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

Liang He, TCA, Institute of Software, Chinese Academy of Sciences; Hong Hu, Pennsylvania State University; Purui Su, TCA / SKLCS, Institute of Software, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences; Yan Cai, SKLCS, Institute of Software, Chinese Academy of Sciences; Zhenkai Liang, National University of Singapore

https://www.usenix.org/conference/usenixsecurity22/presentation/he-liang

This paper is included in the Proceedings of the 31st USENIX Security Symposium.
August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.
**FREEWILL: Automatically Diagnosing Use-after-free Bugs via Reference Miscounting Detection on Binaries**

Liang He¹*, Hong Hu²*, Purui Su¹,³,⁴*, Yan Cai³, Zhenkai Liang⁵

¹TCA / ³SKLCS, Institute of Software, Chinese Academy of Sciences  
²Pennsylvania State University  
⁴School of Cyber Security, University of Chinese Academy of Sciences  
⁵National 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 mainstream 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 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 references as no reference miscounting, we report all dangling pointers as no reference miscounting.

**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), 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.

```c
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;
14.   B* pB=(B*)malloc(N3);
15.   pB->_pA = pA;
16.   gpB=(B*)malloc(N3);
17.   clone(gpB, pB);
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.**
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.

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 16), 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 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 effectivly 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 \( R_i \rightarrow \text{addRef}(); \) \( R_j = R_i \); and \( R_j = R_i; \) \( R_j \rightarrow \text{addRef}() \); in the same program to count a new created reference \( R_j \). 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 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 \( R_1 \) and \( R_j \), if \( R_i.T_c < R_j.T_c < R_i.T_d < R_j.T_d \), we say the two references are overlapped, and - of \( R_1 \) and + of \( R_j \) can be safely omitted. Any reference can only have one overlapping reference.

**OR2: Transmitting-Overlapping Omission.** For any number of references \( R_1, \ldots, R_N \), if for any reference \( R_i \) \((1 < i < N)\), \( R_i - 1 . T_c < R_i . T_c < R_i - 1 . T_d < R_i . T_d \), we say these references are transmitting-overlapped, and in this case, - of \( R_1 \), (+, -) of \( R_i \) \((1 < i < N)\), and + of \( R_N \) can be safely omitted.

**OR3: Containing Omission.** For two references \( R_1 \) and \( R_j \), if \( R_i . T_c < R_j . T_c < R_j . T_d < R_i . T_d \), we say \( R_j \) is contained in \( R_1 \) and (+, -) of \( R_j \) can be safely omitted.

**OR4: Overlapping-Containing Omission.** For three references \( R_i, R_m \) and \( R_j \), if \( R_i \) and \( R_j \) are overlapped based on OR1 or OR2, and \( R_i . T_c < R_m . T_c < R_j . T_c \) and \( R_i . T_d < R_m . T_d < R_j . T_d \), we say \( R_m \) is overlapping-contained in \( R_1 \) and \( R_j \) 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 \( R_0 \)–\( R_4 \) have been detected. Then, based on rule OR1, as \( R_0 \) and \( R_1 \) are overlapped, we can tell the omission of - of \( R_0 \) and + of \( R_1 \) are safe. Similarly, based on rule OR4, \( R_2 \) is overlapping-contained in \( R_0 \) and \( R_1 \), so the omission of (+, -) of \( R_2 \) is allowed. However, as \( R_3 \) is not contained in any other reference, its omission of (+, -) is wrong. Besides, \( R_4 \) 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, refcounting analysis, refcounting detection, and UAF diagnosis.
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.

### 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-tainting 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 X -> 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.

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.
