FreeWill: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries

Liang He[1*] *Hong Hu*[2*] *Purui Su*[1,3,4✉] *Yan Cai*[3] *Zhenkai Liang*[5]

[1]*TCA /* [3]*SKLCS, Institute of Software, Chinese Academy of Sciences*
[2]*Pennsylvania State University*
[4]*School of Cyber Security, University of Chinese Academy of Sciences*
[5]*National University of Singapore*

Abstract

Memory-safety issues in operating systems and popular applications are still top security threats. As one widely exploited vulnerability, Use After Free (UAF) resulted in hundreds of new incidents every year. Existing bug diagnosis techniques report the locations that allocate or deallocate the vulnerable object, but cannot provide sufficient information for developers to reason about a bug or synthesize a correct patch.

In this work, we identified incorrect reference counting as one common root cause of UAF bugs: if the developer forgets to increase the counter for a newly created reference, the program may prematurely free the actively used object, rendering other references dangling pointers. We call this problem *reference miscounting*. By proposing an *omission-aware* counting model, we developed an automatic method, FreeWill, to diagnose UAF bugs. FreeWill first reproduces a UAF bug and collects related execution trace. Then, it identifies the UAF object and related references. Finally, FreeWill compares reference operations with our model to detect reference miscounting. We evaluated FreeWill on 76 real-world UAF bugs and it successfully confirmed reference miscounting as root causes for 48 bugs and dangling usage for 18 bugs. FreeWill also identified five null-pointer dereference bugs and failed to analyze five bugs. FreeWill can complete its analysis within 15 minutes on average, showing its practicality for diagnosing UAF bugs.

1 Introduction

Memory-unsafe languages, like C/C++, are widely used to implement complex software systems, such as operating systems and web browsers, but unfortunately, they have resulted in a large number of exploitable vulnerabilities [37]. As one of the most sophisticated bugs, Use After Free (UAF) bugs are still popular, with 200 to 300 instances founded every year in main stream applications [19]. Researchers have proposed various solutions to detect UAF bugs at different stages of program executions [29,34,38–41,44,46,50,55]. However, diagnosing a reported UAF bug, i.e., identifying the root cause to fix the bug, is still challenging and time-consuming.

Based on the operations involved in these bugs, there could be two reasons leading to UAF bugs. The first reason is that the program mistakenly *uses* a dangling pointer to access a correctly freed object, which we call *dangling usage*. Researchers have proposed several methods to track dangling pointers [33,34] and identify locations of dangling usages [41]. The second reason is that an object is incorrectly *freed*, making remaining references dangling and all future uses invalid. There is not much research for diagnosing incorrect frees, as it is natural to believe that free operations are invoked by developers intentionally and such problems can only be identified and fixed through manual code review.

However, after analyzing a large number of recent UAF vulnerabilities, we observe that UAF bugs caused by incorrect frees are common in real-world programs, especially in programs that rely on reference counting (referred to as *refcounting* hereinafter) to manage memory objects [26]. For each object, when a new reference is created, e.g., being assigned to a member of an object, its reference counter (referred to as *refcounter* hereinafter) should be incremented; when an existing reference is destroyed, e.g., freeing an object containing the reference, its refcounter should be decremented. If the program fails to increase the refcounter properly [7], the program may free objects in a *premature* way, rendering all references as dangling pointers and resulting in UAF bugs.

UAF bugs caused by incorrect refcounting can be abused by attackers to build powerful attacks. For example, a reference miscounting error that exists in Linux for more than 15 years (2006 to 2021) introduces a UAF bug during the booting process (`Commit-8fd0e99` in our evaluation) [10]. Even worse, these bugs are difficult for human analysts to diagnose, which usually leads to incomplete patches and insecure systems. For example, partial patches for MacOS kernel bug `CVE-2016-1828` and PHP script engine bug `BUG-68594` still allow attackers to compromise patched systems.

Problem. We use the term *reference miscounting* to represent

*The two lead authors contributed equally to this work.

the programming error where developers fail to increase the refcounter for a newly created reference. We summarize two types of reference miscountings that can cause UAF bugs. (1) *Mistaken omission*. Considering the high overhead of counting all references [42, 43], coordinated omission of unnecessary refcounting is encouraged to optimize performance [17]. However, if there is a wrong omission, the program will miss the increment of the refcounter and the corresponding object could be prematurely freed. (2) *Inconsistent decrease*. If there is only a decrease to a refcounter without any corresponding increase, the refcounter will prematurely become zero. For example, it is common for the callee function to release the references of its parameters, i.e, decrementing the refcounter. However, if the caller function forgets to increment the refcounter, there is an obvious reference miscounting.

There are two major challenges to the diagnosis of reference miscounting. The first one is the matching problem: it is difficult to match an increment with the corresponding reference creation, as there is no consistent rule mandating the increment before or after a reference creation. Besides, refcounter can be updated using any existing reference. It becomes worse if refcounting is encapsulated by nested non-inlined functions in binary programs. The second challenge is the existence of a large number of count-omission optimizations. For example, if the lifespans of two references `R0` and `R1` are overlapped, developers prefer to use only one increment on `R0` and one decrement on `R1` to keep the object alive. To diagnose reference miscounting, we need to correctly identify all reference lifetimes and calculate their relationships, which cannot be efficiently solved by existing mechanisms.

Approach. In this paper, we propose a trace-based method to efficiently diagnose UAF bugs caused by reference miscounting. First, to calculate the reference relationships, we propose an omission-aware refcounting model to describe the expected counting and omission behaviors. We propose four omission rules to help us to detect reference miscounting. Second, we develop practical methods to automatically identify the lifetimes of UAF objects, all related references and refcounting operations from the execution traces. Then, we use distance-based methods to match the refcounting operation with a proper reference. Finally, we propose an automatic method to detect mismatches between the actual refcounting operations and the expected behaviors. If we confirm there is no reference miscounting, we report all dangling pointers as the diagnosis results. Besides the bug diagnosis, our method can also help synthesize correct patches. Specifically, when we find a missing refcounter increment, our tool will suggest adding the increment back to fix the UAF bug.

Result. We implement a tool called FREEWILL based on dynamic instrumentation and offline trace-based analysis. By extending the hardware emulator *QEMU* [22], our tool can diagnose large-scale GUI-based programs without the source code. We evaluated FREEWILL on diagnosing 76 real-world UAF bugs, including 32 bugs from Internet Explorer (IE),

```
1. typedef struct{ int ref;} A; A* gpA;
2. typedef struct{ A* _pA;} B; B* gpB;
3. void clone(B** b1, B* b2){   /*IE: CVE-2010-0249 */
4.     memcpy(*b1, b2, N2);    /* Mistaken Omission */
5.}
6. void filename_lookup(A* lpA){/*linux:Commit-c3aabf0*/
7.     dec_ref(lpA);         /* Inconsistent Decrease */
8. }
9.void demo_code(){
10.    A* pA=(A*)malloc(N1);     /*R0*/
11.    pA->ref=1;
12.    ...
13.    gpA=pA;                   /*R1*/
14.    B* pB=(B*)malloc(N3);
15.    pB->_pA = pA;             /*R2*/
16.    gpB=(B*)malloc(N3);
17.    clone(&gpB, pB);          /*R3*/
18.    pB->_pA = NULL;
19.    free(pB);
20.    filename_lookup(pA);      /*R4*/   /* Free */
21.}
22.void main(){
23.    demo_code();
24.    dec_ref(gpA);                 /* UAF-1 */
25.    printf(gpB->_pA->ref);        /* UAF-2 */
26.    free(gpB);
27.}
```

Figure 1: **Motivating Example**.

Firefox and Chrome, 21 bugs from Linux and MacOS kernels, 23 bugs from Python and PHP script engines. FREEWILL successfully confirmed that 48 bugs are caused by reference miscounting, 18 bugs are caused by dangling usage and 5 bugs are caused by null-pointer dereference. FREEWILL fails to analyze two bugs due to custom heap managers. Three bugs cannot be reproduced in our instrumented environment. Besides, FREEWILL produced 56 patch suggestions matched with official patches, pinpointed three wrong official patches and gave one wrong suggestion as it cannot handle race problems. FREEWILL completes each bug analysis within 15 minutes on average.

In summary, the main contributions of this paper include:

- **Reference Miscounting Model.** We proposed an omission-aware refcounting model, which can detect two types of reference miscounting bugs.
- **Practical Methods.** We develop several novel practical techniques for binary UAF diagnosis, including taint-based reference analysis, heuristic refcounter identification, and distance-based refcounting matching.
- **Automatic Tool.** We implemented FREEWILL and evaluated it on 76 real-world UAF bugs. The results validated the effectiveness and practicality of our approach.

2 Background

2.1 Motivating Example

Figure 1 contains a motivating example showing how reference miscounting errors lead to UAF bugs. The code defines two structures, `A` and `B`, and two global pointers, `gpA` and `gpB`. `A` uses a refcounter `ref` for its instance life cycle; `B` only has a member `_pA`, a pointer of an `A` instance. It simulates two real-world buggy functions from Linux kernel and IE: `filename_lookup` and `clone`. Then, it uses `demo_code` to call the two buggy functions and related operations. The `main` function invokes `demo_code` and triggers two UAF bugs.

In `demo_code`, the code first creates an `A` instance, say α, and sets its `ref` to **1** as its first reference `R0` is now created and hold by `pA`. Now, the value of `ref` is **1**.

Two Valid Omissions. After allocation and initialization, a global reference `R1` is created and stored into `gpA`. As the lifetime of global `R1` overlaps with the local `R0`, the code safely omits the increase and just uses a single decrease for `R1`. Next, it creates a `B` instance and initializes its member `_pA` with `pA` where a new reference `R2` is created. As the lifetime of `R2` is contained into the one of `R1`, the code omits the refcounting for this new reference. Now, the value of `ref` is still **1**.

Reference Miscounting-1 (RM1). At Line `16-17`, a second `B` instance is created and stored into the global variable `gpB`. Then by calling `clone`, the code copies all memory content pointed by `pB` into the memory pointed by `gpB`. Note that a new reference `R3` will be created after the `memcpy`. As `R3` will be used outside `demo_code`, there is a mistaken omission and the value of `ref` is still **1**.

Reference Miscounting-2 (RM2). At Line `20`, the code fails to call `inc_ref` but directly calls `filename_lookup` in which `dec_ref` is called to decrease the `ref`. Here, a reference `R4` is created on stack and destroyed when the code returns from the function. Now, the value of `ref` is decreased to **0**, and, unfortunately, the instance α will be prematurely freed.

Two Dangling Pointer (DP1&DP2). At Line `18`, `R2` in `_pA` is destroyed by the nullification. Besides, `R0` in `pA` will also be removed by the destruction of the stack frame. However, there are two dangling pointers `gpA` and `gpB->_pA` storing two references, `R1` and `R3`.

