All Use-After-Free Vulnerabilities Are Not Created Equal: An Empirical Study on Their Characteristics and Detectability

Zeyu Chen
University of Delaware
Newark, Delaware, USA
zeyuchen@udel.edu

Daiping Liu
University of Delaware
Newark, Delaware, USA
dpliu@udel.edu

Jidong Xiao
Rensselaer Polytechnic Institute
Troy, New York, USA
xiaoj8@rpi.edu

Haining Wang
Virginia Tech
Arlington, Virginia, USA
hnw@vt.edu

ABSTRACT

Over the past decade, use-after-free (UaF) has become one of the most exploited types of vulnerabilities. To address this increasing threat, we need to advance the defense in multiple directions, such as UaF vulnerability detection, UaF exploit defense, and UaF bug fix. Unfortunately, the intricacy rooted in the temporal nature of UaF vulnerabilities makes it quite challenging to develop effective and efficient defenses in these directions. This calls for an in-depth understanding of real-world UaF characteristics. This paper presents the first comprehensive empirical study of UaF vulnerabilities, with 150 cases randomly sampled from multiple representative software suites, such as Linux kernel, Python, and Mozilla Firefox. We aim to identify the commonalities, root causes, and patterns from real-world UaF bugs, so that the empirical results can provide operational guidance to avoid, detect, deter, and fix UaF vulnerabilities. Our main finding is that the root causes of UaF bugs are diverse, and they are not evenly or equally distributed among different software. This implies that a generic UaF detector/fuzzer is probably not an optimal solution. We further categorize the root causes into 11 patterns, several of which can be translated into simple static detection rules to cover a large portion of the 150 UaF vulnerabilities with high accuracy. Motivated by our findings, we implement 11 checkers in a static bug detector called `Palfrey`. Running `Palfrey` on the code of popular open source software, we detect 9 new UaF vulnerabilities. Compared with state-of-the-art static bug detectors, `Palfrey` outperforms in coverage and accuracy for UaF detection, as well as time and memory overhead.

CCS CONCEPTS

• **Security and privacy** → System security; Bug-finding, debugging.

KEYWORDS

Use-after-free; Static detector; Benchmark

ACM Reference Format:
Zeyu Chen, Daiping Liu, Jidong Xiao, and Haining Wang. 2023. All Use-After-Free Vulnerabilities Are Not Created Equal: An Empirical Study on Their Characteristics and Detectability. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3607199.3607229

RAID '23, October 16–18, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0765-0/23/10…$15.00
https://doi.org/10.1145/3607199.3607229

1 INTRODUCTION

In C/C++ programs, a use-after-free (UaF) bug[1] is triggered when a pointer pointing to freed memory is dereferenced. This pointer is referred to as a *dangling pointer*. Normally, dereferencing a dangling pointer results in data corruption and/or program crashes (denial of services). However, when the dangling pointer and program state can be manipulated by attackers, UaF bugs may lead to much worse unexpected consequences. For example, if attackers can fill the freed memory with deliberately crafted contents, UaF bugs potentially allow attackers to run arbitrary code. UaF vulnerabilities also serve as a popular attack vector to leak the in-memory addresses of certain libraries like `libc` and thus defeat the address space layout randomization (ASLR) in modern systems. Actually, if a UaF bug is present in software, there is a high likelihood that an exploit could be developed to compromise the victim software system [99].

1.1 Motivation

Unfortunately, UaF vulnerabilities are almost inevitable and difficult to detect and fix. Although significant progresses from multiple directions have been made over the past years to address this security challenge, there are still many open, unsolved issues.

Static UaF vulnerability detection. Previous research [49, 54, 57, 62, 69, 88, 102, 103, 107–109] has used static analysis to detect UaF bugs from source code. In static analysis, detectors track memory allocation and pointer propagation. When a dereferenced pointer points to freed memory, a bug is detected. While some promising results have been achieved, it is still far from a complete solution to address UaF bugs. For example, some approaches only consider UaF bugs in specific software suites like Linux device drivers [49]. Also, static detectors often have many false positives [59].

[1]Throughout the paper, we consider that each UaF bug is a potential vulnerability. Thus, we use UaF bug and UaF vulnerability interchangeably.

More importantly, several essential questions remain unanswered: what portion of real world UaF bugs can be detected? How many patterns into which real-world UaF bugs fall? And, is there any UaF bug pattern that has not yet been discovered by existing detectors?

Dynamic software fuzzing. Software fuzzing is a popular dynamic technique to find bugs using automatically generated input data. The program used to generate these inputs is called a fuzzer. Empirical studies show that fuzzing can effectively uncover many common bugs like buffer overflows [81].

Early fuzzers often generate inputs randomly. Therefore, it usually takes a long time, from hours to months, to provide sufficient coverage and find latent bugs. In recent years, the coverage-guided fuzzing has attracted much attention to improve the fuzzing efficacy [51, 52, 55, 65–67, 91, 96]. Coverage-guided fuzzers can effectively reveal many overlooked corner cases, and they have become quite appealing for use in production testing environments. Actually, by virtue of open-source fuzzing tools and frameworks like American fuzzy lop (AFL) [113] and oss-fuzz [17, 92], fuzzing has seen great success in discovering many bugs from a variety of software. When coupled with memory corruption detection tools like Valgrind [85] and ASan [93], fuzzing can also uncover many UaF bugs in large software suites like the Chrome browser.

However, it is still unclear, both quantitatively and qualitatively, how effective the state-of-the-art fuzzers can expose UaF bugs. The major challenge of software fuzzing is the exponential increase in space exploration. The exposure of UaF bugs can be even more challenging, since it requires to trigger a premature memory free and explore all pointer propagation paths. Even worse, many UaF bugs are non-deterministic due to the uncertainty in memory allocators and the concurrency in multi-thread applications. Ultimately, the design of a practical and efficient UaF fuzzer requires a better understanding of the manifestation of real world UaF bugs, with which fuzzers can prioritize the program states and execution paths.

UaF exploit defense. As it is almost impossible to find and fix all UaF bugs during software development, another defense direction is to eliminate dangling pointers, the culprit of UaF vulnerabilities. Along this direction, many modern programming languages have adopted various mechanisms like tracing garbage collection (GC) in Java [18], reference-count GC in Python [19], automatic reference counting (ARC) in Swift [20], smart pointers in C++ [21], and ownership in Rust [22]. There are also solutions to arm legacy C/C++ code with reference counting to address UaF bugs [50, 94]. Meanwhile, based on the insight that UaF vulnerabilities can be avoided when there is no memory reuse, some other methods attempt to avoid page sharing in virtual memory [60, 78, 87]. In essence, these methods follow the rule that a memory block cannot be freed as long as a pointer (may) still points to it.

Several recent works instead follow a different rule to harden C/C++ applications. The rule assumes that all explicit memory frees are what programmers expect, and all dereferences to dangling pointers can be considered as potential exploits. Guided by this rule, one defense mechanism is lock-and-key, in which a unique identifier is generated for each memory allocation (i.e., a lock) and is then associated to pointers (as the key) for accessing the memory [58, 84]. Another approach proactively nullifies the dangling pointers at the time of memory free [72, 76, 101, 110]. These methods can effectively defeat attackers' exploit attempts.

Although mechanisms like GC and dangling pointer nullification show great potential, there are still several open questions. (1) How effective are these mechanisms and what are the caveats to watch out in implementation? (2) How effective is the widely adopted reference count to thwart UaF in C/C++? In addition to programming language features like smart pointers and ownership, there are also ad-hoc implementations of reference count in multiple software suites like Linux kernel and Python. These solutions mostly depend on programmers to correctly understand and use the provided primitives. (3) Is there a better trade-off between performance overhead and protection coverage to facilitate these runtime defenses (e.g., dangling pointer nullification) to be widely adopted in a production environment?

1.2 Contributions

Empirical findings. To address those open problems, this work provides the first comprehensive study on the characteristics of real world UaF bugs. Specifically, we examine bug patterns, manifestations, fix strategies, and several other features. Our study is based on 150 randomly selected real world UaF bugs, collected from 41 large and mature open-source projects like Linux kernel, Python, and Mozilla Firefox. For each bug, we carefully examine its bug report, corresponding source code, related patches, and programmers' discussion, all of which together provide us a relatively thorough understanding of the bug patterns, manifestation conditions, fix strategies, and diagnosis procedures. Our study reveals a number of interesting observations that can provide useful guidelines for UaF bug detection and protection. We summarize our main findings and their implications as follows:

- Heap UaF dominates (148 out of 150), although there are two real UaF bugs on data memory segment (§3.2).
- The majority of UaF bugs are deterministic (115 out of 150), which are generally easier to detect and fix than nondeterministic ones (§3.3).
- Most UaF bugs do not cause crashes when exploited using the proof-of-concept (PoC) code. However, when the applications are hardened with AddressSanitizer (ASan) [93], all UaF bugs will cause crashes. Therefore, fuzzing systems need to couple with ASan to achieve more effective detection performance (§3.5).
- The root causes of the majority bugs can be categorized into 11 patterns. In particular, ~33% follow certain patterns that exhibit strong spatial and temporal relevance. Such bugs are mostly low-hanging fruits and UaF detectors can cover them without much difficulty (§4).
- 26% UaF bugs are caused by errors in reference counting, a popular technique to defend against memory corruption. While it is well known that incorrect reference counts of objects lead to UaF, our study reveals another major cause, the misuse of borrowed references (§4.2).

