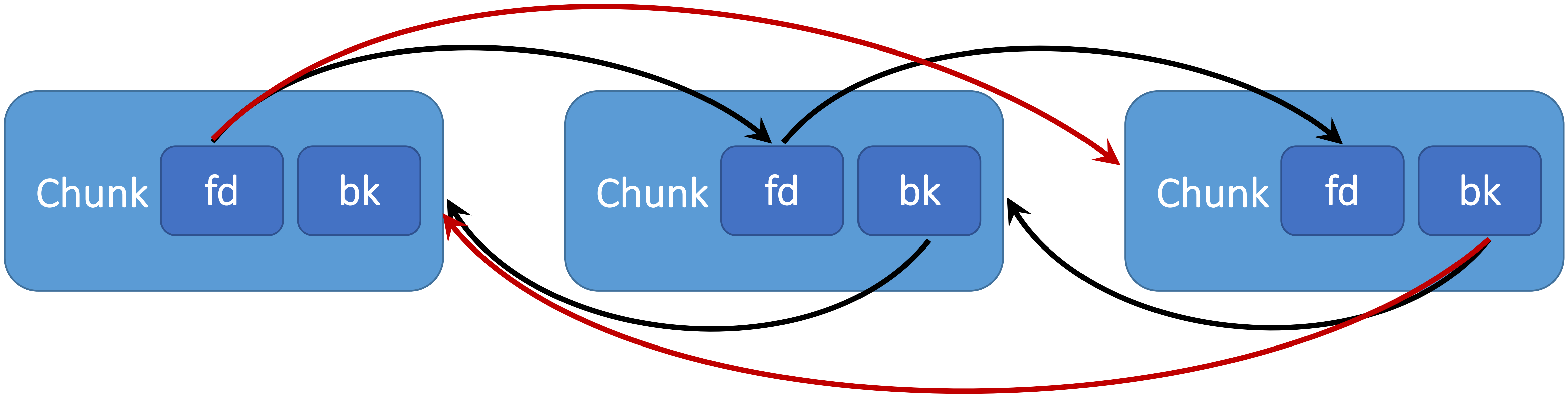
- Used to compromise (in 2019)

Example: unlink() in ptmalloc2



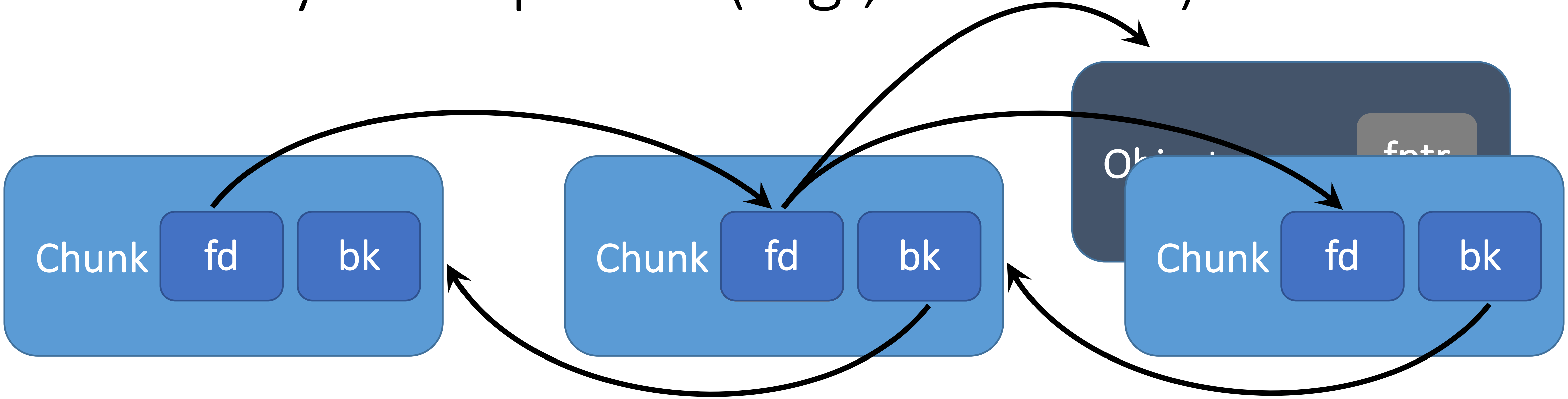

```
unlink():   P->fd->bk = P->bk
            P->bk->fd = P->fd
```

