Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Finding Memory Allocation Bugs Using SeaHorn

Author: Simon Heijungs (2625343)

1st supervisor:	dr. C. Giuffrida
daily supervisor:	prof. dr. W. Fokkink
2nd reader:	dr. K. von Gleissenthall

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 20, 2021

Abstract

Although much effort has been put into finding and fixing software bugs, they are still a major problem. Tools exist to find them, but many such bugs still slip through the cracks. For this thesis, we shall attempt to extend SeaHorn to give it the capability to find use-after-free and double free bugs. SeaHorn is a software verification framework that uses LLVM-IR to perform its analysis on. LLVM is a project developing several compiler tools. It uses a bitcode language called *LLVM-IR*, which is designed to be easy to do analysis and optimizations on. Use-after-free and double free are common types of bugs in programming languages with manual memory allocation like C. We explored several designs to detect double free and use-after-free bugs, and the feasibility of their implementation in SeaHorn. We evaluated their performance using the Juliet Test Suite. We found several positive results. First, we found two different designs that both got a perfect score for the double free category. For use-after-free, our best design gets 58 percent of test cases correct. We indicate some directions for further work.

Contents

Chapter 1

Introduction

1.1 Software Verification

Software verification is the practice of verifying the correctness and absence of bugs in software [11]. Verification can be done in different ways. A common way is testing, but there are some downsides to this. For example, testing can show the presence of bugs, but cannot demonstrate their absence. An alternative is formally verifying software using mathematical proofs. In practice, software is often so complex that it is infeasible to manually write these proofs [6]. For this reason, there are tools that can fully or partially automate this process.

One problem with automatic verification is the fact that most software does not have a specification that is formal enough for these automatic tools to work with. Specifications written in natural language are very common [11], and programs that can interpret natural language are still quite limited. Moreover, natural language is inherently imprecise.

Regardless, there are certain properties that are expected from all software. No interpretation of the specification is necessary for those. This makes them good candidates for automatic verification. An important class of such properties is that of memory safety properties. These are properties related to correctness of memory management. This includes safety from bugs like buffer overflows, use-after-free, double free and NULL pointer dereferences. These types of bugs can lead to program crashes and faulty behavior, and also make programs vulnerable to exploitation. We shall discuss this topic further in Chapter 2.

SeaHorn is a software verification framework that uses LLVM-IR to perform its analysis on. It transforms the properties it has to verify into logical formulas and uses a solver to determine whether the properties always hold. LLVM is a project developing several compiler tools. It uses a bitcode language called *LLVM-IR* that is designed to be easy to do analysis and optimizations on. SeaHorn already has the capability to detect buffer overflows and `NULL` pointer dereferences.

1.2 Objectives

In this thesis, our research question is whether it is feasible to detect use-after-free and double free bugs with a verification framework such as SeaHorn. Use-after-free and double free are common types of bugs in programming languages with manual memory allocation like C. We shall discuss what these types of bugs are in Chapter 2. We have developed four different approaches to finding these types of bugs. They all rely on adding runtime checks for these types of bugs to the code, and then letting SeaHorn check whether these checks can ever be triggered. We shall discuss these methods in further detail in Chapter 5.

1.3 Structure

The structure of this thesis is as follows: we start by discussing the concept of memory safety in Chapter 2. Then, we discuss some prior work in Chapter 3. We discus the tools we used in Chapter 4. Then, we discus the designs we used for our solutions in Chapter 5. We discus how these designs were implemented in Chapter 6. We evaluate how well our solutions work in Chapter 7. Finally, we discus what this all means and how to move forward in Chapter 8.

Chapter 2

Memory Safety

Memory safety is a property of software that holds if the software does not access memory in invalid ways. Examples of invalid ways to access memory include reading memory that has not been allocated, attempting to write to read-only memory and attempting to read from memory that does not physically exist. Memory safety is often understood to be an implicit condition for software correctness that does not need to be explicitly mentioned in the specification.

Violations of memory safety can sometimes cause parts of memory to be overwritten in ways the programmer did not intend. This is called memory corruption. Memory corruption can lead to crashes and unpredictable program behavior. In the C standard, situations where these bugs occur are designated as undefined behavior, which means there are no guarantees regarding what the compiler does with code that contains them. These types of bugs are also commonly abused by attackers to compromise the system [19].

Some high-level programming languages are designed in a way that guarantees memory safety. This tends to come at the cost of low-level control for the programmer, required for systems programming and some types of optimizations. For this reason, languages like C, that do not guarantee memory safety, are still widely used. In fact, as of 2021, C is the most popular programming language [21]. For this thesis, we focus on the C programming language.

A few languages like Rust and Go use a hybrid approach: they normally guarantee memory safety, but include a special `unsafe` keyword that circumvents restrictions that are required for this guarantee. This keyword is often necessary in systems programming. Because these are relatively new languages, a significant amount of legacy C code is still in widespread use.

Memory safety violations often occur in code involving arrays and pointers. We are mainly interested in those that involve pointers. In particular, these often involve dangling pointers. We will explain what these are below.

There are many other types of memory safety violations which we shall not be attempting to find. Common types include `NULL` pointer dereferences, use of uninitialized variables and buffer overflows. The former two are exactly what the name suggests. Buffer overflows are situations where a program reads form or writes to an out-of-bounds array index.

A dangling pointer is a pointer that points to a memory location where an object used to be that is no longer there. While the existence of a dangling pointer does not in itself constitute a memory safety violation, dereferencing or freeing such a pointer does. Hereafter, we shall distinguish three common situations where dangling pointers lead to memory safety violations.

2.1 Use-after-free

A common type of memory safety violation involving dangling pointers is use-after-free. It occurs in dynamically allocated memory. To dynamically allocate memory in C, the programmer can use the function `malloc`. This function receives as an argument the size of the region to be allocated and returns a pointer to the start of the allocated region.

The memory can be read through the pointer. This is called *dereferencing* the pointer. One way the programmer can do this is by putting an asterisk (`*`) in front of the name of the variable.

When the allocated region is not needed anymore, the programmer can deallocate it by passing a pointer to its start address to the function `free`. Doing this causes all pointers that point somewhere within the region, including the one that was passed to `free`, to become dangling pointers; they must not be dereferenced after this point.

If such a pointer is dereferenced anyway, it causes a use-after-free bug. A simple example of a use-after-free bug is shown in Figure 2.1.

```
int main(void){
  int *x=malloc(sizeof(int));
  *x=5;
  free(x); // x is a dangling pointer now
  // x is being read after it has been freed.
  // This is a use-after-free bug.
  printf("*x=%d\n", *x);
}
```

Figure 2.1: A simple use-after-free bug.

Some programmers try to prevent this type of bug by setting each pointer they free to `NULL` immediately after as a rule of thumb. While this may help prevent some simple bugs, it does not eliminate the whole issue of use-after-free because the pointer may be copied, as shown in Figure 2.2.

The dereferenced pointer does not necessarily need to be equal to the allocated and freed pointer. It can also point somewhere in the middle of the allocated region as shown in Figure 2.3. This is called an *interior pointer* and as we shall discuss later, these make detection of use-after-free bugs more difficult.

```
int main(void){
  int *x=malloc(sizeof(int));
  int *y=x;
  *x=5;
  free(x); // x and y are now danglings pointers
  x=NULL; // x is not a dangling pointer anymore
  // y is still a dangling pointer because the memory
  // has been freed through x.
  // This is a use-after-free bug.
  printf("*y=%d\n", *y);
}
```

Figure 2.2: Setting a pointer to `NULL` is not sufficient to completely solve use-after-free problems.

```
int main(void){
  int *x=malloc(8*sizeof(int));
  int *y=&x[5];
  free(x); // x and y are now danglings pointers
  // y is an interior pointer in the region allocated
  // and freed by x. y is a dangling pointer.
  // This is a use-after-free bug.
  printf("*y=%d\n", *y);
}
```

Figure 2.3: Use-after-free bugs can occur with an interior pointer.

2.2 Double Free

Like use-after-free bugs, double free bugs involve dangling pointers as well. In this case, a pointer to a memory region is passed to `free` after the region has already been freed. A simple example of a double free bug is shown in Figure 2.4

```
int main(void){
  int *x=malloc(sizeof(int));
  *x=5;
  free(x); // first free is valid
  free(x);  // double free bug
}
```

Figure 2.4: A simple double free bug.

2.3 Expired Stack Pointers

Bugs similar to use-after-free can also occur on the stack. Stack objects are not freed using explicit `free` calls, but are instead freed automatically at the end of their scope. If a variable with longer scope is made to point to a stack variable, it may become a dangling pointer as shown in Figure 2.5.

```
int main(void){
  int *x;
  for(int i=0;i<10;++i){
        int y=i;
        x=&y;
  } // The scope of y ends here
  // x is now a dangling pointer
  printf("*x=%d\n", *x); // illegal
}
```

Figure 2.5: A simple expired stack pointer deference.