UAF Bugs. In the function `main`, there are two UAF bugs due to the uses of two aforementioned dangling pointers. The first bug happens at Line `24` where the code dereferences the dangling pointer `gpA` to complete a correct `dec_ref` as the pointer will never be used anymore. The second bug is caused by the access of the dangling pointer `gpB->_pA` at Line `25` as it wants to display `ref` of the instance α.

2.2 Limitation of Existing Diagnoses

Several state-of-the-art methods have been proposed to prevent and diagnose UAF bugs. In Table 1, we compare the diagnosis results on our example code from these methods.

Table 1: **Diagnoses from Existing Solutions on Figure 1.** ***DP*** = Dangling Pointer, ***RM*** = Reference Miscounting.

Diagnosis Results	***DP1***	***DP2***	***RM1***	***RM2***
Dangling Access Locations	✔	✘	✘	✘
Dangling Pointers	✔	✔	✘	✘
Refcounting Inconsistencies	✘	✘	✘	✔
Our Goal	✔	✔	✔	✔

Dangling Access Locations. There have been several kinds of UAF prevention schemes [20, 29, 34, 41, 44, 52] owning ability to precisely report dangling access locations. Besides, by encoding detailed information into the object memory region [41] or pointer address [34], they can provide other critical locations, such as object allocation and deallocation. In our example, these methods will stop after identifying the first UAF bug and report *malloc* (Line `10`), *free* (Line `7`) and *reuse* (Line `24`). However, with these locations, developers still have no idea about the existence of reference miscounting. The *reuse* location (Line `24`) can even mislead them as it is a correct access to release `R1` stored in `gpA`.

Dangling Pointers. While the work using pointer tracking or sweeping [20, 23, 34, 44, 50, 55] can provide the details of all dangling pointers, they rely on developers to investigate the root reason of dangling pointers. Besides, dangling pointers may not be the problematic one. For example, `gpA` is correctly used to call `dec_ref`. Finally, the reference miscounting of `R4` (RM2) will be missed as it is not a dangling pointer.

Refcounting Inconsistencies. Statically detecting the inconsistencies between refcounting changes and reference changes [30, 35, 45], or between refcounter increment and decrement [32, 48], can detect reference miscounting. However, considering the limitations of static program analysis, such as alias analysis (especially on binaries), all of them suffer from high false positives and false negatives. In our example, given the source code, they can identify the second reference miscounting as the explicit inconsistent decrease at Line 20. However, they cannot effectively detect mistaken omission at Line `17` (RM1) as the alias pointer (`R3`) in `gpB` is implicitly copied from the alias pointer (`R2`) in `pB`.

2.3 Our Goal and Assumptions

Given a UAF bug, our goal is to determine whether it is caused by reference miscounting; if so, pinpoint the location that misses the refcounting, and suggest a correct patch. If there is no reference miscounting, we should also report the dangling pointers based on our traced-based analysis. Our method should diagnose UAF bugs efficiently to support real-world large programs with or without the source code. Finding new program failures or UAF bugs is out of the scope of this work.

Assumptions. Our approach does not require the source code

of the vulnerable program, but we assume that we can identify heap-related APIs, like *malloc* and *free*. This is possible for most benign programs, which clearly state these APIs in their import descriptors. Further, our solution requires at least one Proof-of-Concept (PoC) input. These inputs are commonly provided by bug reporters, or from fuzzing techniques [1, 25, 57].

2.4 Challenges

To detect reference miscounting issues and suggest practical patches, we have to solve the following challenges:
C1: Precisely Identifying the Lifetimes of UAF Objects. Modern allocators prefer to reallocate freed heap blocks to improve the performance. Therefore, before a UAF bug is triggered, a freed block could have been reallocated many times, which can confuse analysis. Existing solutions mainly rely on source code for detection [41] or tracking pointers of all objects [23]. They cannot efficiently identify the lifetimes of UAF objects, especially for large-scale binary-only programs.
C2: Automatically Identifying Refcounting on Binary Execution Trace. Confirming if the bug is introduced by reference miscounting relies on the fact that the object is managed by refcounting. Unfortunately, there are many similar kinds of counters (e.g., *Loop Counter*, *Object Counter*) that can cause many false positives, especially when no source code can be used to look for their semantic meaning.
C3: Precisely Matching Refcounting Operations with References. First, developers may perform the refcounting on *any* existing reference. It is common to find `Ri->addRef();Rj=Ri;` and `Rj=Ri;Rj->addRef();` in the same program to count a new created reference `Rj`. Second, there is no consistent rule mandating the time to update the refcounter. Ideally, developers should update the refcounter immediately after the creation or destruction of a reference. However, due to the compiler optimization and encapsulation of refcounting, we may distinguish these two types of actions from each other with various distances.

3 Reference Miscounting Detection

Reference Representation. To assist our description, we define a reference as a 4-tuple record `(+,` T_c`, -,` T_d`)`: `+` and `-` are boolean values, indicating whether the increment and the decrement of the corresponding refcounter are identified or not; T_c and T_d are the creation time and the destruction time of the reference, respectively. In the implementation §5.1, we use the instruction ID in the trace to represent T_c and T_d.

Inspired by reference escape analysis [56] which uses the analysis of reference lifetime to optimize (omit) refcounting, we propose an *omission-aware* refcounting model. Our basic rule is, for any interesting reference, the refcounting or

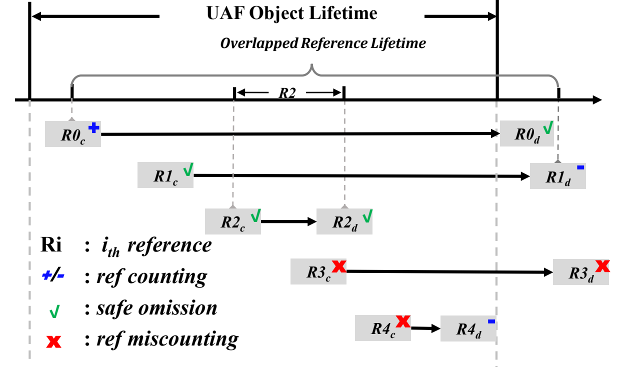

Figure 2: **Reference Miscounting Detection on Figure 1.**

its omission should guarantee the object's liveness until all references are destroyed. We use the following four practical omission rules (ORs) to check if every omission is valid.

OR1: Overlapping Omission. For two references `Ri` and `Rj`, if `Ri.`T_c < `Rj.`T_c < `Ri.`T_d < `Rj.`T_d, we say the two references are overlapped, and `-` of `Ri` and `+` of `Rj` can be safely omitted. Any reference can only have one overlapped reference.

OR2: Transmitting-Overlapping Omission. For any number of references `R1, ..., RN`, if for any reference `Ri` (`1<i⩽N`), `Ri-1.`T_c < `Ri.`T_c < `Ri-1.`T_d < `Ri.`T_d, we say these references are transmitting-overlapped, and in this case, `-` of `R1`, (`+`, `-`) of `Ri` (1<i<N), and `+` of `RN` can be safely omitted.

OR3: Containing Omission. For two references `Ri` and `Rj`, if `Ri.`T_c < `Rj.`T_c < `Rj.`T_d < `Ri.`T_d, we say `Rj` is contained in `Ri` and (`+`, `-`) of `Rj` can be safely omitted.

OR4: Overlapping-Containing Omission. For three references `Ri`, `Rm` and `Rj`, if `Ri` and `Rj` are overlapped based on OR1 or OR2, and `Ri.`T_c < `Rm.`T_c < `Rj.`T_c and `Ri.`T_d < `Rm.`T_d < `Rj.`T_d, we say `Rm` is overlapping-contained in `Ri` and `Rj` and its refcounting (`+`, `-`) can be safely omitted.

Figure 2 explains how we use our model to automatically detect two reference miscountings in the motivating example (Figure 1). First, we assume the lifetimes of references `R0`-`R4` have been detected. Then, based on rule OR1, as `R0` and `R1` are overlapped, we can tell the omission of `-` of `R0` and `+` of `R1` are safe. Similarly, based on rule OR4, `R2` is overlapping-contained in `R0` and `R1`, so the omission of (`+`, `-`) of `R2` is allowed. However, as `R3` is not contained in any other reference, its omission of (`+`, `-`) is wrong. Besides, `R4` is also not overlapped with other references, its `+` omission is also unexpected.

4 FREEWILL Design

Figure 3 shows the design of FREEWILL, which works in four main components: UAF object lifetime identification, reference analysis, refcounting detection, and UAF diagnosis.

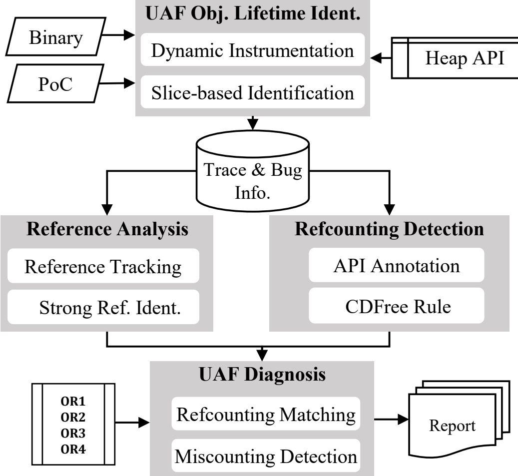

Figure 3: **Overview of FREEWILL.** FREEWILL takes as inputs the vulnerable program and the PoC of the bug, and produces a report of the reference miscounting location.

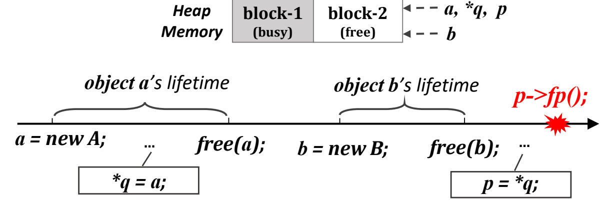

Figure 4: **Heap Block Reallocation.** Current allocators prefer to use one block to satisfy two adjacent same-size requests.

First, given a vulnerable program with a PoC, FREEWILL uses dynamic instrumentation to collect bug-triggered execution trace and identify the lifetimes of UAF objects (§4.1). Then, FREEWILL uses a gray-tainting method to track reference creation, destruction and different ways of reference calculations (§4.2). By identifying the refcounter with the API symbols or our heuristic method (§4.3), FREEWILL matches the refcounting with proper references and compares the actual refcounting behaviors with our omission-aware model to detect reference miscounting issues (§4.4).

4.1 UAF Object Lifetime Identification

FREEWILL is designed to be a trace-based method and we chose to use *Dynamic Code Instrumentation* [8, 15, 51] to record the bug-triggered execution trace and related runtime information, e.g., heap allocations and crash point – the invalid memory access or invalid call instruction causing the program crashed. If there is no crash, we try to manually confirm if the program has produced any crash dump file [2, 6].

