UAFSan: An Object-Identifier-Based Dynamic Approach for Detecting Use-After-Free Vulnerabilities

Anonymous Author(s)

ABSTRACT

Use-After-Free (UAF) vulnerabilities constitute severe threats to software security. In contrast to other memory errors, UAFs are more difficult to detect through manual or static analysis due to pointer aliases and complicated relationships between pointers and objects. Existing evidence-based dynamic detection approaches only track either pointers or objects to record the availability of objects, which become invalid when the memory that stored the freed object is reallocated. To this end, we propose an approach UAFSan dedicated to comprehensively detecting UAFs at runtime. Specifically, we assign a unique identifier to each newly-allocated object and its pointers; when a pointer dereferences a memory object, we determine whether a UAF occurs by checking the consistency of their identifiers. We implement UAFSan in an open-source tool and evaluate it on a large collection of popular benchmarks and real-world programs. The experiment results demonstrate that UAFSan successfully detect all UAFs with reasonable overhead, whereas existing publicly-available dynamic detectors all miss certain UAFs.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Theory of computation → Program analysis**; • **Software and its engineering → Software defect analysis**.

KEYWORDS

Use-After-Free (UAF), double-free, instrumentation, dynamic analysis, object identifier

ACM Reference Format:

Anonymous Author(s). 2021. UAFSan: An Object-Identifier-Based Dynamic Approach for Detecting Use-After-Free Vulnerabilities. In *Proceedings of ISSTA'21, 12-16 July, 2021, Aarhus, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Once a memory object is released, all pointers to the freed memory object become dangling pointers [8, 13, 45, 53, 58]. If any of these dangling pointers is dereferenced to access the data, then a Use-After-Free (UAF) vulnerability occurs. Similarly, using any of these dangling pointers to release the heap memory results in a double-free vulnerability, which is regarded as a special case of UAF.

ISSTA 2021, 12-16 July, 2021, Aarhus, Denmark

ACM ISBN 978-1-4503-XXXX-X/21/07…$15.00
https://doi.org/10.1145/1122445.1122456

As shown in Figure 1, the number of UAFs and double-frees in NVD[1] increases almost every year since 2009. Moreover, most UAFs in NVD are rated as critical or high in severity, which is prone to cause program crashes, silent data leakage, arbitrary code execution, and other serious consequences [43, 56]. Meanwhile, in recent years, memory corruption attacks exploiting UAFs have increased dramatically [25, 51]. In contrast to other memory errors, UAFs in large real-world programs are difficult to detect through manual or static analysis for the following reasons. First, it is difficult to infer all pointer aliases across many data structures. Second, it is challenging to determine which memory object a pointer points to. Last but not least, the path explosion caused by the increase of program size makes inter-procedural analysis rather difficult.

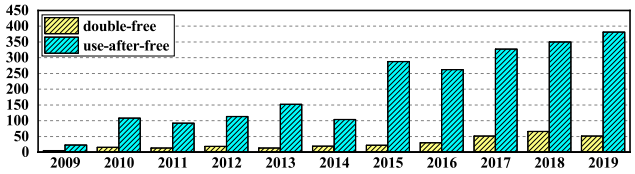


Figure 1: Use-After-Frees and double-frees in NVD

Due to the aforementioned challenges, there are only a few studies on static analysis for UAF detection, e.g., GUEB [18], Tac [55], and CRed [56]. These approaches first transform the target program into an intermediate representation, based on which they perform inter-procedural analysis to track heap operations and pointer propagation while considering pointer alias. They then use the obtained pointer aliases and the points-to relationships between pointers and objects to detect UAFs. Although these static analysis approaches have higher code coverage, they only perform limited inter-procedural analysis and incomplete pointer analysis [47]. Thus, they may generate false positives and false negatives.

The lion's share of attention is focused on the dynamic detection, which falls into two categories: evidence-based approaches [7, 10, 16, 19, 28, 30, 32, 35, 40–42, 46] and prediction-based approaches [9, 22, 30]. The evidence-based approaches [7, 10, 16, 19, 28, 32, 35, 40–42, 46] only track pointers or objects. When a pointer is dereferenced, they use the pointer (object) address to query their auxiliary data structures to determine whether the pointer to an alive object. If not, a UAF is detected. However, if the accessed memory is reallocated, this UAF may be missed. In our analysis of 34 UAFs on 21 real-world programs (cf. Section 4.2), 41.2% (14/34) UAFs are related to memory reallocation, out of which at least 28.6% (4/14) are missed by existing publicly-available evidence-based approaches.

The prediction-based approaches [9, 22, 30] can predict concurrent UAFs introduced by thread scheduling in multi-threaded

[1] https://nvd.nist.gov/

programs, while UAFs in sequential programs go beyond their models [30]. In this paper, we only consider UAFs on the heap while those on the stack are ignored because they are rare [57] and hard to exploit [25]. In addition, we focus on evidence-based UAF detection in sequential C/C++ programs.

To address the limitation of existing evidence-based dynamic detectors (sanitizers), we propose a novel dynamic detector UAFSan. A salient merit of UAFSan is that its detection capability is not affected in the presence of memory reallocation. UAFSan achieves this by checking whether the pointer still points to its intended alive object. Specifically, for each newly-allocated object, UAFSan assigns it a unique identifier, which is also recorded as an attribute of each pointer to the object. Once the object is freed, UAFSan will invalidate the object identifier with the object, while the object identifier with its pointers remains unchanged. Thus, when a memory object is deallocated, or even when the freed memory is reallocated, UAFSan can accurately detect the UAF that occurs because the object identifier with the pointer does not match that with the memory object to which the pointer currently points.

We have implemented UAFSan on top of the LLVM infrastructure [24]. The experimental evaluation on popular benchmarks and real-world programs shows that UAFSan outperforms eight publicly-available evidence-based dynamic detectors [7, 16, 19, 28, 32, 35, 40–42] in terms of the number of UAFs detected.

The contribution of this paper is summarized as follows:

(1) We propose a novel evidence-based technique UAFSan, which is dedicated to comprehensively detecting UAFs including double-frees in C/C++ programs.
(2) Based on LLVM compiler infrastructure [24], we implement UAFSan into an open-source tool[2].
(3) Our experimental evaluation on real-world programs shows that UAFSan can accurately detect UAFs at runtime. In addition, UAFSan finds certain UAFs missed by existing dynamic detectors and those UAFs can be exploited by attackers.

The remainder of this paper is organized as follows. Section 2 uses a running example to motivate our work. Section 3 presents our approach UAFSan. Section 4 conducts the experimental evaluation and comparison. Section 5 reviews the related work and Section 6 concludes the paper.

2 MOTIVATION

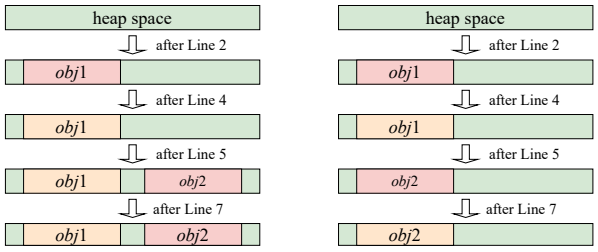
In this section, we use a motivating example to show the limitation of existing evidence-based approaches, and also to illustrate how our approach detects UAFs and overcomes this limitation.

Figure 2(a) shows a C program that involves both UAF and double-free vulnerabilities on the heap. In the program, both pointers p and q become dangling pointers after the object $obj1$ is freed at Line 4. A UAF occurs at Line 6 because the freed object $obj1$ is accessed via pointer q. A double-free occurs at Line 7 because $obj1$ is freed again via q. In some cases, a subtle UAF occurs at Line 8, which needs to be explained in combination with Figure 2(b). Figure 2(b) presents the two possible memory layouts for the program in Figure 2(a) at runtime. As shown on the right scenario of Figure 2(b), if memory reallocation occurs, another object $obj2$ may be allocated on the deallocated memory where the object $obj1$ was stored, that

[2]https://figshare.com/s/d121faab3226633549e6

```
1 int main(){
2   int *p = malloc(40); /*allocate obj1*/
3   int *q = p; /*pointers p and q both point to obj1*/
4   free(p); /*free obj1*/
5   int *r = malloc(40); /*allocate obj2*/
6   *q = 1;
7   free(q); /*free obj1*/
8   *r = 1;
9 }
```

(a) A program written in C



(b) Memory reallocation does not occur and occurs.

Figure 2: A motivating example.

is, $obj2$ uses the same heap address that $obj1$ has used before. Then, the pointer r implicitly becomes a dangling pointer after Line 7, because when q is used to free $obj1$, $obj2$ is freed instead.

Existing evidence-based approaches, such as CETS [32] and ASan [40], only track either pointers or objects to record the availability of objects, but do not distinguish different objects successively-allocated on the same address. When the pointer is dereferenced, they use the pointer (object) address to query their auxiliary data structure to determine whether the accessed memory is currently allocated. If the accessed memory is freed but not reallocated, they can find the UAFs. If the freed memory is reallocated (cf. the right scenario of Figure 2(b)), the UAFs can be missed by these approaches. To address this issue, most existing evidence-based approaches delay memory reallocation by using a quarantine, thereby reducing the chance of UAF detection failures. Unfortunately, this mechanism cannot essentially overcome the limitation of the existing pointer (object)-based approaches. Therefore, they still miss certain UAFs that can be exploited by attackers through some ways [14, 48], causing serious consequences such as arbitrary code execution [2].

