HardsHeap: A Universal and Extensible Framework for Evaluating Secure Allocators

Insu Yun*
KAIST
insuyun@kaist.ac.kr

Woosun Song
KAIST
wireless@kaist.ac.kr

Seunggi Min
KAIST
alex8757@kaist.ac.kr

Taesoo Kim
Georgia Institute of Technology
taesoo@gatech.edu

ABSTRACT

Secure allocators have been extensively studied to mitigate heap vulnerabilities. They employ safe designs and randomized mechanisms to stop or mitigate heap exploitation. Despite extensive research efforts, secure allocators can only be evaluated by with theoretical analysis or pre-defined data sets, which are insufficient to effectively reflect powerful adversaries in the real world.

In this paper, we present HardsHeap, an automatic tool for evaluating secure allocators. The key idea of HardsHeap is to use random testing (i.e., fuzzing) to evaluate secure allocators. To handle the diverse properties of secure allocators, HardsHeap supports an extensible framework, making it easy to write a validation logic for each property. Moreover, HardsHeap employs sampling-based testing, which enables us to evaluate a probabilistic mechanism prevalent in secure allocators. To eliminate redundancy in findings from HardsHeap, we devise a new technique called Statistical Significance Delta Debugging (SSDD), which extends the existing delta debugging for stochastically reproducible test cases.

We evaluated HardsHeap to 10 secure allocators. Consequently, we found 56 interesting test cases, including several unsecure yet underestimated behaviors for handling large objects in secure allocators. Moreover, we discovered 10 implementation bugs. One of the bugs is integer overflow in secure allocators, making them even more invulnerable than ordinary allocators. Our evaluation also shows that SSDD successfully reduces test cases by 37.2% on average without a loss of reproducibility.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Software and application security**.

KEYWORDS

Secure allocators, Fuzzing, Delta debugging, Automatic Exploit Generation

*Corresponding authors

ACM Reference Format:
Insu Yun, Woosun Song, Seunggi Min, and Taesoo Kim. 2021. HardsHeap: A Universal and Extensible Framework for Evaluating Secure Allocators. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460120.3484740

1 INTRODUCTION

Heap vulnerabilities remain prevalent security issues in applications written in memory unsafe languages such as browsers and operating systems. According to Google [16], heap vulnerabilities are directly related to 40% of 0-day (7 out of 18) exploits used in the wild in 2021. Moreover, heap out-of-bounds and use after free — are ranked as the top two vulnerability classes from 2016 to 2019 in Microsoft's system software [26]. The exploitation of these vulnerabilities often leads to serious security implications such as arbitrary code execution or privilege escalation.

Many secure allocators [1, 12, 21, 22, 25, 28, 33, 37, 38, 45] have been proposed to mitigate these heap vulnerabilities. Secure allocators often leverage safe designs (e.g., segregated metadata) along with randomized mechanisms (e.g., random allocations or random reuse) to reduce the reliability of heap exploitation with low performance overhead. Recently, researchers have also worked on secure allocators that support stable spatial memory safety (e.g., `MarkUs` [1] or `ffmalloc` [45]). Thanks to such research efforts, `scudo` [22], a secure allocator developed as a part of the LLVM project, becomes a default allocator from Android 11.

Despite ongoing efforts on secure allocators, their security evaluation is merely theoretical and inflexible. To the best of our knowledge, all studies regarding secure allocators [21, 28, 37, 38] only use theoretical analysis or static workload to evaluate each mechanism. For example, `Guarder` [38] only demonstrated its theoretical entropy for randomness, and `SlimGuard` used 128 B, 1 KB, 64 KB objects to compute the entropy of randomized allocation and reuse. Recently, Entroprise [39] suggested a universal method for evaluating the entropy of secure allocators regardless of their underlying implementations. However, the workload is still statically decided based on a given configuration or program. For example, in its analysis, Entroprise uses 10,000 16-byte objects to calculate the entropy of multiple allocators. Unfortunately, even though these allocators assume a strong adversary that can arbitrarily allocate and deallocate objects, these evaluations are insufficient to demonstrate their security against a powerful attacker. [28]. In another study, `ArcHeap` [49], randomly generates heap actions to discover security issues in allocators without relying on a static workload. However, it only focuses on classical heap exploitation techniques in a normal allocator (e.g., metadata overwrite), in which most secure allocators are invulnerable. Furthermore, ArcHeap cannot support randomized mechanisms, which are prevalent in secure allocators.

In this paper, we propose HardsHeap, an automatic tool for evaluating the security of secure allocators. Unlike static evaluations, HardsHeap randomly generates heap actions (e.g., allocation and deallocation) similar to fuzzing in software testing. To evaluate various security properties in secure allocators (see Table 4), HardsHeap provides an extensible framework to easily build a module for testing each property. Moreover, HardsHeap adopts sampling-based testing to evaluate randomized protections; it repeats multiple experiments and computes the probability of each security property violation. Such a random exploration can effectively explore a large search space [49, 50]; however, its findings often introduce redundancy that impedes further analysis to understand a root cause of security property violation. To overcome this, we devise a technique, called Statistical Significance Delta Debugging (SSDD), to reduce a test case that is only stochastically reproducible. It combines a greedy method with statistical significance to reduce a test case without a loss of reproducibility.

We evaluated HardsHeap with 10 secure allocators including DieHarder [28], Guarder [38], FreeGuard [37], and scudo [22]. To evaluate various security properties, we built seven modules, which only require hundreds of lines to implement thanks to HardsHeap's extensible framework. Using these modules, we identified 56 security violations in these allocators. These results show exceptions that a secure allocator fails to protect. For example, we found that Guarder leads to zero entropy in its random allocation for a large object (> 512 KB), unlike its claim for stable entropy. Moreover, these findings often lead us to discover implementation bugs; we found 10 bugs in seven allocators. More interestingly, a certain type of bugs makes these allocators less secure than an ordinary allocator. For example, we found that four secure allocators (Guarder, FreeGuard, isoalloc, and ffmalloc) suffer from integer overflow. They can return an object whose size is less than the request size if the request size is extremely large (e.g., −8). This vulnerability can render a well-written program insecure by imposing heap overflow. To foster future research, we open-source our prototype of HardsHeap in https://github.com/kaist-hacking/HardsHeap.

In summary, this paper makes the following contributions:

- We build HardsHeap, an extensible framework for evaluating various security properties of secure allocators. HardsHeap adopts sampling-based testing to support randomized algorithms in secure allocators.
- We devise a novel technique called Statistical Significance Delta Debugging (SSDD) to remove redundancy from a stochastically reproducible test case.
- We applied HardsHeap to 10 secure allocators and found 56 interesting test cases. These findings led us to discover several serious yet hidden behaviors that violate security properties. Moreover, they led us to discover 10 implementation bugs in secure allocators.

2 BACKGROUND

2.1 Heap vulnerabilities

A memory allocator supports a set of APIs for dynamic memory management (e.g., malloc and free). The allocator has chosen different design decisions and features for high runtime performance and low memory fragmentation, resulting in various implementations [1, 12, 21, 22, 25, 28, 33, 37, 38, 45]. This dynamic memory (i.e., heap) is required for a long-lived object, which is not suitable for temporary memory (e.g., stack). Such an object is unavoidable in non-trivial applications; therefore, heap and its allocators become essential software components.

Owing to the excessive use of dynamic objects, an application often suffers from various types of heap vulnerabilities. Classically, heap vulnerabilities can be categorized into four types:

- **Overflow:** Writing other objects near the object boundary.
- **Use-after-free:** Using an object that is already freed.
- **Invalid free:** Freeing a non-heap object.
- **Double free:** Freeing an object that is already freed.

Each heap vulnerability provides a unique capability for exploitation. In particular, **overflow** allows the modification data in the adjacent chunks, **use-after-free** allows the control of a freed object if it is successfully reclaimable, **invalid free** allows the modification or allocation of a non-heap object, and **double free** allows the allocation of the same object twice, which can break internal invariants of heap allocators. By exploiting these vulnerabilities, an attacker often causes a more serious security implication, such as arbitrary code execution [26].

2.2 Secure allocators

Many secure allocators have been proposed [1, 12, 21, 22, 25, 28, 33, 37, 38, 45] to prevent these heap vulnerabilities. These allocators employ specific designs to support several security features while incurring low performance overhead. To understand the security features of existing secure allocators, we first manually investigated their security features, as shown in Table 1. Among the allocators, ffmalloc (FF) and MarkUs (MA) are unique; they are specially designed to prevent use-after-free, while others are developed to mount more generic defenses against all heap vulnerabilities. Evidently, these special allocators — ffmalloc and MarkUs— are more secure than others against use-after-free; they support a stable level of security for use-after-free, unlike the randomized allocation in other secure allocators.