Besides the empirical findings, the major contribution of this work lies in three aspects. (1) **UaF benchmark (§4.3).** We constructed a new UaF benchmark `UAFBenchPlus`. Compared to UAFBench [23], `UAFBenchPlus` contains more bugs (120 *v.s.* 14) from more applications (41 *v.s.* 11) with more diverse patterns. (2) **Static**

Source	Deterministic	Non-deterministic	
		Concurrent	Others
Firefox	25	5	0
CPython	27	2	1
Linux	16	10	4
GitHub	47	11	2
Overall	115	28	7

Table 1: Number of UaF bugs collected from three representative software suites and 38 GitHub open source repositories.

detector (§5). To measure the applicability of the detection rules extracted from our empirical study, we built `Palfrey`, a simple static UaF detector. We detected nine *new* UaF bugs in seven production quality open source projects. We also conducted a thorough analysis in term of bug detection ability, performance and false positives with other tools. (3) **Open dataset.** Finally, to foster the future work, we will make our dataset and tools publicly available.

The rest of the paper is organized as follows. Section §2 describes the general problems of UaF and the methodology about the bug collection in the empirical study. Section §3 outlines the general findings from the empirical study. Section §4 details UaF bug patterns, their implications, as well as a UaF benchmark built from the empirical study. Section §5 presents a static tool `Palfrey` and its related experiment results. Section §6 surveys related work, and finally Section §7 concludes this work.

2 METHODOLOGY

2.1 UaF Bug Collection

UaF bugs occur across a wide range of programs that have various features and development cycles. The programs can be commercial/free and closed source/open source. We use the following methodology and principles to compile the datasets for this study:

- **Open source.** In order to thoroughly analyze and understand UaF bugs, we need to access the source code. We leave the extension to closed source programs with only binaries available as our future work.
- **Public bug trackers.** We extracted the UaF bugs from the publicly accessible bug tracking systems of the open source projects. The bugs tracked in these systems are usually well-documented with high quality details. They are often confirmed or disputed by developers. We can also view the discussions on the bug trackers, including the causes and their potential effects, and track how the bugs get resolved like the code patches. Note that although some projects' bug trackers are open, the bugs' details are mostly hidden to the public, e.g., Chromium browser. We consider such cases as closed bug trackers.
- **Mature and actively maintained.** The bugs are selected from mature projects that have stable releases and developers for active maintenance.

Based on these guidelines, we selected a final set of 150 real-world UaF bugs that were randomly sampled from the following sources, as summarized in Table 1.

Three representative software suites. We searched the bug tracking systems from three software suites: Linux Kernel [75], Mozilla Firefox [83], and CPython [90]. They represent three most widely used applications with large code base, more than 10 years development history, and million lines of code. These applications show variations in programming languages (`C` vs `C++`) and memory management mechanisms (manual memory collection vs reference-count based memory collection).

The search function of each bug tracking system was extensively used to find the reported UaF vulnerabilities. In particular, they provided many filters and keyword options to reduce the results to the desired bug reports. The keywords we used include `"uaf"`, `"use-after-free"`, `"dangling pointer"`, `"double free"`, and their variants. To make the bug search comprehensive, we also used the keywords `"data race"`, `"concurrency"`, `"weak reference"`, and `"reference count"`, etc. Then, we checked if such bugs are UaF. From there, we extracted the desired bugs in two steps. First, we cross-checked with the Common Vulnerabilities and Exposures (CVE) database [100]. If a bug has a CVE entry, it is more likely to be a security vulnerability, and thus we prefer such bugs in our study. Then, we randomly sampled a set of results and investigated to verify that the reported bugs were truly confirmed UaF vulnerabilities. In some cases, our search terms were included in the bug reports, but they ended up being incorrect designations by developers. Such cases were ignored during our data collection.

There are three major challenges in the bug report collection. (1) Many bug reports are incomplete. In particular, it is unclear how to trigger the bugs and no PoC is provided. (2) UaF bugs are not easy to understand. Their patterns and manifestations usually involve complicated interactions among multiple software components. (3) Some bugs in different reports originate from the same root cause, e.g., issue 24099 [2] and issue 24101 [3] in Python.

We filtered out a significant number of UaF bugs during the selection process to ensure the quality and relevance of the dataset. Specifically, we excluded bugs that have incomplete reports, miss crucial details, or could not be fully understood or reproduced. In addition, we discarded bugs that were incorrectly classified as UaF or did not meet our defined criteria. As we encountered instances where multiple bugs originated from the same root cause, we made the decision to retain one representative bug to avoid duplications in our analysis. After this meticulous filtering process, we arrived at a final set of 150 real-world UaF bugs for our study, which provided a comprehensive and reliable base for our analysis. The excluded bugs, due to their limited or incomplete information, were not considered in our analysis to maintain the accuracy and reliability of our findings.

38 GitHub open source repositories. In order to complement our dataset with bugs from a more diverse set of projects, we leveraged GitHub [24], one of the largest code hosting services for open source software. The same keywords as above were used to search for UaF bugs on GitHub. Meanwhile, in order to find the bugs conforming to the principles described above, we adopted multiple searching criteria. Specifically, the bugs are limited to the code repositories with active commits within recent six months and more than 1,000 stars. Finally, we manually verified that the repositories contain production level code, instead of teaching or sample code. In total, we chose 60 bugs from 38 repositories. The full list of the repositories is presented in Table 2.

kbd	nng	php	vlc
oboe	dcmtk.	Envoy	KeyDB
mongo	mySQL	swift	uacme
yosys	coturn	Jungle	spead2
SQlite	u-boot	f-stack	gnucash
osquery	electron	i2p.i2p	php-src
systemd	arangodb	goaccess	Open SSH
mosquitto	OMCompiler	oniguruma	profanity.
plan9port	verilator	librdkafka	timescaledb
foundationdb	qpid-dispatch		

Table 2: Github repositories for empirical study

Discussion. We would like to point out that the number of UaF bugs collected from each application does not reflect how secure or insecure an application is or how common UaF is in an application. Our dataset simply represents the UaF bugs that meet our selection criteria and get randomly selected.

2.2 Threats to Validity

As for all empirical studies, the conclusions drawn from our results are subject to some threats to the validity. In the following, we discuss the potential sources of bias and how we address them with our best-efforts.

One limitation of our study is the representativeness of the selected bugs, which may not be representative. There are millions of software packages in the wild, and it is impractical for our study to cover all of them. In particular, our dataset is exclusively collected from open source programs. As a result, our findings might not be generalized to a larger population of commercial software. However, modern open source programs have comparative quality as commercial ones, and many open source programs are widely used in production environments. We carefully select UaF bugs from *diverse* types of open-source programs. And, the bugs are *randomly* sampled from GitHub and bug trackers of popular software, which serve as a great population of UaF bugs. Therefore, we believe that the findings derived from our dataset can provide valuable guidelines for future defense.

We also give higher priority to the latest bugs. Therefore, the findings might not represent the bug characteristics years ago. However, we argue that the latest bugs can provide more valuable and timely insights. Another limitation to the generalizability of our study is the size of our sample set. According to modern statistics, a random sample set with size greater than 30 is larger enough to represent the entire population [95]. Actually, our sample size is comparable to previous similar empirical studies [70, 73, 77, 111].

Finally, our study could suffer from potential observer errors. To minimize the possibility of observer errors, a group of researchers share the same detailed protocol to inspect and classify the bugs, and all bugs are cross-checked by two researchers. In addition, we are familiar with the three targeted programs and some of the randomly sampled applications.

Source	UaF	Non-UaF
Firefox	641	391
Python	1,207	671
Linux	1,312	679
GitHub	1,222	765

Table 3: Average lifespan of UaF and non-UaF bugs, measured in days between when a bug was introduced until a patch was committed.

Memory Segment	# of bugs
Stack	0
Data	2
Heap	148

Table 4: Memory segment of UaF bugs.

3 GENERAL FINDINGS

This section sheds light on some general findings from our study to provide a better understanding on the lifespan and manifestation of UaF bugs.

3.1 Lifespan of UaF Bugs

Table 3 shows the average UaF lifespan, which is defined as the duration from the time when the bug was introduced to when its patch was committed. As a comparison, we also randomly selected 40 non-UaF bugs (with 10 from each source) and analyzed their average lifespan.

Finding 1. *The lifespan of UaF bugs on average is 74.9% longer than non-UaF ones.*

Overall, our results show that UaF bugs have much longer lifespan, implying that UaF bugs are more sophisticated and more difficult to find and fix.

3.2 Where Do Dangling Pointers Point To?

While UaF bugs often occur on heap memory (i.e., heap UaF) and are the focus of most UaF studies, a dangling pointer theoretically can point to prematurely freed memory located on any segments besides heap. Table 4 shows the memory segments of UaF bugs in our dataset.

Finding 2. *While heap UaF dominates (accounting for 98.7%), dangling pointers pointing to other memory segments do exist in the wild.*

First, we did not find a real stack UaF case. A stack UaF occurs when a pointer points to the address of a stack variable after its declaring function has returned. A hypothetical example of stack UaF is presented in Figure 1.

While no stack UaF was found, we did observe two data UaF. As Figure 2 shows, the static variable `module_initialized` is used to guard against multiple initializations. However, when the static module is freed and the corresponding resources get released, the static variable is not cleared. Consequently, if later the static module is initialized again, the runtime system will incorrectly consider

```
int *func_return_stack_addr() {
   char str[32];
   int value = 1024;
   return &value;
}
// stack_ptr is a dangling pointer to a freed stack frame
int *stack_ptr = func_return_stack_addr();
*stack_ptr = 2048;
```

Figure 1: A hypothetical example of stack UaF

the module as initialized, and thus it will use the previously freed resources, resulting in UaF.