If the source code is available, FREEWILL can adopt existing methods, such as *AddressSanitizer* [41] (*ASan*), to identify UAF object lifetimes. Specifically, we recompile the vulnerable program with *ASan*, and run the generated binary with the bug-triggering input. *ASan* adopts a safe memory allocator that saves all freed objects in a quarantined list and records their memory status (freed) in the shadow memory. For every free-memory access, *ASan* will detect it by checking the shadow memory. Note that the quarantined list can efficiently stop the reallocation of the freed memory [41], which means one heap block is usually only supplied to one object. Therefore, as long as the UAF bug is triggered, *ASan* can detect the bug and report the UAF object lifetime.

If source code is not available, we develop a backwards slicing method to efficiently identify the UAF object lifetime. Specifically, we first assume the program has finally reached a crash point where a corrupted function pointer `fp` is accessed. Then, we slice it backwards to find its parent pointer `p` pointing to a heap block from which `fp` is loaded. Without any quarantined list, the heap block could have been reallocated many times for other requests (e.g., `block-2` in Figure 4), which means the same block has more than one lifetime. To identify the proper lifetime, we continue the slicing with `p` backwards to find the proper object, e.g. object `a`. Finally, we identify the object as our target and report its lifetime if the dereference of `p` happens after its destruction. In our experiments, we find the reallocation problem in 18 bugs.

Different from *ASan*, FREEWILL will not terminate the program execution until it meets the crash. If the pointer of the first UAF object can be normal dereferenced, the execution trace may contain other UAF bugs. For example, in Figure 4, if the object pointed by `q` has also been freed before `p=*q` is executed, the final crash is actually caused by the second UAF bug, i.e., `p->fp()`. We call this kind of bugs *Multi-level* UAF (MUAF), which usually happens when dangling pointers can be still normally dereferenced. To analyze these bugs, FREEWILL will repeat above steps with parent pointer (`q`) of current UAF object pointer (`p`) until no UAF bug is found. It will prioritize the diagnosis of the first UAF object. In our experiments, we find 5 bugs that have MUAF problems.

4.2 Reference Analysis

Within an object lifetime, we can use taint analysis to identify locations holding the object address. We first identify such locations as potential references which can be used to create other new references. After we have tracked all locations, we select proper references mainly created by programs and filter out the temporary ones created by compilers.

4.2.1 Potential Reference Tracking

We use the gray-taint tracking method (details in §5.2) to detect potential references in the following three categories.

Reference Creation. We identify a creation of a new reference when an object address is saved to a register or any

memory location, no matter whether this location is on heap, stack or global region. For example, instructions *mov [mem], eax* and *push eax* create two new references if the register `eax` contains an object reference.

Reference Destruction. There are two cases that lead to destruction of a reference: the memory holding the reference is overwritten by a different value, or the memory holding the reference is not being used any more. The second case includes two scenarios: a heap memory is not used due to memory deallocation, or a stack memory is not used due to function return. We inspect memory write operations, function calls to free and return-related instructions to detect them.

Reference Computation. Arithmetic and logical instructions (e.g., *add*, *sub*, *lea*, *and*, *or*, *mul*, *div* and *xor*) can also create or destroy references. Here we use `C-R` to represent a computed reference. We use gray-taint to track the whole propagation of each `C-R` reference, which not only helps us find critical references that lead to UAF bugs, but also discloses valuable reasons and classify these bugs. For example, if `R2` has lifetime `R0->C-R1->R2` and its creation has been miscounted, one possible reason is that `R1` is incorrectly treated as non-reference data during the computation.

4.2.2 Strong Reference Identification

A key issue of the method above is that there are too many temporary references created by compilers. For example, in Figure 1, the source code `gpA=pA` will be compiled into two low-level instructions. One is used to load the reference from `pA` to a register and the second is used to create a new reference in `gpA`. To facilitate our diagnosis, we mark the following references with *strong* tags based on where they are stored.

Heap Reference. These references are usually created by developers and most of the refcounting operations are used to count them. To identify heap references, we first record all heap allocations with their ranges. Then, any reference stored into one of these locations will be marked with a *strong* tag.

Stack Reference. We only track references stored on stacks if they are: (1) *local variables*. We recognize local variable space by checking instructions that manipulate stack frame pointers, and mark all references stored in these spaces as strong. (2) *function arguments*. We treat a reference pushed into the stack as a function argument if a `call` instruction is followed shortly. Note that we will filter out any location that is used in a `push-pop` pair for a temporary storage.

4.3 Refcounting Detection

When the program source code is available, we request software developers to annotate the refcounter increase/decrease locations. Fortunately, most systems adopting refcounting provide dedicated functions for these operations, e.g., *ref_get*, *inc_ref* for increase and *ref_put*, *dec_ref* for decrease. Even if these functions are inlined, we can use the debug information to reliably identify the refcounting instructions.

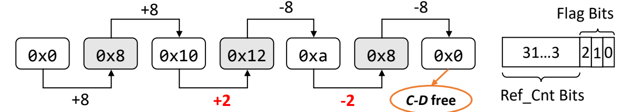

Figure 5: **Detection of *CTreeNode* Refcounter by a Control Dependent Free.**

In case we only have program binaries, we can use heuristics to identify refcounting operations. A recent work [48] proposes to identify refcounters from Linux kernel through the following heuristics, which we call *HR-FixStep*: refcounters are usually increased or decreased by a fixed step (e.g., 1). However, during our testing we find this simple method can be inaccurate. A common reason is that developers use the lowest bits of the refcounter as special flags, and any update to these flags will make this rule fail. Figure 5 demonstrates the value changes of a *CTreeNode* refcounter in the IE browser. Here, IE only uses the highest 29 bits of a 32-bit integer as the refcounter and treats the lowest 3 bits as flags. When the program sets or clears these flags, we will find that the refcounter is updated via different steps, which makes HR-FixStep fail.

To address this limitation, we propose a new heuristic method that relies on the control dependency between the refcounter and the free function to identify refcounters.

HR-CDFree: *If refcounter becomes zero, there should be a free operation that is control-dependent on the refcounter.* Specifically, we collect and build a value sequence for each object field (e.g., 4 bytes for 32-bit systems and 8 bytes for 64-bit systems) in the trace. When any field becomes zero, we use a *coarse-check* method in following execution trace to confirm if there is a control-dependent free: We first check if a *zero-condition-jmp* (e.g., `JZ` and `JNZ`) follows shortly and then detect if a free method is called before the current function returns. In Figure 5, our new method can precisely identify the refcounter field because of the control-dependent free.

We applied both heuristic rules on 543 objects that have different sizes and structures. Based on the ground truth (65 reference-counted and 478 uncounted objects), our new method achieves 98% accuracy and 90% precision, while the previous method only gets 86% accuracy and 43% precision. We will present more detailed evaluation and analysis in §6.3.

4.4 UAF Diagnosis

To diagnose UAF bugs, we first match actual refcounting operations with proper references. For each reference, we compare the refcounting operations with our omission-aware model, and report any violation as a reference miscounting.

4.4.1 Refcounting Matching

Based on our statistics, there are mainly two programming styles to implement refcounting. To achieve higher performance, programs written in C (e.g., system kernels or script engines) prefer to insert an inlined refcounting operation around the reference creation or destruction. However, to achieve better encapsulation and reusability in modular and object-oriented programs (e.g., web browsers), developers like to use additional classes or non-inlined functions (*wrapper*) to implement refcounting. Therefore, we develop two distance-based methods to match refcounting with references.

Matching Based on Execution Distance (ED). For programs using inlined refcounting operations, we develop a simple and efficient distance-based method to match the refcounting with the proper reference. Specifically, in every execution trace, we assign a sequence number `N` to each instruction based on its executing order, denoted as `Instruction ID` (§5.1). Then, as shown on the top of Figure 6, we use $\texttt{ED} = ||\texttt{N}_{count}\text{-}\texttt{N}_{ref}||$ to measure the distance between a refcounting instruction and a reference creation or destruction instruction. Finally, we mark the refcounting and the reference as a matched pair if they have a minimum distance. For example, in Figure 1, it is straightforward to match `pA->ref=1` with the closest `pA=malloc()`. Similarly, the decrease `dec_ref(gpA)` can also be matched with the implicit destruction of the stack frame based on our reference analysis.

Matching Based on Wrapper Distance (WD). For programs using non-inlined functions to implement refcounting, we first identify function boundaries by detecting the entry point `EP` (i.e., *call* instruction) and return point `RP` (i.e., *ret* instruction) in the execution trace. Then, we calculate the wrapper distance `WD` between the `RP` of current wrapper with the `EP` of next wrapper, i.e., $\texttt{WD} = ||\texttt{N}_{RP_Current}\text{-}\texttt{N}_{EP_Next}||$. In Figure 6, to match the + with proper reference, $\texttt{R}_1$ or $\texttt{R}_2$, although there is $\texttt{ED}_1 > \texttt{ED}_2$, considering the existence of wrappers, we will calculate the `WD` and match it with $\texttt{R}_1$ as there is $\texttt{WD}_1 < \texttt{WD}_2$.

If we have the source code, we can use the debug information to annotate the boundaries. For binary-only traces, we can directly use *call-ret* pairing methods to identify the wrappers. Specifically, we assume there will be any number of *call* and *ret* instructions between two wrappers. We first pair successively appeared *call* and *ret* instructions. Then, we mark the last unpaired *ret* and the first unpaired *call* as the boundaries and calculate the $\texttt{ED}_{wrapper} = ||\ \texttt{N}_{ret}\text{-}\texttt{N}_{call}\ ||$ as the temporary wrapper distance. Finally, we check if there is any *call-ret* pair `P` between the two boundaries, if so, we calculate the execution distance $\texttt{ED}_P$ from the *call* to *ret* of `P` and then subtract it from $\texttt{ED}_{wrapper}$ as the final wrapper distance `WD`.