To this end, we propose a novel dynamic detector UAFSan, which can track pointers and objects to accurately and comprehensively detect UAFs, and its detection ability is not affected by memory reallocation. Specifically, UAFSan assigns a unique identifier to each newly-allocated memory object and propagates the object identifier to all pointers to the memory object. Once the memory object is freed, UAFSan will only invalidate the object identifier with the memory object, while keep the object identifiers of its pointers unchanged. In this way, UAFSan can detect the UAF that occurs by matching the object identifier of the pointer and that of the memory object to which the pointer currently points.

We use the program in Figure 2(a) to explain the rationale of UAFSan. After $obj1$ is allocated on the heap, UAFSan assigns it a unique identifier, say $obj1\_id$. Since the pointers p and q both

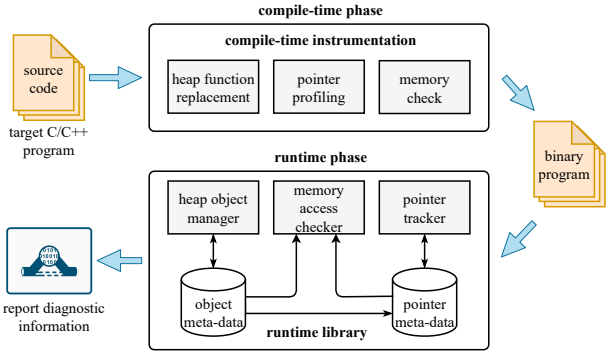


Figure 3: Framework of UAFSan.

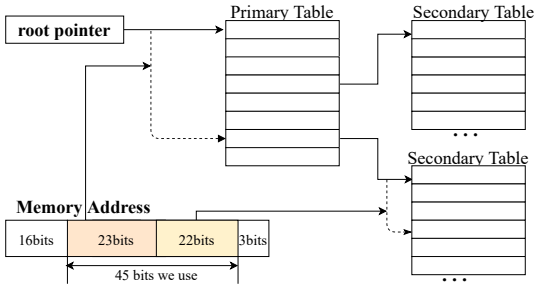


Figure 4: Meta-data organization in the form of two-level lookup table.

point to *obj*1, *obj*1_*id* is also recorded as an attribute of *p* and *q*. When *obj*1 is freed via pointer *p* at Line 4, UAFSan invalidates *obj*1_*id* with *obj*1, and the object identifier of *obj*1 becomes to *invalid_obj*1_*id*. Meanwhile, *p* and *q* still retain *obj*1_*id*. Similarly, for another heap allocation at Line 5, UAFSan assigns another unique identifier *obj*2_*id* to the newly-allocated object *obj*2 and propagates *obj*2_*id* to the pointer *r*. As shown in Figure 2(b), there are two possible memory layouts for the program in Figure 2(a) when the UAF occurs during program execution.

(1) If the freed memory is not reallocated, the current object to which *q* points is an invalid object *obj*1. UAFSan can detect the two UAFs at Lines 6 and 7, because the object identifier *obj*1_*id* with *q* does not match the object identifier *invalid_obj*1_*id* with *obj*1.
(2) If the freed memory is reallocated, the current object to which *q* points becomes to *obj*2. UAFSan can detect the three UAFs at Lines 6, 7 and 8, because the object identifier *obj*1_*id* with *q* does not match the object identifier *obj*2_*id* with *obj*2, and *obj*2_*id* with *r* does not match the invalid object identifier with *obj*2.

Therefore, UAFSan can accurately and comprehensively detect UAFs in programs and is not affected by memory reallocation.

3 UAFSAN

In this section, we elaborate on how UAFSan is designed. Figure 3 overviews UAFSan, which combines compile-time instrumentation and a runtime library. In the compile-time phase (cf. Section 3.2), UAFSan takes the program source code as its input, and outputs an instrumented program. Specifically, the program source code is first compiled into LLVM intermediate representation (IR) [24]. Then, UAFSan instruments the IR code to replace the original heap functions with our wrapper functions, profile the pointer meta-data, and check memory access to determine UAFs.

In the runtime phase, the three components provided by the runtime library interact with the code we instrument. The *heap object manager* (cf. Section 3.3) is responsible for allocating and releasing heap objects. Meanwhile, it manages the meta-data of heap objects, and records the current call stack when a heap object is allocated or released. The *pointer tracker* (cf. Section 3.4) tracks pointers on the entire memory and manages the meta-data of these pointers. It first uses the meta-data of the current heap object pointed to by the original pointer to initialize the meta-data of the original pointer, and then propagates the meta-data of the original pointer to all its derived pointers. The *memory access checker* (cf. Section 3.4) consists of two parts: double-free detection when memory objects are released and UAF detection when pointers are dereferenced. Both of the two parts use the meta-data of the pointer and the meta-data of the memory object to which the pointer currently points.

3.1 Supporting Data Structures

We first explain how to store and retrieve object meta-data and pointer meta-data before elaborating on their structures, and then introduce the interfaces defined to facilitate instrumentation.

The most intuitive way to associate meta-data with objects or pointers is to use embedded meta-data, which puts meta-data and data together and thus avoids the overhead of looking up the meta-data. However, this mechanism changes the structure of pointers and objects and thus may cause serious compatibility issues. Hence, the meta-data should be stored separately. As presented in Figure 4, the lookup table we use has two levels, including one primary table and multiple secondary tables. Each entry in the primary table stores a pointer to the starting address of a secondary table, while each entry in the secondary table stores the meta-data of an object or a pointer. When the instrumented program begins to run, to ensure that there is no address overlap for storing pointer meta-data and object meta-data, UAFSan allocates two lookup tables to store object meta-data and pointer meta-data, respectively. The returned root pointer to the primary table of each lookup table is maintained as a global variable. The reason why we choose the two-tier lookup table is that it can facilitate lookup and reduce memory consumption. The primary table is allocated only once and each secondary table is allocated only when needed, which significantly reduces the virtual memory requirement of UAFSan. In addition, the real physical memory consumption is usually much smaller than the virtual memory consumption.

For each heap object, we regard the starting address of the heap object as its object address and use that to find the position to store its meta-data. For each pointer, we use its memory address (that is, the address of the pointer) to find the position to store its meta-data. Thus, we maintain only one meta-data instance for each pointer or object in its corresponding lookup table. Moreover, for heap objects and pointers, the way to locate their corresponding meta-data is

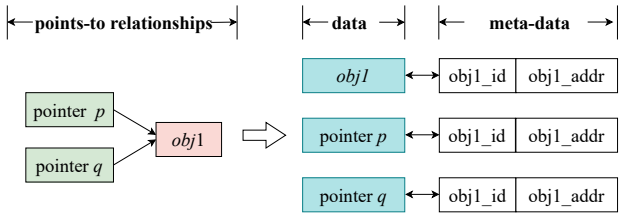


Figure 5: Points-to relationships and the meta-data information.

similar. Generally, the memory address of a heap object is 64 bits on a 64-bit computer, while the first 16 bits and the last three bits are all zero. This also applies to pointers. For each heap object (pointer), its memory address is unique, so the unique middle 45 bits can be used to determine the position for storing the object (pointer) meta-data in the lookup table. As depicted in Figure 4, among the 45 bits we use, the value of the first 23 bits corresponds to the array index in the primary table; the value of the next 22 bits corresponds to the array index in the secondary table.

For pointer meta-data and object meta-data, they share the same data structure and include two attributes. In our implementation, the two attributes are both eight bytes, which facilitates attribute access. For each heap object, its meta-data has two attributes: the object identifier and the memory address of the object. The memory address is used to indicate whether the identifier of the object is invalid, i.e., whether the object has been released (cf. Section 3.4). When a pointer is assigned, the two attributes of the pointer meta-data store the object identifier and memory address (starting address) of the object to which the pointer points. The memory address in the pointer meta-data is used to retrieve the meta-data of the current object to which the pointer currently points.

Let us return to the example in Figure 2(a). After the object *obj*1 is allocated at Line 2, its memory address is obtained, say *obj*1_*addr*; its identifier *obj*1_*id* and its address *obj*1_*addr* are stored as its meta-data. Since pointers *p* and *q* both point to the object *obj*1 after Line 3 and before Line 4, the meta-data of *p* and *q* store the same attributes, namely *obj*1_*id* and *obj*1_*addr*. Figure 5 shows the snapshot of points-to relationship and the meta-data information after Line 3 and before Line 4.

```
1. // Retrieve the pointer metadata for a pointer
2. POmetadata* POpmd_tbl_lookup(ptr_addr)
3. // Retrieve the object metadata for an object
4. POmetadata* POomd_tbl_lookup(obj_addr)
5. // Initializes the metadata for an original pointer
6. void POpmd_tbl_init_as(&original_ptr, original_ptr)
7. // Initializes or updates the metadata for any derived pointer
8. void POpmd_tbl_init_or_update_as(&derived_ptr, &original_ptr)
9. // Initializes or updates the metadata for an object
10.void POomd_tbl_init_or_update_as(obj_addr, obj_id, obj_addr/NULL)
11.// Record the call stacks
12.void POrcs_tbl_update(obj_id, allocated/freed)
13.// Check memory access through pointers
14.void POcheck_mem_access(pointer, pointer_metadata)
```

Figure 6: Interfaces defined to facilitate instrumentation.