Another interesting case is due to the static destruction order fiasco, as shown in Figure 3. In C++, the destruction order of static objects is undefined across different compilation units, and therefore, the destruction order of some static objects, e.g., `alias_table` and `parameter_set` in Figure 3, becomes non-deterministic at runtime. This could lead to UaF. This patch changes them to be static functions that return static member variables, avoiding the problem.

Finally, as expected, about 98.7% (148 out of 150) real-world UaF bugs occur on heap memory. This is because heap memory is usually dynamic and thus prone to be misused due to coding mistakes. In addition, the lifespan and content of objects on heap are easier to be manipulated by attackers. Consequently, heap UaF generally poses more serious threats.

One more issue to mention is that C/C++ supports arithmetic on pointers, and thus a pointer can point to any byte within a memory block at runtime. During our study, we found 1 heap memory case where the dangling pointer points to the internal (instead of the head) of a memory block. For instance in Figure 4, `*_haystack` and `*_needle` point to the memory assigned by `malloc()` since `strup()` calls `malloc()` to allocate memory. The memory will thus be finally released. `ret` points to somewhere within `_haystack` (or null if there is no match) when calling `strstr(_haystack, _needle)`. Since `_haystack` is released, the function then returns a freed memory, which results in UaF.

In summary, while non-heap UaF does exist in the wild, its pattern is relatively simple. By contrast, heap UaF is much more prevalent and complex. In addition, our last example indicates that it is imperative for some dynamic detection to keep track of not only where a pointer points to, but also the object size. Therefore, heap UaF deserves a deeper study and its characterization will be our focus.

3.3 Nondeterminism of UaF

Finding 3. *While the majority of UaF bugs are deterministic, a non-trivial portion (35 out of 150) are non-deterministic.*

Many UaF bugs (35 out of 150) are triggered non-deterministically. First, a large portion (28 out of 35) of non-deterministic UaF bugs are actually concurrency bugs. To highlight the distinctions between UaF and general concurrency bugs, we compare our data with the previous study by Lu et al. [77]. Specifically, in our study all collected UaF concurrency bugs are non-deadlock, while 66% of general concurrency bugs are non-deadlock. In terms of bug patterns, 96% of UaF concurrency bugs are triggered by atomicity-violation

```
static int module_initialized;
static void module_free(void *m) {
    ...
  + module_initialized = 0;
}
static int module_init(void) {
    if (module_initialized != 0) {
        return 0;
    }
    else {
        module_initialized = 1;
    }
}
```

Figure 2: Global UaF due to loss of memory flag update[25]

```
  - static std::unordered_map<...> alias_table;
  - static std::unordered_set<...> parameter_set;
  + static std::unordered_map<...> alias_table();
  + static std::unordered_set<...> parameter_set();
```

Figure 3: Static destruction order fiasco [26]

```
char *strcasestr(const char *haystack, const char *needle) {
    char *ret = NULL;
    char *_haystack = strdup(haystack);
    char *_needle = strdup(needle);
    // _haystack and _needle are assigned by malloc, and
         would be freed
    ret = strstr(_haystack, _needle);
+   if (ret)
+       ret = (char *)haystack + (ret - _haystack);
    free(_haystack);
    free(_needle);
    return ret; // points to freed memory
}
```

Figure 4: Dereference of an strup() object[27]

```
void js_DestroyContext(...){
  js_UnpinPinnedAtom(&atoms) // free
}
void js_DestroyContext(...){
  js_MarkAtom(&atoms,...); // read
}
//error: two functions run concurrently but
     js_UnpinPinnedAtom should happen after js_MarkAtom
```

Figure 5: UaF as a concurrency bug [1]

and order-violation, which is consistent with the observation in [77] (97%). Among the atomicity-violation and order-violation bugs, four are caused by data race [45]. Besides, in our study, the manifestations of all UaF concurrency bugs involve concurrent accesses to a single variable (compared to 66% in [77]). Therefore, it is a good strategy for UaF concurrency bug detectors to focus on atomicity-violation, order-violation, and non-deadlock concurrent accesses to single variables.

Second, the manifestation of UaF could depend on the system state. For instance, when calling certain functions, insufficient memory allocation [28] will result in errors and then cause UaF (see §4.1.3). Another special case is static destruction order fiasco as explained above in Figure 3.

3.4 Line of Free to Line of Dereference

When analyzing and detecting UaF bugs, one important metric is the code range between the creation and dereference of a dangling pointer. This metric is especially valuable and useful for non-concurrent (122 out of 150) UaF analysis. Thus, we measured the lines of code between where the dangling pointer is created and where it is dereferenced for all sequential UaF bugs. We refer this metric as `Line-of-free-to-Line-of-dereference (LOFTLOD)`. `LOFTLOD` does not represent the number of lines of code to execute at runtime. In particular, we do not count the lines of code in the body of a called function, which is counted as one line. And, in case of a conditional statement, each branch is counted in sequence until reaching the last branch or the statement of the dangling pointer dereference.

When computing the `LOFTLOD` metric, the valuable control-flow information is discarded. In order to provide a clearer picture of the UaF code ranges, we augment `LOFTLOD` with the basic block (BB) information. Specifically, we check if the dangling pointer creation and dereference instructions are in the same BB, adjacent BB, or non-adjacent BB. The BB-augmented `LOFTLOD` metric simplifies the code analysis while still providing valuable information.

BB relationship can be easily acquired in Clang if our checker successfully detects a UaF bug. When a specific bug pattern is identified, the `BugReporter()` function is invoked to provide a trace of the UaF bug. To determine whether the code resides within the same BB, we utilize the `CheckerContext::getBlockID()` function, which retrieves the block ID for a given position. If the block IDs of the memory-free and memory-dereference positions are the same, it indicates that they belong to the same basic block. Conversely, if they differ, we employ the `isAdjacent()` function to determine if they are in adjacent basic blocks. In cases where UaF bugs go undetected, we conduct an analysis of the root cause for each bug by identifying where the memory is freed and exploring potential positions for memory dereference. Based on our understanding of the bug and the analysis of the code structure, we compute the closest distance and obtain the corresponding basic block information.

Our analysis results are shown in Table 5. Interestingly, there are 35 cases whose `LOFTLOD` is smaller than 10, and 49 cases whose dangling pointer creation and dereference happen in the same or adjacent basic block. For instance, Figure 6 presents a real-world case in CPython, where `tstate` is freed and then accessed almost immediately. This implies that a fair amount of sequential UaF bugs are relatively easy to detect. Specifically, it could be a high-rewarding strategy for a detector to give a higher weight to the code snippets after the instructions where dangling pointers are created.

	Number of bugs			
LOFTLOD	<10	<50	<200	>200
Same BB	27	6	0	0
Adjacent BB	8	3	0	5
Nonadjacent BB	0	21	8	44
Overall	35	30	8	49

Table 5: Distribution of basic block (BB) augmented LOFTLOD of the bugs in the collected C/C++ applications.

```
tstate_delete_common(tstate); // free
 if (autoInterpreterState &&
      PyThread_get_key_value(autoTLSkey) == tstate)
    // invalid read
    PyThread_delete_key_value(autoTLSkey);
```

Figure 6: Immediate use-after-free

3.5 UaF Reproducibility

We also reproduced some of the collected bugs and observed their behaviors by using AddressSanitizer (ASan) [93]. We aim to understand (1) how likely a UaF bug leads to software crashes and (2) how effective ASan is to detect UaF. To this end, we randomly selected 30 bugs from the data set. Table 12 lists all of the tested 30 bugs. Some bugs were reported along with the proof-of-concept (PoC) code that was directly used in our test. For the rest, we hand-wrote the PoC code to trigger the bugs. For each bug, we ran the PoC code (1) without ASan to see if the bug crashes the software and (2) with ASan to see if ASan could detect the UaF bug.

Application crash serves as the main externally observable behavior that modern fuzzers rely on for bug detection. Our evaluation shows that only 3 out of the 30 tested bugs caused crashes. This implies that most UaF bugs are actually "silent" and fuzzers require auxiliary tools for effective UaF detection.

Finding 4. *Without ASan, only 3 out of 30 UaF bugs caused crashes.*

When a dangling pointer points to a freed and then re-allocated memory block, ASan could have false negatives. To address this issue, ASan implements a quarantine mechanism. After a memory block is freed, it is marked as unaddressable and put into quarantine. The reuse of the freed memory in quarantine will be delayed and will not be allocated by `malloc` in a short time. The quarantine is implemented using a first-in-first-out (FIFO) queue, which holds a fixed amount of memory at any time. When the queue is full, some old memory will be moved out from quarantine and thus will become available for reuse. As a result, the size of the quarantine queue is critical for effective UaF detection, with a small one leading to many false negatives and a large one incurring high performance overhead. Clearly, it is meaningful to evaluate how reliable the quarantine mechanism is in practice, especially with the default quarantine size.

Finding 5. *Given the default quarantine size of 256MB, ASan successfully detected all of the tested 30 UaF bugs.*

Overall, our tests show that the default quarantine size of 256MB is sufficient in real world. Even when the quarantine size is reduced to 64MB, ASan can still reliably detect 27 out of 30 UaF bugs.

Category		Freq.
Immediate UaF	move-free-before-use (P1)	12
	save-before-free (P2)	13
Raise a flag	not or falsely updated (P3)	12
	not checked (P4)	7
Memory resize	improper memory resize (P5)	9
API	API misuse (P6)	14
Double free	free inside a loop (P7)	20
Ref count	use of borrowed ref (P8)	26
	over-decremented ref (P9)	6
	non-decremented ref (P10)	3
	misused ref-count API (P11)	3
Others	other causes (P12)	35

Table 6: The occurrence frequency of each UaF pattern. Note that the total number is more than 150 since there might be overlaps between the two patterns. For example, both flag error and misuse of reference count could lead to double free, and API misuse could result in reference count error.