Another way you can get expired stack pointers is when a function "leaks" addresses of local variables to the outside. This can happen in several ways. Some examples are shown in Figure 2.6 [2].

```
int *glob=NULL;

int *func(int **arg){
  int loc1=42;
  int loc2=8;
  int loc3=17;
  // three ways to leak local addresses
  *arg=&loc1; // through an argument
  glob=&loc2; // through a global variable
  return &loc3; // through the return value
}

int main(void){
  int *x;
  int *y=func(&x)
  printf("*x=%d\n", *x);
  printf("*y=%d\n", *y);
  printf("*glob=%d\n", *glob);
}
```

Figure 2.6: Three more ways expired stack pointers can occur.

There is not that much attention for expired stack pointers. For instance, the Common Weakness Enumeration (CWE) list [23] has no entry for this. The list has two entries that are somewhat related, but do not really capture this issue specifically. The first is CWE-825: Expired Pointer Dereference [24], a much broader category that also includes use-after-free and double free. The second is CWE-562: Return of Stack Variable Address [25], which is one way expired stack pointers can be created, but not the only way. We believe this issue deserves more attention.

Chapter 3

Related Work

Several researchers have already worked on automated methods for finding allocation bugs. We can split their approaches into a few main categories.

3.1 Static Analysis

Some authors [10, 15, 20] have developed ways to detect use-after-free and double free bugs using static analysis. The purpose of these approaches is to allow quality assurance teams or researchers to automatically find these bugs or prove their absence. Our work can be seen as a form of static analysis because it is performed without physically executing the program. However it is different from typical static analysis techniques because we use a verification framework.

Still, our work serves the same purpose as static analysis. A potential advantage of our approach over normal static analysis approaches is that it can take advantage of the flexibility provided by the modular design of SeaHorn. Different verifiers, LLVM passes and types of semantics can be plugged into SeaHorn with relative ease, so improvements in these tools and techniques can be integrated to benefit the performance of our approach with relatively little effort.

Lee et al. [15] and Hong et al. [10] do not just detect defects but automatically fix them in the source code as well. They perform their analysis directly on C source code. This makes their approach hard to adapt to other memory-unsafe languages. Lee [15] et al. go through the lines of the program and maintains information about each allocated object. They show that a correct patch can be found by solving an exact cover problem over this information. Hong et al. [10] use analysis of control flow to summarize a program's heap-related behavior into a graph. They then fix memory errors by solving a graph labeling problem for that graph. Zhu et al. [20] do their analysis on the level of compiled binary code. They use pointer tracking and function summaries to look specifically for use-after-free bugs.

Asryan et al. [3] use symbolic execution with an SMT solver to find execution paths on which it checks for use-after-free bugs.

Falke et al. [8] mathematically defined allocation by axiomatizing it. Sinz et al. [18] created a mathematical model for memory in LLVM-IR. Both of these can be used to translate memory problems to formulas for an SMT-solver to solve. However, doing this in SeaHorn would require implementing a new encoder, which is beyond the scope of this thesis.

3.2 Runtime Checks

Much work has been done on methods to check for use-after-free and double free bugs at runtime [5, 14, 7, 17 26]. There are two different ways these are often used: during testing or fuzzing and during production. Some approaches are more suitable for one and others more for the other. Allocation bugs do not always cause immediately visible problems, making them hard to spot during testing or fuzzing. Runtime checks during testing can report them as they happen and often give more information than normal crash reports, making these bugs easier to find and fix. During production, runtime checks are sometimes used to terminate the program as soon as a bug is detected. This can mitigate exploitation by attackers, although it still leads to a crash.

Our approach relies on runtime checks as well, but not in the traditional way. Our runtime checks are made to be checked by SeaHorn, not to be executed. Because of this difference in goal, the same techniques do not always work well in both scenarios. Several methods that many of these checkers use are problematic for us. First, they often make heavy use of pointer arithmetic, which can be hard for model checkers to deal with. Second, they tend to have optimizations that improve the performance when you run the code, but make the code more complicated for the model checker. Third, some of the existing solutions rely on hardware features like page faults [14, 7]. This also makes them hard to reason about for model checkers.

Caballero et al. [5] developed *Undangle*, which uses taint analysis to detect when dangling pointers are created. They report on long-lived dangling pointers, even if they are not dereferenced. The idea is that long-lived dangling pointers are inherently dangerous: even if no execution can be found where they are dereferenced, such a execution may be introduced in a future update.

Lee et al. [14] developed *DangNull*, which uses pointer tracking to automatically set pointers to `NULL` when they are freed. This goes beyond the pointer that is actually passed to `free`; they set all pointers derived from it to `NULL` as well. This prevents the creation of dangling pointers, preventing use-after-free and double free. An advantage of this approach is that it does not require explicit checks on each dereference. However, their pointer tracking is quite limited as they only track pointers that are stored in the heap.

Van der Kouwe et al. [12] developed *Dangsan*, which uses pointer tracking to invalidate pointers, like in DangNull, but they improve on it in a few ways. First, their pointer tracking is not limited to just pointers stored on the heap. Second, their approach requires less strict synchronization in multi-threaded programs, leading to less overhead.

Dang et al. [7] developed *Oscar*, which creates a new virtual page mapping for each allocation in a page. When `free` is called, it unmaps the used virtual pages and prevents their reuse. When a pointer to this now unmapped page is dereferenced, it causes a hardware exception. Because of this, no explicit checks are needed on dereferences.

Ainsworth et al. [1] developed *MarkUs*, which quarantines freed memory regions until it can ensure that no dangling pointers to it exist. The most problematic consequences of use-after-free and double free bugs, like memory corruption and exploitation, generally only happen when the memory has been reused at the time the bug occurs. Their approach prevents this scenario.

Serebryany et al. [17, 26] developed *Address Sanitizer*, which uses shadow memory to keep track of allocation state. Like our work, this too works on the level of LLVM-IR.

3.3 Fuzzers

We also see some interesting developments with regard to fuzzers. Fuzzers automatically generate test cases and run them to look for behavior that indicates a bug, like program crashes. Some recent fuzzers like *UAFuzz* by Nguyen et al. [16] specifically target use-after-free bugs.

Chapter 4

Background

For the purpose of finding the bugs we are looking for, we utilize a couple of tools. In this chapter, we shall explain what those tools are, how they work and how they can be employed to detect memory allocation errors.

4.1 LLVM

LLVM [22,13] is an open source project developing various modular compiler related tools. It was started in 2000 by Chris Lattner and Vikram Adve as a research project at the University of Illinois Urbana-Champaign. Its original aim was to investigate dynamic compilation. It has since grown to be more generally about compilation and software development.

One of its core features is LLVM IR, an intermediate language that is suitable for automated analysis. The idea is that a compiler is split into a front end which compiles whatever high-level language is used to LLVM IR and a back end that compiles the LLVM IR to whatever assembly language is needed. LLVM's flagship compiler, Clang, uses this approach to compile C, C++ and C$\sharp$ to a plethora of different architectures. A big portion of the optimization and analysis the compiler performs is also done on this intermediate language. LLVM IR can also be used for just in time (JIT) compilation, similar to how the Java virtual machine works.

LLVM IR is useful to us because it is an attractive language to perform automatic formal verification on for several reasons. First, it has nice mathematical properties that make it easier to formally reason about. Second, since many different languages can be compiled to LLVM IR, performing analysis on it allows one to analyze programs written in all those languages. Third, there are tools like *gllvm* that allow one to easily create a single LLVM file from a project that uses a build system like Gnu Make or Ninja. This make it easy to analyze even projects with complicated build structures.

The LLVM framework contains a tool named *Opt*. This tool performs transformations and analyses on LLVM IR. It is made mainly for compiler optimizations and is used internally by Clang. A type of transformation or analysis that can be done by this tool is called a *pass*. The tool contains many useful passes by default and offers a framework in C++ for creating custom passes that can be loaded from shared object files. There is a distinction between *transformation passes*, which may modify the code, and *analysis passes*, which do not.

There are more ways Opt is used besides compiler optimizations. For instance, it is also used to automatically build runtime checks and extra protections into software or to make debugging easier. This is because Opt provides one of the more convenient ways to automatically transform program code. Here, we use a transformation pass to automatically insert checks that trigger SeaHorn's error state if and only if a bug of a type we are looking for is encountered. When we run SeaHorn on the transformed code, it checks whether there is any execution where this error state is triggered. Because this happens exactly when a bug of a type we are looking for is encountered, SeaHorn effectively checks for the presence of those bugs.

4.2 SeaHorn

SeaHorn [9] is a software verification framework that was started in 2015 by Arie Gurfinkel, Temesghen Kahsa and Jorge Navas. Its purpose is to automatically find bugs in software or prove their absence. SeaHorn performs its analysis on the level of LLVM-IR. The user can either directly supply it with an LLVM-IR file or with a C or C++ file. In the latter case, SeaHorn starts by calling a compiler to transform the code to LLVM-IR.