4.4.1 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 execution order, denoted as Instruction ID (§5.1). Then, as shown on the top of Figure 6, we use \( ED = \|N_{count} - N_{ref}\| \) 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\( \rightarrow \text{ref}=1 \) with the closest \( \text{pA}=\text{malloc()} \). Similarly, the decrease \( \text{dec_ref}(\text{gpA}) \) 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., \( WD = \| N_{RP,\text{Current}} - N_{EP,\text{Next}} \| \). In Figure 6, to match the + with proper reference, \( R_1 \) or \( R_2 \), although there is \( ED_1 > ED_2 \), considering the existence of wrappers, we will calculate the \( WD \) and match it with \( R_1 \) as there is \( WD_1 < WD_2 \).

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 \( ED_{\text{wrapper}} = \| N_{ret} - N_{call} \| \) 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 \( ED_P \) from the call to ret of \( P \) and then subtract it from \( ED_{\text{wrapper}} \) 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 ID, 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 \( inc \_set \) (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., \( r'.T_0 < r.T_0 \)). If \( r' \) exists (Lines 11-13), \( r'.- \) and \( r.++ \) are allowed to be False. We will create a virtual reference \( vr: (+, True, .T_0=r'.T_0, -, True, .T_0=r . T_0) \) and add it into \( ov \_set \). We will remove \( r' \) from \( inc \_set \) as it can only be matched with once based on OR1.
4) 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 (R1 in OR2). Based on OR2, it is legitimate to omit \( r'.- \), \(+ \), and \(- \) of intermediate references (i.e., \( R1 \) for \( 1 < i < N \)). We will create the virtual reference, remove \( r' \) from \( inc \_set \) and remove \( R1 \) from \( ref \_set \) (Lines 18-21).
5) 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).
6) 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 \( R0 \_BUG \), 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.
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.

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 < r.T < r',.T then // OR1 |
| 10 vr = (True, r',.T, True, r,.T) |
| 11 add vr into ov_set |
| 12 remove r' from inc_set |
| 13 else if ∃r1 ... RN ∈ ref_set: ∀1<i<N |
| 14 Ri-1.T < ri.Ti < Ri-1.T < Ri.Ti and !(ri,+ and !ri,-), and Ri is r and Ri is r' and r' ∈ inc_set |
| 15 then // OR2 |
| 16 vr = (True, r',.T, True, r,.T) |
| 17 add vr into ov_set |
| 18 remove r' from inc_set |
| 19 remove Ri (1<i<N) from ref_set |
| 20 else |
| 21 report ID BUG; // report |
| 22 add + for r; // patch sugg. |
| 23 foreach r ∈ ref_set: !r,+ and !r,- do |
| 24 if ∃r' ∈ rc_set: r'.T < r.T < r',.T then // OR3 |
| 25 continue to next r |
| 26 else if ∃r' ∈ ov_set: r'.T < r.T < r',.T then // OR4 |
| 27 continue to next r |
| 28 else |
| 29 report NO BUG; // report |
| 30 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 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/Deserialize 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].