Example: unlink() in ptmalloc2




```
unlink():  P->fd->bk = P->bk
           P->bk->fd = P->fd
```

Example: Unsafe unlink() in the presence of memory corruptions (e.g., overflow)




```
unlink():   P->fd->bk = P->bk
```

=> fptr = evil

Security checks are introduced in the allocator to prevent such exploitations

```
unlink():   assert(P->fd->bk == P);
            P->fd->bk = P->bk
```

This check is still *bypassable*,
but it makes HET more *complicated*

Researchers have been studied reusable HETs to handle such complexities

Title : Once upon a free()

Author : anonymous author

Understanding t
breaking it

Project Zero

News and updates from the Project Zero team at Google

All analyses are manual, ad-hoc, and allocator-specific!

Exploiting the wi

From: "Phantasmal Phantasm
Date: Mon, 23 Feb 2004 21:5

Posted by Chris Evans, Exploit Writer Underling to Tavis Ormandy

Problem 1: Existing analyses are highly biased to certain allocators

ptmalloc2 (Linux allocator)

tcmalloc

jemalloc

DieHarder

mimalloc

mesh

scudo

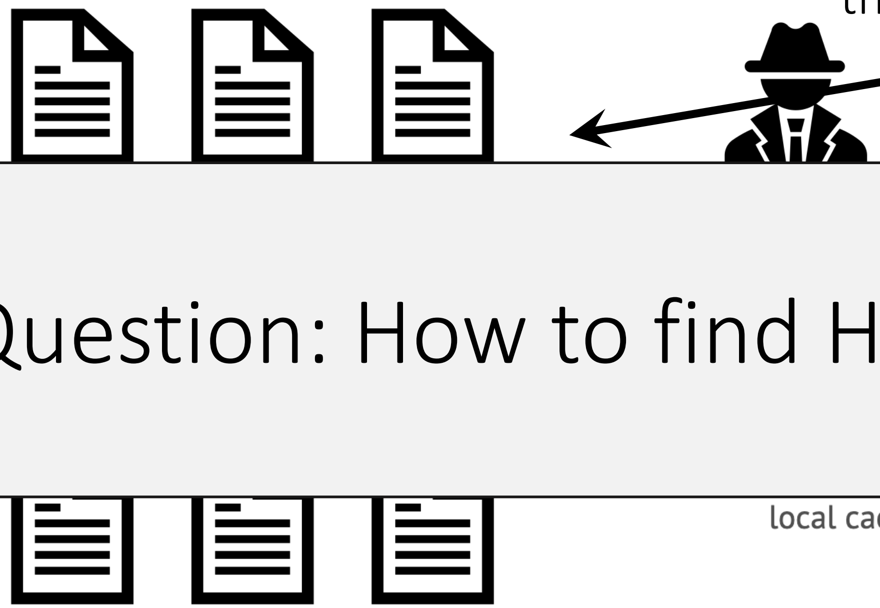
Freeguard

?

Problem2: A manual re-analysis is required in the changes of an allocator's implementation

ptmalloc2 (Linux allocator)