Verifying software requires both the code of the program under analysis and the correctness properties that are to be checked. In SeaHorn, these are not two separate files, but the correctness properties need to be integrated into the code of the program under analysis using an explicit error state. The way this works is that SeaHorn defines a special function called `__VERIFIER_error` and checks whether an execution is possible where it is called. The way to use this is to add runtime checks to the code that make sure the correctness properties are met. If not, one should trigger the error state. To make this convenient, SeaHorn defines an `sassert` macro as `# define sassert(X) if(!(X)) __VERIFIER_error ()`. Such `sassert` statements can be inserted manually and thought of as normal assertions. Changing the program's code for the purpose of analysis is called *instrumentation*.

SeaHorn consists of several modular components. In cases where the input is C or C++, SeaHorn starts by calling Clang to compile the code to LLVM-IR. If more than one source file is given, they are all linked together into one big LLVM-IR file. Some processing like optimizations may be done on this LLVM-IR code. SeaHorn then uses the formal semantics of the LLVM-IR language to generate a set of logical formulas that are satisfiable if and only if an execution exists that violates the correctness properties. These formulas are in a special form called *constrained Horn Clauses*. These formulas are then fed to a solver that searches for a solution. The solver it uses by default is Spacer, which is based on Z3. This structure is shown graphically in Figure 4.1.

It can be a burden to manually insert all the `sassert` statements. To help with that, SeaHorn offers the option to automatically insert certain types of checks as part of the processing on the LLVM-IR code. Since memory safety properties are generally expected to hold in all programs, these are suitable correctness properties for automatic instrumentation. SeaHorn is already capable of instrumenting for some of those like buffer overflows and null pointer dereferences. However, use-after-free and double free are notable properties that it cannot automatically instrument for. In this thesis, we aim to add those capabilities.

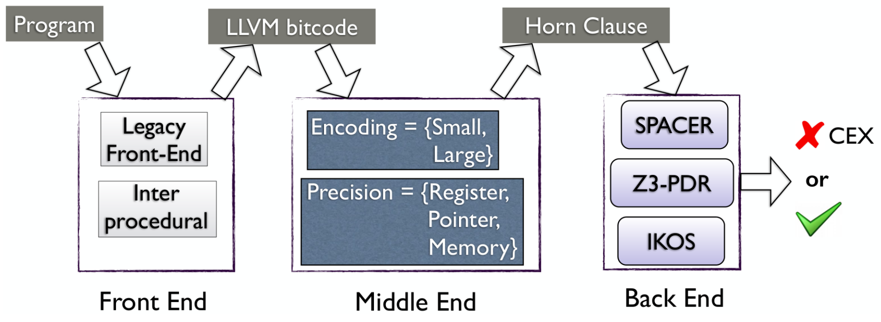

Figure 4.1: SeaHorn's high-level architecture (from [9])

.

4.3 Non-determinism

Unlike normal runtime checks, our instrumentation can also use non-determinism. Any function labeled as `extern` is assumed to return a non-deterministic value. SeaHorn then checks whether the assertions hold for all possible choices of non-deterministic values. Figure 4.2 shows how one can add assertions for a `sort` function without using non-determinism. These assertions test whether each element is less than or equal to the one after it. It is sufficient to only check consecutive values because of the transitivity of an ordering. This requires a loop over all the elements of the array where in each iteration, two consecutive elements are compared. Figure 4.3 shows how this loop can be eliminated using non-determinism. The instrumentation in the two examples is equivalent in the sense that if one of them triggers an assertion, the other does as well. However, the SeaHorn developers told us about how eliminating loops like this often makes verification more efficient, so it is generally a good idea to do this whenever possible.

```
int sort(int A[], int s){
...
}

int main(void){
  int A[SIZE]=...
  sort(A, SIZE);
  for(int i=0;i<SIZE-1;++i)
    sassert(A[i]<=A[i+1]);
}
```

Figure 4.2: Deterministic assertions for sorting.

```
int sort(int A[], int s){
...
}

// since nondet() is labeled as extern, it returns a
// non-deterministic value
extern unsigned int nondet();

int main(void){
  int A[SIZE]=...
  sort(A, SIZE);
  // non-determinisically choose a value for i
  unsigned int i=nondet();
  // make sure the value of i is within range
  assume(i<SIZE);
  // check whether element number i is
  // less than or equal to the next element.
  sassert(A[i]<=A[i+1]);
}
```

Figure 4.3: Non-deterministic assertions for sorting.

Chapter 5

Design

In this chapter, we describe several designs for the instrumentation code. These are numbered chronologically based on when we started developing them.

All approaches follow the same high-level structure. First the program to analyze is compiled to LLVM-IR if it is not in this form already. Then, we use an opt pass to instrument the program around `malloc` and `free` calls and and instructions that dereference pointers. This instrumentation keeps track of the allocation state of memory and adds `sassert` that trigger whenever a use-after-free or double free bug is encountered. SeaHorn's back-end then checks for the reachability of any state where these assertions do not hold. Our four different versions mainly differ in the way that is instrumentation works.

We started with double free bugs because we expected them to be easier to detect than use-after-free bugs. We developed the in-band version for this. When we moved on to use-after-free bugs, we found that the in-band version was not a suitable design for finding those because it cannot deal with interior pointers.

The shadow heap version is an attempt to adapt the in-band version to allow it to deal with interior pointers so it can detect use-after-free bugs. Because we were not satisfied with its performance and did not see promising ways to improve it, we decided to try a different design. The start address array version is the result of this. This did not seem to perform much better though.

During a discussion with the SeaHorn developers, they advised us to use non-determinism. We took this advice to heart and developed the non-deterministic version based on this.

From the moment we started development of the non-deterministic version, we did not spend much time on the shadow heap version and address array version anymore. This is because their initial test results seemed quite poor and the SeaHorn developers advised a different direction. As a result, the shadow heap version and the start address array version are both still quite rudimentary and do not perform well. They are included mostly for completeness.

5.1 In-band Version

This version is based on the way allocators work. To manage allocated memory, most allocators place some metadata at the start of each allocated region. This is called in-band metadata The pointer returned by `malloc` points to the first byte after the metadata, so the allocator can find this metadata by subtracting a fixed offset from the address when it is passed to further calls. The simplest allocators only keep track of the size of the allocated region in this metadata. This is required for the `free` and `realloc` calls because they only receive the starting address as an argument. We can use this metadata to keep track of which memory has been freed for the purpose of finding allocation bugs. This kind of in-band approach is commonly used by run-time checkers.

For this approach, we implemented our own allocator so we can use this metadata for our assertions. We let `malloc` return a `NULL` pointer whenever the allocation size is zero. This means the size metadata will never equal zero, so we can use a zero value to indicate that the region has been freed. This design is shown in Figure 5.1. The grayed out cell shows the allocation has been freed.

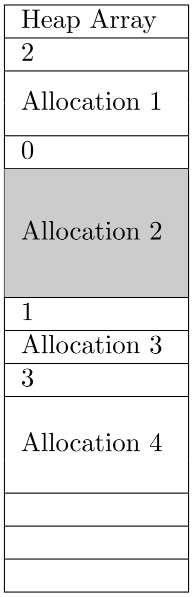

Figure 5.1: Metadata at the start.

The aforementioned approach is sufficient for finding double free bugs. However, use-after-free bugs are problematic for this approach because they can come from interior pointers. If the dereferenced address is an interior pointer, we cannot find the metadata by subtracting a fixed offset from this address. For that, we would need the starting address of the allocated region and we do not have a simple way of finding that using the dereferenced address.

5.2 Shadow heap Version

As mentioned above, the in-band version cannot deal with interior pointers. To solve this, we tried a second approach. For this approach, we created a second global array with the same number of elements as our heap array. Because this is similar to a known technique called *shadow memory*, we call this array the *shadow heap*. In this shadow heap, we write the size of the region at the index corresponding to each byte in the heap array. When the region is freed, we set each byte corresponding to a byte in the freed region to zero. This memory layout is shown in Figure 5.2.

Size array	Heap Array
2	Allocation 1
2	
0	Allocation 2
0	
0	
0	
1	Allocation 3
3	Allocation 4
3	
3	

Figure 5.2: Shadow heap approach.

.

Using the aforementioned approach, we can retrieve the size of the region based on the address of any byte in the region by subtracting the base address and indexing the size array with the result. Subtracting the base address is pointer arithmetic, which SeaHorn sometimes has trouble with. However, it is okay in this case because the result is not converted back to a pointer but used as an array index.

5.3 Start Address Array Version

In the shadow heap version, looping over every byte of the allocation appears make things significantly more difficult for SeaHorn. To get around this, we tried an alternative design. For this new approach, we created a new array of pointers to the starting addresses of all allocations. Because we allocate regions sequentially in an array and we do not reuse memory, this array is guaranteed to be sorted from low to high addresses. Like in the previous approach, we use an array of sizes where zero is written when a region is freed. However, this time each size is only written to this array once. This memory layout is shown in Figure 5.3.