Table 1 shows the trends in the design of secure allocators. First, due to serious security threats from metadata overwrites [36], all allocators employ mechanisms to protect metadata, which are either segregated metadata or metadata encoding. Second, random allocation is the most widely used mechanism for preventing heap overflow. It is sometimes equipped with additional features such as a guard page, overvisioning, and check-on-free (i.e., heap canary). Third, most of allocators adopt random reuse to prevent use-after-free attacks. Moreover, ffmalloc and MarkUs have no other protections for use-after-free because their own mechanisms are believed to be self-sufficient; ffmalloc employs one-time allocation and MarkUs marks dangled objects to exclude them from reclamation. Finally, all secure allocators identify invalid free and double

Abbr.	Allocators
DI	`DieHarder` [28]
FF	`ffmalloc` [45]
FR	`FreeGuard` [37]
GU	`Guarder` [38]
HA	`hardened_malloc` [12]
IS	`isoalloc` [33]
MA	`MarkUs` [1]
MI	`mimalloc-secure` [25]
SC	`Scudo` [22]
SL	`SlimGuard` [21]

(a) Secure allocators and their abbreviations for brevity

Security features	Security Properties	DI	FR	GU	HA	IS	MI	SC	SL	FF	MA
Segregated metadata	Prevent metadata-based attacks	✓	✓	✓	✓	✓			✓	✓	
Metadata encoding	Prevent metadata-based attacks						✓	✓			✓
Random allocations	Reduce reliability of overflow attacks	✓	✓	✓	✓		✓	✓	✓		
Guard pages	Prevent cross-object overflows	✓	✓	✓	✓	✓	✓				
Overvisioning	Reduce the impacts of overflow attacks	✓		✓					✓		
Check-on-free	Timely detect overflow attacks	✓	✓	✓	✓	✓	✓	✓	✓		
Random reuse	Reduce reliability of use-after-free attacks	✓	✓	✓	✓	✓	✓	✓			
Destroy-on-free	Mitigate use-after-free attacks	✓	?	?							
Use-after-free prevention	Prevent reclamation of a dangling object									✓	✓
Detect invalid frees	Prevent invalid free attacks	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Detect double frees	Prevent double free attacks	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓: Supported, ?: Optional

(b) Security features that existing secure allocators support.

Table 1: Existing secure allocators and their security features. We manually investigate them according to their documentation and papers. Since `mimalloc` is a not secure allocator by default, we only consider its secure mode in this paper.

free. If an allocator can manage an object status (e.g., an object is freed or not) in a safe place, it is relatively inconsequential to avoid these vulnerabilities. By using the status, the allocator can detect a suspicious object (e.g., an already freed one) at `free`.

3 DESIGN GOALS

Similar to previous works [21, 38], we analyzed the security properties of secure allocators in Table 1; however, this analysis is insufficient to evaluate their security. In this section, we first discuss limitations of the existing manual analysis. Then, we show the corresponding design goals of HardsHeap to overcome these issues in evaluating secure allocators systematically and thoroughly.

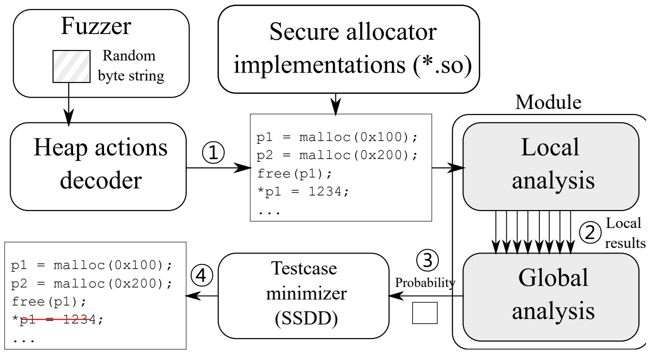
Implementation-agnostic security testing. To evaluate the security properties of secure allocators, we should devise a way to evaluate them regardless of their different implementations. In particular, secure allocators have made unique design decisions to satisfy their own security and performance requirements. As a result, even though many secure allocators have security features in common (see Table 1), their underlying implementations are extremely diverse. For example, allocators individually define a large object that is allocated by `mmap` instead of `sbrk`; `SlimGuard` uses 128 KB as its threshold for large objects where `Guarder` uses 512 KB. Therefore, it is impossible to evaluate various secure allocators with statically and manually crafted test sets. To overcome this, HardsHeap employs random testing (i.e., fuzzing) to arbitrarily call APIs for dynamic memory management and externally observes security properties of secure allocators in order to evaluate them (see §5.2).

Quantitative evaluation. We also need to quantitatively evaluate security properties. Most of the existing security analyses for secure allocators are binary; they only mark whether a certain allocator supports a specific security feature (as shown in Table 1) without further analyzing its quality. However, it is insufficient to evaluate the security of secure allocators. For example, Silvestro et al. [38] show that both `FreeGuard` and `DieHarder` support random allocation. However, their mechanisms are relatively weak because `FreeGuard`'s entropy is extremely low (i.e., 2 bits) while `DieHarder`'s is unstable and varies across object sizes (i.e., larger objects are less randomly allocated). To address this issue, we require a quantitative analysis. To this end, HardsHeap performs sampling-based testing; it reports a security violation with its probability from repeated experiments (see §5.3).

Explainability. Even though random exploration is effective in discovering an erroneous case, its findings naturally bring redundancy, which limits further analysis for understanding the issue. To eliminate this redundancy, delta debugging [49, 51] is widely used; however, it assumes the reproducibility of failures. To apply delta debugging in a stochastic environment, which is prevalent in secure allocators, we devise a new technique called Statistical Significant Delta Debugging (SSDD). This technique combines a greedy method with statistical significance (see §5.4) to further reduce the test case without a loss of reproducibility. According to our evaluation in §8.2, SSDD successfully reduce test cases by 37.2% while preserving their original probability.

4 THREAT MODEL

In this section, we present our threat model for heap vulnerabilities, which is powerful yet realistic to extensively evaluate the security of secure allocators. First, an attacker can allocate objects of an arbitrary size and can deallocate objects in an arbitrary order. This capability highly depends on applications; however, complicated software such as web browsers is often equipped with it. For example, in JavaScript, attackers can allocate a heap object of an arbitrary size using `ArrayBuffer`, and they can also deallocate it by nullifying its reference. Second, the attacker can write arbitrary values to a predefined memory region, which allows them to craft arbitrary fake chunks for abusing allocators. Third, the attacker can repeatedly trigger one of heap vulnerabilities in §2.1 (i.e., overflow, use-after-free, invalid free, or double free). However, it is limited to a single bug type. This simulates a realistic situation for exploitation; The attacker attempts to exploit an application by reusing a single vulnerability multiple times. We also assume that the attacker has no extremely powerful primitive such as arbitrary writes. This demotivates the attacker to launch heap exploitation, enabling easier attacks. We note that this model assumes the worst-case scenario for heap exploitation, which is consistent with other studies on secure allocators [28, 37, 38, 49].



```
p1 = malloc(0x100);
p2 = malloc(0x200);
free(p1);
*p1 = 1234;
...
```

```
p1 = malloc(0x100);
p2 = malloc(0x200);
free(p1);
*~~p1 = 1234~~;
...
```

Figure 1: HardsHeap's overview. Note that gray boxes are dedicated components for each security mechanism, and others are commonly used regardless of its evaluating mechanism.

5 DESIGN

In this section, we illustrate the design of HardsHeap to evaluate various security properties in secure allocators.

5.1 Workflow

Figure 1 shows the high-level design and workflow of HardsHeap.

① HardsHeap relies on a generic binary-based fuzzer, (i.e., AFL [50]) to generate random byte stings. Subsequently, HardsHeap encodes them into heap actions using its decoder [30, 49]. The heap actions consist of four types: allocation, deallocation, memory writes, and bug invocation. We further discuss how HardsHeap generates them in §5.2.

② HardsHeap runs the decoded heap actions with a secure allocator for analysis (§5.3). HardsHeap uses a standard library hooking technique (i.e., `LD_PRELOAD`) to use a secure allocator implementation for its execution. To handle the diversity of security properties, we implement HardsHeap's sampling-based testing for each security property in a special component, referred to as *a module*. This module performs two types of analyses: local and global. The local analysis installs several hooking functions to heap actions to control them and gather information. These hooking functions usually report the occurrence of a security property violation (e.g., adjacent chunks) if such a violation is locally checkable. Otherwise, they gather data and transmit them to the next global analysis.

③ After locally running heap actions multiple times (i.e., sampling), the global analysis calculates the probability of a security property violation. HardsHeap calculates the probability in a straightforward manner — the number of occurrences of security violations divided by the total number of trials.