4.4.2 Reference Miscounting Detection

With our omission-aware model, we design Algorithm 1 to analyze references based on their refcounting behaviors. The algorithm takes the set of identified references `ref_set` as input, and reports the details of reference miscounting issues. We sort references in `ref_set` in the ascending order of $\texttt{T}_c$ and analyze them one by one. Each reference `r` belongs to one of the following four categories:

1) If `r` has both + and -, we add it to `rc_set` (`Line 5`);
2) If `r` only has +, we add it to `inc_set` (`Line 7`);
3) If `r` only has -, we will first check `inc_set` to find an overlapped reference `r'` following rule OR1 such that `r'` is created before `r` (i.e., $\texttt{r'.T}_c < \texttt{r.T}_c$). If `r'` exists (`Lines 11-13`), `r'.-` and `r.+` are allowed to be `False`. We will create a *virtual* reference `vr: (.+=True, .`$\texttt{T}_c$`=r'.`$\texttt{T}_c$`, .-=True, .`$\texttt{T}_d$`=r.`$\texttt{T}_d$`)` and add it into `ov_set`. We will remove `r'` from `inc_set` as it can only be matched with once based on OR1.
   If we cannot find such an `r'`, we will resort to rule OR2 to find an `r'` such that `r'` is the head of the transmitting chain (`R1` in OR2) while `r` is the tail (`RN` in OR2). Based on OR2, it is legitimate to omit `r'.-`, `r.+` and `(+,-)` of intermediate references (i.e., `Ri` for 1<i<N). We will create the virtual reference, remove `r'` from `inc_set` and remove `Ri` from `ref_set` (`Lines 18-21`).
   If we cannot find any `r'` using OR1 nor OR2, our algorithm will report an inconsistent decrement bug `ID_BUG` and suggest an extra + for patching (`Lines 23-24`).
4) If `r` has neither + nor -, we will check `rc_set` to find a reference `r'` that contains `r`. Based on rule OR3, if `r'` exists, we can safely omit the `(+,-)` of `r` (`Line 28`). If `r'` does not exist, we will check `ov_set` to find such a `r'`. If `r'` exists in this case, we can safely omit the `(+,-)` of `r` based on rule OR4 (`Line 31`). If we cannot find any `r'` based on rule OR3 nor OR4, we will report a mistaken omission bug `MO_BUG`, and suggest to add `(+,-)` back as the patch (`Lines 33-34`).

Finally, as shown in Figure 2, we will analyze the *reuse* locations of dangling pointers to detect a special kind of inconsistent decrease bugs (see §6.5.1): a reference `r` having two decrements but only one increment. If no reference miscounting is found, we will report all dangling pointers as our diagnosis result.

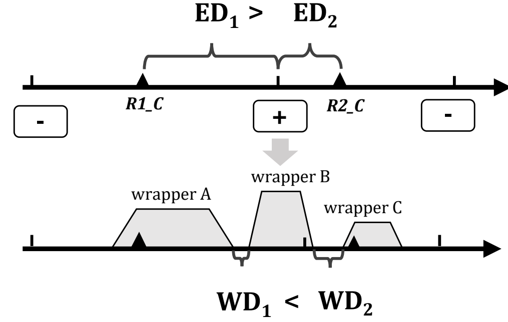

Figure 6: **Distance-based Refcounting Matching.** `R_C` = Ref. Creation, `ED`=Execution Distance, `WD`=Wrapper Distance.

Algorithm 1: UAF Diagnose

```
   Input: ref_set: reference set with matching result
   Output: report: diagnose and patch report
 1 rc_set = ∅ ;                                    // inc&dec set
 2 inc_set = ∅ ;                                       // inc set
 3 ov_set = ∅ ;                                    // overlap set
 4 foreach r ∈ ref_set: r.+ and r.- do
 5     add r into rc_set
 6 foreach r ∈ ref_set: r.+ and !r.- do
 7     add r into inc_set
 8 foreach r ∈ ref_set: !r.+ and r.- do
 9     if ∃r' ∈ inc_set: r'.T_c < r.T_c < r'.T_d < r.T_d
10     then                                                // OR1
11         vr = (True, r'.T_c, True, r.T_d)
12         add vr into ov_set
13         remove r' from inc_set
14     else if ∃R1 ... RN ∈ ref_set, ∀1<i≤N:
15       Ri-1.T_c < Ri.T_c < Ri-1.T_d < Ri.T_d and !Ri.+ and !Ri.-
16       and RN is r and R1 is r' and r' ∈ inc_set
17     then                                                // OR2
18         vr = (True, r'.T_c, True, r.T_d)
19         add vr into ov_set
20         remove r' from inc_set
21         remove Ri (1<i<N) from ref_set
22     else
23         report ID_BUG ;                              // report
24         add + for r ;                            // patch sugg.
25 foreach r ∈ ref_set: !r.+ and !r.- do
26     if ∃r' ∈ rc_set: r'.T_c < r.T_c < r.T_d < r'.T_d
27     then                                                // OR3
28         continue to next r
29     else if ∃r' ∈ ov_set: r'.T_c < r.T_c < r.T_d < r'.T_d
30     then                                                // OR4
31         continue to next r
32     else
33         report MO_BUG ;                              // report
34         add +, - for r ;                         // patch sugg.
```

5 Implementation

Our implementation contains about 10K lines of code, including about 7K lines of C/C++ code to implement instrumentation (QEMU extension and Pin-tool), taint analysis and data slice, and about 3K lines of Python code for UAF diagnosis, including UAF object identification, refcounting detection, refcounting matching and reference miscounting detection.

5.1 Instrumentation

FREEWILL can use any existing instrumentation tools, *Pin* [15] or *DynamoRIO* [8] to collect bug-triggered execution traces. In fact, for command-line programs, such as *Python* script engine, we implement a *Pintool* to collect the traces. However, based on our practical experiments on various large-scale GUI-based programs (e.g., web browsers), it is the 50-100 times overhead [21] that makes many crashes fail to be reproduced by *Pin*. To solve this problem, we implement an instrumentation tool based on a well-performed virtual machine *QEMU* [22]. We fetch the CPU status in the *cpu-exec.c* file and insert our instrument code during the TCG translation, i.e., in the *translate.c* file. Different from *Pin* providing thread-related APIs, we use kernel structures, e.g., *EThread* [9], to extract *thread id* from emulated CPU for Windows programs. In summary, we collect following information:

Instruction Record. We will record *Instruction ID*, *Instruction Address*, *Operation Code*, *Operand Value* and *Thread ID*. Specifically, *Instruction ID* is used to calculate the execution and wrapper distance for our refcounting matching. *Instruction Address* is used to mark the location of reference miscounting. *Operation Code* and *Operand Value* are used to detect all references and build value sequences. *Thread ID* is used to identify multi-thread UAF bugs when *malloc*, *free* and *reuse* are detected in different threads.

Heap Related Record. The heap record contains the *Instruction ID*, *Heap Operation Type*, *Object Address*, *Object Size*. Specifically, *Operation Type* can be one of allocation, deallocation and reallocation. *Object Address* and *Object Size* record the object memory space range. The heap information is used to help detect the UAF bug and identify if there is a refcounter in object fields.

5.2 Gray-Taint Analysis

We first mark with `TAINT` flags the pointers returned from UAF object-related allocations and track them as follows:

Normal Taint Rules. We first analyze movement instructions (e.g., *mov/movsd*, *push*) which can produce new references if the source operand is tainted. To detect destructions of stack references, we will analyze instructions such as *ret N*, *pop*, *leave* and *call* which can destroy them. For heap references, we will check if any tainted cell is freed or overwritten.

Gray Taint Rules. Whenever a tainted reference involves computation, we will first make sure it is not completely destroyed, e.g., *xor eax, eax*. If there is a small change, e.g., a bit operation in *or [ref], 0x1*, we will mark it with a `GRAY` flag and propagate it in the following analysis. When we meet it again, we will check if the final value is equal to object address, if yes, we will treat it as a new reference and mark it with `TAINT`.

Under-Taint and Over-Taint. They are two common problems when we implement our taint engine. We use the reference value, say `ref`, as check conditions to migrate the problems. First, if any instruction containing a `ref` is not tainted, we will add rule(s) to fix this under-taint. Second, when a new reference is detected, we will check if its value is `ref`, if not, we will add rule(s) to fix this over-taint.

6 Evaluation

We mainly evaluate FREEWILL on following aspects:

- **UAF Object Lifetime Identification**: Can FREEWILL identify UAF objects with proper lifetimes?

- **Reference & Refcounting Identification**: How accurate are our reference detection & refcounting identification?
- **Diagnosis & Patch Suggestion**: What is the effectiveness of FREEWILL in root cause diagnosis and patch suggestion for UAF bugs?
- **Lessons from Bugs**: What can we learn from these bugs and wrong (or incomplete) patches (or suggestions)?

6.1 Experiment Setup

Benchmark. Using public search engines, official bug track websites [3–5, 14, 16] and kernel commit logs [11], we collect 76 UAF bugs presented in the first three columns in Table 3. Note that the proportion of different bugs in our dataset could be inconsistent from the ones in the real world as a bug is collected into our dataset only if we can reproduce it. However, to keep the fairness of our selection, we try to cover as many types of bugs as we can and also collect various thorny cases, e.g., custom-defined heap managers and race problems. Our dataset covers five types: mistaken omission (MO), inconsistent decrease (ID), dangling usage (DU), null-pointer dereference (NPD), multi-thread (M) and ten subtypes (see §6.5) of root causes leading to UAF bugs, which makes our dataset representative and convincing.

Environment. We configure FREEWILL to run on a Ubuntu 18.04.1 platform with 8-core 4.00GHz CPU, 64GB RAM. We install 32-bit Windows 7 and Windows 7 SP1 as QEMU guests to execute the various versions of web browsers, and install a 32-bit Ubuntu-16.04 virtual machine to execute other programs. We use our QEMU extension and Pin-tool to instrument and fetch traces.

Kernel Drivers. Due to the inconvenience of kernel instrumentation, we extract all related driver files containing the 21 kernel UAF bugs (15 bugs in 13 versions of Linux and 6 bugs in 5 versions of MacOS) to build user-space dynamic libraries that can be executed and monitored by our tools. For example, we use 74% code of 12 original class files in MacOS *libkern* module for `CVE-2016-1828` and `CVE-2016-4656`. We mainly keep *Serialize*/*UnSerialize* functions and disable unrelated code, e.g., interactions with other kernel drivers. Besides, we use *libc.so*'s *malloc* and *free* to replace the original *kmalloc* and *kfree* in driver files. For other kernel driver bugs, we adopt similar methods to reproduce them and Table 2 gives the details. On average, for each bug, we need about 30 minutes to manually extract all related files and remove unrelated code from source code.

6.2 UAF Object Lifetime Identification

Traces and Crashes. The *Trace* and *Crash* columns in Table 3 show that FREEWILL has successfully reproduced 73 bugs using our instrumentation tools. The average size of the traces for each bug is 24GB containing more than 130 millions of instructions. FREEWILL fails to analyze three bugs and the main reason is the side effects of low-performance instrumentation. For 67 out of 73 bugs, we can directly find the crash points in the last instructions of the traces. However, there are six bugs of Firefox and Chrome whose crash points need to be manually identified with minidump [12] files produced by their *Breakpad* [2] or *Crashpad* [6].

Table 2: **Source Code Extracted for 21 Kernel Bugs.**

Kernel	# Vers.	*.h	*.c/.c++	LoC	Time/Avg.(h)
Linux	13	195	80	50,186	7.5 / 0.5
MacOS	5	60	45	66,650	3 / 0.5

UAF and Reallocations. In the *UAF* column, we can see FREEWILL has successfully identified 66 UAF bugs, including 5 multi-level UAF bugs and 5 multi-thread UAF bugs (*malloc*, *free* and *reuse* have different *thread id*). For five of them, we need the assistance of *ASan* to catch the crash points, otherwise they will be delayed to happen in complicated heap manager functions (*free* or *malloc*). FREEWILL has also identified five null-pointer dereference bugs who have been reported as UAF bugs because nullified references are reused without any null-check. For `CVE-2015-2425` and `CVE-2017-11810`, FREEWILL fails to identify the bugs as we have no custom-defined heap API symbols. Finally, with backwards slicing, FREEWILL identifies 18 UAF bugs whose memory blocks have been reallocated for other objects.