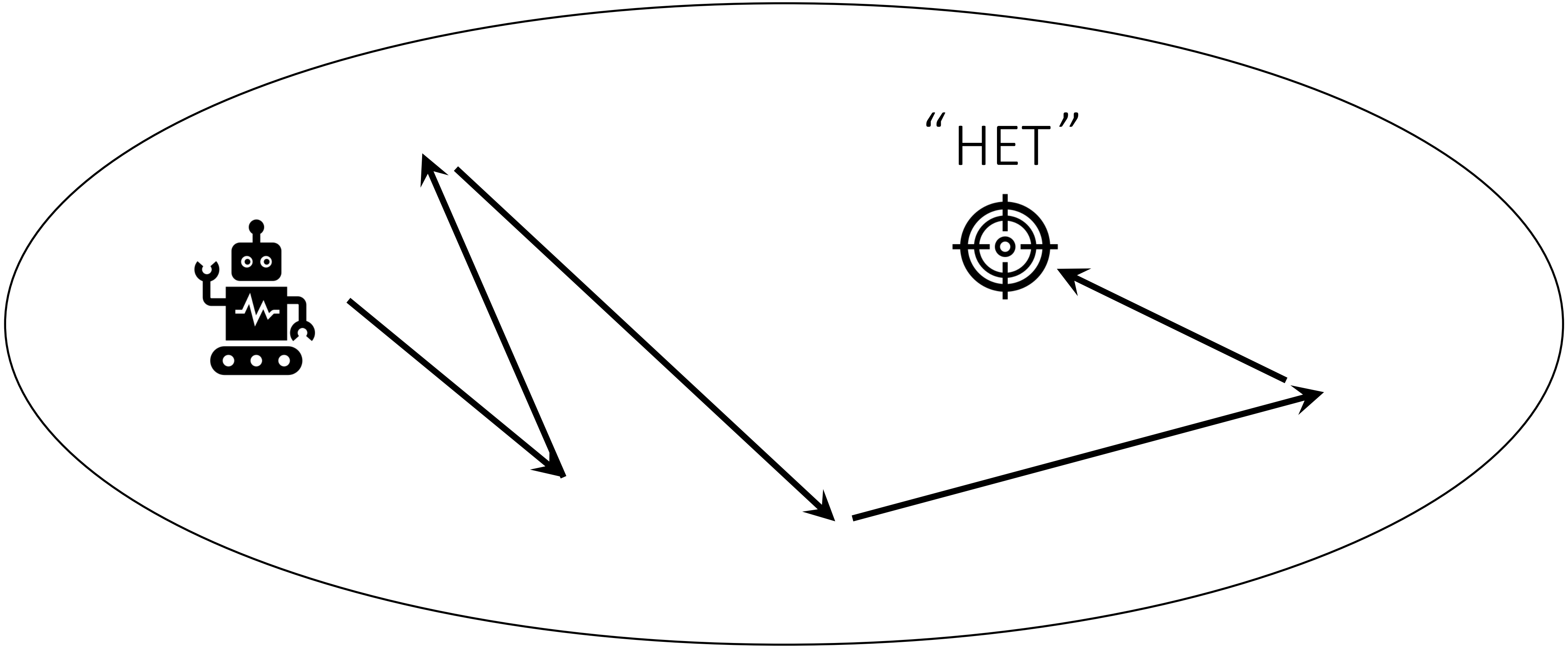
A new feature:
thread-local cache (tcache)



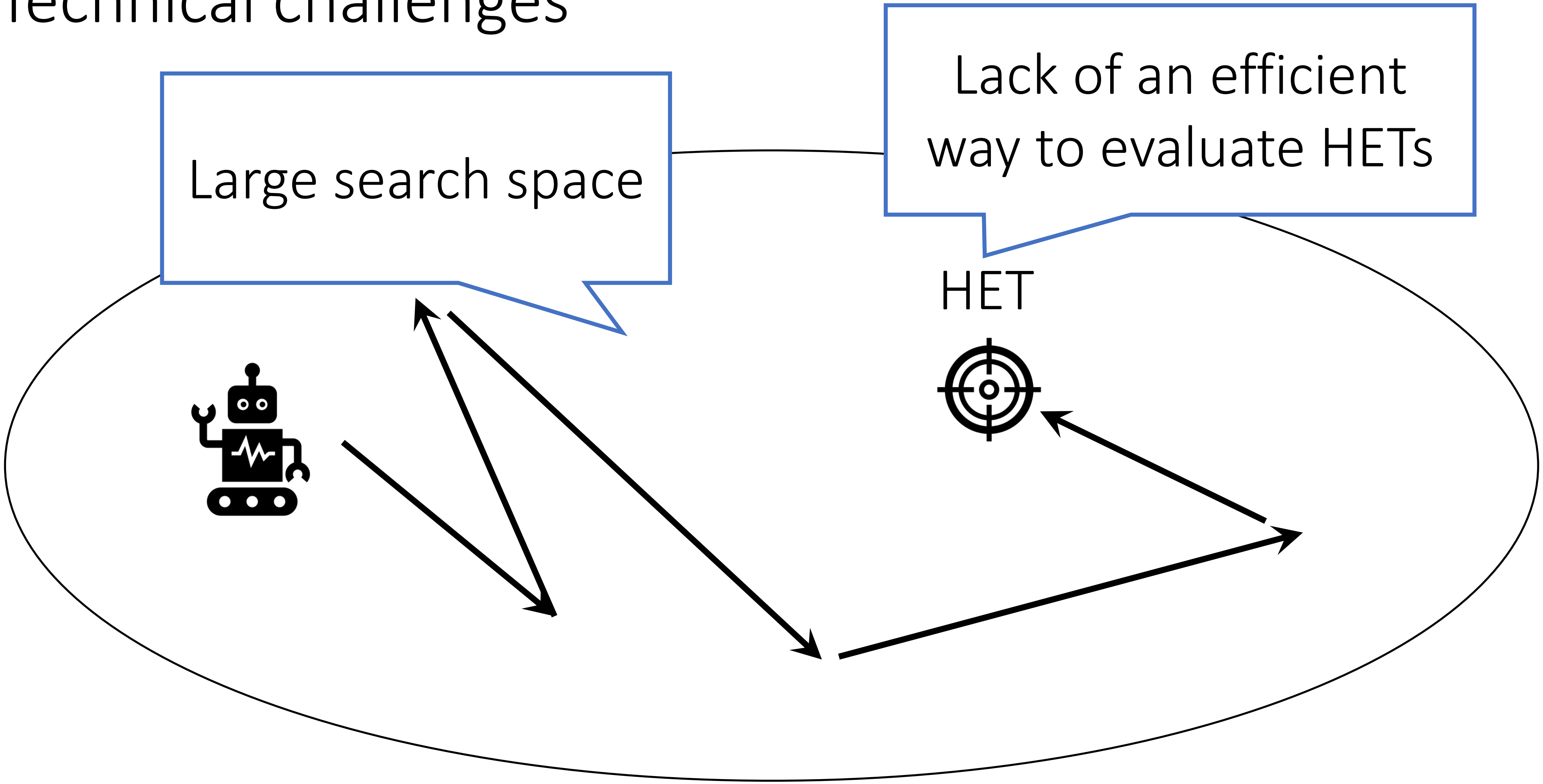
Question: How to find HETs automatically?