④ If the probability is larger than a predefined threshold (0.25 in our prototype), HardsHeap generates Proof-of-concept (PoC) code. Because of its random nature, HardsHeap's test case often contains non-essential actions; therefore, HardsHeap leverages a technique called Statistical Significance Delta Debugging (SSDD) (see §5.4) to reduce it. Finally, it produces a PoC code written in the C language that shows security violations of secure allocators.

5.2 Heap Action Decoding

HardsHeap decodes a random byte string into the corresponding heap actions. It generates four types of heap actions: allocation, deallocation, memory writes, and bug invocation. HardsHeap's decoding logic is directly borrowed from `ArcHeap`'s; therefore, we briefly explain it, providing further detail in [49]. It is worth noting that `ArcHeap` also randomly generates heap actions like HardsHeap; however, it is limited to evaluating a normal allocator because it cannot support various yet randomized security properties of secure allocators.

The following explains each type of heap actions generated by HardsHeap to evaluate secure allocators.

Allocation. HardsHeap allocates memory with randomized sizes using `malloc`. HardsHeap chooses its allocation size carefully to reduce its search space. It has four types of strategies to select allocation sizes: ① random size, ② other chunk's size, ③ special values (e.g., -1 or 0), and ④ offsets between pointers (e.g., heap pointers or its global buffer to craft fake chunks). The last strategy is required to discover integer overflow in handling allocations; others are fairly intuitive strategies for evaluating allocators. For further analysis, HardsHeap also maintains both the request size and actual size of an allocated object; HardsHeap achieves the request size by tracking a `malloc`'s argument and retrieves the actual size using `malloc_usable_size` API.

Deallocation. HardsHeap also randomly picks an object and deallocates it. HardsHeap avoids buggy situations (e.g., double free), which are not controlled by HardsHeap, by tracking an object's allocation status. In particular, HardsHeap maintains a bitmap whose bit represents whether a corresponding object is deallocated or not. Then, HardsHeap simply ignores the execution of a deallocation action if it tries to deallocate already freed objects.

Memory writes. HardsHeap also writes memory with randomly generated values. It has two types of memory to write: heap chunks and a global buffer. HardsHeap uses heap chunks to test whether an allocator is resilient to modify its internal data in heap objects. Moreover, HardsHeap uses a global buffer to check whether an allocator is able to distinguish its heap objects with a fake memory region in buggy situations (e.g., invalid free or use-after-free). Similar to `ArcHeap`, HardsHeap only generates random values that are related to chunk sizes or pointers instead of purely random values to reduce its search space.

Bug invocation. HardsHeap also generates buggy actions to evaluate secure allocators against heap vulnerabilities. HardsHeap simulates four types of vulnerabilities that are common and widely adopted: overflow, write-after-free, invalid free, and double free. Overflow and double free are self-explanatory; write-after-free allows modifying freed objects, and invalid free allows freeing a non-heap region, which is a global buffer in HardsHeap.

5.3 Sampling-based Testing

To evaluate various security properties in a secure allocator, HardsHeap performs sampling-based testing. Because existing security properties are too diverse to assess them using a single method, we rely on a manually crafted *module* for their evaluation. However, thanks to HardsHeap, we only require hundreds of lines of code for each property (see Table 2). It is worth noting that we can reuse heap action generation and test case reduction for any module; we only need to implement its core logic.

With given heap actions, HardsHeap's module performs two types of analyses: local and global. In the local analysis, the module executes heap actions with a secure heap allocator to test and records its behavior. For that, HardsHeap allows locating hooking functions before and after each action (e.g., allocation and deallocation). Moreover, it also allows another hooking functions that will be called at the module's initialization and finalization. These hooking functions can also access the current status of heap objects, which HardsHeap maintains. For example, in a module that checks adjacent chunks, we can place a hooking function after allocation to check whether a returned object is adjacent to current objects. We detail the modules in §6. HardsHeap also provides a method for communicating with its global analysis using shared memory. Therefore, we can transmit the local analysis results to the global analysis for further analysis.

After obtaining local analysis results, HardsHeap's module calculates the probability of security violation. In most cases, this simply counts the number of occurrences of security violations. However, in certain cases, more complicated analysis is required to calculate the probability. For example, if we want to evaluate whether an allocator is resilient to heap spraying, we first need to compute a recurrent address in multiple executions to count its occurrences. This recurrent address cannot be calculated in a single instance; therefore, the global analysis performs this computation by collecting the results of the local analysis (i.e., sampling).

5.4 Statistical Significance Delta Debugging

If the probability is greater than a pre-defined threshold, which is set to 0.25 in our prototype, HardsHeap stores a given set of heap actions and reduces it using Statistical Significance Delta Debugging (SSDD). To reduce test cases, Delta debugging [51] is one of the most widely used techniques; it re-runs an application with a more reduced case to check whether the same failure can occur. By repeating this procedure, delta debugging can reduce the test case for the same failure. In HardsHeap, the failure corresponds to a security violation in secure allocators, and the test case is the set of heap actions. Unfortunately, classical delta debugging is not directly applicable because it assumes that the failure can be reliably reproducible; however, in evaluating secure allocators, security violations only appear stochastically because of their random mechanisms (e.g., adjacent chunks happen in 30%).

One straw-man design for supporting stochastic events is to use a greedy algorithm, which is shown in Algorithm 1. In particular, HardsHeap can compare the probability of a reduced test case, which eliminates one action, with the original probability; HardsHeap can measure both probabilities by sampling multiple events (e.g., 100 in our prototype). If a new probability is greater than or equal to the old one, we believe that it is safe to eliminate the corresponding action. (Line 7). However, because of the fickle nature of probabilities, it is likely to miss opportunities for reducing a test case. Even if a new probability is insignificantly less than the old one, a test case cannot be reduced using this greedy method.

To address this issue, we devise Statistical Significance Delta Debugging (SSDD) by extending the previously mentioned greedy method. Unlike the greedy method, SSDD considers statistical significance not only differences between average. To measure statistical significance, we use Student's t-test [40]; it represents statistical significance if its p-value is less than the threshold, which is usually 0.05. Even if a new probability is less than the old one, SSDD can eliminate a corresponding action if the probability difference is statistically insignificant. It is noteworthy that HardsHeap's modeling satisfies constraints for using Student's t-test; we can use t-test only if an objective distribution follows the normal distribution. In particular, in evaluating secure allocators, each trial for violating a security property (e.g., making an adjacent chunk) is a Bernoulli process, such as coin tossing. Thus, our probability from sampling follows a binomial distribution, which could be approximated as a normal distribution. Thus, we are safe to use Student's t-test to compute statistical significance in SSDD.

Algorithm 1: Greedy & SSDD

```
Input  : actions – actions to minimize
Input  : greedy – use greedy if true, otherwise use SSDD
1  minActions := actions;
2  oldProbs := Sampling(actions);
3  for action ∈ actions do
4      newProbs := Sampling(minActions - action);
5      avgDiff = Avg(newProbs) - Avg(oldProbs);
6      if greedy then
7          if $avgDiff \geq 0$ then
8              minActions = minActions - action;
9          end
10     else
11         pValue := StudentTTest(newProbs, oldProbs);
12         if $avgDiff \geq 0$ or $pValue \geq 0.05$ then
13             minActions = minActions - action;
14         end
15     end
16 end
Output : minActions – minimized actions
```

We optimize this test case reduction procedure for a reliably reproducible case by returning to classical, deterministic delta debugging. Even though a certain security feature is randomized in theory, its violation can be reliably reproducible. For instance, `Guarder`'s allocation is designed to be random; however, it becomes deterministic if we allocate an extremely large object (> 512 KB). In such a situation, we can return to classical delta debugging without SSDD. This is more efficient than SSDD because it requires no repeated experiments to compute the probability in delta debugging.

After reducing a test case, HardsHeap returns a C program as its Proof-of-Concept (PoC) that violates a security property with high probability. It is trivial to create a C program with heap actions because each heap action has one-to-one mapping with a C statement. For example, allocation can be converted into `malloc`, and deallocation can be converted into `free`. In each module, we also add an `assert` statement to check whether a security violation (e.g., adjacent chunks) occurs.