4 UAF CHARACTERIZATION

To better facilitate the manifestation and detection of UaF bugs, we conduct a deeper study for UaF characterization. In particular, we aim to answer the following questions:

- How are UaF bugs introduced? This can provide clues on the best practices to avoid UaF.
- Do UaF bugs have patterns that can provide insights on the design of bug detection tools?

We discover that concurrent UaF has no remarkable differences in root causes from general concurrency bugs studied in previous work [77]. Therefore, we mainly focus on sequential UaF bugs in this section. Meanwhile, as reference counting is a promising technique to address UaF and has been widely adopted, it is desirable to understand its efficacy to prevent UaF in practice. Table 6 shows the occurrence frequency of each pattern. The details are discussed in the rest of this section.

4.1 All UaF Bugs Are Not Created Equal

4.1.1 Immediate-use-after-free. The dangling pointers in 25 UaF bugs get dereferenced only after a few lines of code where the memory is freed. We divide the bugs into two patterns based on their fixes.

Move-free-behind-use (P1): For 12 bugs, the patch is simply to move the memory free after the pointer dereference statements. A typical example can be found in Figure 6. Such simple cases should have been caused by programmers' carelessness. And, because they often do not crash programs and even in some cases the expected data can be read through the dangling pointers, such bugs can be easily overlooked.

Save-before-free (P2): We also observed 13 cases where a memory block is freed, while some internal pointer fields are still needed in the subsequent execution. Figure 7 shows a typical example. After `xc` is freed, the subsequent dereference of `xc->xc_parent` would lead to UaF. Actually, this pattern is more common during list manipulations, e.g., the node deletion in a linked list [29].

```
+ fstWriterContext* const parent = xc->xc_parent;
  free(xc);
- xc->xc_parent->in_pthread = 0;
- pthread_mutex_unlock(&(xc->xc_parent->mutex));
+ parent->in_pthread = 0;
+ pthread_mutex_unlock(&(parent->mutex));
```

Figure 7: A UaF bug of the save-before-free pattern[30]

The fix to such bugs is usually straightforward. A new variable is used to hold the data needed in the subsequent executions, e.g., `parent=xc->xc_parent` in Figure 7.

Implications. Motivated by the fact that the UaF bugs of this "immediate-use-after-free" pattern often do not have complicated code logic and their fixes are usually straightforward, a static checker could be built to accurately detect such bugs with few false positives.

4.1.2 Raise a red flag. A flag variable is often used in software to track the runtime status. If falsely updated or inappropriately used, it could lead to memory corruptions including UaF. We found that 19 out of 150 UaF bugs are due to the inappropriate use of flags, and they fall into two specific flag related operations.

The flag is not or falsely updated (P3). Figure 8 presents an example of this flag related operation. `pci_alloc_consistent()` returns a `NULL` pointer if there is insufficient memory and its return value is checked. If the return value is `NULL`, the control flow will jump to label `create_cq_error`, free the memory and return the variable `ret`. Also, `beiscsi_cmd_cq_create()` returns `0` to `ret` if the execution succeeds. Unfortunately, the flag variable `ret` is not correctly updated in the loop, and this could lead to UaF. For example, when all statements in the first few rounds of the loop succeed and thus `ret` has value `0`. After a few rounds, `pci_alloc_consistent()` fails and thus the control flow jumps to `create_cq_error`. Note that the value of `ret` is still `0` now and the caller of `beiscsi_create_cqs()` would consider the function succeeds to execute. However, the memory has actually been freed by `pci_free_consistent()`. A UaF bug is triggered when the freed memory is accessed in the caller of `beiscsi_create_cqs()`. Similar cases contribute to 12 out of 19 flag UaFs.

The flag is not checked (P4). A flag variable often tracks the status of an object at runtime. When the object is accessed, the program should carefully check the flag value. Otherwise, it could lead to UaF if the object is not initialized or has been freed. For instance, the code in Figure 9 does not check if the audio stream has been initialized. If not, further operations on `mAudioStream` would trigger UaF.

Implications. Previous work has proposed various methods to detect the bugs caused by the missing checks of certain variables (e.g., flags and permissions) in source code [82, 106, 114]. Our empirical study finds that the root causes of flag related UaF are diverse and existing methods cannot cover all of them. This motivates further improvement of these work to provide better detection for UaF bugs. Meanwhile, fuzzing, one of the most popular bug detection approaches, might need to adopt special measures so that the flag related UaF bugs are more likely to get triggered.

```
static int beiscsi_create_cqs(...) {
   int ret = -ENOMEM;
   for (i = 0; i < phba->num_cpus; i++) {
        cq_vaddress = pci_alloc_consistent(...);
        if (!cq_vaddress) {
+           ret = -ENOMEM; // patch to fix
            goto create_cq_error;
        }
   }
create_cq_error:
    for (i = 0; i < phba->num_cpus; i++) {
        if (mem->va)
            pci_free_consistent(...); // free
    }
    return ret;
}
```

Figure 8: UaF due to a falsely updated flag in `beiscsi_create_cqs()` defined in `drivers/scsi/be2iscsi/be_main.c` [7]

```
void AudioCallbackDriver::StateCallback(cubeb_state aState)
     {
-  if (aState == CUBEB_STATE_ERROR) {
+  if (aState == CUBEB_STATE_ERROR && mAudioStream){
    }
}
```

Figure 9: Missing check of a flag variable leads to UaF [10]

4.1.3 Memory resize considered harmful. **Memory resizing is another common root cause of UaF (P5).** Specifically, 9 out of 150 UaF bugs are due to memory resizing, such as reallocation, manipulating resizable objects, and inappropriate memory change. There are a set of functions that can alter the size of memory blocks at runtime, e.g., `realloc()` and `resize()`. However, developers need to bear in mind multiple caveats when using these functions.

Figure 10 shows a typical example. `realloc(p, newsize)` returns a value to the pointer pnew. A non-NULL pnew indicates that a new chunk of memory has been successfully allocated. However, when pnew is NULL, it does not necessarily mean that the function has errors like insufficient memory assignment. Actually, it can also indicate that the memory is successfully freed when `new_size=0` [2]. The freed memory may be accessed later, resulting in UaF, when the special semantics of `realloc(new_size=0)` is ignored.

Even when the `new_size` has a reasonable non-zero integer value, `realloc()` can exhibit three behaviors from the perspective of function callers: (1) simply expand the original memory block "in place" [3]; (2) the new memory is failed to allocate and the old memory is returned; or (3) allocate a new memory block, copy data, and free the original one. Actually, `realloc()` essentially implies

[2]More accurately, according to C18 standard (§7.22.3.5) [39], "If size is zero and memory for the new object is not allocated, it is implementation-defined whether the old object is deallocated."

[3]C18 standard (§7.22.3.5) [39] only defines the behavior of deallocating the old object and returning a pointer to a new object, and has no notion of "in place" expansion. However, it is possible that the memory block is freed and then allocated at the same location. Therefore, to external observers, the behavior looks like "in place" expansion.

```
void *pnew = realloc(p, new_size);
if (pnew) {
    sFreetypeMemoryUsed +=
        FreetypeMallocSizeOfOnAlloc(pnew);
-  } else {
+  } else if (size != 0) {
    sFreetypeMemoryUsed += FreetypeMallocSizeOfOnAlloc(p);
}
```

Figure 10: UaF due to realloc() with a special size parameter 0 in `/gfx/thebes/gfxAndroidPlatform.cpp` [8]

```
static PyObject *
array_fromstring(arrayobject *self, PyObject *args){
    char *str;
    Py_ssize_t n;
    int itemsize = self->ob_descr->itemsize;
    if (!PyArg_ParseTuple(args, "s#:fromstring", &str, &n))
        //str is parsed from args. When an array is passed
        to itself (i.e., args=self), we get str ==
        self->ob_item.
       return NULL;
+   if (str == self->ob_item) {
+       PyErr_SetString(PyExc_ValueError, ...);
+       return NULL;
+   }
    char *item = self->ob_item;
    // When str == self->ob_item, item == str.
    PyMem_RESIZE(item, char, (Py_SIZE(self) + n) *
        itemsize);
    // realloc() is invoked in PyMem_RESIZE(), frees the
        original memory pointed by item and allocates a
        new memory block which may or may not be the same
        as the original one. Therefore, if item == str,
        str may point to freed memory after PyMem_RESIZE().
    memcpy(item + (Py_SIZE(self) - n) * itemsize, str,
        itemsize*n);
    //If str is dangling, a UaF occurs here.
}
```

Figure 11: UaF when a function calls realloc [4]

a memory free [39] and can potentially cause dangling pointers. Figure 11 shows such an example. In Python, `array.fromstring()` appends items from the string, interpreting the string as an array of machine values [40]. This method expects the input string to be a different object from the array. However, the original buggy implementation failed to handle the case when the array itself is passed to `fromstring()`. In such a case, two pointers, `self->ob_item` and the pointer to the argument, point to the same memory block. Later in the method, `PyMem_RESIZE()` - which calls `realloc()` - is invoked to resize the memory pointed by `self->ob_item`. As described above, this operation can trigger the deallocation of the original memory, and thus the pointer to the argument becomes dangling. To fix this UaF bug, Python disallows the array to append to itself.