local caching, a recent addition to glibc malloc.

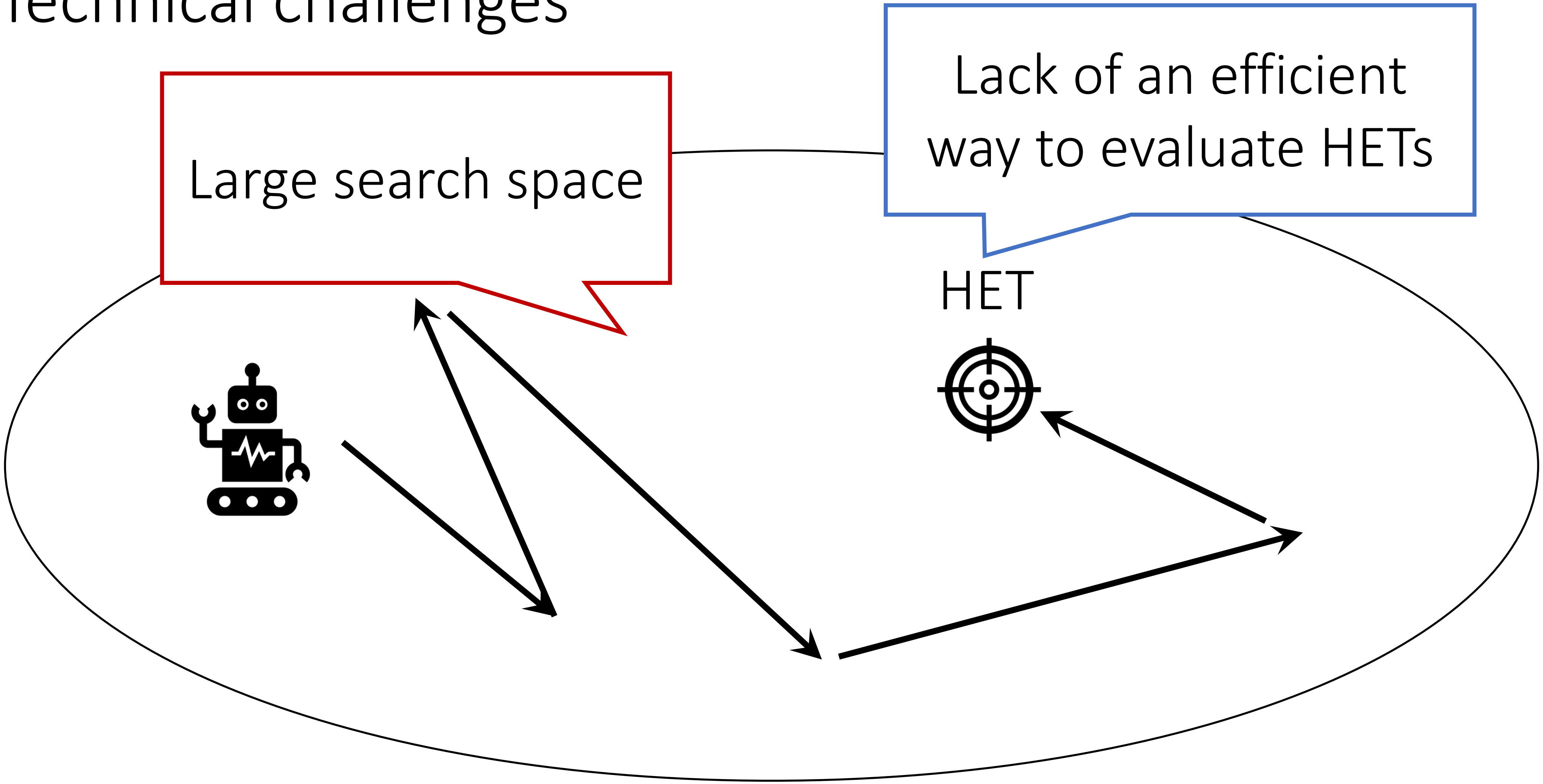
Our key idea: ArcHeap autonomously explore spaces similar to fuzzing!