6.3 Reference & Refcounting Identification

Reference Identification. From the *Ref. Ident.* column, we can see FREEWILL successfully identifies all references of the UAF objects within their lifetimes. On average, each of three web browsers, for the UAF object, creates more than 2,000 potential references, including the ones stored in *heap*, *stack* and *register*. For other programs, each of them also creates more than 100 potential references which is still a challenge for manually debugging. The *Strong* column shows that FREEWILL has filtered out more than 90% of potential references, which can facilitate the diagnoses.

Refcounting Identification. From the + column, we can see that FREEWILL successfully identifies 52 UAF objects using refcounting to manage their lifetimes. We have manually confirmed that there are 14 UAF bugs whose vulnerable objects have no refcounter in their structure or class definitions. From the *Offset* column, we can see different kinds of objects use various positions to store refcounters.

To compare the effectiveness of two heuristic rules (mentioned in §4.3), we manually identify 543 objects (having different sizes and structures) from all traces as our dataset. Based on the ground truth, 65 reference-counted objects and 478 non-reference-counted ones, as shown in Table 4, we use HR-FixStep to identify only 37 counted objects, and use HR-CDFree to totally identify 61 objects.

Table 3: **Diagnosis Results.** ***DMP***=DUMP File, ***RA***=Reallocation Times, ***MU***=Multi-level UAF, ***DF***=Double Free, ***(A)***=ASan, ***(M)***=Multi-thread UAF, *C-/M-/RD-/P-=Calculation*, *Mov*, *Rep Movsd* or *Push* instructions, ***RM***=Reference Miscounting, ***RC***=Reference Counting, ***NPD***=Null-Pointer Dereference, ✘= Wrong or Incomplete Patch (Suggestion).

Prog.	BugId	Ver.	UAF Obj. Ident.				Ref. Analysis				Diagnoses			
			Trace (GB)	Crash	UAF	RA	Ref. Ident.		Ref. Count		RM	Root Cause	Patch Suggestion	Official Patch
							Poten.	Strong	+	Offset				
IE	CVE-2010-0248	8.0.7600.16385	1.6	✔	✔	2	188	31	1	+0x10	✔	*M*-(R_c,)	***Add +/-***	***Add*** R_d
	CVE-2010-0249	8.0.7600.16385	2.7	✔	✔	1	8460	661	56	+0x40	✔	*RD*-(R_c,)	***Add +/-***	✔
	CVE-2010-3971	8.0.7600.16385	3.9	✔	✔	1	52	10	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	CVE-2011-1260	8.0.7600.16385	2.5	✔	***MU***	0	1266	163	1	+0x40	✔	*C-RD*-(R_c,)	***Add +/-***	✔
	CVE-2012-1875	8.0.7600.16385	3.8	✔	✔	0	3273	210	12	+0x4	✔	*M*-(R_c,)	***Add +/-***	***Add*** R_d
	CVE-2012-4787	9.0.8112.16421	1.4	✔	✔	0	313	13	8	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2012-4792	8.0.7600.16385	3.3	✔	✔	0	1026	22	7	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2012-4969	8.0.7600.16385	3.3	✔	✔	1	200	32	3	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2013-0025	8.0.7600.16385	2.7	✔	***MU***	0	2973	265	3	+0x40	✔	*M*-(R_c,)	***Add +/-***	***Add*** R_d
	CVE-2013-1306	9.0.8112.16421	18.0	✔	✔	0	1977	74	7	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
	CVE-2013-1347	8.0.7600.16385	1.7	✔	✔	1	928	18	7	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
	CVE-2013-3163	8.0.7600.16385	1.9	✔	✔	0	2271	322	19	+0x4	✔	*M*-(R_c,)	***Add +/-***	***Add*** R_d
	CVE-2013-3893	8.0.7600.16385	2.4	✔	***MU***	1	5110	763	23	+0x40	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2013-3897	8.0.7600.16385	2.5	✔	✔	1	431	77	1	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2014-0282	8.0.7600.16385	1.8	✔	✔	0	4104	508	29	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔
	CVE-2014-1776	8.0.7600.16385	6.2	✔	***MU***	1	3660	463	26	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2014-1815	8.0.7600.16385	2.5	✔	✔	0	5021	962	65	+0x4	✔	*P*-(R_c,)	***Add +/-***	✔
	CVE-2015-2425	11.0.9600.16428	6.8	✔	–	–	–	–	–	–	–	–	–	–
	CVE-2017-11810	11.0.9600.16428	3.1	✔	–	–	–	–	–	–	–	–	–	–
	CVE-2018-8174	11.0.9600.16428	3.7	✔	✔	0	217	36	9	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
Linux	Commit-81b9de4 (2018)	4.16.0-rc1	0.1	✔	***DF(A)***	0	43	10	1	+0xc	✔	*M*-(+R_c, -, -)	***Stop -***	✘***(Add +)***
	Commit-c3aabf0 (2019)	5.2.0-rc1	0.1	✔	***DF(A)***	0	32	9	1	+0x8	✔	*M*-(R_c, -)	***Add +***	✔
	Commit-0cef13d (2020)	5.9.0-rc7	0.1	✔	***DF(A)***	0	60	15	1	+0xc	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	Commit-8cc0dcf (2020)	5.10.0-rc3	2.5	✔	✔	0	258	60	2	+0xc	✔	*M*-(R_c, -)	***Add +***	✔ ***(Delay Free)***
	Commit-ba34c3d (2020)	5.10.0-rc7	2.7	✔	✔	0	258	55	2	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-dbfa04e (2020)	5.11.0-rc1	2.1	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	Commit-beb691e (2021)	5.11.0	2.5	✔	✔	0	171	48	2	+0x0	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	Commit-357a07c (2021)	5.12.0-rc7	2.5	✔	✔***(M)***	0	40	7	1	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-8d43259 (2021)	5.12.0	2.5	✔	✔	0	239	58	3	+0x4	***No***	***Reuse Counted R***	***Stop Reuse***	✔
	Commit-ae4393d (2021)	5.12.0	2.3	✔	✔	0	21	7	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Commit-c81d3d2 (2021)	5.13.0-rc1	2.3	✔	✔	0	119	28	1	+0x0	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-a4f0377 (2021)	5.13.0-rc3	2.1	✔	✔***(M)***	0	214	55	1	+0x64	✔	*M*-(R_c,)	***Add +/-***	✔
	Commit-8fd0e99 (2021)	5.13.0-rc4	2.5	✔	✔	0	73	25	1	+0x0	✔	*M*-(R_c, -)	***Add +***	✔
	Commit-af35fc3 (2021)	5.14.0-rc3	2.6	✔	✔	0	203	52	1	+0xfc	✔	*C*-(R_c,)	***Add +/-***	✔ ***(Delay Free)***
	Commit-ff11764 (2021)	5.14.0-rc3	2.3	✔	✔***(M)***	0	82	22	1	+0x10	✔	*M*-(R_c,)	✘***(Add +/-)***	***Add Lock***
MacOS	CVE-2016-1828	10.11.4	0.5	✔	✔	0	112	38	2	+0x4	✔	*M*-(R_c,)	***Add +/-***	✘***(Delay Free)***
	CVE-2016-4656	10.11.6	0.5	✔	✔	0	120	40	1	+0x4	✔	*M*-(R_c,)	***Add +/-***	✔***(Delay Free)***
	CVE-2017-2545	10.13.2	4.0	✔	✔	0	91	22	1	+0x8	✔	*M*-(R_c, -)	***Add +***	***Remove*** R_c
	CVE-2018-4083	10.13.2	4.3	✔	***DF(A)(M)***	0	77	27	1	+0x4	✔	*M*-(+R_c, -, -)	***Stop -***	***Add Lock***
	CVE-2019-6225	10.14.2	2.3	✔	✔	0	340	67	3	+0x8	✔	*M*-(R_c, -)	***Add +***	✔
	CVE-2020-9892	10.15.5	2.3	✔	✔	0	117	18	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
Python	Issue-24091 (2015)	3.4.3	40.0	✔	✔	1	123	46	6	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔
	Issue-24092 (2015)	3.4.3	40.0	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	Issue-24093 (2015)	3.4.3	40.0	✔	✔	1	357	81	10	+0x8	✔	*M*-(R_c, -)	***Add +***	✔
	Issue-24094 (2015)	3.4.3	30.0	✔	✔	1	183	61	8	+0x8	✔	*M*-(R_c,)	***Add +/-***	***Remove*** R_c
	Issue-24099 (2015)	3.4.3	24.0	✔	✔	1	159	67	6	+0x8	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	Issue-24100 (2015)	3.4.3	24.0	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	Issue-38588-01 (2019)	3.8.0	15.0	✔	✔	0	81	21	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38588-02 (2019)	3.8.0	15.0	✔	✔	0	53	14	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38588-03 (2019)	3.8.0	15.0	✔	✔	0	56	15	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38610-01 (2019)	3.8.0	15.0	✔	✔	0	65	17	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38610-02 (2019)	3.8.0	15.0	✔	✔	0	68	18	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-38610-03 (2019)	3.8.0	15.0	✔	✔	0	68	18	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-39421 (2020)	3.8.0	15.0	✔	✔	0	77	19	3	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-39453 (2020)	3.8.0	15.0	✔	✔	0	61	18	2	+0x10	✔	*C-P*-(R_c,)	***Add +/-***	✔
	Issue-39510 (2020)	3.8.0	40.0	✔	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
PHP	BUG-68594 (2014)	5.4.36	42.0	✔	✔***(A)***	0	84	5	1	+0x8	✔	*M*-(R_c,)	***Add +/-***	✘***(Delay Free)***
	BUG-68710 (2015)	5.4.36	42.0	✔	✔***(A)***	0	84	5	1	+0x8	✔	*M*-(R_c,)	***Add +/-***	✔***(Delay Free)***
	BUG-73392 (2016)	7.0.14	2.4	✔	✔	1	55	13	2	+0x0	✔	*M*-(R_c,)	***Add +/-***	***Remove*** R_c
	OSS-17903 (2019)	7.4.0	–	**No**	–	–	–	–	–	–	–	–	–	–
	OSS-24436 (2020)	8.0.0	2.7	✔	✔	0	103	20	1	+0x0	✔	*M*-(R_c,)	***Add +/-***	✔
	BUG-79922 (2020)	7.3.22	2.4	✔	✔	0	93	26	4	+0x0	✔	*M*-(+R_c, -, -)	***Stop -***	✔
	BUG-66783 (2021)	7.4.0	2.8	✔	✔	0	127	29	0	–	No	***Dangling (No RC)***	***Stop Reuse***	✔
	BUG-80927 (2021)	7.4.0	2.5	✔	✔	1	31	6	0	–	No	***Dangling (No RC)***	***Stop Reuse***	✔
Firefox	CVE-2011-0065	3.0	14.0	✔	✔	1	14	6	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	CVE-2013-0753	13.0.1	3.0	✔	✔	0	539	80	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	CVE-2016-9899	50.0	–	***No***	–	–	–	–	–	–	–	–	–	–
	CVE-2017-5404	52.0a2	272	✔ **(DMP)**	***NPD***	–	–	–	–	–	***No***	***NPD***	***Null-Check***	✔
	CVE-2018-18492	63.0	40	✔ **(DMP)**	✔	0	82	24	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	***Use RC***
Chrome	Issue-1033759 (2019)	80.0.3987.0	22.0	✔ **(DMP)**	✔	0	7134	423	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Issue-1041406 (2020)	79.0.3945.0	61.0	✔	✔***(M)***	3	130	16	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	***Remove R***
	Issue-1062091 (2020)	81.0.4044.0	294.0	✔	✔	0	22247	517	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Issue-1111737 (2020)	86.0.4214.0	–	***No***	–	–	–	–	–	–	–	–	–	–
	Issue-1137630 (2020)	86.0.4214.0	27.0	✔ **(DMP)**	***MU***	0	113	2	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	***Delay Free***
	Issue-1142675 (2020)	87.0.4280.0	97.0	✔ **(DMP)**	✔	2	423	16	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔
	Issue-1155426 (2020)	81.0.4044.0	358.0	✔ **(DMP)**	✔	3	7177	485	0	–	***No***	***Dangling (No RC)***	***Stop Reuse***	✔