**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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2010-0248</td>
<td>8.0.7600.16385</td>
<td>1.6</td>
<td>✓</td>
<td>2</td>
<td>188 31 1 1+0x10</td>
<td>✓</td>
<td>M(R, r)</td>
</tr>
<tr>
<td>CVE-2010-0249</td>
<td>8.0.7600.16385</td>
<td>2.7</td>
<td>✓</td>
<td>1</td>
<td>8460 661 56+0x40</td>
<td>✓</td>
<td>RD(R, r)</td>
</tr>
<tr>
<td>CVE-2010-3971</td>
<td>8.0.7600.16385</td>
<td>3.9</td>
<td>✓</td>
<td>1</td>
<td>52 10 0</td>
<td>No</td>
<td>Dangling (No RC) Stop Reuse</td>
</tr>
<tr>
<td>CVE-2011-11630</td>
<td>8.0.7600.16385</td>
<td>2.5</td>
<td>✓</td>
<td>MU</td>
<td>0</td>
<td>1266 163 1+0x40</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2012-1875</td>
<td>8.0.7600.16385</td>
<td>3.8</td>
<td>✓</td>
<td>0</td>
<td>273 210 12+0x40</td>
<td>✓</td>
<td>M(R, r)</td>
</tr>
<tr>
<td>CVE-2012-4787</td>
<td>9.0.8112.16421</td>
<td>1.4</td>
<td>✓</td>
<td>0</td>
<td>313 18 3 1+0x40</td>
<td>✓</td>
<td>M(R, r)</td>
</tr>
<tr>
<td>CVE-2012-4792</td>
<td>8.0.7600.16385</td>
<td>3.3</td>
<td>✓</td>
<td>0</td>
<td>1026 22 7+0x40</td>
<td>✓</td>
<td>M(R, r)</td>
</tr>
<tr>
<td>CVE-2012-4969</td>
<td>8.0.7600.16385</td>
<td>3.3</td>
<td>✓</td>
<td>0</td>
<td>200 32 3+0x40</td>
<td>✓</td>
<td>P(R, r)</td>
</tr>
<tr>
<td>CVE-2013-0025</td>
<td>8.0.7600.16385</td>
<td>2.7</td>
<td>✓</td>
<td>MU</td>
<td>0</td>
<td>2973 265 3+0x40</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2013-1306</td>
<td>9.0.8112.16421</td>
<td>18.0</td>
<td>✓</td>
<td>0</td>
<td>1977 74 7+0x40</td>
<td>No</td>
<td>Reuse Counted R Stop Reuse</td>
</tr>
<tr>
<td>CVE-2013-1344</td>
<td>8.0.7600.16385</td>
<td>1.7</td>
<td>✓</td>
<td>0</td>
<td>928 18 7+0x40</td>
<td>No</td>
<td>Reuse Counted R Stop Reuse</td>
</tr>
<tr>
<td>CVE-2013-3163</td>
<td>8.0.7600.16385</td>
<td>1.9</td>
<td>✓</td>
<td>0</td>
<td>2271 322 19+0x40</td>
<td>✓</td>
<td>M(R, r)</td>
</tr>
<tr>
<td>CVE-2013-3893</td>
<td>8.0.7600.16385</td>
<td>2.4</td>
<td>✓</td>
<td>MU</td>
<td>1</td>
<td>5110 763 23+0x40</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2013-3897</td>
<td>8.0.7600.16385</td>
<td>2.5</td>
<td>✓</td>
<td>1</td>
<td>431 77 1+0x40</td>
<td>✓</td>
<td>P(R, r)</td>
</tr>
<tr>
<td>CVE-2014-0282</td>
<td>8.0.7600.16385</td>
<td>1.8</td>
<td>✓</td>
<td>0</td>
<td>4104 508 29+0x40</td>
<td>✓</td>
<td>M(R, r)</td>
</tr>
<tr>
<td>CVE-2014-1776</td>
<td>8.0.7600.16385</td>
<td>6.2</td>
<td>✓</td>
<td>MU</td>
<td>1</td>
<td>3660 463 26+0x40</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2014-1815</td>
<td>8.0.7600.16385</td>
<td>2.5</td>
<td>✓</td>
<td>0</td>
<td>5021 962 65+0x40</td>
<td>✓</td>
<td>P(R, r)</td>
</tr>
<tr>
<td>CVE-2015-9600</td>
<td>11.0.9600.24285</td>
<td>6.8</td>
<td>✓</td>
<td>0</td>
<td>243 16 0+0x40</td>
<td>✓</td>
<td>P(R, r)</td>
</tr>
<tr>
<td>CVE-2017-1180</td>
<td>11.0.9600.16428</td>
<td>3.1</td>
<td>✓</td>
<td>0</td>
<td>190 32 1+0x40</td>
<td>✓</td>
<td>P(R, r)</td>
</tr>
<tr>
<td>CVE-2018-8714</td>
<td>11.0.9600.16428</td>
<td>3.7</td>
<td>✓</td>
<td>0</td>
<td>217 36 9+0x40</td>
<td>No</td>
<td>Reuse Counted R Stop Reuse</td>
</tr>
</tbody>
</table>

**Commit-819964 (2018)**
- 4.16.0+rc1
  - DF(A): 0
  - DF(A): 0
- Add +/– Add

**Linux**
- 5.12.0
  - DF(A): 0
  - DF(A): 0
- Add +/– Add

**MacOS**
- 10.11.6
  - DF(A): 0
  - DF(A): 0
- Add +/– Add

**Python**
- 3.8.0
  - DF(A): 0
  - DF(A): 0
- Add +/– Add

**PHP**
- 7.4.0
  - No
  - 0
  - 0
  - 0
- Add +/– Add

**Firefox**
- 9.0.4067.0 (2020)
  - 81.0.0+rc1
  - 81.0.0+rc1
  - 81.0.0+rc1
  - 81.0.0+rc1
- Add +/– Add

**Chrome**
- 68.0.3597.0
  - 67.0.3396.0
  - 67.0.3396.0
  - 67.0.3396.0
- Add +/– Add

**USENIX Association**
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 takes 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., 0x80000 or 0xffffffff) to the refcounter before they free the objects.