Figure 6 lists the interfaces defined to facilitate instrumentation. `POmetadata` in Figure 6 represents the returned meta-data. For example, the argument of the function `POomd_tbl_lookup()` is the memory address of an object, that is, the value of an original pointer or the object address in the meta-data of the original pointer and its derived pointers. A call to this function returns the object meta-data of the current object to which the pointer currently points. If a pointer corresponds to a non-zero offset within an object, the meta-data of this pointer records the starting address of the memory object, which can be used to retrieve the meta-data of the memory object the pointer currently points to.

3.2 Compile-time Instrumentation

The compile-time instrumentation includes three aspects:

(1) The original heap allocation and deallocation functions (e.g., `malloc()`, `free()`) are replaced with our wrapper functions such that the *heap object manager* can manage the meta-data of heap objects at runtime.
(2) Before and/or after the instructions related to pointer propagation, specific tracking code is instrumented to interact with the *pointer tracker* at runtime.
(3) Certain memory checks are instrumented before the instructions related to accessing memory objects through pointer dereference.

Heap function replacement. To facilitate UAF detection, we need to monitor and modify function calls related to heap functions in the program, which is a common practice of almost all existing dynamic detection approaches [7, 19, 28, 32, 35, 40–42, 46]. There are two ways to achieve this: wrapping and replacing. Wrapping allows the original heap functions to execute, but it adds a prologue and epilogue where certain code is added to monitor the heap allocations and deallocations. Replacing uses custom heap functions and does not execute any of the original heap functions. We choose to wrap the original heap functions to add some code that manages the meta-data of the heap objects at runtime.

Specifically, we need to find all function calls related to heap operations in the program, and replace them with calls to our implemented wrapper functions. We find that the custom heap functions in the original program are wrappers of the standard C/C++ heap functions. Therefore, we only need to replace the function calls to the original heap functions with the calls to our wrapper functions. Since the function names are preserved in the LLVM IR, we can find the corresponding function calls through their names.

Figure 7 presents our instrumented program for the program in Figure 2(a). At Lines 2 and 7 in Figure 7, the two calls to our function `wrapped_malloc()` provided by the *heap object manager* replace the calls to the function `malloc()`. Similarly, at Lines 6 and 12, the two calls to our function `wrapped_free()` replace the calls to the function `free()`.

Pointer profiling. Once a pointer is assigned, we need to initialize or update the two attributes of the pointer meta-data. We take the following steps to achieve this. First, we find all original pointers in the program and assign the meta-data to them. By checking whether an instruction is similar to the instruction $p = \&a$, where a is a stack variable, a global variable, or a dynamically allocated heap object, we can find all the instructions that assign original pointers. For each of these instructions, we instrument a call to the function `POpmd_tbl_init_as()` into the program to propagate the meta-data of the object to the original pointer that points to the

```
1. int main(){
2.   int *p = wrapped_malloc(40); //heap function replacement
3.   POpmd_tbl_init_as(&p, p); //pointer profiling
4.   int *q = p;
5.   POpmd_tbl_init_update_as(&q, &p); //pointer profiling
6.   wrapped_free(p); //heap function replacement
7.   int *r = wrapped_malloc(40); //heap function replacement
8.   POpmd_tbl_init_as(&r, r); //pointer profiling
9.   POcheck_mem_access(q, POpmd_tbl_lookup(&q)); //memory check
10.  *q = 1;
11.  wrapped_free(q); //heap function replacement
12.  POcheck_mem_access(r, POpmd_tbl_lookup(&r)); //memory check
13.  *r = 1;
14.}
```

Figure 7: Our instrumented program for the program in Figure 2(a).

object. The function `POpmd_tbl_init_as()` has two parameters: The first one is the address of the original pointer and the second one is the address of the memory object.

As depicted in Figure 7, there are two original pointers p and r, which are initialized at Lines 2 and 7, respectively. Therefore, we instrument two calls to our function `POpmd_tbl_init_as()` into the program at Lines 3 and 8 to initialize the pointer meta-data of p and r, respectively.

Next, we propagate the meta-data of original pointers to all their derived pointers. For pointer propagation through pointer assignment including pointer arithmetic, the meta-data of the original pointers is read first, and then is copied to the meta-data of the derived pointers. This is implemented in our instrumented function `POpmd_tbl_init_or_update_as()`. As presented at Line 5 in Figure 7, we instrument a call to `POpmd_tbl_init_or_update_as()` to propagate the meta-data of p to its derived pointer q.

For pointer propagation through function calls, it is essentially implicit pointer assignment, but requires special treatment. To this end, we introduce a shadow stack that mirrors the call stack to propagate the meta-data of pointers to the derived pointers. If the actual arguments include pointers, the caller can store the meta-data of pointers (actual arguments) into the shadow stack such that the callee can obtain this information just as it obtains actual arguments (pointers). If the return value is a pointer, the callee can also store the meta-data of the pointer into the shadow stack such that the caller can obtain this information just as it obtains the return value. In our implementation, we first instrument code before the call site to store the meta-data of the pointer arguments into the shadow stack. Then, we instrument the function body to retrieve the meta-data of the input pointers from the shadow stack, and to store the meta-data of the return value (pointer) into the shadow stack. We also instrument code after the call site to retrieve the meta-data of the return pointer from the shadow stack.

Memory check. We first find all instructions related to pointer dereference by analyzing whether the memory is accessed through pointers. Then, we instrument calls to our implemented function `POcheck_mem_access()` to detect possible UAFs during program execution. As depicted in Figure 7, we instrument two calls to our function `POcheck_mem_access()` at Lines 9 and 12 because the pointers p and r are dereferenced at Line 10 and 13, respectively. Since our work focuses on the UAFs on the heap, there is no need to instrument memory checks when pointers dereference to non-heap objects. Therefore, we conduct a static intra-procedural backward

```
1. void * wrapped_malloc(size){
2.   void *ptr = malloc(size);
3.   size_t obj_id = unique_id++;
4.   POomd_tbl_init_or_update_as(ptr, obj_id, ptr);
5.   POrcs_tbl_update(obj_id, allocated);
6.   return ptr;
7. }
```

Figure 8: An illustration of heap allocation management: the wrapper function `wrapped_malloc(size)` of `malloc(size)`.

data-flow analysis to identify pointers to non-heap objects, and avoid instrumenting memory checks when such pointers are dereferenced. Specifically, given a memory object pointed to by the pointer, we find the instruction that creates this object from which we can determine whether the memory object is a non-heap object. Since the static analysis performed is conservative, we may still instrument some memory checks when pointers dereference to non-heap objects.

3.3 Heap Object Manager

The *heap object manager* consists of three parts: heap allocation management, heap deallocation management, and heap reallocation management. Before introducing these three parts, we first discuss how to generate object identifiers for heap objects.

Initialization. In our approach, every allocated heap object is assigned a unique object identifier, which is an non-negative integer. All pointers to the same object inherit the identifier of the object. We use the following rules to assign object identifier to non-heap objects and their pointers:

(1) "Zero" is assigned as the object identifiers of null pointers or pointers returned from thirty-party functions whose source code is unavailable.
(2) "One" is assigned to pointers to global variables that always exist during program execution, e.g., static variables and constants.
(3) "Two" is assigned to pointers to stack variables that exists only if the stack frame is valid.

The reason why we also assign identifiers to non-heap objects is that the static backward data flow analysis used at compile-time to eliminate unnecessary memory checks (cf. Section 3.2) is conservative and intra-procedural. Thus, we further leverage the object identifier of the pointer to avoid unnecessary memory checks when pointers dereference to non-heap objects at runtime.

For heap objects, we use a static variable named *unique_id* as a counter and increase it each time to generate a new unique identifier for each newly-allocated heap object. The variable *unique_id* is initialized to "three". There is no need to worry about the situation where the unique identifier are used up because its length is eight bytes and thus a sufficient number of identifiers can be generated.

Heap allocation management. There are several heap allocation functions provided by the C and C++ languages, i.e., `malloc()`, `calloc()`, `mmap()`, `operator new()`, and `operator new[]()`. Our wrapper functions for these heap allocation functions are similar. As an illustration, Figure 8 presents the heap allocation management in the wrapper function `wrapped_malloc(size)` of `malloc(size)`:

```
1. void wrapped_free(ptr){
2.   POcheck_mem_access(ptr, POpmd_tbl_lookup(&ptr));
3.   //the value of ptr is the object address
4.   POmetadata *obj_metadata = POomd_tbl_lookup(ptr);
5.   POomd_tb1_init_or_update_as(ptr, obj_metadata->obj_id, NULL);
6.   POrcs_tbl_update(obj_metadata->obj_id, freed);
7.   free(ptr);
8. }
```

Figure 9: An illustration of heap deallocation management: the wrapper function `wrapped_free(ptr)` of `free(ptr)`.

```
1. void * wrapped_realloc(ptr, size){
2.   POcheck_mem_access(ptr, POpmd_tbl_lookup(&ptr));
3.   POmetadata *old_obj_metadata = POomd_tbl_lookup(ptr);
4.   POomd_tb1_init_or_update_as(ptr, old_obj_metadata->obj_id, NULL);
5.   POrcs_tbl_update(old_obj_metadata->obj_id, freed);
6.   ptr = realloc(ptr, size);
7.   size_t obj_id = unique_id++;
8.   POomd_tb1_init_or_update_as(ptr, obj_id, ptr);
9.   POrcs_tbl_update(obj_id, allocated);
10.  return ptr;
11.}
```

Figure 10: An illustration of heap reallocation management: the wrapper function `wrapped_realloc(ptr,size)` of `realloc(ptr, size)`.