Moreover, many functions in practice implicitly assume that the input parameters are not resizable objects. For instance, the function `PySlice_GetIndicesEx()` in Python (`Objects/sliceobject.c`)

```
- auto& subscriber = subscriber_it->second;
+ auto subscriber = subscriber_it->second;
  ef.event_subs_.erase(subscriber_it);
  subscriber->tearDown();
  subscriber->state(EventState::EVENT_NONE);
```

Figure 12: auto and auto&[42]

expects to operate on a non-resizable sequence. Otherwise, the sequence can be resized, e.g., calling `PyNumber_AsSsize_t()`, after passing its length to `PySlice_GetIndicesEx()`. This can lead to returning indices out of the length of the sequence, and thus accessing that part of the sequence will trigger UaF [11]. To avoid such UaF, the parameters to such functions should be carefully checked.

Finally, memory resize has to be carefully handled in concurrent systems. Specifically, a memory block could get resized at any time during the execution of racy code, e.g., the resizable buffer in `SNDRV_RAWMIDI_IOCTL_PARAMS` ioctl [13]. Same as other concurrency bugs, synchronization such as locking is required to protect accesses to resizable memory.

Implications. Obviously, memory resize operations deserve special attention to detect and avoid UaF bugs. Specifically, a static checker can be built to detect the potential misuses of memory resize operations. In addition, static bug detection systems need to take the fact that parameters could be potentially resizable into consideration.

4.1.4 Misuse of APIs and features. Another common UaF root cause is **the misuse of specific APIs and features (P6)**. Developers often misunderstand the semantics of these APIs and features. The `realloc()` and `resize()` described above fall into this category. Our study also finds multiple other APIs and features whose misuses have resulted in UaF in practice.

First, the misuse of `list.erase(it)` and `list.remove(*it)` has caused multiple UaF bugs. In C++, `list.erase(it)` [31] only deletes one object from the list, while `list.remove(*it)` [32] can delete multiple objects with the same value as `*it` from the list. The misuse of `list.remove()` - where `list.erase()` should be used - could lead to UaF because more objects might get deleted unexpectedly [33]. A similar UaF bug occurred in [41], where the developers misuse two user-defined functions and free the whole chain of `rte_mbuf`, instead of one segment. `list.erase(it)` and `list.remove(*it)` are frequently misused because developers often get confused about their semantics. In particular, some applications have overridden the two methods using custom implementations with different semantics. For instance, `list.remove(*it)` in Python only removes the first matching element from the list.

Another case is the reference operator `&` in C++, which denotes an alias for an object. Developers sometimes get confused about a reference and a pointer. However, a reference behaves quite differently from a pointer, and the misuse of a reference could lead to UaF bugs. Figure 12 shows an example of the UaF bug caused by the misuse of references. `auto& subscriber` is a reference to `subscriber_it->second`. Once `subscriber_it` is freed, `auto& subscriber` points to a freed object and thus causes UaF. Finally, the misuses of some APIs in various reference counting implementations could also cause UaF. We will describe these cases in §4.2.

Implications. As an immediate improvement for UaF detection, we can model the semantics for all of the APIs described above and build a static checker to detect the UaF bugs that are caused by the misuses of these APIs. Furthermore, we can explore to extend to more sensitive APIs. However, there are two main challenges to address. First, it requires a method that can systematically identify all potentially sensitive APIs. Second, the semantics of these APIs need to be modeled automatically. Actually, there has already been some work on sanitizing API usages [112]. Further research is needed to evaluate if and how such methods can be extended to detect UaF bugs caused by the API misuses.

4.1.5 Double free. **Double free (P7)** is a special type of UaF. It occurs when a memory block is freed more than once. Our study finds 20 double free UaF. For instance, the bug in [15] forgets to set the pointer `sock->sk` to `NULL` after the memory gets freed. As a result, the same memory could get repeatedly freed, resulting in double free. Moreover, we find that, multiple double free UaF happens because `free()` is put inside a loop.

Implications. Loop is one major location for the occurrence of double free. Static analysis should focus on the memory tracking inside the body of the loop to detect double free. Basically, to avoid double free, a memory block should not have been freed when calling `free()`. For example, memory allocators can provide a new API `safe_free()`, which checks the memory status before calling `free()`. This helps to mitigate all nine double free bugs in our study. Therefore, it deserves further research on how to achieve efficient checks in `safe_free()` without noticeably degrading the performance of memory allocators.

4.1.6 Miscellaneous cases. Finally, there are 35 bugs in our dataset that are mainly caused by the faulty code logic. Such logic errors are usually application-specific, and thus no meaningful patterns could be derived. For example, in plan9port [34], `thread0` is a special thread that is used for the context switches back to the scheduler, and it cannot be freed until the end of the scheduler function. If `thread0` gets freed unexpectedly, the UaF bug would be triggered during context switches. Another similar case occurs in Linux Kernel, where the unexpected free of a special memory block `block#0` could cause UaF [14]. Due to space limit, we will not enumerate all cases here.

4.2 Reference Counting, A Panacea for UaF?

In order to achieve memory safety, it is a common idea to use reference counters to manage the lifetime of objects. The core idea is to maintain a counter per object, which gets incremented whenever a new reference to the object is taken and decremented when a reference is released. An object can only be released when the counter reaches zero. Many software suites such as Linux kernel and Python have adopted reference counting. In the best case, reference counters can eliminate UaF bugs because no dangling pointer could be generated. Unfortunately, we find that a bunch of UaF bugs - 38 out of 150 - are caused by the implementation mistakes or misuse of reference counters. In particular, around 66% of UaF bugs in Python are caused by the errors related to reference counters. We refer to these bugs as *refcount UaF* and we present their common patterns in this section.

It is risky to use something borrowed (P8). Reference counters in most implementations such as Python [35] and Linux kernel [80] require explicit manipulations, e.g., using the API pair `Py_INCREF()` / `Py_DECREF()` in Python. A safe practice is to always increment the reference counter of an object whenever getting a reference to the object and and decrement after the reference is released. But it is actually not necessary for every reference, especially when we are sure that the reference counter must be larger than zero during the whole lifetime of the reference. For instance, the reference counting implementation in Python guarantees to increment the reference counter for every object passed as a function argument at the entry point of the called function. In Python, the references that do not increment the reference counter of the referred object (*referent*) are called *borrowed references* [36]. In our study, we extend this definition to all reference counting implementations in C/C++ [4].

It is error-prone to use borrowed references in practice. Our study reveals that the majority of refcount UaF bugs - 29 out of 39 - are due to the misuse of borrowed references. For example, in Figure 14, `PyList_GET_ITEM(slotnames, i)` returns a borrowed reference `name` to the object at index `i` in list `slotnames`. However, `name` cannot prevent the referent from being freed. If the referent gets freed before or during the execution of `PyObject_GetAttr(obj, name)`, `name` becomes a dangling reference and the dereferencing operations will trigger UaF. Actually, it is a common pitfall to grab a borrowed reference to an object in `lists`, `arrays` or `tuples` and hold on to it of a while without incrementing the reference counter. In addition, developers sometimes are aware that a borrowed reference is unsafe to operate and the reference counter should be incremented to avoid UaF, but they fail to do it properly. Figure 13 shows a typical example. `kvm_ioctl_create_device()` creates and initializes a device `dev` that holds a borrowed reference `kvm` to a virtual machine (VM) object. Then, `anon_inode_getfd()` transfers the ownership of the reference `dev` to the file descriptor table of the caller of `kvm_ioctl_create_device()`. After the ownership transfer, developers call `kvm_get_kvm()` to increment the reference counter. Unfortunately, the borrowed reference is still misused here and the reference counter should be incremented before `anon_inode_getfd()`. Specifically, an attacker can close the file descriptor, which triggers the code logic that automatically drops the borrowed reference `kvm`. This can cause the reference counter of the VM object referred by `kvm` to drop to zero and the VM object will be freed prematurely.

Reference counters get-over decremented (P9). Another common error that developers make is that the API to decrement reference counters could be unexpectedly and redundantly invoked, causing reference counters to reach zero prematurely. In case there are still in-use references to the freed objects, UaF occurs. For instance, the function `sctp_assoc_update()` in Figure 15 calls `sctp_auth_asoc_init_active_key()` to generate a new key and delete the old one. For key deletion, the function `sctp_auth_key_put()` is called to decrement the reference counter of the old key by one. When the reference counter reaches zero, the memory associated with the key gets freed. In this buggy case, developers mistakenly call `sctp_auth_key_put()` twice, one in `sctp_assoc_update()` and the other in `sctp_auth_asoc_init_active_key()`.

[4]Note that the borrowed reference in Rust has different semantics [37].

```
static int kvm_ioctl_create_device(...){
      dev->kvm = kvm;
      ops->init(dev);
    +  kvm_get_kvm(kvm);
      ret = anon_inode_getfd(...);
      if (ret < 0) {
    +     kvm_put_kvm(kvm);
     }
    -  kvm_get_kvm(kvm);
 }
```

Figure 13: A refcount UaF occurs because a borrowed reference is not properly converted to a non-borrowed one [16]

```
Py_LOCAL(PyObject *) _PyObject_GetState(...) {
    // name is a borrowed reference, and the referent
        might get unexpectedly freed.
    name = PyList_GET_ITEM(slotnames, i);
+   Py_INCREF(name);
    value = PyObject_GetAttr(obj, name);
    // error: misused borrowed reference
+   Py_DECREF(name);
 }
```

Figure 14: UaF due to a misused borrowed reference [5]

```
void sctp_assoc_update(...){
-   sctp_auth_key_put(asoc->asoc_shared_key);
    sctp_auth_asoc_init_active_key(...);
}
int sctp_auth_asoc_init_active_key(...){
   sctp_auth_key_put(asoc->asoc_shared_key);
}
```

Figure 15: UaF due to reference counter over-decrement [6]

Reference counters are not decremented (P10). While over decrementing can cause UaF, missed decrements can sometimes cause UaF too. Reference counters are usually represented using integers, e.g., `atomic_t` in Linux kernel. When developers forget to decrement a reference counter, the reference counter could overflow and become zero. Therefore, the referent would be freed prematurely. For example, the buggy code in Figure 16 gets a reference `keyring` and increments the reference counter by calling `find_keyring_by_name()`. Suppose that the reference counter should be decremented at the exit of `join_session_keyring()`; but under certain condition (`keyring == new->session_keyring`), the code jumps to label `error2` and fails to decrement the reference counter. Attackers can then repetitively trigger this buggy code and cause the reference counter to overflow.