HR-CDFree FP/FN Analysis: The reason of false positives is that it cannot 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 0xffffffff as the free condition (this has been fixed). Nevertheless, 6% false negative and 98% accuracy rate for more than 50 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, CFiatMarkupPointer, 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 (Rc, -) 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 48 UAF bugs caused by mistaken omission. We use (Rc, -) to mean the inconsistency problems as there is only - but no corresponding +, which is the main reason for 6 UAF bugs. We use (+Rc, -, -) 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-(Rc, ) 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 1. 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 callee 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+, -) to represent the first type, e.g., the KM2 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-, -), 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-2016-3971), 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-Checks, 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 pro-
gram 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 in-
structions 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], FreeSen-
try [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. CRCCount [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 ex-
ploration. 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. Ref-
erever [30], Pungi [45] and LinKRI
d [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 pos-
tives. Compared with Referee, Pungi introduces the borrow and steal concepts (i.e., refcounting omission), but it does not give any efficient solution. Recent work, LinKRI
d, 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 track-
ing, LinKRI
d 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. How-
ever, 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. Be-
sides, 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. 20191111 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


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, Ref-counting 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.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>BugId</th>
<th>Ref. Type</th>
<th>RCM</th>
<th>DP</th>
<th>Bug</th>
<th>CR</th>
<th>UI</th>
<th>Diag.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Commit-5189dc6</td>
<td>10 0</td>
<td>1 0 1 9 2</td>
<td>ID</td>
<td>8x</td>
<td>0.4s (A)</td>
<td>0.6s</td>
<td>9s</td>
<td></td>
</tr>
<tr>
<td>MacOS</td>
<td>Commit-3a9db0</td>
<td>8 1</td>
<td>0 1 0 0 7</td>
<td>ID</td>
<td>8x</td>
<td>0 1s (A)</td>
<td>0.5s</td>
<td>9s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-ac1c1f1d</td>
<td>15 0</td>
<td>0 1 0 14 2</td>
<td>ID</td>
<td>8x</td>
<td>0.1s (A)</td>
<td>0.6s</td>
<td>9s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-8c83cdcc</td>
<td>59 1</td>
<td>1 1 1 57 13</td>
<td>ID</td>
<td>1m13s</td>
<td>1s</td>
<td>2s</td>
<td>1m28s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-ba34cd3c</td>
<td>53 2</td>
<td>0 0 0 51 10</td>
<td>ID</td>
<td>2m49s</td>
<td>1s</td>
<td>3s</td>
<td>3m3s</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>Commit-b6f09bfe</td>
<td>46 2</td>
<td>1 1 1 46 11</td>