Table 4: **Accuracy of Refcounting Identification.**

Rules	TP	TN	FP	FN	Acc.	Prec.	Recall
HR-FixStep	37	428	50	28	86%	43%	57%
HR-CDFree ✔	61	471	7	4	98%	90%	94%

Ground Truth Extraction. When source code is available, we can extract the ground truth directly from the programs. When we only have binaries, e.g., IE, we extract the ground truth as follows: First, we use the reverse engineering tool, *IDA Pro*, with the official symbol (PDB [13]) files to fetch the object name based on the object allocation instruction address in our traces and collect all its related function names (like *CTreeNode::XXX*) in the *Function Name* window of *IDA Pro*. Second, we can extract the ground truth based on the existence of refcounting function names like *CDoc::AddRef* or *CTreeNode::AddRef*. Finally, we will find out in these *AddRef*-like functions the critical instructions, such as *inc [this+0x4]*, which can be used to infer the refcounter offsets (e.g., *+0x4*). While the reversing time mainly depends on the size of the target binary module (e.g., *mshtml.dll*), in our evaluations, it tooks about 5-10 minutes to fetch all function names and extract the ground truth for each object.
HR-FixStep FP/FN Analysis. The main reason of false positives is that it cannot distinguish other kinds of counters, such as *Loop Counter*. The reason of false negatives is that it cannot efficiently handle the diversity of the steps, e.g., the refcounting of *CTreeNode* (explained in Figure 5). Besides, many programs will set a marked *flag* value (e.g., `0x8000` or `0xffff`) to the refcounter before they free the objects.
HR-CDFree FP/FN Analysis: The reason of false positives is that it can not handle status flags, e.g., `isActive`, `isBusy`, which will be first enabled and then disabled to call the *free*. Four false negatives are introduced by decrement functions who have no control-dependence on zero, e.g., MacOS kernels use `0xffff` as the free condition (this has been fixed). Nevertheless, 6% false negative and 98% accuracy rate for more than 500 various objects have proved the efficiency of HR-CDFree. In fact, HR-CDFree helps us to automatically and precisely identify lots of useful refcounters in IE objects, e.g., the *CTreeNode*, *CTreePos*, *CBase*, *CLayout*, *CSecurityThunkSub*, *CFlatMarkupPointer*, etc,. As a result, we advise to use the HR-CDFree rule first in practical usage.

6.4 Diagnoses and Patch Suggestions

UAF Diagnoses. From the *Diagnoses* main column, we can see that there are totally 48 UAF bugs caused by reference miscounting and 18 bugs are caused by dangling usage. Within the dangling usage bugs, there are 4 bugs caused by the reuse of counted references and there are no refcounting in the other 14 bugs. In the *Root Cause* column, we use *(R_c,)* to mean a UAF bug is caused by mistaken omission. Specifically, from the analysis result, we can see that there are totally 36 out of

```
MSHTML-8.0.7600.16385 (Bug Version)
.text: 74D79D49   call   ?GetmarkupPtr@CElement
.text: 74D79D4E   push   [ebp+arg_0]            P-(Rc, )
.text: 74D79D51   push   eax
.text: 74D79D52   call   ?onCssChange@CMarkup
.text: 74D79D57   pop    ebp

MSHTML-8.0.7601.18446 (Patch Version)
if ( *(this + 7) & 0x200 ){                     Add +/-
    v4 = CElement::GetMarkupPtr(this);          (FreeWill)
    (*(*v4 + 0xE0)) (v4);  // AddRef()
    v2 = CMarkup::OnCssChange(v4, a2);
    (*(*v4 + 0xE4)) (v4);  // Release()
}                                               Offical Patch
```

Figure 7: ***P-(R_c,)* Mistaken Omission, Patch Suggestion, and the Official Patch (CVE-2014-1776).**

48 UAF bugs caused by mistaken omission. We use *(R_c, -)* to mean the inconsistency problems as there is only - but no corresponding +, which is the main reason for 6 UAF bugs. We use *(+R_c, -, -)* to mean another kind of inconsistency decreases for 6 bugs in which the problematic references have been decreased twice. It is noted that FREEWILL uses the prefix, *C*, *M*, *RD* and *P* for all reference miscounting caused bugs to mean, *Calculation*, *Mov*, *Rep Movsd* and *Push*, different ways of reference creation, which is very useful to classify the bugs and suggest effective patches. Finally, FREEWILL has identified five null-pointer dereference (*NPD*) bugs (their parent object are still alive) reported as a special kind of UAF bugs because the nullified references are still accessed without any *Null-Check*. We give more details of our diagnoses and the performance in Appendix §A.1.

Patch Suggestions. Based on above diagnosis reports, FREEWILL can automatically conclude bug types and produce effective patch suggestions for UAF bugs. For example, for the typical *P-(R_c,)* bug shown in Figure 7, FREEWILL will suggest to *add +/-* around the reference creation, which is exactly matched with the official patch. In fact, from the last two columns of Table 3, we can conclude that FREEWILL can help developers to effectively fix different kinds of UAF bugs. Specifically, there are totally 56 out of 71 patch suggestions which are matched with the official patches and marked with ✔ in the table. Besides, for the reference miscounting caused UAF bugs, FREEWILL can give more concrete patch suggestions and there are totally 37 out of 48 patch suggestions that are exactly the same with the official patches. Third, while not considering multi-thread bugs in our omission-aware model, from the result, FREEWILL can even give two correct patch suggestions and we will give the details in the next section. Finally, while it is conservative, FREEWILL can give more reliable patch suggestions as we have identified three wrong or incomplete official patches but only one from our tool.

6.5 Lessons From Bugs

6.5.1 Mistaken Omission and Inconsistent Decrease

There are totally 36 mistaken omissions caused UAF bugs and we can conclude two main reasons (subtypes) for these bugs. One reason may be that when many programmers participate in the development of a large, complicated program, the one writing a copy functionality does not know how the pointed object manages its lifetime. For example, in `CVE-2011-1260`, the developers, to implement *clone*-like library functions, only simply copy the memory content by calling *memcpy* and fail to increase its member's refcounter properly as they have not cared about the object content. Another reason can be explained by `CVE-2014-1776` in Figure 7. Specifically, when developers pass an `In/Out` object pointer into a large and complicated function, they fail to increase the refcounter but the callee function still decreases the refcounter. Finally, when the caller continues to access the passed out (dangling) pointer, there will be a UAF bug. In fact, this is a common kind of mistaken omission that happens in kernels and script engines.

FREEWILL has detected 12 UAF bugs caused by inconsistent decrease and they can also be classified into two subtypes. We use *(R_c, -)* to represent the first type, e.g., the *RM2* in Figure 1. Considering the obvious reference miscounting, for these bugs, FREEWILL suggests to use *add* + to fix them. As a result, FREEWILL produces five out of six suggestions exactly matched with the official patches and the programmers remove the problematic code to prevent reference creation for the sixth bug (`CVE-2017-2545`). The second subtype, represented by *(+R_c, -, -)*, is more complicated as we cannot precisely tell which decrease is wrong, e.g., the bugs happened in Linux, MacOS, Python, and PHP. While it is less rigorous for our suggestions to prevent the second decrease, there are still five out of six official patches that are matched.

6.5.2 Dangling Usage and Null-Pointer Dereference

FREEWILL confirms 18 UAF bugs caused by two subtypes of dangling usage. First, for the UAF objects whose data structures are simple *Array* and *Buffer* (e.g., `CVE-2020-9892`, `CVE-2010-3971`), FREEWILL does not identify any refcounting during their whole life cycles. Second, if references who have been counted are accessed after the object is freed (e.g., `CVE-2018-8174`), they are also diagnosed as dangling pointers. For both of the dangling bugs, FREEWILL suggests to *stop reuse* related pointers. It is noted that FREEWILL finds no reference miscounting but only dangling usage in *Firefox* and *Chrome*. Based on our manual analysis, both of them have adopted SmartPointer [18] to prevent reference miscounting and the UAF bugs are mainly caused by other raw pointers who have not been protected.

Null-pointer dereference (e.g., `CVE-2017-5404`) can also be treated as a special subtype of dangling usage as a nullified pointer is a special kind of dangling pointer. In fact, all five null-pointer dereference bugs detected by FREEWILL are reported as UAF bugs in the real world. Finally, for these bugs, we suggest adding the corresponding *Null-Check*s, which are all matched with official patches.

6.5.3 Wrong or Incomplete Patches

When there is a problematic reference in the patched program, created and accessed to trigger a UAF bug as in the original code, FREEWILL could confirm the patches to be wrong or incomplete. In this way, FREEWILL identifies three wrong or incomplete patches and one wrong patch suggestion.

Linux (`Commit-81b9de4`) Wrong Patch. This bug is caused by two continuous decrements (as shown in Table 3) and the official patch tries to increase the refcounter before the first decrement. However, the developers wrongly use a reference of another object and this is an obvious wrong patch as the two continuous decrements still exist without any increment.