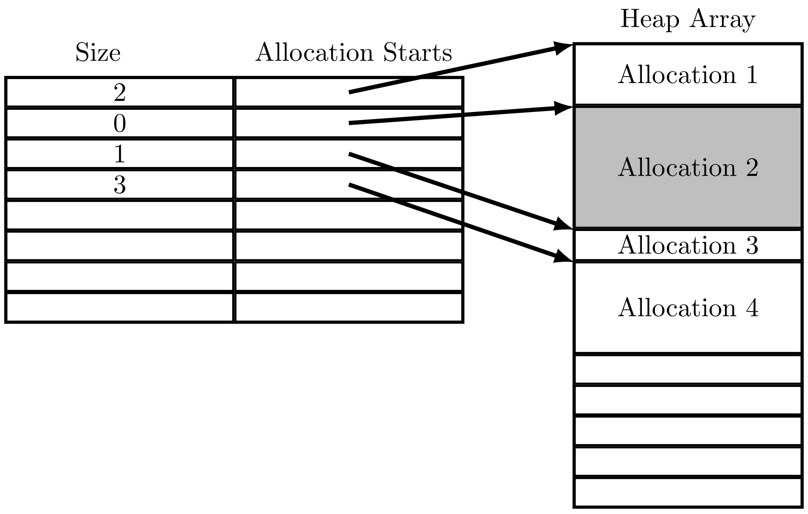

Figure 5.3: Start address array.

Since each allocation adds one element to the start addresses array and one to the sizes array, the same index refers to the same allocation in both arrays. When our `free` or use instrumentation receives an address, we can find this index by scanning through the array of start addresses to find the last one less than or equal to the address our instrumentation received. This yields the index of the region containing the received address.

This approach removes the loops we had over the size of the region in the `malloc` and `free` instrumentation. However, it adds new loops in the `free` and use instrumentation for finding the right segment, and those also seem to give SeaHorn trouble.

5.4 Non-deterministic Version

The start address array version was designed to eliminate the loops introduced in the shadow heap version, but introduced other loops instead. To get rid of the loops altogether, we tried a non-deterministic approach. In this version, we dropped the idea of allocating to a global array and instead tried calling the normal `malloc` and `free` in our instrumentation functions. Our instrumentation chooses an allocated region non-deterministically. Its begin and end addresses are saved and we have a single Boolean to keep track of whether the chosen region has been freed. We assume the chosen region is allocated at the highest address of all allocated regions. We use `assume` statements to limit ourselves to the cases where this holds. Because SeaHorn models `malloc` as a function

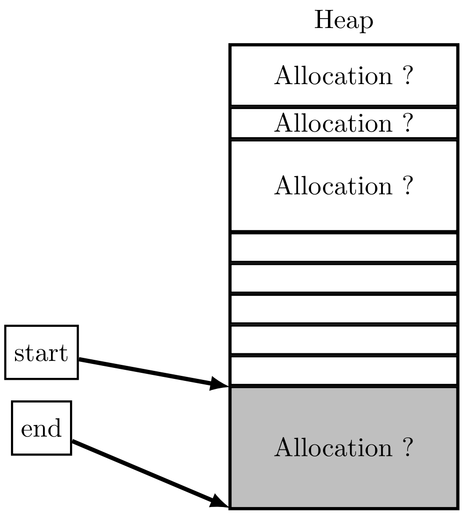

Figure 5.4: A non-deterministic approach.

that non-deterministically returns an address, this is always a possibility. This assumption allows us to check whether a call to our `free` or use instrumentation is in the chosen region by simply checking whether the pointer is greater than the saved starting address. This memory layout is shown in Figure 5.4.

When our instrumentation finds that the given address is indeed in the chosen region, it asserts the Boolean that keeps track of whether it has been freed is set to `false`. The `free` instrumentation then sets this Boolean to `true`.

We make the non-deterministic choice what region to check at allocation time. Each time `malloc` is called, we check a global Boolean variable that keeps track of whether we have already chosen a region. If not, we generate a non-deterministic Boolean that determines whether the newly allocated region will be chosen. Since any allocated region could be chosen this way, the absence of an execution that violates our assertions proves that no use-after-free bug can occur in any region, making the program safe.

This non-deterministic approach eliminates all loops, so it seems the most promising in terms of performance. However, it seems the least intuitive and it is the furthest removed from any instrumentation that normal runtime checkers use. This makes it harder to convince oneself of its correctness.

An additional issue we had to deal with is the fact that we cannot statically determine where heap allocated memory is used. LLVM uses the `load` and `store` instructions to access heap memory, but they may also be used to access stack and global variables. In a first attempt, we tried to address this by having our instrumentation check each received address against the bounds of the heap array. This led to incorrect results. We did not manage to track down the reason why this happens. We now partially solve the issue by replacing all stack allocations with heap allocations, which we explicitly free at the end of the function. This has the additional benefit that it allows us to catch some stack-based dangling pointer bugs with our analysis.

Chapter 6

Implementation

The existing instrumentation passes in SeaHorn simply add all instrumentation code inline wherever they are needed. While this is possible in our case as well, it would be quite burdensome because the code we insert during instrumentation is more complicated than that in their existing instrumentation. It would also be hard to get it to work when function pointers to `malloc` and `free` are used. This inline instrumentation would require us to instrument each call instruction where `malloc` or `free` is called, and those are hard to statically detect when done with a function pointer.

Because of this, we opted to insert our own functions to replace `malloc` and `free`. Rather than instrument the call sites, we could replace the symbols that refer to `malloc` and `free` to make them refer to our own functions. This makes sure that even when a function pointer would be set to `malloc` or `free`, it is set to our replacement function instead. To insert those functions, we link an LLVM-IR file where those functions are implemented with the LLVM-IR of the code under analysis. The capability to link with extra files not supplied by the user as steps in the analysis was not yet part of SeaHorn.

In the non-deterministic approach mentioned in Chapter 5, it was convenient to call the original malloc from inside the function that replaces it. Since we want the original malloc there, we cannot have the instrumentation code replace it. This is a problem because replacing the symbol necessarily applies everywhere. To solve this problem, we defined an external function that takes the same arguments as malloc and call this as if it were malloc. Then after linking and calling the instrumentation pass where malloc gets replaced, it is linked with another file where this external function is implemented as a simple malloc call with the same arguments. Since this malloc call gets introduced after the instrumentation pass is done, the instrumentation pass cannot touch it, so this malloc call is left intact like we wanted.

We integrated our work into SeaHorn for LLVM 10.0.1. Integrating into SeaHorn's modular structure required modifying it in several places. First, we had to add the files for the instrumentation pass and the linked library code into SeaHorn's source directories and modify the build files to get them properly compiled. The library code had to be compiled both in 32 bits and in 64 bits to allow its use for both kinds of binaries.

Then we had to modify the *seapp* tool to make it call our instrumentation pass when given a flag. This *seapp* tool is the preprocessor of SeaHorn, and this is the part that is responsible for instrumentation. This tool normally does a couple of optimization passes by default, but makes an exception when some instrumentation passes that add assertions are used. These optimization passes created problems for us, so we added our instrumentation pass to the list of exceptions.

The process of using SeaHorn is often a sequence of several steps. First, the code gets compiled to LLVM-IR. Then transformations are performed on this. Then, constrained Horn clauses are generated. Finally, the satisfiability of those constrained Horn clauses is checked. To make this easier for the user, SeaHorn has a Python script called *Sea*, which can call all these steps automatically for a number of predefined common use cases.

We added a sequence for our instrumentation. In writing this, linking the library files we discussed took a bit of extra work. None of the existing sequences in SeaHorn require this kind of step, so we had to implement it into the Python script ourselves.

We implemented some flags into our instrumentation pass that control its behavior. One option allows the user to choose between the four designs discussed in Chapter 5. Another turns off use-after-free checks to only do the more light-weight double free checks. We also have an option to replace all `memset` and `memcpy` calls with dummy code. This exists because these functions exist in many test cases we use for evaluation and they can cause problems for SeaHorn. Finally, we had to check the flag that signifies whether we analyze 32 or 64 bit code. This determines the size of pointers, which is important for some of our instrumentation.

Chapter 7

Evaluation

We evaluated using the Juliet Test Suite [4]. This is a collection of over 81000 synthetic C, C++ and Java programs called *test cases*. These are made specifically for the purpose of testing verification tools like ours. Each test case has two variants: one with and one without a bug. These two variants are in the same file and can be selected using compiler flags. The test cases are divided over directories based on the CWE number of the bug in the variant that has one. We specifically used the test cases that are written in C from CWE-415: Double Free and CWE-416: Use After Free.

We wrote bash scripts to automatically compile the test cases to LLVM-IR and run SeaHorn on them. We wrote one script testing double free detection and one for use-after-free detection. These scripts can be found in Appendix C.

The scripts compile each test case in the relevant category twice with different flags to get a version with a bug and one without a bug. These are then fed into SeaHorn using the `sea uaf` command we added. The test cases are all run with each of the four versions. The double free test cases were run using the `--doubleFreeOnly` flag and the use after free test cases with the `--replaceMemset` flag. They both got the `--horn-use-euf-gen=false` flag, which is a workaround for a bug in SeaHorn that I reported. Without this flag, SeaHorn sometimes crashes.