6 MODULES

Modules	LoC	Mode	Security features	Description
`Adjacent`	135		Random allocation,	Check if chunks can be adjacent
		Small	+ Guard page	Check if adjacent chunks happen for small objects (< 1K bytes)
		Cross		Check if adjacent chunks have different object sizes
`Reclaim`	119		Random reuse	Check if a dangling chunk is reclaimable
		Small	Use-after-free prevention	Check if chunk reclaimiation happens for small objects (< 1K bytes)
`CheckOnFree`	89		Check-on-free	Check if an allocator terminates when deallocates a corrupted chunk
`Uninitialized`	78		Segregated metadata	Check if an allocator leaves metadata in uninitialized memory
`Spray`	64		(Resilient to heap spray attack)	Check if subsequent allocations have no recurrent address
`SizeCheck`	61		(Resilient to integer overflow)	Check if an allocated size is greater or equal to requested one
`ArcHeap`	574		(Resilient to heap vulnerabilities)	[49]

Table 2: Modules that we implemented using HardsHeap. Modules can have specific modes to discover more interesting cases during their analyses. In particular, the Small mode focuses on small objects whose sizes are less than 1 KB, and the Cross mode focuses on adjacent chunks whose sizes are different from each other.

```
1  capabilities = [ ALLOC, DEALLOC ]
2
3  def post_allocate(shm, hmgr, index):
4    '''
5    :param hmgr: A collection of heap chunks
6    :param index: An index for an allocated object
7    :param shm: Shared memory for storing local analysis results
8    '''
9    obj = hmgr[index]
10   for i, other_obj in enumerate(hmgr):
11     if i == index:
12       continue
13
14     if is_adjacent(obj, other_obj):
15       shm.write(index)
16       shm.write(i)
```

(a) A local analysis of the `Adjacent` module

```
1  collector = defaultint(int)
2
3  def analyze_single(shm):
4    while not shm.empty()
5      i = shm.read()
6      j = shm.read()
7      collector[(i,j)] += 1
8
9  def calculate_prob(n_runs):
10   '''
11   :param n_runs: The total number of runs
12   '''
13   return max(collector.values()) / n_runs
```

(b) A global analysis of the `Adjacent` module

Figure 2: Pseudocode for the HardsHeap's `Adjacent` module. For a brief explanation, we used Python to represent pseudocode; however, our actual prototype is written in C to avoid accidental uses of heap allocators.

Using HardsHeap, we implemented seven modules that evaluate various security properties in secure allocators. As shown in Table 2, their *security properties* are highly related to *security features* in secure allocators; however, they are externally feasible to evaluate unlike these features. Thanks to HardsHeap's extensible framework, it is easy to build a module; we only need hundreds of lines for building modules. The most complex module is `ArcHeap` [49], which discovers heap exploitation techniques (e.g., overlapping chunks) in the presence of heap vulnerabilities. It shows the extensibility of HardsHeap by supporting a complex module such as `ArcHeap` in our framework.

In the following, we explain each module and show how to implement it using the `Adjacent` module as an example. Our repository also contains our prototype implementations for other modules.

Adjacent. The `Adjacent` module checks whether an attacker can achieve adjacent chunks reliably. We present its pseudocode in Figure 2. Many secure allocators make it challenging to achieve adjacent chunks to mitigate overflow in corrupting sensitive data (e.g., a function pointer) in the next chunk. To evaluate this security property, the module's local analysis installs a hooking function at allocation (Figure 2a). This function iterates over all chunk objects and checks whether the other chunks are adjacent to the just allocated one. If it discovers such chunks, it records their indices to the shared memory for the global analysis (Lines 15–16). At the end of the local analysis, the global analysis records the number of adjacent chunks (Lines 3–7). Our analysis is aware of chunk indices; they are required to identify a victim and a vulnerable object in exploitation. As noted in §5.3, HardsHeap allows the module to specify its actions to generate. For efficient exploration, the `Adjacent` module only generates allocation and deallocation (Line 1) After sampling n experiments ($n = 100$ in our prototype), the module computes the probability of adjacent chunks; it simply divides the number of executions of adjacent chunks by the total number of executions (Line 13). If this probability is beyond our threshold (0.25), HardsHeap stores it and later generates the PoC code after reduction.

Reclaim. The `Reclaim` module validates whether a dangling object is reliably reclaimable in secure allocators. To use memory efficiently, secure allocators should reclaim unused memory; however, an attacker can abuse this behavior to exploit use-after-free bugs. In particular, an attacker can reclaim a dangling object to modify its data and use it to cause undefined behaviors. To stop this, secure allocators often prohibit reclaiming of a dangling object (e.g., `MarkUs` and `ffmalloc`) or make reclamation unreliable. To test whether this mechanism works as intended, our module checks whether a newly allocated object can occupy the already freed object while randomly allocating and deallocating objects. Note that all freed objects in HardsHeap are naturally dangled; HardsHeap holds these objects in its data structure for analysis regardless of whether the objects are freed or not. Therefore, if a secure allocator can distinguish a dangling object from a normal one (e.g., MarkUs), it should consider an object from HardsHeap as dangled.

CheckOnFree. The CheckOnFree module validates whether secure allocators can detect overflow at free. Even though this mechanism is limited in blocking overflow due to its specific spot to check (i.e., free), it is quite widely used thanks to its straightforward implementation and low performance overhead. Similar to the stack canary, we can support this mechanism by placing a random value (i.e., canary) between heap objects and checking their corruptions by inspecting the canary at free. This also helps to debug by early detecting overflows; however, HardsHeap only focuses on its usefulness in mitigating exploitation. For that, HardsHeap randomly allocates objects, deallocates them, and triggers an overflow. At allocation, it zeros out an object. Then, before deallocation, it checks whether the object is still filled with zeros. Otherwise, it implies that this object is corrupted due to overflow, which is only permitted to modify heap object's contents. Subsequently, CheckOnFree verifies whether a program is still running even after free; this indicates that an allocator fails to detect this corruption, which shows failures in the check-on-free mechanism.

Uninitialized. The Uninitialized module checks whether an attacker can leak metadata from uninitialized memory. Such a technique is widely used in exploitation because the metadata often contain secret values for security (e.g., canary). To evaluate this, this module randomly allocates objects and deallocates memory while checking whether the allocated memory is zero-initialized. It relies on the internal behavior of Linux (or other operating systems) that newly allocated memory from the kernel (i.e., pages) is always zero-initialized. Therefore, if a memory contains a non-zero value, we conclude that it is part of the metadata from an allocator. Unfortunately, HardsHeap cannot determine the severity of this leakage. Therefore, this finding could have no security implication unlike the other module's one. We discuss such an invulnerable case in §10.

Spray. The Spray module checks whether an attacker can guess any recurrent address from the secure allocators. In this regard, the module randomly allocates and deallocates objects and checks whether a recurrent address exists among multiple executions. Unlike other modules, which can validate a security violation locally, it requires global analysis because the recurrent address is only calculable with multiple instances. To this end, in its local analysis, this module records only object information — its start address and size. Then, in its global analysis, it computes the recurrent address and its corresponding probability.

SizeCheck. The SizeCheck module checks whether an allocator holds its intrinsic invariant; the actual size of a heap object should be greater than or equal to the request size. This invariant is often broken if an allocator is vulnerable to an integer overflow. It is worth noting that HardsHeap maintains both the request size and the actual size of an object in its data structure for analysis. To check this invariant, this module hooks the allocation and checks whether the invariant holds using HardsHeap's data structure for sizes.

ArcHeap. We also port ArcHeap as a module of HardsHeap. In short, ArcHeap attempts to detect the violation of other intrinsic invariants of heap allocators; heap allocators should not 1) modify non-heap regions and 2) return a chunk that overlaps with other memory regions. To verify this, the ArcHeap module randomly generates all heap actions (i.e., allocation, deallocation, memory writes, and bug invocation) while validating whether the invariants hold using shadow memory and HardsHeap's object information [49].

Component	LoC	Language
Core Library	1,697	C/C++
Minimizer	269	Python
Modules	1,306	C/C++
AFL modification	191	C
Total	3,463	

Table 3: HardsHeap's components and their Lines of Code (LoC). We further describe each module's LoC in Table 2.

7 IMPLEMENTATION

We implemented the HardsHeap's prototype in 3,463 lines of code. Table 3 shows HardsHeap's components with corresponding lines of code; each module's complexity can be estimated through its lines of code in Table 2. As specified in §5.2, HardsHeap's core library is built on ArcHeap [49], which supports heap action generation. We used American Fuzzy Lop (AFL) [50] as our underlying binary fuzzer. HardsHeap implemented the SSDD minimizer in Python. We utilized scipy [43] for statistical analysis, such as Student's t-test. It is worth noting that it is important to limit heap usage in modules except for intended behaviors for reproducibility. More specifically, implicit heap usage may differentiate the behavior of allocators in an analysis phase and its PoC code, thereby making PoC difficult to reproduce. Thus, we used C to write our module to eliminate the accidental use of dynamic memory. In a global analysis, we support C++ and its Standard Template Library (STL) for convenient development. Moreover, instead of using a shared library, we built each module as a dedicated binary because we found that a shared library leverages a large volume of dynamic allocations that affect an allocator's behaviors. In AFL, we increased its default timeout from 1 second to 10 seconds because one module executes multiple local analyses internally (i.e., 100 by default) for sampling; therefore, we found that HardsHeap occasionally suffers from numerous timeouts and fails to observe meaningful behaviors.