MacOS (`CVE-2016-1828`) and PHP (`BUG-68594`) Incomplete Patches. These two bugs are very similar as both of them involve the *Unserialize* functions. They are triggered as follow: First, the UAF object, created and held by reference R0, is stored with a key into a table or array without refcounting (refcounter is still 1). Second, a new object, owning the same key with the existing object in the table or array, will be used to *replace* the old object who will be then released (freed as the refcounter becomes 0). Finally, R0 becomes dangling and its access triggers the UAF bug. While this is a classic reference miscounting, both of the official patches chose to prevent the *replace* by adding special check conditions. However, their conditions are incomplete and they can be passed in other new execution paths.

FREEWILL Wrong Patch Suggestion (`Commit-ff11764`) and Multi-thread UAF. The bug is caused by a race problem (one subtype), where an unlocked decrease can concurrently happen in another thread. As FreeWill does not consider races, it cannot produce suggestions to prevent these bugs. However, FreeWill produces two useful Linux patch suggestions (`Commit-357a07c` and `Commit-a4f0377`) almost matched with official patches. In the two bugs (another subtype), threads rely on a synchronized *Producer-Consumer* queue to pass objects. In the wrong case, the *Producer* thread firstly puts the object into the queue and then increases the refcounter; and the *Consumer* firstly fetches the object and then decreases the refcounter. As a result, the increase by Producer and the decrease by Consumer can be executed concurrently, causing a UAF bug. The suggestion to increase refcounter before the object is added into the queue (i.e., a new reference creation) can fix the bugs as increase and decrease can be executed in order. Finally, there is a special simple multi-thread UAF bug (third subtype) in which a dangling pointer is accessed in another thread after the object is freed. In fact, FREEWILL detects this kind of bugs in Chrome and suggests to stop reuse the dangling pointer.

7 Limitation and Future Work

New Bug Detection. FREEWILL is mainly used to diagnose a UAF bug based on a bug-triggering input. However, by concluding the patterns and lessons of reference miscounting, we will focus on finding new bugs in future work.

Race Problem Diagnosis. Based on our evaluations, FREEWILL cannot handle the race problems and it is the main limitation of the current version. While there have been concurrent UAF bug detection methods [24, 31], diagnosing these bugs can be an important future work.

Custom Heap Recognition. FREEWILL failed to analyze two bugs, `CVE-2015-2425` and `CVE-2017-11810`, as we have no idea of their allocation and destruction functions. Therefore, to strengthen FREEWILL, we can adopt reversing methods to recognize custom-defined heap managers.

8 Related Work

8.1 General Root Cause Analysis

Trace Comparison. Miller et al. [36] utilize the binary program analysis toolkits *BitBlaze* [47] to compare benign and malicious traces for root causes analysis. While they can find out some valuable diversities of two traces, it still needs lots of handwork and security expert experience. A recent work, *Aurora* [49], is proposed to automatically identify the diversity of benign and malicious traces to capture the instructions causing behavioral differences which is used as the root cause. However, for UAF bugs, crashing or non-crashing the programs depends on if it chose a buggy path in which the dangling pointer is accessed, and this is not efficient to explain the creation of dangling pointer, i.e., the root cause of UAF.

Data Dependency Analysis. *POMP* [54], *CREDAL* [53], *RETTracer* [28] and *REPT* [27] are proposed to automatically identify the connection between the crash point and the memory corruptions, e.g., the buffer overflow bugs. While by utilizing the core dump and backward taint analysis, these methods are very efficient to find out the reasons why there is a *data dependency mismatch* – treated as the root cause, all of them assume there should be a data dependency between the data creation and final usage. For reference miscounting caused UAF bugs, there is no such data dependency.

8.2 UAF Prevention and Diagnosis

Prevention. Considering the serious impacts of UAF bugs, plenty of solutions have been proposed to prevent the bugs and related attacks. *Undangle* [23], *DangNull* [33], *FreeSentry* [55], *DangSan* [50] and *pSweeper* [34] try to prevent dangling pointers by invalidating them soon after the object is freed. *CETS* [40] and *Oscar* [29] are two representative solutions to detect dangling access by making freed object memory inaccessible. *CRCount* [44] and *MarkUs* [20] delay free operation until they can confirm the object is not pointed by any pointers. *FFmalloc* [52] is designed to be a secure allocator based on the one-time allocation to defeat UAF exploitation. During the prevention, most of these methods can identify all dangling pointers (e.g., *Undangle* and *DangNull*). However, they still rely on lots of manual analysis to find out the reason for dangling creation and access, i.e., the root cause of UAF bugs

Diagnosis. *ASan* [41] and *pSweeper* [34] not only detect the UAF bugs, but also provide as diagnosis results the call stack information for different stages, including the *malloc*, *free* and *reuse*. While these results are helpful for simple bugs (e.g., one used in *pSweeper*), they may fail or even mislead developers in diagnosing reference miscounting caused UAF bugs (explained in §2.2).

8.3 Refcounting Bug Detection

Refcounting and Reference Change Inconsistency. *Referee* [30], *Pungi* [45] and *LinKRID* [35] are proposed to detect refcounting bugs by statically identifying mismatch between refcounting and reference changes. As the wide existence of refcounting omissions, all of them suffer from high false positives. Compared with *Referee*, *Pungi* introduces the *borrow* and *steal* concepts (i.e., refcounting omission), but it does not give any efficient solution. Recent work, *LinKRID*, tries to identify *internal* references which should not be counted during the refcounting invariant checking. Different from our omission-ware model and complete trace-based pointer tracking, *LinKRID* only statically identifies two specific patterns and adopts function summary, which can lead to high false positives and false negatives (explained in §2.2).

Increase and Decrease Inconsistency. *RID* [32] and *CID* [48] are proposed to detect refcounting bugs by statically checking if the consistency of refcounter increase and decrease is broken. Based on their experiments, both of them can detect lots of bugs caused by refcounting errors. However, they can only analyze inconsistent decreases with source code, but are not able to handle mistaken omission bugs in binary-only programs as there is no mismatch problem. Besides, they have not analyzed the reference relationships and the refcounting matching problems.

9 Conclusion

We proposed FREEWILL to automatically diagnose UAF bugs caused by reference miscounting. With an omission-aware refcounting model, FREEWILL incorporates several practical techniques to find the root cause. We prototyped FREEWILL and evaluated it with 76 UAF bugs from real-world programs. The experimental results show that FREEWILL is effective and practical on automatically diagnosing UAF bugs caused by reference miscounting.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Antonio Bianchi, for their helpful feedback. We also thank Chao Zhang for helpful discussion on the heuristic rules. We also thank Yanhang Wang and Xiangkun Jia for discussion on paper's motivation and the omission model. This research was supported, in part, by National Natural Science Foundation of China (Grand No. U1936211, U1836117, U1836113, 61902384 and 62102406), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (Grand No. 2019111 and 2017151), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDC02020300), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006) and Ministry of Education, Singapore (Grant No. MOE2018-T2-1-142). All opinions expressed in this paper are solely those of the authors.

References

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] Breakpad. https://chromium.googlesource.com/breakpad/breakpad/+/master/docs/getting_started_with_breakpad.md.

[3] Bugzilla. https://bugzilla.mozilla.org.

[4] Chromium Bugs. https://bugs.chromium.org/p/chromium/issues/list.

[5] Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[6] CrashPad. https://chromium.googlesource.com/crashpad/crashpad/+/refs/heads/main/README.md.

[7] CWE-911: Improper Update of Reference Count. https://cwe.mitre.org/data/definitions/911.html.

[8] DynamoRIO - Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/.

[9] ETHREAD Structure. https://www.nirsoft.net/kernel_struct/vista/ETHREAD.html.

[10] Kernel Boot Problem. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0711f0d7050b9e07c44bc159bbc64ac0a1022c7f.

[11] Linux Kernel Commit Log. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/.

[12] Minidump Files. https://docs.microsoft.com/en-us/windows/win32/debug/minidump-files.

[13] PDB File. https://devblogs.microsoft.com/cppblog/whats-inside-a-pdb-file/.

[14] PHP Bug Tracking System. https://bugs.php.net.

[15] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[16] Python Bug Tracker. https://bugs.python.org/.

[17] Rules For Managing Reference Counts. https://docs.microsoft.com/zh-cn/windows/desktop/com/rules-for-managing-reference-counts.

[18] Smart Pointer. https://en.wikipedia.org/wiki/Smart_pointer.

[19] Zero Day Initiative. https://www.zerodayinitiative.com/.

[20] Sam Ainsworth and Timothy Jones. MarkUs: Drop-in Use-after-free Prevention for Low-level Languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[21] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium*, 2018.

[22] Fabrice Bellard. QEMU, A Fast And Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[23] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection Of Dangling Pointers In Use-After-Free and Double-Free Vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2012.

[24] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[25] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform For In-vivo Multi-path Analysis Of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[26] George E. Collins. A Method For Overlapping And Erasure Of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[27] Weidong Cui, Xinyang Ge, Baris Kasikei, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: Revrse Debugging Of Failures In Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[28] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P.Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[29] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme For Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[30] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying Reference Counting Implementations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[31] Jeff Huang. UFO: Predictive Concurrency Use-After-Free Detection. In *Proceedings of the 2018 ACM/IEEE 40th International Conference on Software Engineering*, 2018.

[32] Mao Junjie, Chen Yu, Xiao Qixue, and Shi Yuanchun. Rid: Finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architecture Support for Programming Lagnuages and Operating Systems*, 2016.

[33] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free With Dangling Pointers Nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, 2015.

[34] Daiping Liu, Mingwei Zhang, and Haining Wang. A Robust And Efficient Defense Against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, 2018.

[35] Jian Liu, Lin Yi, Weiteng Chen, Chenyu Song, Zhiyun Qian, and Qiuping Yi. LinKRID: Vetting Imbalance Reference Counting in Linux kernel with Symbolic Execution. In *Proceedings of the 31th USENIX Security Symposium*, 2022.

[36] Charlie Miller, Juan Caballero, Noah M. Johnson, Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis with bitblaze. In *Blackhat*, 2010.

[37] Matt Miller. Trends, Challenges, And Strategic Shifts In The Software Vulnerability Mitigation Landscape. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, 2019. BlueHat IL.

[38] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware For Safe And Secure Manual Memory Management And Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[39] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety For C. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[42] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down For The Count? Getting Reference Counting Back In The Ring. In *ACM SIGPLAN Notices*, volume 47, pages 73–84. ACM, 2012.

[43] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. Taking off the gloves with reference counting immix. In *ACM SIGPLAN Notices*, volume 48, pages 93–110. ACM, 2013.

[44] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. CRCount: Pointer Invalidation With Reference Counting To Mitigate Use-after-free in Legacy C/C++. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium*, 2019.

[45] Li Siliang and Tan Gang. Finding reference-counting errors in python/c programs with affine analysis. In *Proceedings of European Conference on Object-Oriented Programming*, 2014.

[46] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring The Spatial And Temporal Memory Safety Of C At Runtime. *Softw. Pract. Exper.*, 43(1), 2013.