We ran the tests with a time-out of 900 seconds (15 minutes). Most test cases ran much faster than that. We present our results using confusion matrices with an added time-out column.

For the use-after-free category, we ran into an issue where most programs in the test suite used `memset`, which SeaHorn has trouble dealing with. We overcame this issue by replacing each `memset` call with our own function during instrumentation.

7.1 In-band Version

The in-band version is based on metadata saved at the start of each allocated block. This version is not suitable for use-after-free, so we do not have results for that. The results for double free are shown in Table 7.1.

Table 7.1: Confusion matrix of the double free results for the in-band version

	given answer		
true answer	positive	negative	time-out
positive	228	0	0
negative	0	228	0

As you can see, all results are either true positives or true negatives. That means it got everything correct. Based on these results, the in-band version seems to work very well. Considering what the SeaHorn developers told us about the importance of non-determinism, it is a bit surprising that this version works sow well without it. However, it is limited to double free, and we would like to detect use-after-free as well.

7.2 Shadow Heap Version

For the shadow heap version, we use a separate array where for each allocated byte, our instrumentation saves the size of the region in which it is contained. The results for double free are shown in Table 7.2 and those for use-after-free in Table 7.3.

Table 7.2: Confusion matrix of the double free results for the shadow heap version.

	given answer		
true answer	positive	negative	time-out
positive	0	228	0
negative	0	228	0

As you can see, the shadow heap version reported only negative results for double free and only positive results for use-after-free. This is not a good score; it shows no ability to distinguish programs that contain a bug from those that do not. This does not seem like a good approach.

The SeaHorn developers told us that these kinds of assertions where all allocated regions are checked at the same time often require complex proofs involving quantified invariants, and SeaHorn has trouble with these. This may be a reason for this poor performance.

Table 7.3: Confusion matrix of the use-after-free results for the shadow heap version.

	given answer		
true answer	positive	negative	time-out
positive	138	0	0
negative	138	0	0

7.3 Start Address Array Version

The start address array version is based on an array of pointers to the start of allocated regions on the heap. Its results for double free are shown in Table 7.2 and those for use-after-free in Table 7.3

Table 7.4: Confusion matrix of the double free results for start address array version.

	given answer		
true answer	positive	negative	time-out
positive	0	228	0
negative	6	222	0

As you can see, the start address array version reported only negative results for double free. For use-after-free, it reported almost only positive results, and the few negative results were incorrect. Like the shadow heap version, this shows no ability to distinguish programs that contain a bug from those that do not. This does not seem like a good approach.

It probably suffers from the same issues as the shadow heap version. Its assertions also rely on loops, which likely makes them more difficult for SeaHorn to check.

Table 7.5: Confusion matrix of the use-after-free results for the start address array version.

	given answer		
true answer	positive	negative	time-out
positive	138	0	0
negative	138	0	0

7.4 Non-deterministic Version

The non-deterministic version uses non-determinism to choose an allocated region to keep track of. Its results for double free are shown in Table 7.6 and those for use-after-free in Table 7.7.

Table 7.6: Confusion matrix of the double free results for the non-deterministic version.

	given answer		
true answer	positive	negative	time-out
positive	228	0	0
negative	0	228	0

Table 7.7: Confusion matrix of the use-after-free results for the non-deterministic version.

	given answer		
true answer	positive	negative	time-out
positive	58	80	0
negative	31	95	12

As you can see, all of the non-deterministic version's results for double free are either true positives or true negatives. This means that like the in-band version, this version too got a perfect score here.

For use-after-free, it got a total of 153 correct results (true positive or true negative), 111 incorrect results (false positive or false negative) and 12 unknown results. While far from perfect, this does show some ability to find use-after-free bugs.

All in all, this version shows good potential. Its use of non-determinism instead of loops to check all cases is likely a reason for this performance.

We do still see a significant number of incorrect results in use-after-free test cases though. This is quite surprising to us because this version's instrumentation uses only a small amount of metadata and contains no loops, so it should be easy to reason about for SeaHorn. It is not yet clear to us why these incorrect results are generated. We believe the workaround we had to do for `memset` and `memcpy` could be responsible for this. This workaround changes program behavior, which could introduce problems when the program's control flow depends on the results of the those calls.

7.5 Comparison

For each version, we computed what percentage of the results was correct. This was computed by adding the number of true positives to the number of true negatives and dividing the result by the total number of test cases in the relevant category. The results are shown in Table 7.8. From this table, we can see that

Table 7.8: Percentage of results that are correct. Time-outs are excluded in this calculation.

Version	Double Free	Use-after-free
In-band	100%	-
Shadow Heap	50%	50%
Start Address Array	49%	50%
Non-deterministic	100%	58%

the in-band version and the non-deterministic version both got a perfect result in the double free category. On the use-after-free side, the non-deterministic version is the only one that looks somewhat promising. The shadow heap version and the start address array version got poor results overall. They are no better than random guesses. We conclude that the non-deterministic version shows the most promise.

It is unclear whether incorrect results are to be attributed to faults in our instrumentation code or in SeaHorn itself. A possible clue is given by warnings given by SeaHorn for some of the test cases. These warnings read "WARNING: main function not found so program is trivially safe." This is weird because manual inspection of these test cases reveals that a main function is in fact present.[1] This indicates a problem that may well be responsible for a number of the false negatives.

[1] This issue may be related to the a known issue at `https://github.com/seahorn/seahorn/issues/65`. Further investigation is needed.

Chapter 8

Discussion

8.1 Limitations

We assume that the program we analyze uses `malloc` and `free` calls for its memory management. Our approach does not give correct results for programs that use functions with different names for their memory management. We also do not support `malloc` and `realloc`.

SeaHorn cannot properly reason about dereferences to pointers that were derived from casts from integers. SeaHorn also has trouble with programs that use the `memset` and `memcpy` functions. These limitations are not the topic of our research, but they did pose problems for us in evaluation because our benchmark for use-after-free used `memset` and `memcpy` a lot. As a result, we had to replace `memset` calls with dummy functions in the use-after-free test cases to get them to work with SeaHorn. The fact that we use dummy functions that do not actually write to the memory may affect the program's behavior and it could be the culprit for some of our wrong results. However, actually implementing the memset/memcpy calls seems to slow down analysis significantly.

Our deterministic instrumentation code does not reuse freed memory. This causes programs to run out of memory at some point after repeatedly allocating and freeing memory. However, this is not much of a problem because it generally only happens in very long execution paths that we would not be able to verify anyway.

8.2 Future Work

Despite several attempts, the performance of our instrumentation on use-after-free test cases is not yet satisfactory. It is not entirely clear to us whether these issues arise from problems with the instrumentation code or from problems with SeaHorn. Further investigation is needed.

One approach that may prove fruitful would be to track the LLVM-IR instruction `getelementptr` to track the base address. This instruction is the most common way to compute interior pointers from base addresses. Using pointer tracking from there, one could pass the base address to any later load or store instruction. This could eliminate the problems of interior pointers, and thus allow methods that would otherwise only be suitable for double free to be used for use-after-free as well. The `getelementptr` is not the only way an interior pointer can be computed, so this approach is not completely air tight. Most cases would be caught though, and a 100% success rate is not necessary to make a result useful.

Both the in-band version and the non-deterministic version of our instrumentation got great results for double free in the Juliet test suite. However, their scalability on real-world software remains to be tested.

It may also be possible to find memory leaks using our approaches by implementing a function that gets called right before termination. This function would need to assert that all tracked allocated regions have been freed.

To decrease runtime, it could help to do some static analysis at instrumentation time and omit instrumentation in places where this analysis can statically determine that allocation bugs do not occur.

This work could also be adapted to work with C++. This would entail writing instrumentation for the `new` and `delete` functions. What makes this a bit harder is the fact that different types all have their own `new` and `delete` functions, so they would likely all need to be instrumented individually.

Expanding to C++ would also make it possible to look for another type of bug. When an object is allocated using `malloc` and freed using `delete`, or allocated using `new` and freed using `free`, this is mismatch is a bug. We expect this not to be too difficult to detect if the metadata keeps track of what allocation function is used.

8.3 Additional Considerations

One of the most difficult aspects of this research was debugging the instrumentation pass and the code it inserts. This inserted code does not stand on its own, so traditional debug tools cannot be used. It is not run, so we cannot rely on debug output from print statements either.

Another problem we ran into is that not everything in SeaHorn seems to be well documented. This is not too surprising considering the fact that SeaHorn is an academic project that is moreover still in progress. We could not find a centralized documentation section on the SeaHorn website and SeaHorn does not come with a manual page. Some information is shown when the SeaHorn executables are called with the `--help` flag, but it appears to be incomplete. On top of that, SeaHorn consists off several executables that are called internally by the *sea* tool. These all give their own information with `--help`, making it hard to know where to look. For instance, we could make SeaHorn output the invariants it inferred when proving something, but we could not find information on how to interpret the syntax of this output anywhere.