Technical challenges



Technical challenges



Search space consisting of heap actions is enormous

2^{64} → `malloc(sz)`
Allocation

`free(p)`
Deallocation

`size(p)` x 2^{64} → `p[i]=v`
Heap write

`buf[i]=v`
Buffer write

Search space can be reduced using model-based search based on *common designs* of allocators!

Overflow

Write-after-free

Double free

Arbitrary free

Buggy actions

Common design 1: Binning

- Specially managing chunks in different size groups
  - Small chunks: Performance is more important
  - Large chunks: Memory footprint is more important
- e.g., ptmalloc
  - fast bin (< 128 bytes): no merging in free chunks
  - small bin (< 1024 bytes): merging is enabled

- Sampling a size uniformly in the 2^{64} space ➔ P(fast bin) = 2^{-57}

ArcHeap selects an allocation size aware of binning

- Sampling in exponentially distant size groups

- ArcHeap partitions an allocation size into four groups:
  $(2^0, 2^5]$, $(2^5, 2^{10}]$, $(2^{10}, 2^{15}]$, and $(2^{15}, 2^{20}]$

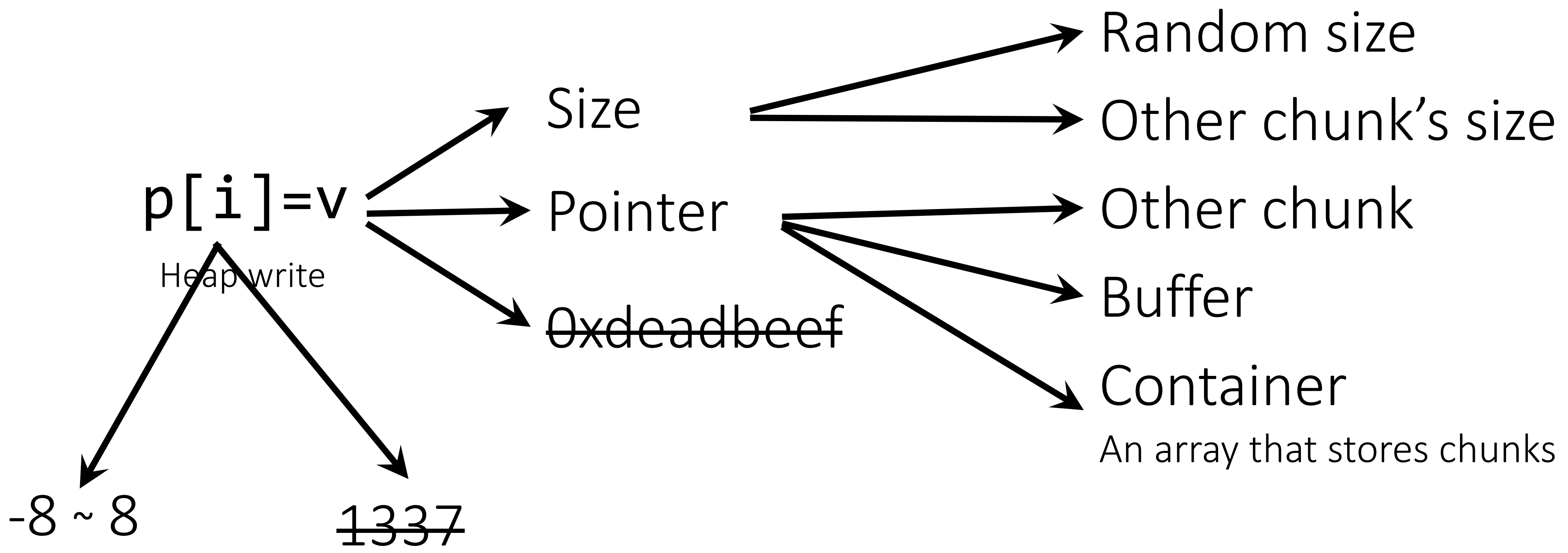
- Then, it selects a group and then selects a size in the group uniformly
  - e.g., P(fast bin) > P(selecting a first group) = ¼