(1) The wrapper function `wrapped_malloc()` calls the heap allocation function `malloc()` to allocate a heap object, and obtains a pointer *ptr* to the newly-allocated heap object.
(2) It obtains a unique identifier which is assigned to this newly-allocated heap object and then increases *unique_id*.
(3) It stores the object identifier and object address (starting address) as object meta-data in the lookup table by calling the function `POomd_tb1_init_or_update_as()`.
(4) It records the current call stack to facilitate the diagnosis.
(5) It returns the pointer *ptr*.

Heap deallocation management. The heap deallocation functions provided by standard C and C++ languages include `free()`, `unmmap()`, `operator delete()`, and `operator delete[]()`. Our wrapper functions for these heap deallocation functions are similar. As an illustration, Figure 9 presents the heap deallocation management in the wrapper function `wrapped_free(ptr)` of `free(ptr)`:

(1) The wrapper function `wrapped_free()` calls our memory check function `POcheck_mem_access()` to determine whether a UAF (double-free) occurs (cf. Section 3.4).
(2) It uses the object address to retrieve the object meat-data of the current object from the lookup table.
(3) It invalidates the object identifier with the current object by changing the object address in the object meta-data of the current object to NULL.
(4) It records the current call stack to facilitate the diagnosis.
(5) It calls the original heap deallocation function `free()` to release memory object.

Heap reallocation management. If the program calls the heap reallocation function, i.e., `realloc()`, there are four possible cases:

(1) The function call only allocates a memory object.
(2) The function call only releases a memory object.
(3) The function call only changes the memory range of the existing object, not the starting address of the object.

```
1. void POcheck_mem_access(ptr, POpmd_tbl_lookup(&ptr)){
2.   if (ptr == NULL) return;
3.   POmetadata* ptr_metadata = POpmd_tbl_lookup(&ptr);
4.   if (ptr_metadata->obj_id == 0 || ptr_metadata->obj_id == 1 ||
     ptr_metadata->obj_id == 2)
5.     return;
6.   POmetadata *obj_metadata = POomd_tbl_lookup(ptr_metadata->obj_addr);
7.   if (obj_metadata->obj_address == NULL || obj_metadata->obj_id !=
     ptr_metadata->obj_id)
8.     reportUAFError(ptr_metadata->obj_id);
9. }
```

Figure 11: An illustration of UAF detection.

(4) The function call first allocates a new object, then copies the contents of the old object to the new object, and finally releases the old object.

For the first and second cases, the code in our wrapper function of `realloc()` is similar to that in Figure 8 and Figure 9, respectively. For the third case, similar to the existing approaches [51, 53], we do not need to instrument any instructions, because the memory object still exists. The last case corresponds to the combination of object allocation and deallocation. Figure 10 presents the code snippet to handle the last case in the wrapper function `wrapped_realloc(ptr,size)` of `realloc(ptr, size)`. According to Linux programmer's manual, we differentiate the four cases of heap reallocation through the actual arguments and return value of the function call, which is implemented in our wrapper function.

3.4 Pointer Tracker and Memory Access Checker

At runtime, the *pointer tracker* manages the meta-data of the created pointers by interacting with the tracking code we instrumented. Specifically, if an original pointer is created, its meta-data is initialized with the meta-data of the object to which the pointer points. If its derived pointers are created, no matter how complicated, they directly or indirectly inherit the meta-data of the original pointer. For `p=malloc(8)` and `q[i+2]=p`, p and $q[i+2]$ point to the same object, so $q[i+2]$ inherits the meta-data of p. Therefore, during program execution, the meta-data of the original pointer and all its derived pointers record the same attributes.

Once a memory object is accessed or released, the *memory access checker* determines whether a UAF occurs by matching the object identifier with the pointer and that with the current object to which the pointer currently points.

(1) If the memory object is still alive, the object address in the object meta-data is not NULL, and the object identifier in the object meta-data is equal to the object identifier of the pointer. In this case, there is no UAF.
(2) If the freed memory is not reallocated, the object address in the object meta-data is NULL, which indicates the object identifier with current object is invalid. Thus, the object identifier in the current object meta-data does not match the object identifier of the pointer. In this case, a UAF is detected.
(3) If the freed memory is reallocated, although the object address in the object meta-data is not NULL owing to memory reallocation, the object identifier in the current object meta-data is not equal to the object identifier of the pointer. In this case, a UAF is detected.

Thus, no matter whether the freed memory is reallocated, UAFSan can detect UAFs including double-frees in programs.

Figure 11 presents the code we instrument for detecting UAFs and double-frees:

(1) `POcheck_mem_access()` checks whether the pointer is a null pointer. If so, the function returns directly.
(2) It obtains the object identifier with the pointer, based on which it determines whether the pointer points to a non-heap object. If the object identifier with the pointer is "zero", "one", or "two", the function returns directly. Note that for pointers returned from functions without being instrumented, the object identifiers with them are "zero".
(3) It obtains the object address with the pointer, based on which it queries the lookup table to obtain the meta-data of the current object to which the pointer currently points.
(4) It determines whether a UAF or a double-free occurs based on the consistency between object identifiers. If so, it reports the diagnostic information about this UAF and then terminates the program execution.

4 EVALUATION

We have implemented UAFSan based on the LLVM 7.1 [24] compiler infrastructure. The compile-time instrumentation in UAFSan is performed on the LLVM IR. Our evaluation aims to answer the following research questions (RQs).

- **RQ1**: *Effectiveness* - Can UAFSan accurately detect UAFs in the benchmarks and real-world programs? What is its advantage over existing dynamic detectors?
- **RQ2**: *Performance* - What is the runtime overhead and memory overhead of UAFSan? Is the performance of UAFSan comparable to that of existing dynamic detectors?

4.1 Experimental Setup

Methodology. To answer RQ1, we first evaluate UAFSan on popular benchmarks. We further evaluate UAFSan on large real-world programs with known UAFs, and compare it with eight state-of-the-art evidence-based dynamic detectors (sanitizers) that are publicly-available: Valgrind [35], Dr.Memory [7], TSan [41, 42], CETS [32], EffectiveSan [16], QASan [19], ASan [29] and DoubleTake [8]. To answer RQ2, we take all the above tools for comparison.

Benchmarks. JTS benchmark[3] is a collection of C/C++ programs with known UAFs, covering various data types and control flows. We select 960 small C/C++ programs with UAFs or double-frees from the JTS benchmark. Since the programs in JTS benchmark do not involve memory reallocation, we adapt them by adding additional heap allocations so that the UAFs (including double-frees) in these programs occur after memory reallocation. In this way, we obtain another 960 programs, i.e., UAFBench benchmark, to validate the effectiveness of UAFSan. To further evaluate the effectiveness of UAFSan and compare it with existing dynamic detectors, we collect 21 large real-world programs from GitHub (cf. Table 2) containing 34 known UAFs and their corresponding proof-of-concepts (PoCs). We select these real-world programs by considering different categories, different sizes, and popularity. To answer RQ2, we first use the real-world programs in Table 2 to evaluate the performance of UAFSan and existing dynamic detectors. Since these real-world programs are not provided with sufficient test suites and some dynamic detectors do not work well on them (cf. Section 4.2.3), we further use MiBench benchmark [21] to answer RQ2 more comprehensively. MiBench benchmark is a free and commercially representative benchmark used by many previous studies [10, 26, 46]. We select 13 representative programs from the MiBench benchmark.

All experiments were performed under the 64-bit Ubuntu 18.04.4 with 2 quad-core Intel Core i7-7700 CPUs and 16GB memory. All benchmarks and real-world programs are compiled with "O0" optimization level and are publicly-available[4].

[3]https://samate.nist.gov/SRD/testsuite.php/

Table 1: The Effectiveness of UAFSan on JTS and UAFBench

Category	Vulns Type	Reused	Language	# Vulns	F-measure
JTS	UAF	No	C	132	100%
	UAF	No	C++	268	100%
	double-free	No	C	168	100%
	double-free	No	C++	392	100%
UAFBench	UAF	Yes	C	132	100%
	UAF	Yes	C++	268	100%
	double-free	Yes	C	168	100%
	double-free	Yes	C++	392	100%
Total	-	-	-	**1920**	**100%**

4.2 RQ1: Effectiveness

4.2.1 Experiment on JTS and UAFBench. We first use JTS benchmark and UAFBench benchmark to validate the effectiveness of UAFSan. Each program in these two benchmarks consists of 100 to 500 lines of code, and contains only one UAF or double-free. Table 1 summarizes the experimental results, in which the column "Reused" indicates whether the UAF (double-free) occurs when the program reallocates the freed memory.