8 EVALUATION

To evaluate HardsHeap, this section attempts to answer the following questions.

- How effective is HardsHeap in evaluating the security properties of secure allocators? (§8.1)
- How effective HardsHeap's SSDD in reducing test cases? (§8.2)
- How many PoCs of HardsHeap can be reproduced? (§8.3)

Evaluation Setup. We performed every experiment on Intel Xeon Gold 6248R with 256 GB RAM running on Ubuntu 20.04. We used 16 random strings for our seed files, and according to our experience, their values are unimportant in our evaluation because HardsHeap

Abbr.	Allocators	Version	Patch (LoC)
DI	`DieHarder` [28]	`6cf204ec`	
FF	`ffmalloc` [45]	`9e1e5825`	
FR	`FreeGuard` [37]	`bfdf6d9a`	+17
GU	`Guarder` [38]	`9e85978a`	+17
HA	`hardened_malloc` [12]	`v5`	
IS	`isoalloc` [33]	`a683f427`	
MA	`MarkUs` [1]	`4c75ffd5`	
MI	`mimalloc-secure` [25]	`v1.7.0`	
SC	`Scudo(non-standalone)` [22]	`v11.0.0`	
SL	`SlimGuard` [21]	`237d842a`	+36, -1

Table 4: Secure allocators that are used for evaluating HardsHeap. To specify versions for allocators, we use `git` commit hash or their own version strings, which start with `v`. As we can see from the Patch column, we insert(+) or delete(-) codes in some allocators to support `malloc_usable_size()` API or to optimize initialization for evaluation.

quickly converges because of its smaller search space compared to that of classical software testing.

Secure allocators. To evaluate HardsHeap, we used 10 secure allocators as shown in Table 4. To comprehensive evaluate HardsHeap, we attempted to include as many secure allocators as possible; we collected them from both academic projects [1, 21, 28, 37, 38, 45] and industrial projects [12, 22, 25, 33]. Because our tool, HardsHeap, can only analyze a standalone allocator without any program dependencies, we excluded secure allocators that are customized for special applications, such as `PartitionAlloc` [11] for the Chromium browser. Moreover, we also exclude memory safety solutions that require modification of programs (i.e., instrumentation) [20, 35, 42], which are also outside the scope of this project.

We patched some allocators (see Table 4) before the evaluation for several reasons. First, we patched allocators with no support for `malloc_usable_size` API, which HardsHeap used to obtain an object's size. This API is fairly straightforward to implement because all allocators already have internal routines to determine an object size from a heap object for `realloc`. In particular, allocators use an object size to optimize `realloc` by avoiding additional allocation when an old object's size is already greater than a newly requested size. Therefore, we simply modified allocators to call the internal routine in the `malloc_usable_size` API. Second, we patched `SlimGuard` to improve its initialization. The original `SlimGuard`'s initialization is extremely slow because it writes NULL to its data structures, which invokes many page copies because of copy-on-write mechanisms in Linux. To resolve this, we modify `SlimGuard` to write NULL to the data structure's field only if its old value was not NULL. By doing so, we could avoid redundant page copies, thereby improving `SlimGuard`'s initialization by an order of magnitude. Finally, we also patched a weak randomness issue in `Guarder` and `FreeGuard`. We found that this rendered our analysis inaccurate due to its unusual random behavior. These allocators are globally random but not locally within one second because they use timestamps as their random source. This seriously violates their security guarantee. We further discuss this issue in §9.2.

```c
// 0x80000 = 512KB
// 0x80000 + 0x1000(+1 page) - 0x10(metadata)
void* p0 = malloc(0x81000 - 0x10);
void* p1 = malloc(0x81000 - 0x10);
assert(p1 + 0x81000 == p0);
```

Figure 3: A way to make adjacent chunks reliably in `Guarder`.

8.1 Evaluating Security Properties

To evaluate security properties in secure allocators, we applied HardsHeap's modules (Table 2) to 10 secure allocators and one baseline allocator (Table 4) for 24 hours. For the baseline allocator, we used the default allocator in Ubuntu 20.04 that is `ptmalloc2` in glibc 2.31. Table 5 shows the maximum probability of each module's finding. We also show whether all findings of HardsHeap can be reproduced deterministically (i.e., in 100%). This allows us to understand the impact of each violation and whether the violation is probabilistic, deterministic, or both.

In total, HardsHeap identified 56 interesting test cases in 10 secure allocators. Most of our findings demonstrate security issues in secure allocators. However, since HardsHeap only observes the external behaviors of allocators without reasoning, it is completely possible that HardsHeap's results have no security implications. We discuss this limitation in §10, which is related to the `Uninitialized` module.

For the baseline allocator (`ptmalloc2`), HardsHeap found deterministic test cases in all modules, except for `SizeCheck` and `Spray`. This is reasonable because 1) `ptmalloc2` has no randomized mechanisms, resulting in deterministic behaviors, and 2) unexpected behaviors from `SizeCheck` and `Spray` often relate to implementation bugs, which `ptmalloc2` is unlikely to have. In particular, unexpected behaviors from the `SizeCheck` module imply an integer overflow in an allocator, and those from the `Spray` module imply the weak entropy of ASLR in heap.

In the remainder of this section, we discuss our findings in secure allocators and their underlying reasons for all modules.

Adjacent. HardsHeap found that all secure allocators cannot stop adjacent chunks completely [38]; it still allows making adjacent chunks reliably in several extraordinary situations. This happens for two reasons. First, we found that most secure allocators fail to provide sufficient entropy in large objects. It is a well-known behavior of `DieHarder` [38]; `DieHarder` does not provide protection for large object allocations, as these are just diverted to `mmap` and `munmap`. However, we discovered that even `Guarder` suffers from this issue despite its claim of stable entropy (see Figure 3). In particular, `Guarder` uses a raw `mmap` without specifying its start address if an object size is larger than 512 KB. Unfortunately, `mmap` returns the adjacent addresses in subsequent calls in Linux. Note that the `mmap`'s behavior is dependent on the underlying operating system. Linux, which we used for our evaluation, returns the adjacent addresses in the subsequent `mmap` calls, while OpenBSD returns random addresses to ensure enough entropy among mappings. However, as mentioned in `DieHarder` [28], we believe that secure allocators need to work securely regardless of their underlying operating systems. Therefore, we believe that the secure allocator is responsible for this issue. Recently, Entroprise [39] evaluated `Guarder`'s allocation entropy; however, it failed to discover this issue due to its fixed workload (i.e., 32-byte objects) even though Entroprise could have

		ptmalloc2		DieHarder		FreeGuard		Guarder		MarkUs		SlimGuard		ffmalloc		hardened_malloc		isoalloc		mimalloc		scudo	
		P	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**	**P**	**D**
`Adjacent`		100	✓	100	×	100	×	100	×	100	✓	100	×	100	✓	100	×	100	✓	100	×	100	×
`Adjacent`	Cross	100	✓	100	×	100	✓	100	×	100	✓	100	✓	100	×								
`Adjacent`	Small	100	✓							100	✓	100	✓	100	✓	40	×	100	✓	44	×		
`Reclaim`		100	✓	100	✓	100	✓	100	✓	100	×	100	✓			100	×	100	×	100	✓	100	×
`Reclaim`	Small	100	✓									100	✓							54	×	100	✓
`CheckOnFree`		100	✓	100	×	100	×	100	✓	100	✓	100	×			52	×	100	✓	73	×		
`Uninitialized`		100	✓									100	✓					100	✓	100	×		
`SizeCheck`						100	×	100	×			100	✓	100	✓			100	✓				
`Spray`						42	×							61	×					100	×		
`ArcHeap`		100	✓	100	×							100	×							98	×		

P: The maximum probability (%), **D**: A mark that will be set if every test case is deterministic

Table 5: Summary of HardsHeap's security evaluation for one baseline allocator (`ptmalloc2`) and 10 secure allocators using multiple modules.

```
const bool BypassQuarantine =
    !Size || (Size > QuarantineChunksUpToSize);
```

(a) A condition for quarantine in `scudo`.

```
p0 = malloc(0);
free(p0);
p1 = malloc(16);
```

(b) Reclamation for a small object

```
p0 = malloc(0x1000);
free(p0);
p1 = malloc(0x1000);
```

(c) Reclamation for a large object

Figure 4: Two ways to reliably reclaim objects (p0 → p1) in `scudo`. They satisfy each condition to bypass quarantine in Figure 4a, respectively.

discovered this issue if it uses a configuration to specify larger object sizes (> 512 KB). Thanks to HardsHeap's random exploration, HardsHeap can automatically discovered this issue by allocating an extremely large object without using any specific configurations. Second, we also confirmed that three allocators (`MarkUs`, `ffmalloc`, and `isoalloc`) have no random allocation support, which allows us adjacent chunks, as shown in Table 1.