Other common designs: Cardinal data and In-place metadata

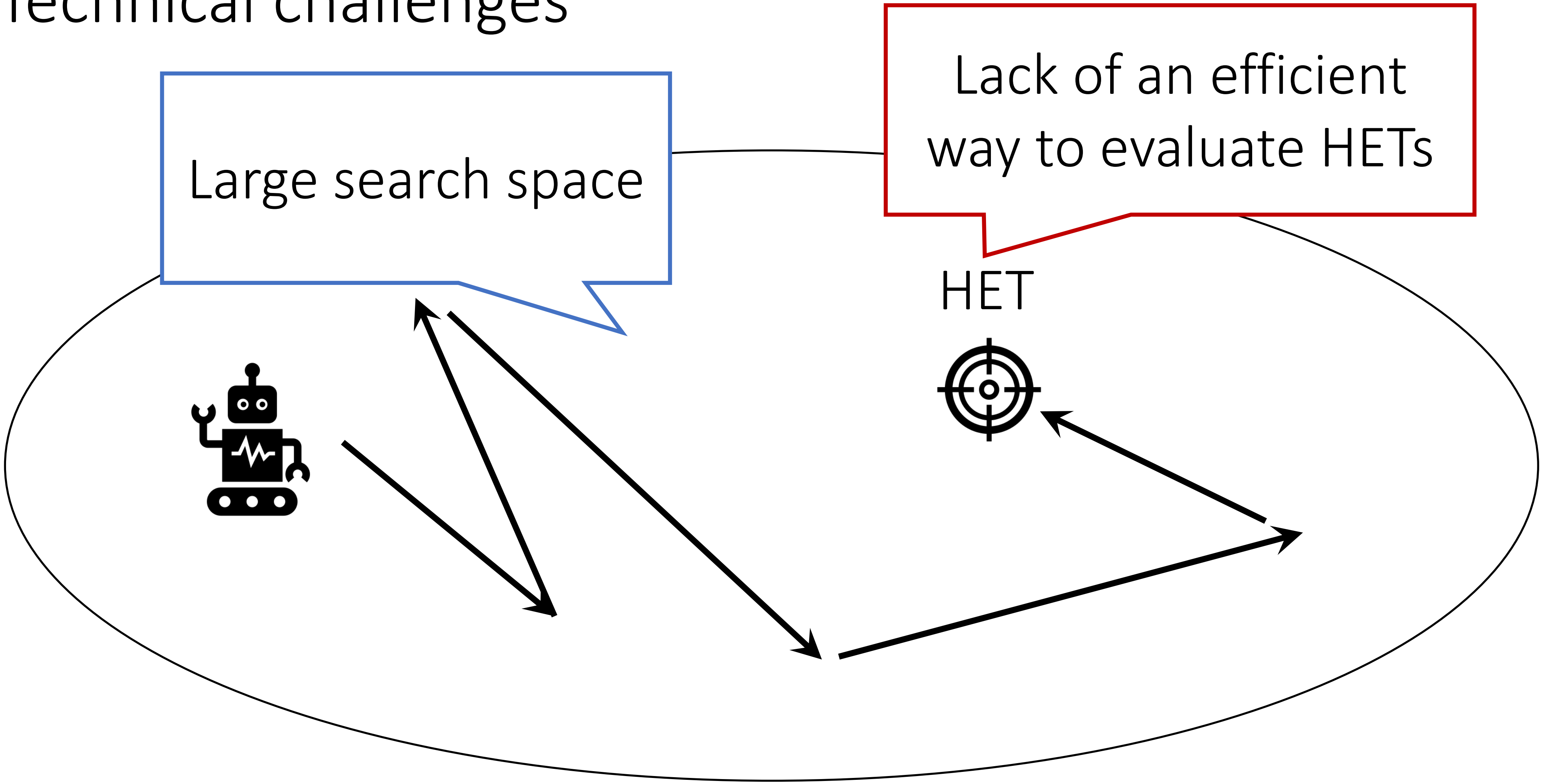
- Cardinal data: Metadata in a chunk are either sizes or pointers, but not other random values