Fortunately, the SeaHorn developers have been very helpful. They gave some recommendations, especially with regard to the use of non-determinism. When we reported a crash, they told us they would fix it and they immediately informed us about a flag to use as a workaround.

8.4 Conclusion

In this thesis, we describe instrumentation passes and library code that we designed with the purpose of allowing SeaHorn to detect use-after-free and double free bugs. We created four different versions. The in-band version was only designed to detect double free bugs, while the other three were designed to detect use-after-free bugs as well. For the detection of double free bugs, this has been quite successful, with two of the four versions scoring perfect marks on the Juliet test suite. For the detection of use-after-free bugs, the non-deterministic version seems the most promising, but there is still room for improvement.

Acknowledgments

I would like to thank Wan Fokkink, Cristiano Giuffrida and Klaus von Gleissenthall for their valuable feedback and great support during regular meetings, helping me stay on track.

I would also like to thank the SeaHorn developers Arie Gurfinkel and Jorge Navas for taking their time to answer some questions and giving good directions on approaches to try, especially with regards to the use of non-determinism.

Finally, I would like to thank my parents, who helped me stay focused and provided feedback as outsiders, helping improve readability.

Literature

1. S. Ainsworth, T. M. Jones: MarkUs: Drop-in use-after-free prevention for low-level languages. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 578–591, 2020, doi: 10.1109/SP40000.2020.00058

2. Z. Alzamil: Application of Computational Redundancy in Dangling Pointers Detection. In: 2006 International Conference on Software Engineering Advances (ICSEA'06), pp. 30, 2006, doi: 10.1109/ICSEA.2006.261286.

3. S. A. Asryan, S. S. Gaissaryan, S. F. Kurmangaleev, A. M. Aghabalyan, N. G. Hovsepyan, S. S. Sargsyan: Dynamic Detection of Use-After-Free Bugs. In: Programming and Computer Software. 45(7), pp. 365–371, 2019, doi: 10.1134/S0361768819070028

4. T. Boland, P.E. Black: Juliet 1.1 C/C++ and Java Test Suite. In: Computer, 45(10), pp. 88–90, 2012, doi: 10.1109/mc.2012.345

5. J. Caballero, G. Grieco, M. Marron, A. Nappa: Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In: ISSTA 2012 Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 133–143, 2012, doi: 10.1145/2338965.2336769

6. E.M. Clarke, T.A. Henzinger, H. Veith: Introduction to Model Checking. In: E.M. Clarke, T.A. Henzinger, pp. 1–26, 2018, H. Veith, R.Bloem: Handbook of Model Checking, Springer

7. T.H.Y. Dang, P. Maniatis, D. Wagner: Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 815–832, 2017

8. S. Falke, F. Merz, and C. Sinz: A Theory of C-Style Memory Allocation. Proceedings SMT, 2011

9. A.Gurfinkel, T.Kahsai, A. Komuravelli, J.A. Navas. The SeaHorn Verification Framework. In: CAV 2015. LNCS 9206, pp. 343–361, 2015, doi: 10.1007/978-3-319-21690-4_20

10. S. Hong, J. Lee, J. Lee, H. Oh: SAVER: scalable, precise, and safe memory-error repair. in: ICSE 2020: pp. 271–283, doi: 10.1145/3377811.3380323, 2020

11. J. Laski, W. Stanley: Software Verification and Analysis, Springer, 2009

12. E. van der Kouwe, V. Nigade, C. Giuffrida: DangSan: Scalable Use-after-free Detection. In: EuroSys '17: Proceedings of the Twelfth European Conference on Computer Systems, pp. 405–419, 2017

13. C. Lattner, V.S. Adve: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, pp. 75–88, 2004

14. B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, W. Lee: Preventing Use-after-free with Dangling Pointers Nullification. In: Proceedings of the 2015 Network and Distributed System Security Symposium, 2015

15. J. Lee, S. Hong, H. Oh: MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 95–106, 2018

16. M. Nguyen, S. Bardin, R. Bonichon, R. Groz, M. Lemerre: Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), pp. 47–62, 2020

17. K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov: AddressSanitizer: A Fast Address Sanity Checker. In: USENIX ATC'12: Proceedings of the 2012 USENIX conference on Annual Technical Conference, pp. 26, 2012

18. C. Sinz, S. Falke, F. Merz: A Precise Memory Model for Low-Level Bounded Model Checking. Proceedings SSV, 2010

19. L. Szekeres, M. Payer, T. Wei and D. Song, SoK: Eternal War in Memory, 2013 IEEE Symposium on Security and Privacy, pp. 48–62, doi: 10.1109/SP.2013.13, 2013

20. K. Zhu , Y. Lu, H. Huang: Scalable Static Detection of Use-After-Free Vulnerabilities in Binary Code. In: IEEE Access 8, pp. 78713–78725, 2020

21. TIOBE programming community index, TIOBE Software BV, `https://www.tiobe.com/tiobe-index/`

22. The LLVM Compiler Infrastructure, LLVM Foundation, `https://llvm.org/`

23. `https://cwe.mitre.org/` (accessed 23 July 2021)

24. `https://cwe.mitre.org/data/definitions/825.html` (accessed 15 July 2021)

25. `https://cwe.mitre.org/data/definitions/562.html` (accessed 15 July 2021)

26. Address Sanitizer. `https://clang.llvm.org/docs/AddressSanitizer.html` (accessed 15 July 2021)

Appendix A

Usage

The code for this project is based on a SeaHorn fork and is publicly available at `https://github.com/SHeijungs/seahorn/tree/uaf`. The main command to call our code is `sea uaf [FLAGS] [FILES]`. The available flags are the following:

- `-UAF`[version]: select the version of instrumentation as described in Chapter 5. Available versions are 1, 2, 3 and 4.
- `--doubleFreeOnly`: Omit use-after-free checks and only check for double free.
- `--moveStackAllocs`: Move all stack allocations to the heap. This is currently required for correctness when the doubleFreeOnly flag is not given.
- `--replaceMemset`: Replace `memset` and `memcpy` by dummy functions. Some solvers have trouble with `memset` and `memcpy`. This option is there to get our code to work on some benchmarks where `memset` and `memcpy` are used.

Appendix B

Instrumentation Code

These are the files we made from scratch to add to SeaHorn. Existing SeaHorn files we edited for integration are not shown here. The full source of SeaHorn with our adaptations can be found at `https://github.com/SHeijungs/seahorn/tree/uaf`.