As shown in Table 1, UAFSan successfully detects all 960 UAFs on the JTS, including 560 double-frees. This indicates that UAFSan can accurately detect UAFs and double-frees in the programs if these programs do not reallocate the freed memory. UAFSan also successfully detects all 960 UAFs on the UAFBench, including 560 double-frees. This implies that UAFSan can still accurately detect UAFs and double-frees in the programs even when these programs reallocate the freed memory. Beside, UAFSan does not find other UAFs, indicating that no false positives are reported. Altogether, the recall and precision of its analysis are both 100%. Thus, the F-measure is also 100%.

In summary, the experiments on the benchmarks demonstrate that UAFSan can accurately and comprehensively detect UAFs and double-frees in small C/C++ programs.

4.2.2 Experiment on Real-World Programs. We then use 21 real-world programs containing 34 known UAFs to evaluate the effectiveness of UAFSan and to verify the prevalence of the UAFs that are relevant to memory reallocation in real-world programs. Table 2 shows the experimental results.

As shown in Table 2, UAFSan successfully detects all 34 known UAFs (including double-frees) in these 21 real-world programs. In addition, UAFSan does not report any false positives. Among the

[4]https://figshare.com/s/fa35ded20e679aa1270b

Table 2: The Detection Results of Different Evidence-based Detectors on 21 Real-World Programs with 34 Known UAFs

Program	Version	LoC	Vulnerabilities	Vulns Type	Reused	UAFSan	ASan	Valgrind	Dr.Memory	DoubleTake	TSan	QASan	EffectiveSan	CETS
GoHttp	-	0.5k	CVE-2019-12160	double-free	NO	√	√	√	√	×	×	√	×	√
nasm	2.14.02	102k	CVE-2019-8343	UAF	No	√	√	√	√	×	√	√	√	√
			CVE-2018-20535	UAF	Yes	√	√	√	√	×	√	√	√	√
			CVE-2018-20538	UAF	No	√	√	√	√	×	√	√	√	√
			CVE-2017-10686	UAF	Yes	√	√	√	√	×	√	√	√	√
binaryen	1.38.22	98k	CVE-2019-7703	UAF	No	√	√	×	×	×	×	×	–	–
nasm	2.14rc02	101k	CVE-2018-19216	UAF	No	√	√	√	√	×	√	√	√	√
			CVE-2017-17820	UAF	Yes	√	√	√	√	×	√	√	√	√
			CVE-2017-17817	UAF	Yes	√	√	√	√	×	√	√	√	√
			CVE-2017-17816	UAF	No	√	√	√	√	×	√	√	√	√
			CVE-2017-17814	UAF	Yes	√	√	√	√	×	√	√	√	√
			CVE-2017-17813	UAF	Yes	√	√	√	√	×	√	√	√	√
optipng	0.6.4	100k	CVE-2015-7801	double-free	Yes	√	×	√	√	×	√	×	√	×
readelf	2.31.1	1781k	CVE-2018-20623	UAF	Yes	√	√	√	√	×	√	√	×	×
readelf	2.28	1781k	CVE-2017-6966	UAF	Yes	√	√	√	√	√	×	√	×	×
jpegoptim	1.4.5	23k	CVE-2018-11416	UAF	No	√	√	√	√	×	×	√	√	×
yara	3.5.0	75k	CVE-2016-10211	UAF	No	√	√	√	√	×	√	×	–	×
			CVE-2017-5924	UAF	Yes	√	√	√	√	×	√	×	–	×
recutils	1.8	43k	CVE-2019-6455	UAF	No	√	√	×	×	×	√	×	–	×
jasper	1.900	53k	CVE-2015-5221	UAF	No	√	√	√	√	×	√	√	√	×
gravity	0.2.8	21k	CVE-2017-1000172	double-free	Yes	√	×	×	×	×	×	×	×	–
mxml	2.12	33k	CVE-2018-20005	UAF	Yes	√	×	√	√	×	√	√	–	–
			CVE-2018-20592	UAF	No	√	×	√	√	×	×	√	–	–
ghostscript	9.20	1693k	CVE-2016-7978	UAF	No	√	√	√	√	×	×	√	–	–
			CVE-2016-10217	UAF	Yes	√	√	√	√	×	×	√	–	–
openh264	1.8.0	143k	UAF-issue-1	UAF	No	√	√	×	×	×	×	×	–	–
cflow	1.6	50k	UAF-issue-2	UAF	No	√	√	√	√	×	×	√	×	×
GraphicsMagick	1.3.26	112k	UAF-issue-3	UAF	No	√	√	√	×	×	×	√	×	–
yasm	1.3.0	65k	UAF-issue-4	UAF	No	√	√	√	√	×	×	√	√	√
			UAF-issue-5	UAF	No	√	√	√	√	×	×	√	√	√
			UAF-issue-6	UAF	No	√	√	√	√	×	×	√	√	√
lua	5.3.5	30k	CVE-2019-6706	UAF	No	√	√	√	√	×	√	√	×	–
giflib	-	43k	UAF-issue-7	UAF	No	√	√	×	×	×	√	×	–	√
libheif	1.4.0	12k	CVE-2019-11471	UAF	Yes	√	√	×	×	×	×	×	–	–
Total	-	**6156k**	**34**	-	**14**	**34**	**30**	**28**	**27**	**1**	**19**	**25**	**16**	**15**

34 UAFs, UAFSan finds that 14 (14/34 = 41.2%) of them are related to memory reallocation. This indicates that UAFs that are relevant to memory reallocation are common in practice, and UAFSan can accurately detect UAFs in real-world programs no matter whether the freed memory is reallocated.

4.2.3 Comparison with Existing Dynamic Detectors. We finally compare UAFSan with eight state-of-the-art dynamic detectors. To this end, we also apply the eight dynamic detectors to the 21 real-world programs. The experimental results are listed in Table 2, where "√", "×", and "-" represent that the UAF is detected, the UAF is not detected, and the program cannot be instrumented, respectively.

Table 2 shows none of these eight dynamic detectors successfully detect all 34 known UAFs; they miss some UAFs, more or less.

It follows that DoubleTake has the worst UAF detection capability because it only detects one real UAF. The reasons why DoubleTake fails to find 33 UAFs are analyzed as follows: 1) It replaces the first 128 bytes of each freed memory object with canary values, and can detect UAFs once the canary values are modified. Therefore, it misses certain UAFs, each of which reads the freed memory object and does not modify canary values. 2) It can only find UAFs on the first 128 bytes of the freed memory object, but misses the UAFs that occur to the rest of the freed memory object. 3) It cannot detect UAFs relevant to memory reallocation, because in this case the canary values on the freed object no longer exists.

TSan misses 15 UAFs because TSan focuses on detecting data races and thus fails to detect UAFs irrelevant to data races. CETS and EffectiveSan cannot find 19 and 18 real UAFs, respectively, because the following two reasons. First, the prototypes of CETS and EffectiveSan are not robust enough to instrument some real-world programs, and thus they miss ten and eleven UAFs, respectively. Second, they only track pointers to record the availability of memory objects. Hence, they cannot distinguish different memory objects of the same type allocated on the same memory address, which causes them to miss UAFs that are related to memory reallocation.

```
    //gravity_lexer.c
527. gravity_lexer_t * gravity_lexer_create(...){
528.   gravity_lexer_t *lexer = mem_alloc(sizeof(...));
538.   return lexer;
539. }
    //gravity_parser.c
681. static gnode_t * parse_analyze_literal_string(...){
789.   gravity_lexer_t *sublexer = gravity_lexer_create(...);
804.   gravity_lexer_free(sublexer);
828. }
2508.static uint32_t parser_run(...){
2522.  gravity_lexer_t *lexer = CURRENT_LEXER;
2523.  gravity_lexer_free(lexer);
2533.}
```

Figure 12: A UAF in *gravity* which is missed by the eight publicly-available dynamic detectors.