Reference counting APIs are misused (P11). Some implementations of reference counting provide rich APIs, which can give developers more flexibility to trade off some runtime checks for better performance. Unfortunately, some APIs could be misused and thus lead to UaF. Figure 17 shows an example in Linux kernel.

```
long join_session_keyring(const char *name){
   // Look for an existing keyring of this name.
   // If the keyring exists and the reference is returned,
        its reference counter is incremented by one.
   keyring = find_keyring_by_name(name, false);
   if (PTR_ERR(keyring) == -ENOKEY) {
   } else if (IS_ERR(keyring)) {
   } else if (keyring == new->session_keyring) {
+     key_put(keyring);
      ret = 0;
      goto error2;
   }
  key_put(keyring);
}
```

Figure 16: UaF due to the missing decrement for a reference counter [9]

```
   spin_lock_bh(&net->nsid_lock);
   peer = idr_find(&net->netns_ids, id);
   if (peer)
-     get_net(peer);
+     peer = maybe_get_net(peer);
   spin_unlock_bh(&net->nsid_lock);
```

Figure 17: UaF due to a misused reference counting API [12]

The API `maybe_get_net(struct net *net)` checks if the current reference counter for the struct `net` is zero. If so, it means that the struct has been freed and `NULL` is returned. Otherwise, a reference to `net` is returned and the reference counter is incremented. By contrast, the API `get_net(struct net *net)` simply increments the reference counter and returns a reference to `net`. Obviously, a misused `get_net()` - where `maybe_get_net()` should be used - will cause UaF as shown in [12]

Summary. Our study shows that refcount UaF bugs are quite prevalent. However, there is only limited research on detecting reference counting bugs and all of these methods have certain shortcomings. The refcount tracing and balancing techniques in Firefox enable developers to track down the leak of referents at runtime [38]. The detection capability of such techniques highly depends on the quality of inputs. Unfortunately, it is often quite challenging to generate inputs with high quality. Meanwhile, researchers have also proposed multiple static analysis methods to detect reference counting bugs in source code [61, 74, 79, 98]. These methods are mostly limited in their scopes. Specifically, they fail to handle the unprotected borrowed references and misuses of APIs. More effort is needed to improve the coverage of refcount UaF bugs.

4.3 UaF Benchmark

Nguyen et al. [23] built a fuzzing benchmark UAFbench comprising 14 real bugs from 11 applications. However, UAFbench has several limitations. First, UAFbench only considers C applications and no C++ UaF bug is included. Second, all of the included UaF bugs are sequential. Our empirical study shows that concurrent UaF bugs comprise a non-trivial portion and should not be overlooked. Third, there are limited patterns of the bugs and the buggy code is relatively simple in terms of the patch size. Motivated by the above limitations and our empirical findings, we construct a new benchmark `UAFBenchPlus` that contains 120 out of the 150 collected UaF bugs. `UAFBenchPlus` only includes bugs that support recent releases of Clang [46] which are commonly used to build bug detectors. The detailed comparison between UAFBench and `UAFBenchPlus` is shown in Table 7. It shows `UAFBenchPlus`'s specification in terms of the number of applications, lines of code, and bug patterns. It is by far the most comprehensive benchmark for UaF.

	UAFBenchPlus	UAFBench
# of Applications	41	11
Application Size†	90K∗ ∼ 2479M∗∗	9K ∼ 502M
Application Language	C, C++	C
# of Bugs	120	14
Patch Size ‡	1 ∼ 215	1 ∼ 13
# of Patterns	All	P2 ∼ P5, P7

Table 7: Comparison of `UAFBenchPlus` and UAFBench [86]. †Application size is measured in lines of code. ‡Patch size is measured in lines of code added and deleted. ∗K for Thousand. ∗∗M for Million.

Note that `UAFBenchPlus` only contains 120 out of the 150 collected bugs. We have to exclude 30 bugs because the bugs may involve different versions of code in a long time span and some bugs exist in the old versions of applications that cannot be successfully configured in current environments. In addition, some software only supports GCC but not Clang.

5 PALFREY: A PATTERN-BASED UAF DETECTOR

Our results in §4 demonstrate that a large portion of UaF bugs follow specific patterns, some of which can be translated into simple static checkers. For instance, as described in §4.1.3, when calling the function `realloc()`, developers should carefully handle the case where the parameter size equals 0, to avoid potential UaF. To measure the applicability of our findings, we built a pattern-based static UaF detector, `Palfrey`. Next, we discuss how `Palfrey` is implemented and how effective it is for discovering UaF bugs.

5.1 Design and Implementation of `Palfrey`

Overall, we extracted seven rules from our empirical study and developed 11 checkers in `Palfrey`. These rules are quite straightforward. For example, to handle unprotected borrowed references, the checker conducts analysis on a list of manually crafted functions that return borrowed references, and then report a bug if a borrowed reference returned by these functions is not protected by primitives such as `Py_INCREF`.

Our tool includes a framework for Abstract Syntax Tree (AST) and Control Flow Graph (CFG) analysis, a module for pointer alias analysis, and a module for pattern analysis. Our checkers are mainly based on the AST, call graphs, and CFG of programs. First, the AST is generated from the source code of a program, and then the call graphs and CFGs are constructed based on the AST. Next, after

the CFG, AST, and call graphs are constructed, a pattern matching engine is used to identify if UaF bugs exist. The inter-procedure analysis is performed to determine if a memory free operation happens, while the AST analysis is used to decide if specific functions are called. Subsequently, the data flow analysis is performed locally for the alias analysis, and the pattern matching engine is eventually used to report UaF.

The procedure to apply those rules are composed of the analysis on function call and return, the parameter analysis, and memory tracing. For example, to detect the use of unprotected weak reference in Figure 14, the AST analysis is first conducted to locate the functions of interest (in this case, `PyList_GET_ITEM(slotnames, i)`), and then the function return value `name` is further traced to see if it is protected before it is dereferenced.

`Palfrey` is built upon the `LLVM` (12.0.0) compiler infrastructure [47], which provides great support for both C and C++ code. Our checkers conduct path-insensitive analysis and they are mostly straightforward to implement. In particular, we leverage the data flow analysis provided by `LLVM` to detect the code snippets that match the bug patterns.

5.2 Evaluation of `Palfrey`

In order to demonstrate the effectiveness and efficacy of `Palfrey`, we first conducted evaluations on the 150 collected bugs. In particular, we compared `Palfrey` against four well-known static detectors, the open-source Infer version v1.1.0 [43], commercial PC-lint Plus [44], Clang [46], and GCC static analyzer [48]. All detectors support `C/C++` and claim to be able to find UaF bugs. Then, we applied `Palfrey` to the latest releases of a set of applications to evaluate its capability of finding new bugs. We ran the experiments on an x86-64 desktop with six-core Intel i7-9750 2.6G Hz processors and 16GB memory on Ubuntu 18.04.5. Our evaluation results are detailed below.

Effectiveness on 150 collected bugs. We successfully ran the five tools on only 75 bugs due to the system environment problems. For example, Infer relies on Clang to compile the target code, and it might fail if the target code cannot be compiled with Clang. Unfortunately, our dataset contains some bugs that cannot work with Clang. For instance, the Linux kernel cannot support Clang until 2016. Therefore, we only presented the results for the 75 successful cases.

Table 8 lists the results. All five tools successfully detected some of the bugs, especially those with simple patterns. However, `Palfrey` significantly outperforms Infer, PC-lint, Clang, and GCC static analyzer. A further investigation reveals that the superiority of `Palfrey` stems from the new bug patterns that the other four tools do not have. None of the other tools can reliably handle the UaF bugs caused by the misused borrowed references. While Infer can detect static initialization order fiasco, it fails to cover the static destruction order fiasco. Moreover, `Palfrey` is able to find some more complicated UaF bugs through the parameter analysis on certain functions.

We also evaluated the efficiency of `Palfrey` in terms of memory overhead and how long it takes to check a codebase. Due to space limit, we only present the results on five selected applications in Table 9 and the results on other applications are roughly similar. Overall, `Palfrey` consumes 23.6% ∼ 97.2% less memory than Infer, PC-lint, Clang, and GCC static checker. `Palfrey` also takes much less time to check a codebase, with 28.6% ∼ 98.3% less time than Infer, PC-lint, Clang, and GCC static checker. `Palfrey` is much more efficient to detect UaF in terms of memory consumption and time overhead because pattern analysis is much more efficient to locate possible bugs than general syntactic analysis.

	C	C++	Overall
Infer	14/53	2/22	16/75
PC-lint	10/53	3/22	13/75
GCC	7/53	2/22	9/75
Clang	16/53	5/22	21/75
`Palfrey`	34/53	6/22	40/75
`Palfrey` ∩ Infer	9/53	2/22	11/75
`Palfrey` ∩ PC-lint	7/53	2/22	9/75
`Palfrey` ∩ GCC	7/53	1/22	8/75
`Palfrey` ∩ Clang	12/53	2/22	14/75

Table 8: Detection results of `Palfrey`, Infer, and PC-lint on our collected UaF bugs. Infer and PC-lint failed to run on the other 75 bugs, due to various system environment and configuration problems.