B.1 Instrumentation Pass

```
/* based on tutorial at https://www.cs.cornell.edu/~
   asampson/blog/llvm.html*/

#include "llvm/Pass.h"
#include "llvm/Transforms/Instrumentation.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"
#include <vector>
#include <stack>
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/InstIterator.h"
#include "llvm/IR/InstrTypes.h"
#include "llvm/Support/CommandLine.h"

using namespace llvm;

cl::opt<bool> doubleFreeOnly ("doubleFreeOnly", cl::
   desc("Only check for double free bugs, not use-
   after-free."));
cl::opt<bool> moveStackAllocs ("moveStackAllocs", cl::
   desc("Move all stack allocations to the heap"));
cl::opt<bool> replaceMemset ("replaceMemset", cl::desc
   ("Replace memset and memcpy calls with code that
    simulates the memory use. Currently only for
    version 4"));


enum bitWidthEnum {
  m32, m64
};

enum versionEnum {
  UAF1, UAF2, UAF3, UAF4
};

/*enum memsetWidthEnum {
  memset, m64
};*/

cl::opt<bitWidthEnum> bitWidth(cl::desc("Choose the
    bit width:"),
  cl::values(
    clEnumVal(m32, "32 bits"),
    clEnumVal(m64, "64 bits")
  )
);

cl::opt<versionEnum> UAFversion(cl::desc("Choose
    instrumentation version:"),
  cl::values(
    clEnumVal(UAF1, "Version 1"),
    clEnumVal(UAF2, "Version 2"),
    clEnumVal(UAF3, "Version 3"),
    clEnumVal(UAF4, "Version 4")
  )
);

namespace {
  struct instrUAF : public ModulePass {
    static char ID;
    instrUAF() : ModulePass(ID) {}
    private:
    //BuilderTy builder;
    Function *use_prehook;
    Function *instrumentedMalloc;
    Function *mallocPostHook;
    Function *instrumentedFree;
    Function *instrumentedMemset;
    Function *instrumentedMemcpy;
```

```
void instrumentUse(Value *ptr, IRBuilder<> builder
    ){
  if(isa<Constant>(ptr))
    return;
  auto *ArgPtr=builder.CreateBitCast(ptr, builder.
     getInt8PtrTy());
  //errs() << ArgPtr << "\n";
  builder.CreateCall(use_prehook, ArgPtr);
}

CallInst *instrumentAlloca(AllocaInst *AI,
    IRBuilder<> builder, Module &M){
  Type *allocatedType=AI->getAllocatedType();
  //allocatedType->print(errs());
  DataLayout layout(&M);
  int typeSize = layout.getTypeAllocSize(
     allocatedType);
  //errs() << "\ntype size: " << typeSize << "\n";
  CallInst *callInst;
  if(bitWidth==m32){
    FunctionType *ft=FunctionType::get(builder.
       getInt8PtrTy(), builder.getInt32Ty(), false
       );
    callInst = builder.CreateCall(ft,
       instrumentedMalloc, ConstantInt::get(M.
       getContext(), APInt(32, typeSize, 10)));
  } else {
    FunctionType *ft=FunctionType::get(builder.
       getInt8PtrTy(), builder.getInt64Ty(), false
       );
    callInst = builder.CreateCall(ft,
       instrumentedMalloc, ConstantInt::get(M.
       getContext(), APInt(64, typeSize, 10)));
  }
  Value *castedCall;
  if(allocatedType == builder.getInt8Ty()) //
     typecast if a different type is needed
    castedCall = callInst;
  else
    castedCall = builder.CreateBitCast(callInst,
       allocatedType->getPointerTo());
  AI->replaceAllUsesWith(castedCall);
  return callInst;
}
```

```
void freeStackVariables(IRBuilder<> builder, std::
    stack<CallInst*> &freeList){
  //if(freeList.empty())
    //errs() << "Nothing to free in this function\
        n";
  while(!freeList.empty()){
    CallInst* allocedVar = freeList.top();
    //errs() << "Created free\n";
    freeList.pop();
    builder.CreateCall(instrumentedFree,
        allocedVar);
  }
}

/*void instrumentMalloc(CallBase *CB, IRBuilder<>
    builder){
  Value *args[2];
  args[0]=CB;
  args[1]=CB->getCalledOperand();
  builder.CreateCall(mallocPostHook, args);
}*/

public:
int instrumentFunction(Function &F, Module &M){
  //errs() << "Instrumenting function" << F.
      getName();
  std::stack<CallInst*> freeList;
  for (auto i = inst_begin(F), e = inst_end(F); i
      != e; ++i) {
    Instruction *I = &*i;
    IRBuilder<> builder(I);
    builder.SetInsertPoint(I);

    if(!doubleFreeOnly){
      if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
        instrumentUse(LI->getPointerOperand(),
            builder);
      }

      else if(StoreInst *SI = dyn_cast<StoreInst>(
          I)){
        instrumentUse(SI->getPointerOperand(),
            builder);
      }
    }
    if(moveStackAllocs){
      if(AllocaInst *AI = dyn_cast<AllocaInst>(I))
          { //we want to move all allocations to
         the heap so we can analyse them using our
          shadow memory
        freeList.push(instrumentAlloca(AI, builder
           , M));
      }
      else if(ReturnInst *RI = dyn_cast<ReturnInst
         >(I)) {
        freeStackVariables(builder, freeList);
      }
    }
  }
  return 0;
}
virtual bool runOnModule(Module &M) {
  unsigned version;
  DataLayout layout(&M);
  switch(UAFversion){
    case UAF1:
      instrumentedMalloc = M.getFunction("
         __seahorn_UAF_malloc");
      use_prehook = M.getFunction("
         __seahorn_UAF_use_prehook");
      instrumentedFree = M.getFunction("
         __seahorn_UAF_free");
      printf("using version 1\n");
      break;
    case UAF2:
      instrumentedMalloc = M.getFunction("
         __seahorn_UAF_malloc2");
      use_prehook = M.getFunction("
         __seahorn_UAF_use_prehook2");
      instrumentedFree = M.getFunction("
         __seahorn_UAF_free2");
      printf("using version 2\n");
      break;
    case UAF3:
      instrumentedMalloc = M.getFunction("
         __seahorn_UAF_malloc3");
      use_prehook = M.getFunction("
         __seahorn_UAF_use_prehook3");
      instrumentedFree = M.getFunction("
         __seahorn_UAF_free3");
      printf("using version 3\n");
      break;
  case UAF4:
    instrumentedMalloc = M.getFunction("
        __seahorn_UAF_malloc4");
    use_prehook = M.getFunction("
        __seahorn_UAF_use_prehook4");
    instrumentedFree = M.getFunction("
        __seahorn_UAF_free4");
    printf("using version 4\n");
    break;
  default:
    printf("unknown version\n");
    break;
}
//mallocPostHook = M.getFunction("
    __seahorn_UAF_mallocPostHook");
instrumentedMemset = M.getFunction("
    __seahorn_UAF_memset_fake");
instrumentedMemcpy = M.getFunction("
    __seahorn_UAF_memcpy_fake");
if(Function *malloc = M.getFunction("malloc")){
  malloc->replaceAllUsesWith(instrumentedMalloc)
      ;
  malloc->eraseFromParent();
}
if(replaceMemset){
  if(Function *memset = M.getFunction("llvm.
      memset.p0i8.i32")){
    memset->replaceAllUsesWith(
        instrumentedMemset);
    memset->eraseFromParent();
  }
  if(Function *memset = M.getFunction("llvm.
      memset.p0i8.i64")){
    memset->replaceAllUsesWith(
        instrumentedMemset);
    memset->eraseFromParent();
  }
  if(Function *memcpy = M.getFunction("llvm.
      memcpy.p0i8.p0i8.i32")){
    memcpy->replaceAllUsesWith(
        instrumentedMemcpy);
    memcpy->eraseFromParent();
  }
  if(Function *memcpy = M.getFunction("llvm.
      memcpy.p0i8.p0i8.i64")){
            memcpy->replaceAllUsesWith(
                instrumentedMemcpy);
            memcpy->eraseFromParent();
          }
        }
        /*if(Function *calloc = M.getFunction("calloc"))
          calloc->replaceAllUsesWith(M.getFunction("
              __seahorn_UAF_calloc"));
        if(Function *realloc = M.getFunction("realloc"))
          realloc->replaceAllUsesWith(M.getFunction("
              __seahorn_UAF_realloc"));*/
        if(Function *free = M.getFunction("free")){
          free->replaceAllUsesWith(instrumentedFree);
          free->eraseFromParent();
        }
        //use_prehook = M.getFunction("
            __seahorn_UAF_use_prehook");
        for (auto& F : M) {
          if(!F.getName().startswith("__seahorn")){ //
              don't instrument our own instrumentation
              functions
            //if(F.getName().startswith("llvm."))
              //errs()<<F.getName()<<"\n";
            if(UAFversion==UAF4&&F.getName().startswith
                ("main")){
              auto i = inst_begin(F);
              Instruction *I = &*i;
              IRBuilder<> builder(I);
              builder.CreateCall(M.getFunction("
                  __seahorn_UAF_init"));
            }
            instrumentFunction(F, M);
          }
        }
        return true;
      }
    };
}

namespace seahorn {
Pass *createUAFPass() {
  return new instrUAF();
}
} // namespace seahorn

char instrUAF::ID = 0;
```

```
static RegisterPass<instrUAF> X("instrUAF", "
   instrument for detection of UAF and double free
   bugs",
                              false /* Only looks at
                                 CFG */,
                              false /* Analysis Pass
                                 */);
```