ASan, Valgrind, Dr.Memory and QASan detect 30, 28, 27, and 25 UAFs, respectively. They have better UAF detection capabilities because they all delay memory reallocation with the help of a quarantine. The reason ASan performs better than the other three dynamic detectors lies in that the default quarantine of ASan is 256MB, which is larger than those of the others. However, ASan still

Table 3: The Performance of Different Evidence-based Detectors on Large Real-World Programs

Program	Version	UAFSan		ASan		Valgrind		Dr.Memory		DoubleTake		TSan		QASan		EffectiveSan		CETS	
		TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR
nasm	2.14.02	4.20	1.34	8.00	2.93	126.20	13.93	80.20	12.08	3.20	2.66	4.40	3.83	68.00	5.79	1.20	1.29	3.20	3.14
binaryen	1.38.22	3.71	1.65	3.14	5.24	144.43	16.44	124.29	13.31	1.43	2.91	3.57	7.06	19.14	4.90	–	–	–	–
nasm	2.14rc02	3.60	1.84	8.40	3.98	124.80	19.61	82.00	17.08	3.00	3.76	3.60	5.46	18.80	7.06	1.20	1.86	2.80	4.25
optipng	0.6.4	6.93	1.41	3.73	4.00	34.13	28.98	10.53	30.24	6.93	3.40	3.07	6.89	39.13	10.22	7.13	1.55	6.60	2.22
readelf	2.31.1	2.16	1.93	1.80	7.90	12.18	19.30	3.47	22.22	1.16	4.38	2.29	5.56	3.53	9.92	1.35	1.46	2.33	2.26
readelf	2.28	2.00	1.86	1.67	8.06	11.80	19.71	4.20	22.68	1.22	4.49	2.06	5.69	3.80	9.47	1.26	1.50	2.34	2.24
jpegoptim	1.4.5	7.87	1.06	3.07	1.74	58.13	7.74	1.53	7.48	14.33	1.56	3.73	5.03	35.40	2.64	1.53	1.13	7.27	1.81
yara	3.5.0	1.92	1.11	1.77	1.82	91.54	6.47	52.31	3.84	14.08	1.48	3.08	4.39	26.15	1.34	–	–	7.92	1.25
recutils	1.8	2.16	1.20	1.98	1.45	10.48	6.22	8.22	6.02	1.62	2.12	1.85	2.50	8.30	1.30	–	–	1.75	1.31
jasper	1.900	1.03	1.37	1.02	3.69	5.34	24.65	1.60	18.92	1.02	2.88	1.12	6.31	4.22	7.73	1.99	1.79	1.04	2.30
gravity	0.2.8	6.00	1.74	9.00	6.01	256.67	19.55	286.67	17.34	2.33	3.91	5.00	6.76	20.00	5.59	–	–	–	–
mxml	2.12	2.50	1.37	2.50	2.86	295.50	26.51	261.50	23.78	42.50	5.11	4.50	5.50	23.50	5.83	–	–	–	–
ghostscript	9.20	3.21	1.35	2.84	2.46	45.21	3.44	10.96	2.15	2.68	1.95	9.97	4.10	20.22	2.04	–	–	–	–
openh264	1.8.0	2.33	1.37	2.02	1.63	9.72	4.03	3.20	3.94	1.85	1.24	2.02	5.17	5.80	1.77	–	–	–	–
cflow	1.6	9.58	1.61	4.67	3.79	55.33	21.57	23.17	18.78	1.25	2.81	2.17	5.75	40.08	7.03	1.33	1.73	8.17	1.68
GraphicsMagick	1.3.26	3.50	1.52	12.25	2.25	239.50	6.76	56.75	4.69	27.25	1.66	4.75	2.67	164.75	2.34	7.00	1.45	–	–
yasm	1.3.0	2.33	2.56	2.33	4.66	223.67	20.92	60.00	16.37	1.33	2.77	4.00	5.94	26.00	6.71	1.33	2.33	34.67	3.44
lua	5.3.5	3.42	3.14	3.08	3.35	56.75	20.76	20.92	18.64	8.83	2.35	4.67	5.45	29.25	5.12	1.92	1.90	–	–
giflib	–	5.67	1.76	3.75	6.26	46.75	29.60	10.58	23.72	1.50	3.57	3.67	9.53	7.67	6.50	–	–	8.50	1.64
libheif	1.4.0	6.69	1.72	3.46	3.05	91.54	9.94	66.92	7.91	21.54	2.26	2.85	2.62	75.38	2.05	–	–	–	–
Geomean	–	**3.47**	**1.59**	**3.26**	**3.41**	**57.09**	**13.66**	**21.40**	**11.73**	**3.74**	**2.66**	**3.25**	**5.04**	**19.55**	**4.36**	**1.91**	**1.61**	**4.56**	**2.15**

misses four UAFs. The above four dynamic detectors miss real UAFs due to the fact that they only track objects to record the availability of memory objects and thus cannot distinguish different memory objects allocated on the same memory address. When the freed memory is reused due to quarantine exhaustion, they miss real UAFs that occur on the reallocated freed memory.

Figure 12 illustrates a real UAF that can be exploited to crash the program[5] and is missed by the eight detectors. At runtime, the program first frees the memory object at Line 804, which also drains the quarantine of existing detectors. Then, at Line 528, a new memory object is allocated on the freed memory. As a result, the eight dynamic detectors all miss the UAF that occurs at Line 2523.

In summary, UAFSan is more precise than existing dynamic detectors that only track either pointers or objects in real-world programs. By tracking pointers and objects using the object identifiers we introduce, UAFSan can detect more UAFs.

4.3 RQ2: Performance

4.3.1 Experiment on Real-World Programs. We first evaluate the runtime overhead and memory overhead incurred by UAFSan on the large real-world programs in Table 2, while the program *GoHttp* is excluded because it is a server program that runs continuously and does not exit during normal execution. We measure the runtime overhead in terms of the execution time ratio (TR), which is the ratio of the execution time of the instrumented program to that of the original program. Similarly, we measure the memory overhead in terms of the memory ratio (MR), that is, the memory consumption of the instrumented program over that of the original program. The execution time of a program refers to the time span from program beginning to program termination, and its memory consumption refers to the maximum resident set size (RSS) during program execution. We use Linux's *time* to collect the execution time and memory consumption of the program, in seconds and kilobytes, respectively. To ensure accuracy, we repeat the experiment 10 times and average the collected execution time (memory consumption). For comparison, we also apply the eight publicly-available dynamic detectors to these real-world programs. The performance results are listed in Table 3.

It follows from Table 3 that UAFSan exhibits runtime overhead ranging from 1.03x to 9.58x with a geomean of 3.47x, and memory overhead ranging from 1.06x to 3.14x with a geomean of 1.59x. By contrast, the geomean runtime overhead of Valgrind, Dr.Memory and QASan is 57.09x, 21.40x, and 19.55x, respectively; the geomean memory overhead is 13.66x, 11.73x and 4.36x, respectively. Therefore, UAFSan outperforms Valgrind, Dr.Memory and QASan in terms of both runtime overhead and memory overhead. The reasons are analyzed as follows. First, UAFSan uses compile-time instrumentation, and thus avoids the overhead of dynamic instrumentation. Second, UAFSan only requests additional memory space when necessary, whereas the three detectors request a large memory at the very beginning. The runtime overhead and memory overhead of UAFSan is comparable to that of the other five dynamic detectors, including ASan. The geomean runtime overhead of ASan is 3.26x, which is slightly lower than that (3.47x) of UAFSan; the geomean memory overhead of ASan is 3.41x, which is much higher than that (1.59x) of UAFSan. Overall, the performance of UAFSan is better than that of ASan because the runtime overhead is similar but ASan introduces more memory overhead. Specifically, for the three programs (*optipng*, *gravity*, and *mxml*) from which ASan does not find UAFs, the geomean runtime overhead (memory overhead) of UAFSan and ASan on the three programs is 4.70x (1.49x) and 4.37x (4.09x), respectively. Notably, for *gravity*, both runtime overhead and memory overhead of UAFSan is lower than that of ASan. These indicate that UAFSan can find some missing UAFs that ASan fails to find and it does not have a performance penalty.

The performance difference between UAFSan and ASan is mainly due to the two different instrumentation algorithms used. UAFSan

[5]https://github.com/marcobambini/gravity/issues/144

Table 4: The Performance of Different Evidence-based Detectors on MiBench Benchmark