- In-place metadata: Allocators place metadata near its chunk’s start or end for locality

Cardinal data and In-place metadata reduce search space in data writes



Technical challenges



Automatically synthesizing full exploits is inappropriate in evaluating HETs

- Difficult: e.g., In the DAPRA CGC competition, ***only one heap bug*** was successfully exploited by the-state-of-the-art systems

- Inefficient: Takes a few seconds, minutes, or even hours for one try

- Application-dependent: A HET, which is not useful in a certain application, may be useful in general

Our idea: Evaluating impacts of exploitations (i.e., detecting broken invariants that have security implications)

1. Allocated memory should not be overlapped with pre-allocated memory
   - Overlapping chunks: Can corrupt other chunk's data
   - Arbitrary chunks: Can corrupt global data

Easy to detect: Check this at every allocation

2. An allocator should not modify memory, which is not under its control (i.e., heap)
   - Arbitrary writes
   - Restricted writes

How about this?
(NOTE: should be efficient)

Shadow memory can detect arbitrary writes and restricted writes

- Maintain external consistency
- Check divergence

`container[i] = malloc(sz)`

$container_{shadow}$`[i] = malloc(sz)`

Allocation

`buf[i]=v`

buf_{shadow}`[i]=v`

Buffer write

`malloc(sz)`

Allocation

`free(p)`

Deallocation

`i]=v`

ap write

`buf[i]=v`

Buffer write

Divergence can only happen in the internal of allocators

CHECK: `equal(container, `$container_{shadow}$`)`
`equal(buf, `buf_{shadow}`)`

ArcHeap provides a minimized PoC code for further analysis

- Proof-of-Concept code: Converting actions into C code
  - Trivial, because they have one-to-one mapping

- Minimize the PoC code using delta-debugging
  - Idea: Eliminate an action, which is not necessary for triggering the impact of exploitations
  - Details can be found in our paper

Evaluation questions

1. How effective is ArcHeap in finding new HETs, compared to the existing tool, HeapHopper?

2. How general is ArcHeap's approach?

ArcHeap discovered five new HETs in ptmalloc2, which cannot be found by HeapHopper

- Unsorted bin into stack: Write-after-free → Arbitrary chunk
  - Requires fewer steps (5 steps vs 9 steps)
- House of unsorted einherjar: Off-by-one write → Arbitrary chunk
  - No require heap address leak

All HETS *cannot be discovered* by HeapHopper because of its scalability issue (i.e., symbolic execution + model checking)

- Fast bin into other bin: Write-after-free → Arbitrary chunk

ArcHeap is generic enough to test various allocators

- Tested 10 different allocators
  - Cannot find HETs in LLVM Scudo, FreeGuard, and Guarder, which are “secure allocators”

Allocators	P	I	Impacts of exploitation			AW
	✓	✓	O		WF	AF, OV, WF
	✓	✓	O			
	✓	✓	O		WF	AF, OV, WF
mu	✓	✓	OV	AF, OV, WF	AF, OV, WF	AF, OV, WF
jema 2.1						
tcmalloc 7		✓	OV, DF	OV, WF, DF	OV	OV
mimalloc-1.0.8		✓	OV, WF, DF		OV, WF	WF
mimalloc-secure-1.0.8		✓	DF			
DieHarder-5a0f8a52			DF			
mesh-a49b6134			DF, NO			

Even found HETs in “secure” allocators

Works for ptmalloc2-unrelated allocators

N: New techniques compared to the related work, HeapHopper [17]; only top three allocators matter. **NO**: No bug is required, i.e., incorrect implementations. **I**: In-place metadata, **P**: ptmalloc2-related allocators.