<td>ID</td>
<td>2m47s</td>
<td>8s</td>
<td>2s</td>
<td>2m57s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-357a07c</td>
<td>6 1</td>
<td>0 1 1 5 3</td>
<td>MO</td>
<td>2m19s</td>
<td>8s</td>
<td>3s</td>
<td>2m30s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-54d3f259</td>
<td>55 3</td>
<td>3 0 0 55 9</td>
<td>DU</td>
<td>2m38s</td>
<td>7s</td>
<td>3s</td>
<td>2m48s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-ac435d9a</td>
<td>6 1</td>
<td>– – – –</td>
<td>MO</td>
<td>2m9s</td>
<td>8s</td>
<td>3s</td>
<td>2m20s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-c8143d42</td>
<td>26 0</td>
<td>1 1 1 26 8</td>
<td>MO</td>
<td>2m7s</td>
<td>3s</td>
<td>3s</td>
<td>2m13s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-4a0f377</td>
<td>52 3</td>
<td>0 1 1 53 16</td>
<td>ID</td>
<td>2m15s</td>
<td>7s</td>
<td>2s</td>
<td>2m24s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-d68f699</td>
<td>23 2</td>
<td>0 1 1 23 5</td>
<td>ID</td>
<td>2m18s</td>
<td>7s</td>
<td>2s</td>
<td>2m27s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-at25a7e</td>
<td>47 1</td>
<td>0 1 1 56 20</td>
<td>MO</td>
<td>3m1s</td>
<td>8s</td>
<td>2s</td>
<td>3m10s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-f17764</td>
<td>20 2</td>
<td>0 1 1 20 13</td>
<td>MO</td>
<td>2m38s</td>
<td>8s</td>
<td>3s</td>
<td>2m49s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24091</td>
<td>40 6</td>
<td>5 1 1 39 11</td>
<td>ID</td>
<td>1m38s</td>
<td>1s</td>
<td>3s</td>
<td>1m54s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24092</td>
<td>– – – –</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24093</td>
<td>60 21</td>
<td>8 2 2 69 13</td>
<td>ID</td>
<td>1m38s</td>
<td>1s</td>
<td>2s</td>
<td>1m51s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24094</td>
<td>47 10</td>
<td>7 1 1 53 10</td>
<td>ID</td>
<td>1m31s</td>
<td>3s</td>
<td>3s</td>
<td>1m35s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24099</td>
<td>50 17</td>
<td>5 1 1 61 13</td>
<td>ID</td>
<td>1m32s</td>
<td>1s</td>
<td>3s</td>
<td>1m54s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24140</td>
<td>– – – –</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-38588-01</td>
<td>15 6</td>
<td>1 1 2 16 2</td>
<td>MO</td>
<td>3m52s</td>
<td>0.8s</td>
<td>5s</td>
<td>3m53s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-38588-02</td>
<td>11 3</td>
<td>1 1 1 11 8</td>
<td>MO</td>
<td>3m12s</td>
<td>0.7s</td>
<td>0.7s</td>
<td>3m14s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-38588-03</td>
<td>11 4</td>
<td>1 1 1 12 9</td>
<td>MO</td>
<td>3m10s</td>
<td>0.9s</td>
<td>0.4s</td>
<td>3m12s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-38610-01</td>
<td>12 5</td>
<td>1 1 1 14 9</td>
<td>MO</td>
<td>3m12s</td>
<td>0.9s</td>
<td>0.7s</td>
<td>3m14s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-38610-02</td>
<td>12 6</td>
<td>2 1 1 14 9</td>
<td>MO</td>
<td>3m12s</td>
<td>0.8s</td>
<td>0.6s</td>
<td>3m13s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-38610-03</td>
<td>12 6</td>
<td>2 1 1 14 9</td>
<td>MO</td>
<td>3m12s</td>
<td>0.9s</td>
<td>0.7s</td>
<td>3m14s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-39421</td>
<td>13 6</td>
<td>2 1 1 16 10</td>
<td>MO</td>
<td>3m16s</td>
<td>0.8s</td>
<td>0.7s</td>