Reclaim. Because a random reuse mechanism is strongly related to random allocations, most underlying reasons are equal to those in adjacent chunks — failures in large allocations. However, we also found several interesting results regarding reclamation in secure allocators. First, HardsHeap successfully identified boundary conditions to enable reliable reclamation in `scudo`. In particular, it discovered two types of test cases that could reliably reclaim objects by satisfying the conditions in Figure 4a; Figure 4b allocates a zero size to satisfy the first condition (`!Size`), and Figure 4c exploits the second condition where `QuarantineChunksUpToSize` is 2048 as default (`Size > QuarantineChunksUpToSize`). It is worth noting that this issue was fixed in the standalone `scudo`, but not in the non-standalone `scudo` that we tested. Second, unlike its security feature for defending against use-after-free vulnerabilities, `MarkUs` allows unsafe reclamation with dangling pointers. We further describe this in §9.3.

CheckOnFree. We also found that eight allocators failed to properly support the check-on-free (i.e., canary) mechanism for various reasons. First, five allocators (`DieHarder`, `Guarder`, `SlimGuard`, `hardened_allocator`, and `mimalloc`) do not check the canary for a large object. Second, `isoalloc`'s case is special. Even though it claims that it supports canary in its documentation, its canary is different from others; it randomly places the canary between chunks, causing it to fail to detect several corrupted chunks in `free`. This implies that we require an automatic analysis like HardsHeap by showing the insufficiency of the checklist-based, manual analysis for security properties similar to Table 1. Finally, `ffmalloc` and `MarkUs` are only designed to defend against use-after-free vulnerabilities; they have no support this mechanism for overflow.

Uninitialized. HardsHeap found three interesting cases in the `Uninitialized` module. First, `SlimGuard` leaves its heap pointer to construct a linked list in a deallocated object. This behavior can be abused in exploitation by leaking a heap pointer to break ASLR. Second, `mimalloc` leaves its metadata for large objects. We have reported this issue to developers; however, they conclude that it has no security implication because the metadata become obsolete in deallocating large objects. We further discuss this in §10.3. Third, HardsHeap claims `isoalloc`'s security violation for the `Uninitialized` module; however, it turns out to be a false positive. To prevent attacks, `isoalloc` fills an object with a magic byte (0xde) when the object becomes deallocated. Unfortunately, HardsHeap has no assumption about the shape of metadata for implementation-agnostic testing. Thus, it incorrectly considers this magic byte as metadata leakage.

SizeCheck. These findings are related to integer overflow bugs in allocators. We further discuss this issue in §9.1.

Spray. HardsHeap also found that three allocators — `FreeGuard`, `ffmalloc`, and `mimalloc`— suffer from heap spraying. Through post-analysis, we figured out that this occurs due to the blind use of `MAP_NORESERVE` in `mmap` for allocating a large object. In particular, it causes Linux to use memory overcommit; it allows to map an extremely large size of memory (> TB), resulting in a recurrent address among multiple executions. This issue becomes more serious in `mimalloc` due to its low entropy for an initial memory address. As a result, we could discover a deterministic case for `mimalloc`. We further discuss this in §9.4.

ArcHeap. HardsHeap also discovered three security violations in the `ArcHeap` module. In particular, we found that `SlimGuard` fails to check invalid free properly due to its implementation bug. Moreover, HardsHeap found that both `mimalloc` and `DieHarder` are vulnerable to double free, resulting in overlapping chunks. `DieHarder`'s bug is equivalent to the `ArcHeap`'s finding [49]; This is related to `DieHarder`'s non-protection of large objects. `mimalloc`'s violation is also similar to the old `ArcHeap`'s finding; however, we found that `mimalloc` has introduced a stochastic defense to mitigate this. This defense successfully stops a 100% reliable attack; however, it still allows overlapping chunks in a high probability (∼ 50%). Thanks to our sampling-based testing, HardsHeap successfully discovered this issue unlike `ArcHeap`, which only works in a non-random environment.

8.2 Statistical Significance Delta Debugging

To demonstrate the effectiveness of our delta debugging, we applied three types of delta debugging techniques — classical, greedy, and SSDD— to reduce test cases in §8.1. We exclude test cases that are always reproducible in our evaluation because all of the techniques switch to using the classical method for optimization, as explained in §5.4.

Figure 5 shows the results of our test case reduction using the above techniques. This shows that SSDD successfully reduces test cases by 37.2% on average, helping further root cause analyses. Moreover, SSDD outperforms other methods, and it it further reduces test cases without losing reproducibility. In particular, SSDD can produce smaller test cases than the greedy method. According to our evaluation, the SSDD's reduced test cases are 13.8% smaller than those from the greedy method on average (see Figure 5 (a)). The classical method can reduce test cases further; however, it significantly reduces reproduction probabilities due to its careless analysis of stochastic findings. As a result, the classical method results in 48.7% lower reproducibility compared to SSDD, whereas SSDD's probabilities are similar to the original ones (see Figure 5 (b)).

It is worth noting that the greedy method sometimes produces smaller cases than SSDD (e.g., see `SL AD` in Figure 5). At first glance, these results appear to be unusual because SSDD's approach is more aggressive than the greedy approach. Essentially, the greedy method's condition for reduction is sufficient to satisfy the SSDD's. However, this is still explainable because of the nature of stochasticity. In particular, it is possible that a single action can satisfy the greedy method's stronger condition for the reduction but not the SSDD's weaker condition because both methods rely on distinct sampling results. Moreover, HardsHeap sometimes fails to reduce test cases from `Guarder` and `FreeGuard` properly. This is because these allocators pre-allocate huge memory for their dynamic memory management (e.g., 64 TB in `FreeGuard`), resulting in system-level instability.

8.3 Reproducibility

The HardsHeap's findings are highly reproducible thanks to its straightforward analysis. To measure the reproducibility of HardsHeap, we created and ran PoCs, which are obtained from §8.1. Note that HardsHeap's PoC is equipped with assertions to ensure security violations. For example, the `Adjacent` module asserts that two chunks are adjacent using their pointers and sizes, as shown in Figure 3. Similarly, the `SizeCheck` module checks whether the actual size of a chunk is smaller than the request size (see Figure 6). Only exception is the `Spray` module; its PoC has no proper assertions because the current prototype of HardsHeap lacks a global analysis in building a PoC program; this analysis is required for the `Spray` module to identify the recurrent address. Therefore, we manually verified PoC programs for the `Spray` module whether they are reproducible.

Total	Failures (Generation)	Failures (Reproduction)	Success
3,343	684	34	2,625 (78.5%)

Table 6: The reproducilbity of HardsHeap. We could reproduce 78.5% of test cases found by HardsHeap thanks to HardsHeap's straightforward analysis.

As shown in Table 6, HardsHeap successfully generated 2,659 PoCs among 3,343 cases, and we successfully reproduced 2,625 test cases among them, which accounted for 78.5% of the total test cases. Most of the failures in generating PoCs come from incomplete PoC code because HardsHeap could be terminated due to system failures in allocators (e.g., memory exhaustion). Moreover, PoCs fail to be reproduced because 1) HardsHeap's memory layout could be different from a standalone PoC program because of its data structure, and 2) PoC could terminate early due to our `assert` to check security violations; however, in the analysis phase, HardsHeap can further explore for finding next security violations. Note that HardsHeap successfully discovered other variants that are successfully reproducible for finding security violations of secure allocators. We also believe that HardsHeap's reproducibility is fairly high (78.5%), which is sufficient for evaluating secure allocators.