Case study1: Double free → Overlapping chunks in DieHarder and mimalloc-secure

```
// [PRE-CONDITION]
//    lsz : large size (> 64 KB)
//    xlsz: more large size (>= lsz + 4KB)
// [BUG] double free
// [POST-CONDITION]
//    p2 == malloc(lsz);
void* p0 = malloc(lsz);
free(p0);
void* p1 = malloc(xlsz);

// [BUG] free 'p0' again
free(p0);

void* p2 = malloc(lsz);
free(p1);

assert(p2 == malloc(lsz));
```

Double free large chunk → Overlapping chunk

Same thing happens in both DieHarder and mimalloc

Interestingly, these issues are irrelevant

Me: Is mimalloc related to DieHarder?

↓

Mimalloc developer: No!

`free(`p_{large}`)`

DieHarder `unmap(`p_{large}`)` No check!

mimalloc `check(`p_{large}`)` Wrong check!

Our PoC has been added in a mimalloc's regression test

```
+ static void double_free2() {
+   void* p[256];
+   uintptr_t buf[256];
+   // [INFO] Command buffer: 0x327b2000
+   // [INFO] Input size: 182
+   p[0] = malloc(712352);
+   p[1] = malloc(786432);
+   free(p[0]);
+   // [VULN] Double free
+   free(p[0]);
+   p[2] = malloc(786440);
+   p[3] = malloc(917504);
+   p[4] = malloc(786440);
+   // [BUG] Found overlap
+   // p[4]=0x433f1402000 (size=917504), p[1]=0x433f14c2000 (
+   fprintf(stderr, "p1: %p-%p, p2: %p-%p\n", p[4], (uint8_t*
786432);
+ }
```

Case study 2: Overflow → Arbitrary chunk in dlmalloc-2.8.6

- dlmalloc: ancestor of ptmalloc2 but has been diverged after its fork

```
void* p0 = malloc(sz);
void* p1 = malloc(xlsz);
void* p2 = malloc(lsz);
void* p3 = malloc(sz);


// [BUG] overflowing p3 to overwrite top chunk
struct malloc_chunk *tc = raw_to_chunk(p3 + chunk_size(sz));
tc->size = 0;


void* p4 = malloc(fsz);
void* p5 = malloc(dst - p4 - chunk_size(fsz) \
                  - offsetof(struct malloc_chunk, fd));
assert(dst == malloc(sz));
```

Looks complicated…

Its root cause is more complicated!

```
// Make top chunk available
void* p0 = malloc(sz);
// Set mr.mflags |= USE_NONCONTIGUOUS_BIT
void* p1 = malloc(xlsz);
// Current top size < lsz (4096) and no available bins, so dlmalloc calls sys_alloc
// Instead of using sbrk(), it inserts current top chunk into treebins
// and set mmapped area as a new top chunk because of the non-continous bit
void* p2 = malloc(lsz);
void* p3 = malloc(sz);
// [BUG] overflowing p3 to overwrite treebins
struct malloc_chunk *tc = raw_to_chunk(p3 + c
tc->size = 0;
// dlmalloc believes that treebins (i.e., top chunk) has enough size
// However, underflow happens because its size is actually zero
void* p4 = malloc(fsz);
// Similar to house-of-force, we can allocate an arbitrary chunk
void* p5 = malloc(dst - p4 - chunk_size(fsz) \
                - offsetof(struct malloc_chunk, fd));
assert(dst == malloc(sz));
```

Easy to miss by manual analysis
➔ Shows benefits of automated methods!

Discussion & Limitations

- Incompleteness: Unlike HeapHopper that is complete under its model
  - But HeapHopper’s model cannot be complete because of its scalability issue
- Overfitting: Our strategy might not work for certain allocators
  - In practice, our model is quite generic: found HETs in seven allocators out of ten except for secure allocators
- Scope: ArcHeap only finds HETs and does not generate end-to-end exploits for an application

Conclusion

- Automatic ways to discover HETs
  - Model-based search based on common designs of allocators
  - Shadow-memory-based detection
- Five new HETs in ptmalloc2 and several ones in other allocators
  - Including secure allocators, DieHarder and mimalloc secure
- Open source: https://github.com/sslab-gatech/ArcHeap

Thank you!