Program	UAFSan		ASan		Valgrind		Dr.Memory		DoubleTake		TSan		QASan		EffectiveSan		CETS	
	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR	TR	MR
basicmath (s)	1.18	1.09	2.09	2.20	71.82	19.97	12.73	15.59	1.18	2.41	2.18	4.29	14.00	4.82	1.09	2.16	1.27	1.08
basicmath (l)	1.08	1.12	1.36	2.19	25.97	19.67	3.05	15.36	1.04	2.39	1.46	4.27	8.03	4.92	1.03	2.15	1.06	1.13
bitcount (s)	1.79	1.30	1.64	3.05	43.79	29.74	8.86	23.66	1.21	3.38	3.36	6.19	10.93	6.24	1.50	1.41	1.93	1.30
bitcount (l)	2.05	1.28	1.32	2.99	16.89	29.06	2.15	22.99	1.08	3.32	3.39	6.07	10.89	6.94	1.44	1.47	2.11	1.27
qsort (s)	3.00	1.66	2.60	3.55	109.60	16.92	33.80	10.65	2.20	4.53	5.80	7.92	15.00	6.54	1.20	3.07	649.00	3.96
qsort (l)	2.06	1.42	1.81	2.44	101.81	11.09	4.97	7.01	1.42	2.99	2.53	5.47	9.33	4.70	1.25	3.08	1172.39	2.55
susan (s)	5.58	1.15	14.42	2.64	143.17	20.37	30.25	15.84	1.33	2.44	15.25	4.54	17.00	5.37	1.83	5.67	4.33	1.15
susan (l)	6.61	1.06	2.34	2.36	28.63	14.75	4.92	11.49	1.10	2.14	25.63	4.95	14.21	3.95	2.58	4.64	7.95	1.21
adpcm (s)	2.02	1.35	1.32	2.99	29.02	30.17	5.28	24.01	3.72	3.08	2.64	6.26	6.66	7.09	1.43	1.30	1.74	1.34
adpcm (l)	2.43	1.34	1.29	3.13	9.36	30.91	1.59	24.62	3.95	3.16	2.87	6.48	6.11	7.33	1.54	1.30	1.98	1.37
CRC32 (s)	3.94	1.31	1.52	3.09	27.63	30.68	6.55	24.40	1.19	3.47	2.99	5.80	14.50	6.40	1.88	1.33	3.55	1.39
CRC32 (l)	3.63	1.30	1.57	3.10	24.71	30.68	5.84	24.47	1.13	3.47	2.74	5.77	12.68	6.37	1.78	1.36	3.27	1.44
FFT (s)	2.71	1.12	2.36	2.23	103.64	19.73	18.36	15.41	1.57	2.39	2.50	4.42	4.79	4.03	1.50	1.78	1.86	1.13
FFT (l)	3.41	1.10	1.34	2.16	35.25	17.70	4.70	13.85	1.02	2.29	1.86	4.66	1.75	3.61	1.16	1.89	1.67	1.17
gsm (s)	8.58	1.40	2.25	3.12	114.00	19.85	72.33	12.98	1.92	5.52	5.67	6.25	12.25	6.26	1.58	1.66	3.67	1.51
gsm (l)	6.37	1.45	1.65	3.32	40.33	21.04	22.34	13.78	1.07	5.85	6.69	7.00	9.54	6.71	1.82	1.71	5.17	1.60
sha (s)	2.60	1.25	2.60	2.99	98.40	29.26	22.20	23.27	1.20	3.47	6.40	6.14	14.20	6.76	1.80	1.43	2.40	1.26
sha (l)	3.76	1.34	2.21	3.18	31.72	30.76	4.83	24.51	1.07	3.48	7.83	6.43	12.62	6.46	2.21	1.41	3.62	1.41
blowfish (s)	9.19	1.44	2.96	3.79	112.85	31.22	53.46	24.89	1.38	6.06	6.04	6.76	9.92	6.69	1.77	1.70	15.15	1.54
blowfish (l)	8.48	1.31	3.57	3.44	64.04	28.60	39.15	22.81	1.71	5.50	7.60	6.14	9.59	7.20	2.34	1.68	8.11	1.37
stringsearch (s)	1.50	1.31	2.00	2.92	218.00	30.39	51.50	24.14	2.00	3.44	3.50	5.78	21.00	6.35	1.50	1.40	1.50	1.38
stringsearch (l)	1.25	1.37	1.75	2.84	112.00	29.76	28.25	23.68	1.75	3.38	3.00	5.89	12.00	7.19	1.50	1.38	1.50	1.51
dijkstra (s)	4.73	1.48	2.33	3.43	37.07	30.10	9.73	23.88	1.93	3.44	7.47	6.68	11.20	12.32	2.20	1.51	3.87	1.67
dijkstra (l)	6.32	1.47	2.70	5.78	17.94	30.46	5.38	24.22	1.45	3.52	7.91	6.72	11.42	37.99	2.36	1.45	5.25	2.56
patricia (s)	2.60	2.52	1.67	2.26	61.00	17.51	10.33	13.92	1.53	2.69	2.40	5.54	12.27	10.29	1.60	1.40	1.87	3.49
patricia (l)	3.13	3.48	2.18	1.60	41.32	9.26	4.51	4.89	1.54	2.05	2.46	4.72	14.91	12.97	1.17	1.43	2.14	5.37
Geomean	**3.19**	**1.39**	**2.10**	**2.87**	**50.38**	**23.04**	**10.75**	**17.38**	**1.49**	**3.30**	**4.23**	**5.74**	**10.47**	**6.84**	**1.61**	**1.78**	**4.44**	**1.60**

tracks pointers and objects using the object identifiers we introduce, while ASan only tracks objects by implementing the one-level lookup table using a large shadow space with a large amount of unused memory. Therefore, ASan is generally faster than UAFSan, but at the cost of consuming much more memory. However, since UAFSan maintains only one meta-data instance for each object and pointer, whereas ASan maintains one meta-data instance for every 8 bytes of each object, UAFSan is sometimes better than ASan in terms of both runtime overhead and memory overhead.

4.3.2 Experiment on MiBench. Since the large real-world programs in Table 2 are not provided with sufficient test suites and some dynamic detectors do not work well on them, we then use MiBench to more fully evaluate the performance of the dynamic detectors for comparison. The small (s) and large (l) input files are provided as program input. We also repeat the experiment 10 times and average the collected execution time (memory consumption) to ensure accuracy.

Table 4 lists the performance results. As shown in Table 4, UAFSan is better than Valgrind, Dr.Memory and QASan in terms of both runtime overhead and memory overhead. The reasons have been mentioned in Section 4.3.1. On MiBench benchmark, the performance of UAFSan is still comparable to or even better than that of the other five dynamic detectors, including ASan. The geomean runtime overhead of ASan is 2.10x, which is lower than that (3.19x) of UAFSan, but the geomean memory overhead of ASan is 2.87x, which is much higher than that (1.39x) of UAFSan. Overall, the performance of UAFSan is not worse than that of ASan.

In summary, UAFSan is better than existing publicly-available dynamic detectors because UAFSan is more effective without sacrificing performance.

4.4 Discussion

Limitations. Although our approach theoretically applies to binaries if the type information is available [15, 54], our prototype currently uses compile-time instrumentation, so the program source code is required. As a result, similar to existing dynamic detection approaches [7, 28, 35, 40], UAFSan cannot instrument the C libraries and the third-party libraries that are not open-source, and thus may miss some UAFs.

UAFSan may report false positives due to the following two reasons: 1) Programs may have type casting between pointer types and non-pointer types, resulting in special aliases that affect the proper propagation of pointer meta-data. Existing approaches [16, 27, 51, 57] also suffer from this limitation. Fortunately, since it is rare in practice, its impact on UAFSan is slim. 2) Programs may have functions with variable number of arguments. Thus, similar to existing pointer-based approaches [10, 32, 46], our approach can hardly propagate pointer meta-data to such functions.

Memory check optimizations. We can perform some optimizations to eliminate redundant memory checks, thus improving efficiency without sacrificing accuracy. The potential optimizations are as follows: 1) For loops, only one memory check is instrumented if the same memory object is accessed in the loop. 2) If no function call occurs between multiple consecutive access to the same object, only one memory check is instrumented. Some redundant memory checks cannot be eliminated if a function is invoked between

multiple access to the same object, because the function call may change the state of the memory object or the pointer.

5 RELATED WORK

In this section, we review the related work on detecting and mitigating UAFs in C/C++ programs.

5.1 UAF Detection

Dynamic evidence-based detectors [7, 16, 19, 28, 32, 35, 40–42, 46] only track either pointers or objects to record the status of each memory object, but do not distinguish different objects successively-allocated on the same address. Consequently, they all miss certain UAFs when the freed memory is reallocated. Valgrind [35], QASan [19] and Dr.Memory [7] leverage different DBI frameworks to dynamically instrument programs and thus incur heavy runtime overhead. DoubleTake [28] uses library interposers to dynamically instrument programs and can only detect UAFs that write to the freed memory. ASan [40] instruments programs at compile time and thus incurs lower runtime overhead. EffectiveSan [16] uses fat pointers to record memory addresses that pointers can safely access. However, EffectiveSan changes the memory layout and inevitably introduces compatibility issues. To this end, CETS [32], MemSafe [46] and MOVEC [10] employ disjoint meta-data storage to track pointers. TSan [41, 42] aims to detect data races and thus may fail to detect UAFs irrelevant to data races. Compared to these existing dynamic detectors, UAFSan is more precise as it can detect UAFs no matter whether the freed memory is reallocated. UAFL [52] is orthogonal to UAFSan because it focuses on generating test cases for sake of finding UAFs based on dynamic detectors.

The prediction-based detectors, i.e., UFO [22] and ConVul [9, 30] are designed to predict concurrent UAFs. They do not consider UAFs in sequential C/C++ programs [30].

There are a few static analysis approaches for detecting UAFs [18, 55, 56]. GUEB [18] uses dedicated value analysis to track heap operations and pointer propagation. However, since its inter-procedural analysis is based on function in-lining, it may repeat analyzing the same function multiple times. CRed [56] detects UAFs based on the demand-driven pointer analysis and spatial-temporal context reduction. Similar to CRed, Tac [55] is also based on the demand-driven pointer analysis. The difference is that Tac uses machine learning to reduce false positives. Although the above approaches achieve higher code coverage, they are not suitable for large programs and may generate false positives and false negatives. In contrast to these approaches, UAFSan reports fewer or no false positives.

5.2 Runtime UAF Protection

Our work is also related to UAF exploit mitigation, which has received much attention [1, 3–6, 8, 11–13, 17, 20, 23, 25, 31, 33, 34, 37–39, 44, 45, 51, 53, 57, 58].

FreeSentry [57], DangNULL [25], and DangSan [51] prevent UAF exploits by explicitly invalidating dangling pointers. They keep track of pointers to the same object and invalidate these pointers once the memory object is released. pSweeper [27] differs from the above approaches in that it uses an extra thread to invalidate all dangling pointers and thus achieves lower runtime overhead.

DangDone [53] and MPChecker [38] protect against UAF exploits by inserting intermediate pointers between objects and their pointers. All operations performed on memory objects through their pointers must be performed via intermediate pointers. Once a memory object is freed, they invalidate the intermediate pointer. Thus, dangling pointers can no longer re-access the freed memory.