9 CASE STUDY

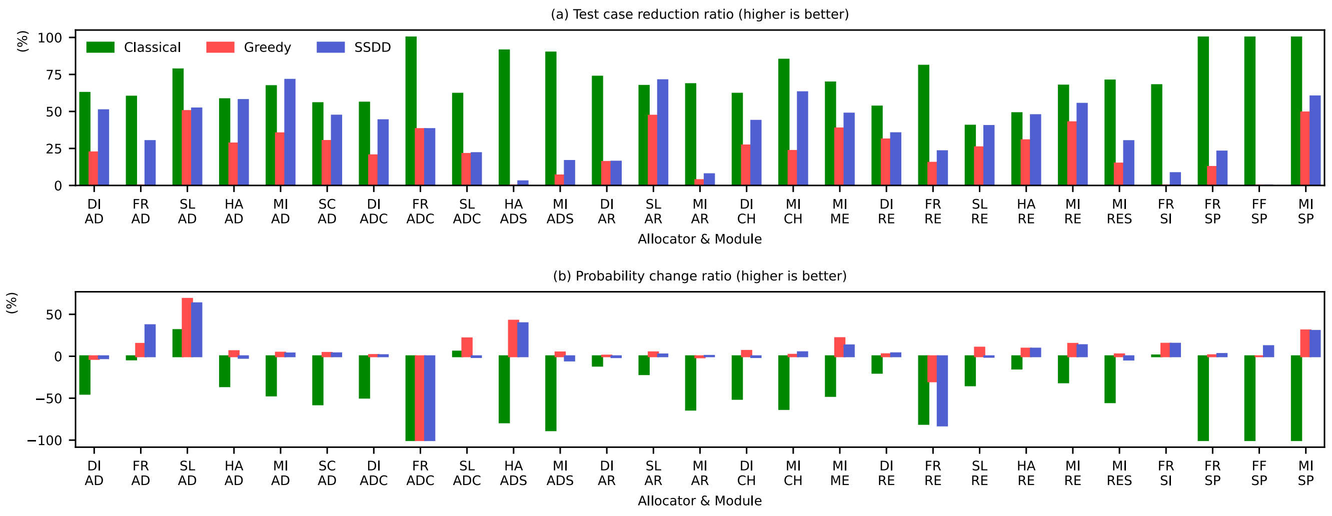
Although many test cases from HardsHeap are caused by intended trade-offs between security and performance, HardsHeap also found several implementation bugs, which show unexpected behaviors in secure allocators, as shown in Table 7. In the following, we describe each implementation bug in detail.

9.1 Integer Overflows in Multiple Allocators

HardsHeap found an integer overflow in memory allocation in several allocators. It is quite straightforward to trigger integer overflows in allocators (e.g., `malloc(-8)`). However its impact is huge; this bug can render a safe code under an ordinary allocator vulnerable. Let us assume that we have a program that reads a size, allocates a heap object with the size, and reads data up to the size. Although the program is correctly implemented, an attacker can trigger a heap overflow by giving an extremely large size (e.g., `-8`) to cause internal overflow of the allocator. Then, the allocator returns an object whose actual size is much less than the requested one due to its integer overflow, resulting in a heap overflow.

9.2 Predictable Seeds in `FreeGuard` and `Guarder`

While analyzing strange cases from the `Adjacent` module in `FreeGuard` and `Guarder`, we discovered their critical randomness issue. In particular, `FreeGuard` and its successor `Guarder` rely on predictable seeds in their pseudo-random generator (i.e., `srand(time(NULL))`). They use this random generator in multiple places for their secure behaviors, such as allocations and guard



AD: `Adjacent`, **ADC**: `Adjacent` (Cross), **ADS**: `Adjacent` (Small), **AR**: `ArcHeap`,
CH: `CheckOnFree`, **UN**: `Uninitialized`, **RE**: `Reclaim`, **RES**: `Reclaim` (Small), **SI**: `SizeCheck` **SP**: `Spray`

Figure 5: Test case reduction ratio and probability difference ratio in different delta debugging techniques. SSDD outperforms the greedy method by significantly reducing test cases. Moreover, SSDD can preserve probabilities of original test cases unlike the classical method.

Allocator	Module	Description	Status
`Guarder` `FreeGuard`	`Adjacent`	Insufficient randomness due to predictable seeds	**R** **R**
`MarkUs`	`Reclaim`	Unsafe reclamation in mmapped memory Unsafe reclamation due to failed allocation	**P** **P**
`mimalloc`	`Spray`	Heap spray is possible due to memory overcommit	**P**

`Guarder`
`FreeGuard`
`isoalloc`
`ffmalloc`
	`SizeCheck`	Integer overflow in memory allocation	**A**
			A
			P
			P

`SlimGuard`	`ArcHeap`	Insufficient check for invalid free	**R**

R: Reported, **A**: Acknowledged, **P**: Patched

Table 7: Implementation bugs in secure allocators that are discovered by HardsHeap.

```
int main() {
  void* p0 = malloc(-1);
  assert(malloc_usable_size(p0) < -1);
}
```

Figure 6: PoC code that triggers an integer flow in `ffmalloc` found by the `SizeCheck` module.

page placement, thereby making them predictable. We could discover this problem thanks to HardsHeap's sampling-based testing; HardsHeap reported that a certain test case from these allocators can always be reproducible in a specific time window. After investigating its root cause, we concluded that it occurs because of this weak random seed based on the current timestamp. We reported this issue to the developer and await their response.

9.3 Unsafe reclamation in `MarkUs`

HardsHeap also found unsafe reclamation in `MarkUs`; it is a secure allocator that is specially designed for preventing use-after-free (UAF). `MarkUs` prevents UAF attacks by forbidding the reallocation of an object with its dangling pointer. Unlike its theoretical guarantee, HardsHeap found that `MarkUs` still allows reclaiming memory for a dangling pointer. In particular, HardsHeap found two cases for unsafe reclamation in `MarkUs`. The first bug was caused by the simple error of omitting the mapped memory in their analysis; however, the second bug is more interesting. Figure 7 shows the simplified PoC code for the second `MarkUs` bug. This PoC first triggers an oversized allocation by calling `malloc(-1)` (Line 4). This large allocation caused `sbrk` failure, and `MarkUs` shifted to use `mmap` for further allocation. Unfortunately, `MarkUs`' security mechanism stopped working after this exceptional state and allowed unsafe

```c
int main() {
  void* p0 = malloc(-1);
  void* p1 = malloc(0x80000);
  free(p1);
  void* p2 = malloc(0x40000);

  // p2 reclaimed p1's region even p1 is dangling
  assert(p1 <= p2 && p2 < p1 + 0x80000);
}
```

Figure 7: PoC code that triggers unsafe reclamation in `MarkUs` even with a dangling pointer (i.e., `p1`).

reclamation. To fix it, the latest version of `MarkUs` disables this `mmap` feature because it will only occur with this oversized allocation, which rarely happens in normal applications. This shows the effectiveness of HardsHeap in discovering unexpected bugs in secure allocators.

9.4 Heap spray in `mimalloc`

HardsHeap also found that `mimalloc` is vulnerable to heap spraying attacks if an attacker can control the allocation size. For example, if we request 4 TB size memory in mimalloc, it successfully returns a valid object, which always includes `0x7FFFFFFF000` in its address range regardless of randomization. It happens that mimalloc always turns on memory overcommit by setting `MAP_NORESERVE` in its `mmap` call. As previous work has demonstrated [29], memory overcommit should be carefully used; otherwise, it can allow us to break ASLR. Thanks to our reporting, `mimalloc` fixed this issue by restricting the maximum allocation size for sufficient entropy in allocation [48]. We found that `Guarder` and `FreeGuard` have similar issues; however, they are much better than `mimalloc` because their memory addresses have higher entropy than `mimalloc`'s.

9.5 Other issues

HardsHeap also found that `SlimGuard` is insufficient to validate invalid free, which is similar to `ptmalloc2` [3, 36]. Even though `SlimGuard` claims its safety in invalid free, HardsHeap successfully found a counterexample for it.

10 DISCUSSION & LIMITATIONS

10.1 Usefulness of Secure Allocators

It is worth noting that our findings do not imply that secure allocators are useless. In fact, HardsHeap shows that secure allocators are effective in defending against several types of heap vulnerabilities. In particular, our evaluation shows that most secure allocators work well in securing small objects, which are prevalent in normal applications (see §8.1). Basically, HardsHeap only demonstrates that secure allocators are no silver bullet for heap vulnerabilities. They have several limitations particularly for large objects and elastic objects whose sizes are controllable. Moreover, they could be incorrectly implemented similar to other software. However, these facts do not completely eliminate the advantages of secure allocators. Therefore, *we should use secure allocators for better security.*

10.2 Windows support

We believe that HardsHeap can be extended to support other platforms such as Windows. Microsoft Windows is particularly interesting compared to other platforms; it supports a hardened allocator at the system level, which is known as Low-Fragmentation Heap (LFH) [24]. To this end, HardsHeap requires considerable changes to support Windows because of its significant differences from Linux. In particular, HardsHeap requires to use unique APIs for dynamic memory management such as `HeapCreate`, `HeapAlloc`, and `HeapFree`. Even though Windows also supports the standard C APIs such as `malloc` and `free`, they are just wrappers of the aforementioned APIs and are insufficient to evaluate unique features in Windows. Moreover, HardsHeap needs to use different system-level mechanisms such as signals and process creations, which are tightly coupled with the underlying platforms. For example, HardsHeap currently uses SIGUSR2 to notify its finding to the fuzzer; however, it is no longer usable in Windows.