New bugs. To evaluate `Palfrey`'s capability to find new bugs, we applied it to the latest releases of Python (v3.8.12), Linux Kernel (5.14), Firefox (90.0), and GitHub `C/C++` projects with more than 1K stars as of 2021-10-01. `Palfrey` successfully reveals nine bugs in seven applications, three of which we reported and received confirmations from developers (details in Table 11 Appendix B). We were able to test six applications using Infer, seven using PC-lint, nine using Clang and GCC static analyzer. Infer can find four out of the nine bugs, PC-lint can find three, GCC static analyzer can find three, and Clang can find six.

False positives and false negatives. There are eleven false positives by `Palfrey` in all of the tested applications. We found that the false positives are mainly caused by inaccurate pointer alias analysis. Compared to many other static bug detectors, `Palfrey` generates much fewer false positives. This is because our checkers target specific patterns and enforce strict constraints, which can reduce false positives.

Table 10 shows detection accuracy of `Palfrey` in various bug patterns. A single checker is specifically designed to detect unprotected borrowed references in CPython written in C, while two checkers are created for each of the other five bug patterns to analyze C and C++ programs. False negatives occur due to three reasons: (1) pointer alias analysis is inaccurate and the checker cannot support pointer arithmetic analysis, for example, the code in Figure 4; (2) the borrowed reference checker only applies to CPython, but some cases in Linux require deeper analysis to locate possible borrowed references; (3) our double free checker is able to detect repeated free in a loop but unable to detect all general cases.

Overall, our evaluation demonstrates that `Palfrey` is effective and efficient. In particular, leveraging the bug patterns identified in our study, `Palfrey` significantly outperforms existing static UaF

	Palfrey		Infer		PC-lint		GCC		Clang	
	Time†	Mem‡	Time	Mem	Time	Mem	Time	Mem	Time	Mem
swig	12	0.17	130	1.85	150	1.96	44	0.67	22	0.33
Python	68	0.44	570	4.03	482	3.95	211	2.02	101	0.79
Linux	1,682	1.2	4,867	3.12	5,016	3.91	2583	3.4	2699	2.4
u-boot	440	0.16	2,170	4.65	2,448	6.03	730	1.53	573	0.32
z-3	40	0.26	2,059	3.76	1,830	3.12	140	1.14	54	0.34

Table 9: Comparison of the time and memory overhead of three tools. † Time is measured in seconds. ‡ Mem is the memory overhead measured in gigabytes (GB).

Detected bug patterns	Accuracy
Immediate-use-after-free	20/25
Unprotected borrowed references	20/26
Unsafe memory resize	7/7
Double free	4/20
Parameter analysis for sensitive functions	6/9
Functions return illegal stack memory	0/0

Table 10: Detection accuracy of Palfrey in various bug patterns

detectors with much less overhead. Although Palfrey is still preliminary, we envision the promising results can motivate and inspire more future work.

6 RELATED WORK

There is a large body of previous empirical studies on software bug characteristics. To the best of our knowledge, this work is the first large-scale analysis to understand the characteristics of UaF bugs. The manual analysis allowed us to categorize the root causes of real-world UaF bugs into 11 patterns, based on which we implement a static UaF bug detector. Next, we will discuss related work from two perspectives.

Empirical study on bug characteristics. Several previous studies have focused on specific types of bugs other than UaF. Lu et al. [77] studied 105 concurrency bugs from four mature open source applications. They found that 97% of the studied concurrency bugs belong to two simple bug patterns. Jin et al. [70] conducted a comprehensive study of 109 real-world performance bugs from five software suites. Their findings led to the creation of 25 efficiency check rules that uncovered 332 previously unknown performance problems. Our work instead focused on the patterns and potential detection opportunities of UaF bugs. There are also a bunch of studies [63, 89, 97, 111] on generic bugs and failures in various applications. By contrast, our study found that UaF bugs have more diverse unique patterns. Different from the aforementioned work that focused on the buggy and vulnerable code itself, Holzinger et al. [68] studied the security vulnerabilities in Java platforms from the perspective of the exploits collected in the wild. The collection and analysis of UaF exploits in the wild will be our future work. Lastly, another line of empirical study that is marginally related to ours is the fault tolerance and recovery of software in face of triggered errors and bugs [56, 104]. In summary, while there have been a bunch of existing empirical studies on software bugs, none of them have focused on UaF.

Static bug detection. Static bug detectors have been popular among software developers. GUEB [64] is a static tool for detecting UaF vulnerabilities in binary code. It tracks heap operations and address transfers to identify program locations that allocate or free heap memory. By analyzing the resulting graph, GUEB identifies UAF vulnerabilities, offering better coverage than dynamic tools. However, GUEB's points-to analysis is imprecise, affecting its soundness and precision. Additionally, the path analysis and value set analysis in GUEB are resource-intensive and not easily scalable to large programs. DCUAF [49] applied the local-global analysis to detect concurrency UaF bugs in Linux device drivers. The tool has successfully confirmed 95 real cases that are previously unknown. Palfrey is instead built upon a set of patterns learnt from real bugs. In particular, Palfrey demonstrates that many UaF bugs can be detected without using sophisticated techniques. Also, our study provides guidelines for future work to build better static UaF detectors. CID [98] uses two dimension checking to detect INC-DEC inconsistency and DEC-DEC inconsistency, but it is unable to detect borrowed reference bugs since the cause of such bugs is due to the missing ownership not reference count inconsistency. There are many other detectors for various specific types of bugs and applications, e.g., API misuse [112], missing checks [106], buffer overflows [71], semantic bugs [82], violations of inferred programmer beliefs [63], configuration errors [105], and bugs in JavaScript bindings [53]. However, none of them target UaF bugs.

7 CONCLUSION

This paper provides a comprehensive study of real-world UaF bugs, examining their patterns, manifestations, fixes, and other characteristics. Our study is based on 150 UaF bugs, with 115 deterministic and 35 non-deterministic ones, collected from a diverse set of applications such as Linux Kernel, CPython, etc. Our study reveals a number of interesting findings that can provide valuable guidelines for UaF detection and protection. For instance, a main finding is that the root causes of UaF bugs can generally be categorised into 11 patterns. To demonstrate the applicability of these findings, we build a simple static UaF detector, Palfrey. Our evaluation shows that Palfrey outperforms existing tools in terms of both effectiveness and efficiency. In particular, Palfrey successfully detects nine new UaF bugs. Finally, to foster future research, we will make our dataset and tools publicly available.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the U.S. Office of Navy Research (ONR) grant N00014-23-1-2158. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] 2001. https://bugzilla.mozilla.org/show_bug.cgi?id=133773. (2001).
[2] 2015. https://bugs.python.org/issue24099. (2015). Accessed: 2020-11-01.
[3] 2015. https://bugs.python.org/issue24101. (2015). Accessed: 2020-11-01.
[4] 2015. https://bugs.python.org/issue24613. (2015).
[5] 2015. https://bugs.python.org/issue24097. (2015).
[6] 2015. https://bugzilla.redhat.com/show_bug.cgi?id=1196581. (2015).
[7] 2016. https://bugzilla.kernel.org/show_bug.cgi?id=188941. (2016).
[8] 2016. https://bugs.python.org/issue27867. (2016).
[9] 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2016-0728. (2016). Accessed: 2020-07-12.
[10] 2017. https://hg.mozilla.org/releases/mozilla-beta/rev/fb00d84ec825. (2017).
[11] 2017. https://github.com/python/cpython/commit/4d3f084c035ad3dfd9f8479886c41b1b1823ace2. (2017). Accessed: 2020-07-12.
[12] 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15129. (2017).
[13] 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=39675f7a7c7e7702f7d5341f1e0d01db746543a0. (2018). Accessed: 2020-07-12.
[14] 2018. (2018).
[15] 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9060cb719e61b685ec0102574e10337fa5f445ea. (2019).
[16] 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6974. (2019).
[17] 2020. Available online: https://google.github.io/oss-fuzz/. (2020). Accessed: 2020-7-19.
[18] 2020. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. (2020). Accessed: 2020-07-12.
[19] 2020. https://devguide.python.org/garbage_collector/. (2020). Accessed: 2020-07-12.
[20] 2020. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html. (2020). Accessed: 2020-07-12.
[21] 2020. https://en.cppreference.com/book/intro/smart_pointers. (2020). Accessed: 2020-07-12.
[22] 2020. https://doc.rust-lang.org/1.30.0/book/2018-edition/ch04-00-understanding-ownership.html. (2020). Accessed: 2020-07-12.
[23] 2020. https://github.com/strongcourage/uafbench. (2020). Accessed: 2022-01-12.
[24] 2020. https://github.com/. (2020). Accessed: 2020-11-01.
[25] 2020. https://bugs.python.org/issue40294. (2020). Accessed: 2020-11-01.
[26] 2020. https://github.com/microsoft/LightGBM/pull/2743. (2020). Accessed: 2020-11-01.
[27] 2020. https://github.com/ndilieto/uacme/commit/de76f1926f405a6d884dc0f5bb5001ee34a1e5ad. (2020).
[28] 2020. https://github.com/coturn/coturn/issues/601. (2020).
[29] 2020. https://github.com/scala-native/scala-native/pull/2072. (2020).
[30] 2020. https://github.com/verilator/verilator/commit/f98782c061e1f1718677090e4adc1c7576377b68. (2020).
[31] 2020. https://www.cplusplus.com/reference/list/list/erase/. (2020). Accessed: 2020-07-12.
[32] 2020. https://www.cplusplus.com/reference/list/list/remove/. (2020). Accessed: 2020-07-12.
[33] 2020. https://github.com/DCMTK/dcmtk/commit/7c3ca88c6197af3d03b8376aeb46e6cc7d7c3724. (2020).
[34] 2020. https://github.com/9fans/plan9port/commit/2991442aef1cf020ffde43673433ee97ef322a53. (2020).
[35] 2020. https://docs.python.org/3/c-api/intro.html. (2020). Accessed: 2020-07-12.
[36] 2020. https://pythonextensionpatterns.readthedocs.io/en/latest/refcount.html. (2020). Accessed: 2020-07-12.
[37] 2020. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html. (2020). Accessed: 2020-07-12.
[38] 2020. Refcount Tracing and Balancing. https://firefox-source-docs.mozilla.org/performance/memory/refcount_tracing_and_balancing.html. (2020).
[39] 2021. https://www.iso.org/standard/74528.html. (2021). Accessed: 2021-09-09.
[40] 2021. https://docs.python.org/2/library/array.html#array.array.fromstring. (2021). Accessed: 2021-09-09.
[41] 2021. https://github.com/F-Stack/f-stack/pull/565. (2021).
[42] 2021. https://github.com/osquery/osquery/pull/6880/files. (2021).
[43] 2021. https://fbinfer.com/. (2021). Accessed: 2021-01-12.
[44] 2021. https://www.gimpel.com/. (2021). Accessed: 2021-01-12.
[45] 2022. https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html. (2022). Accessed: 2022-01-17.
[46] 2023. https://clang.llvm.org/. (2023).
[47] 2023. https://llvm.org/. (2023).
[48] 2023. https://gcc.gnu.org/. (2023).
[49] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. 255–268.
[50] H. Boehm, A. Demers, and M. Weiser. 2002. A garbage collector for c and c++. https://www.hboehm.info/gc/.
[51] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 2329–2344.
[52] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.
[53] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (S&P '17)*. 559–578.
[54] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 133–143.
[55] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy (S&P '15)*. 725–741.
[56] Subhachandra Chandra and Peter M. Chen. 2000. Whither Generic Recovery from Application Faults? A Fault Study Using Open-Source Software *(DSN '00)*.
[57] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International Conference on Software Engineering and Formal Methods*. Springer, 233–247.
[58] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *26th USENIX Security Symposium (USENIX Security '17)*. 815–832.
[59] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. 2015. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. 166–177.
[60] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN '06)*. 269–280.
[61] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. 2009. Verifying Reference Counting Implementations. In *Tools and Algorithms for the Construction and Analysis of Systems*, Stefan Kowalewski and Anna Philippou (Eds.).
[62] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 237–252.
[63] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. 73–88.
[64] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques* 10, 3 (2014), 211–217.
[65] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *2008 Network and Distributed System Security Symposium (NDSS '08)*.
[66] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *22nd USENIX Security Symposium (USENIX Security '13)*. 49–64.
[67] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. 2018. Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing. In *2018 Network and Distributed System Security Symposium (NDSS '18)*.
[68] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. 2016. An In-Depth Study of More Than Ten Years of Java Exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 779–790.
[69] Shin Hong and Moonzoo Kim. 2013. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software* 86, 2 (2013), 377–388.
[70] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 77–88.
[71] D. Larochelle and David Evans. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *10th USENIX Security Symposium (USENIX Security '01)*. 177–190.