Overwriting virtual table pointers is the most widely used technique to exploit UAFs. SafeDispatch [23], VTpin [39], VTV [49], and Vip [17] protect virtual table pointers to mitigate UAF exploits. Although these approaches introduce low runtime overhead, they can only protect pointers that are virtual table pointers. As a result, when attackers target other pointers instead of virtual table pointers, these approaches are ineffective.

DieHard [4], DieHarder [36] and Archipelago [29] assume that the heap space is infinite, and randomly allocate memory objects on the heap. Thus, attackers cannot easily allocate objects on the freed memory. ZEUS [58] improves DieHard in that it introduces additionally random prefix offsets into a single object. However, the above mitigation approaches only provide probabilistic security, which can be bypassed by some ways [14, 48].

Cling [3] and Type-after-Type [50] only allow the freed memory to be reallocated to heap objects of the same type. In contrast to Cling, Type-after-Type uses static analysis to infer the object type instead of performing object type inference at runtime. However, they can only defend against UAFs between objects of different types and cannot prevent UAFs between objects of the same type.

Oscar [13] employs isolated heap allocation to prevent UAF exploits. For each newly allocated memory object, it allocates a new virtual page, and thus the newly-allocated memory object cannot be accessed through a dangling pointer. However, the isolated heap allocation mechanism may incur larger runtime overhead in memory-intensive programs.

6 CONCLUSIONS

Use-After-Free (UAF) vulnerabilities, including double-free vulnerabilities, pose serious threats to C/C++ programs. Existing dynamic detectors (sanitizers) all miss certain UAFs on sequential programs because their detection capabilities are compromised when the freed memory is reallocated. To this end, we propose a novel technique UAFSan, which is dedicated to accurately and comprehensively detecting UAFs including double-frees. UAFSan achieves this by assigning each newly-allocated heap object a unique identifier and propagating this unique identifier to all pointers to the object. Once a heap object is freed, UAFSan invalidates the unique identifier with the object, while pointers of the object still retain this unique identifier. When a dangling pointer is dereferenced, the current object on that accessed memory is either a freed object or a new object, and in either case, the identifier of the current object does not match the identifier of the dangling pointer. Thus, UAFSan can detect the UAF that occurs. Experimental results on popular benchmarks and real-world programs demonstrate that UAFSan is more precise than existing publicly-available evidence-based dynamic detectors as it detects more UAFs with reasonable overhead. The future work can study how the new UAFs detected in popular software could be exploited to launch attacks.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (2009), 4:1–4:40.

[2] Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the IEEE Symposium on Security and Privacy, SP'20, San Francisco, CA, USA, May 18-21*. 578–591.

[3] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium, Washington, DC, USA, August 11-13*. 177–192.

[4] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14*. 158–168.

[5] Hans Boehm, Alan Demers, and Mark Weiser. [n.d.]. *A garbage collector for C and C++*. http://hboehm.info/gc/

[6] Hans-Juergen Boehm and Mark D. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exp.* 18, 9 (1988), 807–820.

[7] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr.Memory. In *Proceedings of the 9th International Symposium on Code Generation and Optimization, CGO'11, Chamonix, France, April 2-6*. 213–223.

[8] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA'12, Minneapolis, MN, USA, July 15-20*. 133–143.

[9] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'19, Tallinn, Estonia, August 26-30*. 706–717.

[10] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'19, Beijing, China, July 15-19*. 341–351.

[11] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. 2003. CCured in the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'03, San Diego, California, USA, June 9-11*. 232–244.

[12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods, SEFM'12, Thessaloniki, Greece, October 1-5*. 233–247.

[13] Thurston H. Y. Dang, Petros Maniatis, and David A. Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17), Vancouver, BC, Canada, August 16-18*. 815–832.

[14] Mark Daniel, Jake Honoroff, and Charlie Miller. 2008. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies, WOOT'08, San Jose, CA, USA, July 28*.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *Proceedings of the IEEE Symposium on Security and Privacy, SP'20, San Francisco, CA, USA, May 18-21*. 1497–1511.

[16] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'18, Philadelphia, PA, USA, June 18-22*. 181–195.

[17] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'17, Santa Barbara, CA, USA, July 10 - 14*. 329–340.

[18] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2014. Statically detecting use after free on binary code. *J. Comput. Virol. Hacking Tech.* 10, 3 (2014), 211–217.

[19] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *Proceedings of the IEEE Secure Development, SecDev'20, Atlanta, GA, USA, September 28-30*. 23–30.

[20] Robert Gawlik and Thorsten Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC'14, New Orleans, LA, USA, December 8-12*. 396–405.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings 4th annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538), Austin, TX, USA, December 2*. 3–14.

[22] Jeff Huang. 2018. UFO: predictive concurrency use-after-free detection. In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). 609–619.

[23] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS'14, San Diego, California, USA, February 23-26*.

[24] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization, CGO'04, 20-24 March, San Jose, CA, USA*. 75–88.

[25] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS'15, San Diego, California, USA, February 8-11*.

[26] Beichen Liu, Pierre Olivier, and Binoy Ravindran. 2019. SlimGuard: A Secure and Memory-Efficient Heap Allocator. In *Proceedings of the 20th International Middleware Conference, Middleware'19, Davis, CA, USA, December 9-13*. 1–13.

[27] Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS'18, Toronto, ON, Canada, October 15-19*. 1635–1648.

[28] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering, ICSE'16, Austin, TX, USA, May 14-22*. 911–922.

[29] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2008. Archipelago: trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'08, Seattle, WA, USA, March 1-5, 2008*, Susan J. Eggers and James R. Larus (Eds.). 115–124.

[30] Ruijie Meng, Biyun Zhu, Hao Yun, Haicheng Li, Yan Cai, and Zijiang Yang. 2019. CONVUL: An Effective Tool for Detecting Concurrency Vulnerabilities. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE'19, San Diego, CA, USA, November 11-15*. 1154–1157.

[31] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA'12, June 9-13, Portland, OR, USA*. 189–200.

[32] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISSM'10, Toronto, Ontario, Canada, June 5-6*. 31–40.

[33] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526.

[34] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'02, Portland, OR, USA, January 16-18*. 128–139.

[35] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'07, San Diego, California, USA, June 10-13*. 89–100.

[36] Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security, CCS'10, Chicago, Illinois, USA, October 4-8*. 573–584.

[37] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS'15, San Diego, California, USA, February 8-11*.

[38] Weizhong Qiang, Weifeng Li, Hai Jin, and Jayachander Surbiryala. 2019. Mpchecker: Use-After-Free Vulnerabilities Protection Based on Multi-Level Pointers. *IEEE Access* 7 (2019), 45961–45977.

[39] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: practical VTable hijacking protection for binaries. In *Proceedings of the 32nd Annual Computer Security Applications Conference, ACSAC'16, Los Angeles, CA, USA, December 5-9, 2016*. 448–459.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference, USENIX ATC'12, Boston, MA, USA, June 13-15*. 309–318.

[41] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the the workshop on binary instrumentation and applications, New York, USA, December 12*. ACM, 62–71.

[42] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for ThreadSanitizer. In *Proceedings of the 2nd International Conference on Runtime Verification, RV'11, San Francisco, CA, USA, September 27-30*. 110–114.

[43] Rasool Sharifi and Ashish Venkat. 2020. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA'20, Valencia, Spain, May 30 - June 3*. 762–775.

[44] Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In *Proceedings of the Annual Computer Security Applications Conference, ACSAC'20, Virtual Event/Austin, TX, USA, 7-11 December*. 454–465.

[45] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS'19, San Diego, California, USA, February 24-27*.

[46] Matthew S. Simpson and Rajeev Barua. 2013. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Softw. Pract. Exp.* 43, 1 (2013), 93–128.

[47] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *Proceedings of the IEEE Symposium on Security and Privacy, SP'19, San Francisco, CA, USA, May 19-23*. 1275–1295.

[48] Alexander Sotirov. 2007. Heap feng shui in javascript. http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html.

[49] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium, USENIX Security'14, San Diego, CA, USA, August 20-22*. 941–955.

[50] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC'18, San Juan, PR, USA, December 03-07*. 17–27.

[51] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the 12th European Conference on Computer Systems, EuroSys'17, Belgrade, Serbia, April 23-26*. 405–419.

[52] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE'20, Seoul, South Korea, 27 June - 19 July*. 999–1010.

[53] Yu Wang, Fengjuan Gao, Lingyun Situ, Lingzhang Wang, Bihuan Chen, Yang Liu, Jianhua Zhao, and Xuandong Li. 2018. DangDone: Eliminating Dangling Pointers via Intermediate Pointers. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware, Internetware'18, Beijing, China, September 16*. 6:1–6:10.

[54] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20, Lausanne, Switzerland, March 16-20*. 133–147.

[55] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC'17, Orlando, FL, USA, December 4-8*. 42–54.

[56] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering, ICSE'18, Gothenburg, Sweden, May 27 - June 03*. 327–337.

[57] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium, NDSS'15, San Diego, California, USA, February 8-11*.

[58] Mingbo Zhang and Saman A. Zonouz. 2018. Use-After-Free Mitigation via Protected Heap Allocation. In *Proceedings of the IEEE Conference on Dependable and Secure Computing, DSC'18, Kaohsiung, Taiwan, December 10-13*. 1–8.