[47] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach To Computer Security Via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[48] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *Proceedings of the 30th USENIX Security Symposium.*, 2021.

[49] Blazytko Tim, Schlogel Moritz, Aschermann Cornelius, Abbasi Ali, Frank Joel, Worner Simon, and Holz Thorsten. Aurora: Statistical crash analysis for automated root cause explanation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[50] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[51] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-Independent Dynamic Information Flow Tracking. In *22nd International Conference on Compiler Construction*, 2013.

[52] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[53] Jun Xu, , Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. CREDAL: Towards Locating A Memory Corruption Vulnerability With Your Core Dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[54] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[55] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, 2015.

[56] Park Young Gil and Goldberg Benjamin. Reference escaple analysis: Optimizing reference counting based on the lifetime of references. In *ACM/IFIP Conference on Partial Evaluation and Program Manipulation*. ACM, 1991.

[57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

A Appendix

A.1 More Details And Performance

We present more details and performance in Table 5. The left half of the table presents the details of *Reference Type*, *Refcounting Matching*, *Dangling Pointers*, and *Bug Type*. As the diagnosis result, we have marked the problematic references with red color. From the *Performance* result, we can see that, on average, FREEWILL can even make a correct root analysis in no more than 15 minutes.

Table 5: **Details of Diagnoses and Performance.** ***DP***=Dangling Pointer, ***RCM***=Refcounting Matching, ***MO***=Mistaken Omission, ***ID***=Inconsistent Decrease, ***DU***=Dangling Usage, ***CR***=Crash Reproduction, ***UI***=UAF Identification. **N** means the UAF bug is caused by one of these references.

Prog.	BugId	Ref. Type		RCM				DP	Bug	Performance			
		Stack	Heap	(+, -)	(+,)	(, -)	(,)			CR	UI	Diag.	Total
IE	CVE-2010-0248	29	**2**	1	0	0	**30**	1	MO	1m41s	51s	18s	2m50s
	CVE-2010-0249	503	**158**	44	12	12	**593**	7	MO	2m43s	1m1s	13m57s	17m41s
	CVE-2010-3971	**9**	1	–	–	–	–	**3**	DU	58s	1m46s	0.6s	2m45s
	CVE-2011-1260	106	**57**	0	1	1	**161**	10	MO	4m17s	1m37s	1m5s	6m59s
	CVE-2012-1785	200	**10**	12	0	0	**198**	8	MO	3m30s	4m20s	5m28s	13m18s
	CVE-2012-4787	9	**4**	8	0	0	**5**	1	MO	1m27s	1m13s	1m27s	4m7s
	CVE-2012-4792	15	**7**	6	1	1	**14**	5	MO	2m46s	1m21s	3m39s	7m46s
	CVE-2012-4969	**27**	5	3	0	0	**29**	8	MO	2m46s	30s	4m16s	7m32s
	CVE-2013-0025	131	**134**	1	2	2	**260**	2	MO	2m28s	1m8s	3m31s	7m7s
	CVE-2013-1306	67	**7**	**7**	0	0	67	**6**	DU	8m12s	10m25s	10m21s	28m58s
	CVE-2013-1347	12	**6**	**7**	0	0	11	**5**	DU	1m59s	1m23s	1m56s	5m18s
	CVE-2013-3163	302	**20**	19	0	0	**303**	7	MO	3m11s	8m10s	3m16s	14m37s
	CVE-2013-3893	**630**	133	23	0	0	**740**	12	MO	2m13s	2m28s	6m7s	10m48s
	CVE-2013-3897	**73**	4	1	0	0	**76**	8	MO	3m10s	1m30s	1m12s	5m54s
	CVE-2014-0282	484	**24**	28	1	1	**478**	8	MO	1m57s	48s	6m10s	8m55s
	CVE-2014-1776	**409**	54	19	7	7	**430**	9	MO	3m50s	5m23s	17m57s	27m10s
	CVE-2014-1815	**908**	54	62	3	3	**594**	10	MO	2m42s	1m37s	15m52s	20m11s
	CVE-2015-2425	–	–	–	–	–	–	–	–	3m57s	–	–	–
	CVE-2017-11810	–	–	–	–	–	–	–	–	2m12s	–	–	–
	CVE-2018-8174	29	**7**	**6**	3	3	25	**3**	DU	1m47s	2m32s	6m32s	10m51s
Linux	Commit-81b9de4	**10**	0	1	0	**1**	9	2	ID	8s	0.4s (A)	0.6s	9s
	Commit-c3aabf0	**8**	1	0	1	**1**	0	7	ID	8s	0.1s (A)	0.5s	9s
	Commit-0cef13d	**15**	0	1	0	**1**	14	2	ID	8s	0.1s (A)	0.6s	9s
	Commit-8cc0dcf	**59**	1	1	1	**1**	57	13	ID	1m13s	13s	2s	1m28s
	Commit-ba34c3d	**53**	2	2	0	0	**51**	10	MO	2m49s	11s	3s	3m3s
	Commit-dbfa04e	–	–	–	–	–	–	–	NPD	2m16s	–	–	–
	Commit-beb691e	46	**2**	1	1	**1**	46	**11**	ID	2m47s	8s	2s	2m57s
	Commit-357a07c	6	**1**	0	1	1	**5**	3	MO	2m19s	8s	3s	2m30s
	Commit-8d43259	55	**3**	**3**	0	0	55	**9**	DU	2m38s	7s	3s	2m48s
	Commit-ae4393d	**6**	1	–	–	–	–	**5**	DU	2m9s	8s	3s	2m20s
	Commit-c81d3d2	27	**1**	0	1	1	**26**	8	MO	2m7s	3s	3s	2m13s
	Commit-a4f0377	52	**3**	0	1	1	**53**	16	MO	2m15s	7s	2s	2m24s
	Commit-8fd0e99	23	**2**	0	1	**1**	23	5	ID	2m18s	7s	2s	2m27s
	Commit-af35fc3	**51**	1	0	1	1	**50**	20	MO	3m1s	8s	2s	3m10s
	Commit-ff11764	20	**2**	0	1	1	**20**	13	MO	2m38	8s	3s	2m49s
MacOS	CVE-2016-1828	36	**2**	2	0	0	**36**	14	MO	26s	0.3s	0.9s	28s
	CVE-2016-4656	39	**1**	1	0	0	**39**	13	MO	25s	0.3s	0.9s	27s
	CVE-2017-2545	21	**1**	0	1	**1**	20	7	ID	4m15s	8s	3s	4m26s
	CVE-2018-4083	26	**1**	1	0	**1**	26	10	ID	4m19s	11s	9s	4m39s
	CVE-2019-6225	66	**1**	1	2	**2**	62	5	ID	2m8s	12s	3s	2m23s
	CVE-2020-9892	17	**1**	–	–	–	–	**6**	DU	2m7s	11s	2s	2m20s
Python	Issue-24091	**40**	6	5	1	1	**39**	11	MO	18m38s	13s	3s	18m54s
	Issue-24092	–	–	–	–	–	–	–	NPD	18m37s	–	–	–
	Issue-24093	60	**21**	8	2	**2**	69	13	ID	18m38s	13s	2s	18m51s
	Issue-24094	**47**	14	7	1	1	**52**	11	MO	13m23s	13s	3s	13m39s
	Issue-24099	50	**17**	5	1	**1**	61	13	ID	12m38s	13s	3s	12m54s
	Issue-24100	–	–	–	–	–	–	–	NPD	13m15s	–	–	–
	Issue-38588-01	**15**	6	1	2	2	**16**	12	MO	3m52s	0.8s	0.5s	3m53s
	Issue-38588-02	**11**	3	1	1	1	**11**	8	MO	3m12s	0.7s	0.7s	3m14s
	Issue-38588-03	**11**	4	1	1	1	**12**	9	MO	3m10s	0.9s	0.4s	3m12s
	Issue-38610-01	**12**	5	1	1	1	**14**	9	MO	3m12s	0.9s	0.7s	3m14s
	Issue-38610-02	**12**	6	2	1	1	**14**	9	MO	3m12s	0.8s	0.6s	3m13s
	Issue-38610-03	**12**	6	2	1	1	**14**	9	MO	3m12s	0.9s	0.7s	3m14s
	Issue-39421	**13**	6	2	1	1	**16**	10	MO	3m16s	0.8s	0.7s	3m18s
	Issue-39453	**12**	6	1	1	1	**15**	9	MO	3m14s	0.7s	0.5s	3m15s
	Issue-39510	–	–	–	–	–	–	–	NPD	18m38s	–	–	–
PHP	BUG-68594	3	**2**	0	1	1	**3**	2	MO	21m38s	1s (A)	3s	21m42s
	BUG-68710	3	**2**	0	1	1	**3**	2	MO	21m35s	1s (A)	2s	21m38s
	BUG-73392	**12**	1	1	1	1	**10**	3	MO	2m14s	3s	2s	2m19
	OSS-17903	–	–	–	–	–	–	–	–	–	–	–	–
	OSS-24436	18	**2**	0	1	1	**18**	7	MO	2m42s	5s	2s	2m49s
	BUG-79922	20	**6**	3	1	**1**	22	6	ID	2m48s	4s	2s	2m54s
	BUG-66783	21	**8**	–	–	–	–	**5**	DU	2m11s	3s	3s	2m17s
	BUG-80927	4	**2**	–	–	–	–	**2**	DU	2m38s	5s	3s	2m46s
Firefox	CVE-2011-0065	4	**2**	–	–	–	–	**5**	DU	6m25s	2m11s	43s	9m19s
	CVE-2013-0753	**76**	4	–	–	–	–	**6**	DU	2m32s	23s	11s	3m6s
	CVE-2016-9899	–	–	–	–	–	–	–	–	–	–	–	–
	CVE-2017-5404	–	–	–	–	–	–	–	NPD	105m20s	–	–	–
	CVE-2018-18492	**16**	8	–	–	–	–	**24**	DU	22m39s	1m10s	21s	24m10s
Chrome	Issue-1033759	392	**31**	–	–	–	–	**8**	DU	4m15s	1m39s	21s	6m15s
	Issue-1041406	11	**5**	–	–	–	–	**2**	DU	13m11s	5m12s	32s	18m55s
	Issue-1062091	413	**104**	–	–	–	–	**79**	DU	118m10s	19m23s	18s	137m51s
	Issue-1111737	–	–	–	–	–	–	–	–	–	–	–	–
	Issue-1137630	0	**2**	–	–	–	–	**1**	DU	5m32s	1m54s	23s	7m49s
	Issue-1142675	14	**2**	–	–	–	–	**2**	DU	21m54s	3m19s	19s	25m32s
	Issue-1155426	432	**53**	–	–	–	–	**43**	DU	128m13s	23m15s	12s	151m39s
Total:		**36 MO, 12 ID, 18 DU, 5 NPD**									**Average:**		11m58s