<td>3m18s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-39453</td>
<td>13 6</td>
<td>1 1 1 15 9</td>
<td>MO</td>
<td>3m14s</td>
<td>0.7s</td>
<td>0.5s</td>
<td>3m15s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-39510</td>
<td>– – – –</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-685894</td>
<td>3 2</td>
<td>0 1 1 3 2</td>
<td>MO</td>
<td>2m38s</td>
<td>1s (A)</td>
<td>3s</td>
<td>2m42s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-687504</td>
<td>3 2</td>
<td>0 1 1 3 2</td>
<td>MO</td>
<td>2m35s</td>
<td>1s (A)</td>
<td>2s</td>
<td>2m38s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-73392</td>
<td>12 1</td>
<td>1 1 1 10 3</td>
<td>MO</td>
<td>2m14s</td>
<td>3s</td>
<td>2s</td>
<td>2m19s</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>Issue-17903</td>
<td>– – – –</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-24436</td>
<td>18 2</td>
<td>0 1 1 18 7</td>
<td>MO</td>
<td>2m42s</td>
<td>5s</td>
<td>2s</td>
<td>2m49s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-79222</td>
<td>20 6</td>
<td>3 1 1 22 6</td>
<td>ID</td>
<td>2m48s</td>
<td>4s</td>
<td>2s</td>
<td>2m54s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-66783</td>
<td>21 6</td>
<td>– – – –</td>
<td>DU</td>
<td>2m11s</td>
<td>3s</td>
<td>2s</td>
<td>2m17s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-80927</td>
<td>4 2</td>
<td>– – – –</td>
<td>DU</td>
<td>2m38s</td>
<td>5s</td>
<td>3s</td>
<td>2m46s</td>
<td></td>
</tr>
<tr>
<td>Firefox</td>
<td>Issue-10053</td>
<td>11</td>
<td>– – – –</td>
<td>EU</td>
<td>6m25s</td>
<td>2m11s</td>
<td>43s</td>
<td>9m19s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-1093753</td>
<td>76 4</td>
<td>– – – –</td>
<td>DU</td>
<td>2m32s</td>
<td>3s</td>
<td>11s</td>
<td>3m68s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-2016-9699</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-2017-5404</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-2018-18492</td>
<td>16 8</td>
<td>– – – –</td>
<td>DU</td>
<td>2m39s</td>
<td>1m10s</td>
<td>21s</td>
<td>2m40s</td>
<td></td>
</tr>
<tr>
<td>Chrome</td>
<td>Issue-1035759</td>
<td>392 31</td>
<td>– – – –</td>
<td>DU</td>
<td>4m15s</td>
<td>1m39s</td>
<td>21s</td>
<td>6m15s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-104a0106</td>
<td>11 5</td>
<td>– – – –</td>
<td>DU</td>
<td>1m11s</td>
<td>5m22s</td>
<td>32</td>
<td>18m55s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-1062091</td>
<td>413 104</td>
<td>– – – –</td>
<td>DU</td>
<td>1m18s</td>
<td>9m23s</td>
<td>18s</td>
<td>13m73s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-111777</td>
<td>– – – –</td>
<td>– –</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-1137630</td>
<td>0 2</td>
<td>– – – –</td>
<td>DU</td>
<td>5m32s</td>
<td>1m54s</td>
<td>23s</td>
<td>7m49s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-1142675</td>
<td>14 2</td>
<td>– – – –</td>
<td>DU</td>
<td>2m54s</td>
<td>3m19s</td>
<td>19s</td>
<td>2m32s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue-1155424</td>
<td>432 53</td>
<td>– – – –</td>
<td>DU</td>
<td>12m31s</td>
<td>23m15s</td>
<td>12s</td>
<td>15m139s</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>36</td>
<td>12</td>
<td>18</td>
<td>5</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>11m38s</td>
<td></td>
</tr>
</tbody>
</table>