[72] 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 *2015 Network and Distributed System Security Symposium (NDSS '15)*.
[73] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 517–530.
[74] Siliang Li and Gang Tan. 2014. Finding reference-counting errors in Python/C programs with affine analysis. In *European Conference on Object-Oriented Programming*. Springer, 80–104.
[75] Linux Kernel Organization, Inc. 2020. https://bugzilla.kernel.org/. (2020).
[76] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A Robust and Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 1635–1648.
[77] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*. 329–339.
[78] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Archipelago: Trading Address Space for Reliability and Security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*. 115–124.
[79] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. 2016. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 531–544.
[80] Paul E. McKenney. 2007. Overview of Linux-Kernel Reference Counting.
[81] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44.
[82] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 361–377.
[83] Mozilla Foundation. 2020. https://bugzilla.mozilla.org. (2020).
[84] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA '12)*. 189–200.
[85] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan Notices* 42, 6 (2007), 89–100.
[86] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*. 47–62.
[87] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2007. Exterminator: Automatically Correcting Memory Errors with High Probability. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 1–11.
[88] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. 2014. Coccinelle: tool support for automated CERT C secure coding standard certification. *Science of Computer Programming* 91 (2014), 141–160.
[89] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. 305–318.
[90] Python Software Foundation. 2020. https://bugs.python.org/. (2020).
[91] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *2017 Network and Distributed System Security Symposium (NDSS '17)*, Vol. 17. 1–14.
[92] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. https://github.com/google/oss-fuzz.
[93] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*. 309–318.
[94] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Y. Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Network and Distributed System Security Symposium (NDSS '19)*.
[95] C. Spatz. 1981. Basic Statistics: Tales of Distributions.
[96] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *2016 Network and Distributed System Security Symposium (NDSS '16)*, Vol. 16. 1–16.
[97] Gang Tan and Jason Croft. 2008. An Empirical Security Study of the Native Code in the JDK. In *Proceedings of the 17th Conference on Security Symposium (USENIX Security '08)*. 365–378.
[98] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. 2021. Detecting Kernel Refcount Bugs with Two-Dimensional Consistency Checking. In *30th USENIX Security Symposium (USENIX Security '21)*. 2471–2488.
[99] The MITRE Corporation. 2019. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html. (June 2019). Accessed: 2019-7-1.
[100] The MITRE Corporation. 2019. Search CVE List. https://cve.mitre.org/cve/search_cve_list.html. (Jan. 2019). Accessed: 2019-7-24.
[101] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-Free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. 405–419.
[102] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. 391–402.
[103] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '07)*. 205–214.
[104] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. 2003. Characterization of linux kernel behavior under errors. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings. (DSN '03)*.
[105] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 619–634.
[106] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. 499–510.
[107] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*. 42–54.
[108] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE '18)*. 327–337.
[109] Jiayi Ye, Chao Zhang, and Xinhui Han. 2014. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. 1529–1531.
[110] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers.. In *2015 Network and Distributed System Security Symposium (NDSS '15)*.
[111] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. 249–265.
[112] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. 2016. APISAN: Sanitizing API Usages through Semantic Cross-Checking. In *25th USENIX Security Symposium (USENIX Security '16)*. 363–378.
[113] Michal Zalewski. 2017. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. (Nov. 2017). Accessed: 2019-7-31.
[114] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel. In *28th USENIX Security Symposium (USENIX Security '19)*. 1205–1220.

A DETAILS OF 30 REPRODUCED BUGS

Table 12 shows the details of the 30 reproduced UaF bugs.

B NEW BUGS DETECTED BY PALFREY

Table 11 lists the nine new bugs detected by `Palfrey`.

Application	Buggy function	Pattern	Unique? †	Confirmation
u-boot	membuff_uninit [*li/membuff.c:389*]	P7	Yes	Reported
u-boot	setjmp [*lib/efi_loader efi_boottime.c:2930*]	P2	No	Reported
Z3	ziplistPush [*src/quicklist.c:511*]	P2	No	Reported
dcmtk	base_clear [*ofstd/include/dcmtk/ofstd/ofmem.h:261*]	P2	No	Reported
dcmtk	insertSequenceItem [*dcmdata/libsrc/dcitem.cc:4203*]	P2	No	Reported
dcmtk	destroy [*ofstd/include/dcmtk/ofstd/ofmem.h:261*]	P1	Yes	Reported
osquery	deregisterEventSubscriber [*osquery/events/eventfactory.cpp:190*]	P2	No	Confirmed
dandere2x	get_diamond_search_points [*block_plugins/block_matching/DiamondSearch.cpp:97*]	P2	Yes	Confirmed
yosys	module->remove [*src/database.c2022*]	P2	No	Confirmed

Table 11: Summary of new UaF bugs detected by `Palfrey`. †This column indicates if the bug is uniquely detected by `Palfrey`.

Bug ID	Application	Crash	Pattern
issue24613	CPython	Yes	P5
issue18328	CPython	No	P1
issue25388	CPython	No	P4
issue24100	CPython	No	P8
issue24552	CPython	Yes	P1
issue29028	CPython	No	P5
issue29438	CPython	No	P2
issue24096	CPython	No	P8
issue28275	CPython	No	P4
Bug 1338772	Firefox	No	P1
Bug 750820	Firefox	No	P3
Bug 1388243	Firefox	No	P4
Bug 803853	Firefox	No	P7
Bug 634961	Firefox	No	P4
Bug 1033006	Firefox	No	P2
Bug 1417405	Firefox	No	P8
Bug 999274	Firefox	No	P7
Bug 557174	Firefox	No	P7
Bug 1338772	Firefox	No	P1
CVE-2019-8912	Linux	No	P3
Bug 199403	Linux	No	P4
Bug 200001	Linux	Yes	P4
Bug 199839	Linux	No	P4
#7614	frr	No	P2
#6469	Apache	No	P1
#601	coturn	No	P4
CVE-2018-10685	lrzip	No	P5
CVE-2018-11416	jpegoptim	No	P2
CVE-2018-20623	GNU Binutils	No	P5
#2072	scala-native	No	P2

Table 12: List of the reproduced 30 bugs.