10.3 Lack of reasoning

Even though HardsHeap's approach is applicable to diverse allocators owing to its implementation-agnostic approach, it is occasionally insufficient to understand the security implications of its findings due to a lack of reasoning. During our evaluation, HardsHeap found metadata leakage in `mimalloc`; we can leak metadata by deallocating a large object. Although this finding surprises the developers of `mimalloc`, we found that it has no security implication after further analysis. In more detail, leaked metadata from HardsHeap become obsolete when `mimalloc` marks its underlying pages as free. This happens in large object deallocations, which are required for leakage. Unfortunately, HardsHeap fails to reason this because it works without understanding the allocator's implementation.

10.4 Incompleteness

Similar to classical fuzzing, HardsHeap cannot guarantee any completeness in the security of secure allocators. In other words, HardsHeap only indicates us the existence of security violations but cannot prove their non-existence. Moreover, it is impossible to argue that the existing modules in HardsHeap are sufficient for evaluating security allocators. It happens that we empirically designed and implemented them without formal definitions of secure allocators. We believe that it is still debatable what security properties are sufficient to secure allocators. Thus, we leave it as future work to formally define these security properties against heap vulnerabilities.

11 RELATED WORK

11.1 Security Analysis of Secure Allocators

Many secure allocators have been developed to mitigate heap vulnerabilities. `DieHarder` [28] designs a secure allocator that supports several secure mechanisms, including segregated metadata and randomized allocations after formally analyzing existing attacks and allocators. `FreeGuard` [37] further reduces runtime overhead even though its security guarantee could be weaker than that of `DieHarder`. Moreover, `Guarder` [38] addresses the unstable security of previous works with low overheads. Microsoft also has employed several security mechanisms in its default allocator [7]. Moreover, `scudo` [22], a hardened allocator in the LLVM project, becomes a default allocator in Android's native code.

Despite such efforts to develop secure allocators, their security evaluations still remain ad-hoc. Most existing studies rely on manual analysis of security experts or theoretical analysis [5, 7, 28, 38]. There have been several studies for automatically evaluating the security of allocators. Heelan et al. proposes pseudo-random black box search for discovering adjacent chunks. HeapHopper [10] adopts bounded model checking to evaluate the security of allocators, while ArcHeap [49] leverages random testing. However, they are limited to non-secure allocators and cannot support randomization, which is essential in evaluating secure allocators. In comparison, HardsHeap can successfully evaluate secure allocators automatically and thoroughly thanks to its extensibility for supporting various security properties as well as sampling-based testing for handling randomization. Recently, Entroprise [39] evaluates the entropy of randomized allocations in secure allocators; however, it only supports pre-defined object sizes unlike HardsHeap. Entroprise can determine the actual entropy of memory as allocated by actual applications, which HardsHeap cannot support.

11.2 Delta Debugging

Delta debugging [19, 51] is a widely used technique for minimizing failing test cases; it launches a program with a smaller input and checks whether the failure still occurs with this reduced one. By repeating this process, delta debugging can find the minimum input for failure. HDD [27], C-Reduce [31], and Perses [41] expand this idea by exploiting a hierarchical structure in programming languages. Groce et al. [13, 14] adopt delta debugging to speed up software testing even without failures. Unfortunately, none of them assumes the stochastic failures that HardsHeap attempts to find. Choi et al. [9] and Hammoudi et al. [15] deal with stochastic failures using record and replay. However, record and replay is limited in HardsHeap because it cannot measure the probability of bad events (e.g., adjacent chunks), which is important for the evaluation of secure allocators. Thus, we devise another technique called SSDD, which repetitively samples to achieve the current probability in reduction.

11.3 Automatic Exploit Generation

There has been a line of research works for automatic exploit generation [4, 6, 17, 18, 23, 32, 34, 44]. Avgerinos et al. and Schwartz et al. [2, 34] explore fully automated exploit generation for stack overflow and format string bugs. To address the complexity of heap vulnerabilities, Repel et al. [32] and Heelan et al. [18] leverage modular approaches. Moreover, FUZE [47] and KOOBE [8] successfully demonstrate automatic exploit generation to a more complex target, Linux Kernel, for use-after-free and out-of-bounds vulnerabilities, respectively. However, these studies only focus on default allocators with limited security mechanisms or rely on domain-specific knowledge for exploitation. Even though HardsHeap cannot support end-to-end automatic exploit generation like these works, we believe that HardsHeap's findings can be used as a part of automatic exploit generation to secure allocators similar to other works for discovering useful exploit primitives [10, 23, 46, 49].

12 CONCLUSION

In this paper, we present HardsHeap, a new framework for automatically evaluating secure allocators. HardsHeap supports an extensible framework that makes it easy to build an analysis for each security property. Moreover, HardsHeap employs sampling-based testing and Statistical Significance Delta Debugging (SSDD) to support randomized security mechanisms. We applied HardsHeap to 10 secure allocators to show that HardsHeap's approach is effective in evaluating the security in secure allocators. Using HardsHeap, we also successfully discovered 10 implementations bugs that seriously harm the security properties of the allocators.

13 ACKNOWLEDGMENT

We thank the anonymous reviewers, and our shepherd, David Chisnall, for their helpful feedback. We also thank to Kostya Kortchinsky for discussion on the `scudo` allocator. We thank to Emery Berger and Michael Steranka for discussion on `DieHarder` and Entroprise. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE grant under Agreement No. HR00112090034 ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google. This work was also supported by Institute for Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2019-0-01343, Regional strategic industry convergence security core talent training business).

REFERENCES

[1] Sam Ainsworth and Timothy M Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[2] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2011. AEG: Automatic exploit generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[3] blackngel. 2009. Malloc Des-Maleficarum. http://phrack.org/issues/66/10.html.

[4] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[5] Silvio Cesare. 2020. Breaking Secure Checksums in the Scudo Allocator. https://blog.infosectcbr.com.au/2020/04/breaking-secure-checksums-in-scudo_8.html.

[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[7] Wei Chan. 2019. Heap Overflow Exploitation on Windows 10 Explained.

[8] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[9] Jong-Deok Choi and Andreas Zeller. 2007. Isolating failure-inducing thread schedules. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

[10] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. Heaphopper: Bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[11] Google. 2013. Partition Alloc Design. https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md.

[12] GrapheneOS. 2018. Hardened malloc. https://github.com/GrapheneOS/hardened_malloc.

[13] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA.

[14] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: delta debugging, even without bugs. *Software Testing, Verification and Reliability* 26, 1 (2016), 40–68.

[15] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. 2015. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 10th European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Bergamo, Italy.

[16] Ben Hawkes. 2019. 0day "In the Wild". https://googleprojectzero.blogspot.com/p/0day.html.

[17] Sean Heelan, Tom Melham, and Daniel Kroening. 2018. Automatic Heap Layout Manipulation for Exploitation. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[18] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.

[19] Lukas Kirschner, Ezekiel Soremekun, and Andreas Zeller. 2020. Debugging inputs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*.

[20] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[21] Beichen Liu, Pierre Olivier, and Binoy Ravindran. 2019. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Proceedings of the 20th International Middleware Conference*.

[22] LLVM Project. 2019. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[23] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[24] Microsoft. 2018. Low-fragmentation Heap. https://docs.microsoft.com/en-us/windows/win32/memory/low-fragmentation-heap.

[25] Microsoft. 2019. mimalloc. https://github.com/microsoft/mimalloc.

[26] Matt Miller. 2020. Pursuing Durably Safe Systems Software. In *SSTIC*.

[27] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. Shanghai, China.

[28] Gene Novark and Emery D Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL.

[29] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX.

[30] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Beijing, China.

[31] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China.

[32] Dusan Repel, Johannes Kinder, and Lorenzo Cavallaro. 2017. Modular Synthesis of Heap Exploits. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*. Dallas, TX.

[33] Chris Rohlf. 2020. https://github.com/struct/isoalloc.

[34] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy.. In *Proceedings of the 20th USENIX Security Symposium (Security)*. San Francisco, CA.

[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA.

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

[37] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. Freeguard: A faster secure heap allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[38] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A tunable secure allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[39] Michael Steranka and Emery Berger. 2019. Entroprise. https://github.com/plasma-umass/entroprise.

[40] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.

[41] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden.

[42] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. Belgrade, RS.

[43] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[44] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada.

[45] Brian Wickman, Hong Hu, Insu Yun Daehee Jang, JungWon Lim Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. (Aug. 2021).

[46] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[47] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[48] Insu Yun. 2021. mimalloc issue # 372. https://github.com/microsoft/mimalloc/issues/372.

[49] Insu Yun, Dhaval Kapil, and Taesoo Kim. 2020. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[50] Michal Zalewski. 2014. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[51] Andreas Zeller. 1999. 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.