B.2 Instrumentation Library

```
#include "seahorn/seahorn.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>
const size_t __seahorn_UAF_HEAP_SIZE=1LL<<20; //
   SeaHorn can reason about absurdly large arrays.
const size_t __seahorn_MAX_ALLOCATIONS=100;
char __seahorn_UAF_heap[__seahorn_UAF_HEAP_SIZE];
size_t __seahorn_UAF_heap_sizes[
   __seahorn_MAX_ALLOCATIONS];
char __seahorn_UAF_shadow_heap[__seahorn_UAF_HEAP_SIZE
   ];
uint64_t __seahorn_UAF_firstFree=0;
size_t __seahorn_allocationCount=0;
char* __seahorn_allocationStarts[
   __seahorn_MAX_ALLOCATIONS];
extern bool __seahorn_UAF_nondet(void);
extern char *__seahorn_UAF_nondet_ptr(void);
/*based on implementations from https://git.busybox.
   net/uClibc/tree/libc/stdlib/malloc-simple/alloc.c*/
extern char *__seahorn_UAF_malloc_redir(size_t size);
bool __seahorn_UAF_active=0;
char *__seahorn_UAF_bgn;
char *__seahorn_UAF_end;
bool __seahorn_UAF_freed=0;

void __seahorn_UAF_init(void){
  __seahorn_UAF_bgn=__seahorn_UAF_nondet_ptr();
  __seahorn_UAF_end=__seahorn_UAF_nondet_ptr();
}

char *__seahorn_UAF_malloc(size_t size){
        char *result;
        if(size == 0)
          return NULL;
        if (__seahorn_UAF_nondet()) {
                /*As per the specification malloc can
                    fail at any time.
                We model this non-deterministically*/
                return NULL;
        }
  if(size>__seahorn_UAF_HEAP_SIZE-
      __seahorn_UAF_firstFree-sizeof(size_t))
  /*safely check whether we have enough memory.*/
    return NULL;
  result=&__seahorn_UAF_heap[__seahorn_UAF_firstFree];
  __seahorn_UAF_firstFree+=size+sizeof(size_t);
        * (size_t *) result = size;
        return(result + sizeof(size_t));
}

char *__seahorn_UAF_malloc2(size_t size) __attribute__
    ((optnone)){
        char *result;

  if(size==0)
    return NULL;
  /*safely check whether we have enough memory.*/
  assume(size<=__seahorn_UAF_HEAP_SIZE-
      __seahorn_UAF_firstFree);
  result=&__seahorn_UAF_heap[__seahorn_UAF_firstFree];
  __seahorn_UAF_heap_sizes[__seahorn_UAF_firstFree]=
      size;
  for(int i=0;i<size;++i){
          __seahorn_UAF_shadow_heap[
              __seahorn_UAF_firstFree+i]=1;
        }
  __seahorn_UAF_firstFree+=size;
        return(result);
}

char *__seahorn_UAF_malloc3(size_t size){
        char *result;

  if(size==0)
    return NULL;
  assume(size<__seahorn_UAF_HEAP_SIZE-
      __seahorn_UAF_firstFree);
  assume(__seahorn_allocationCount<
      __seahorn_MAX_ALLOCATIONS);
  result=&__seahorn_UAF_heap[__seahorn_UAF_firstFree];
  __seahorn_UAF_heap_sizes[__seahorn_allocationCount
      ]=1;
  __seahorn_allocationStarts[__seahorn_allocationCount
      ]=result;
  __seahorn_UAF_firstFree+=size;
  ++__seahorn_allocationCount;
  __seahorn_allocationStarts[__seahorn_allocationCount
      ]=NULL;
  printf("memory allocated at address %p\n", result);
        return(result);
}

char * __seahorn_UAF_malloc4(size_t size){
  assume(size<__seahorn_UAF_HEAP_SIZE-
      __seahorn_UAF_firstFree);
  /*char *result;
  result=&__seahorn_UAF_heap[__seahorn_UAF_firstFree];
  __seahorn_UAF_firstFree+=size;
  */
  char *result=__seahorn_UAF_malloc_redir(size);
  if(!__seahorn_UAF_active&&__seahorn_UAF_nondet()){
    __seahorn_UAF_active=1;
    assume(__seahorn_UAF_bgn==result);
    assume(__seahorn_UAF_end=result+size);
  } else {
    assume(result<__seahorn_UAF_bgn);
  }
        return result;
}

char *__seahorn_UAF_calloc(size_t nmemb, size_t lsize)
{
        char *result;
        size_t size = lsize*nmemb;

        /* guard vs integer overflow, but allow nmemb
         * to fall through and call malloc(0) */
        if(nmemb&&lsize != (size/nmemb)){
                return NULL;
        }
        result = __seahorn_UAF_malloc(size);/*
        if (result != NULL) { // either slows down the
            analysis too much or gets turned into
            memset by the compiler
                  for(char *i=result; i<result+size; ++i
                     )
                    *i=0;
        }*/
        return result;
}

void __seahorn_UAF_free(char*ptr){
  if (ptr == NULL)
                  return;
        ptr -= sizeof(size_t);
        sassert(* (size_t *) ptr!=0);
        *(size_t *)ptr = 0;
}

void __seahorn_UAF_free2(char*ptr){
  if (ptr == NULL)
                  return;
        //ptr -= sizeof(size_t);
        size_t index = ptr-__seahorn_UAF_heap;
        size_t size=__seahorn_UAF_heap_sizes[index];
        sassert(size!=0);
        for(int i=0;i<size;++i){
          sassert(__seahorn_UAF_shadow_heap[index+i
             ]!=0);
          __seahorn_UAF_shadow_heap[index+i] = 0;
        }
}

void __seahorn_UAF_free3(char*ptr){
  printf("freeing memory at address %p\n", ptr);
  if (ptr == NULL)
                  return;
  int index=0;
  while((size_t)ptr>=(size_t)
      __seahorn_allocationStarts[index]&&index<
      __seahorn_allocationCount){
    ++index;
  }
  --index;
        sassert(ptr==__seahorn_allocationStarts[index
           ]);
        size_t size=__seahorn_UAF_heap_sizes[index];
        sassert(size!=0);
        __seahorn_UAF_heap_sizes[index]=0;
```

```
}

void __seahorn_UAF_free4(char*ptr){
  if(__seahorn_UAF_active&&ptr==__seahorn_UAF_bgn){
    assume(ptr<=__seahorn_UAF_end);
    sassert(!__seahorn_UAF_freed);
    __seahorn_UAF_freed=true;
  }
}

char *__seahorn_UAF_realloc(void *ptr, size_t size)
{
        void *newptr = NULL;

        if(!ptr)
                return __seahorn_UAF_malloc(size);
        if(!size){
                __seahorn_UAF_free(ptr);
                return __seahorn_UAF_malloc(0);
        }

        newptr = __seahorn_UAF_malloc(size);
        if(newptr){
                size_t old_size = *((size_t *) (ptr -
                   sizeof(size_t)));
                memcpy(newptr, ptr, (old_size < size ?
                    old_size : size));
                __seahorn_UAF_free(ptr);
        }
        return newptr;
}

void __seahorn_UAF_use_prehook(char*ptr){
  //printf("using address %p\n", ptr);
        //if(ptr < &__seahorn_UAF_heap[0] || ptr > &
           __seahorn_UAF_heap[__seahorn_UAF_HEAP_SIZE
           -1])
        ptr -= sizeof(size_t);
        sassert(* (size_t *) ptr!=0);
}

void __seahorn_UAF_use_prehook2(char*ptr){
  //printf("using address %p\n", ptr);
        //if(ptr < &__seahorn_UAF_heap[0] || ptr > &
           __seahorn_UAF_heap[__seahorn_UAF_HEAP_SIZE
           -1])
        sassert(__seahorn_UAF_shadow_heap[ptr-
            __seahorn_UAF_heap]!=0);
}

void __seahorn_UAF_use_prehook3(char*ptr){
  printf("using address %p\n", ptr);
  unsigned int index=0;
  bool t=0;
  while((size_t)ptr>=(size_t)
      __seahorn_allocationStarts[index]&&index<
      __seahorn_allocationCount){
    ++index;
    t=1;
  }
  sassert(t);
  --index;
  size_t size=__seahorn_UAF_heap_sizes[index];
        sassert(size!=0);
}

void __seahorn_UAF_use_prehook4(char*ptr)
   __attribute__ ((optnone)){
  if(__seahorn_UAF_active&&ptr>=__seahorn_UAF_bgn&ptr
     <=__seahorn_UAF_end){
    assume(ptr<=__seahorn_UAF_end);
    sassert(!__seahorn_UAF_freed);
  }
}

void __seahorn_UAF_memset_fake(char *str, char c,
   size_t n, bool x) __attribute__ ((optnone)){
  __seahorn_UAF_use_prehook4(str);
}

void __seahorn_UAF_memset_real(char *str, char c,
   size_t n, bool x) __attribute__ ((optnone)){
  __seahorn_UAF_use_prehook4(str);
  for(int i=0;i<n;++i)
    str[i]=c;
}

void __seahorn_UAF_memcpy_fake(char *str, char *str2,
   size_t n, bool x) __attribute__ ((optnone)){
  __seahorn_UAF_use_prehook4(str);
  __seahorn_UAF_use_prehook4(str2);
}
```

```
void __seahorn_UAF_memcpy_real(char *str, char *str2,
    size_t n, bool x) __attribute__ ((optnone)){
  __seahorn_UAF_use_prehook4(str);
  __seahorn_UAF_use_prehook4(str2);
  for(int i=0;i<n;++i)
    str[i]=str2[i];
}
```

Appendix C

Evaluation Scripts

C.1 Double Free Evaluation Script

```
#!/bin/bash

shopt -s extglob

oPath=evaluation/DF$1
mkdir $oPath

for uafVersion in {1..4}; do
  mkdir $oPath/version$uafVersion
  cat /dev/null > $oPath/version$uafVersion/
     truePositives.txt
  cat /dev/null > $oPath/version$uafVersion/
     falsePositives.txt
  cat /dev/null > $oPath/version$uafVersion/
     trueNegatives.txt
  cat /dev/null > $oPath/version$uafVersion/
     falseNegatives.txt
  cat /dev/null > $oPath/version$uafVersion/
     positiveUnknown.txt
  cat /dev/null > $oPath/version$uafVersion/
     negativeUnknown.txt
  for testCase in Juliet/C/testcases/
     CWE415_Double_Free/s??/CWE415_*[[:digit:]]?(a).c;
      do
    testCase="${testCase%.c}"
    if testcase=~*a; then
      testCase="${testCase%a}"
    fi
```

```
echo "running bad test for $testCase"
flist=""
for fname in $testCase?([[:alpha:]]).c; do #
   Combine test cases with the same number
  fname="${fname%.*}"
  flist+=" "$fname".bc"
  clang-10 -c -I Juliet/C/testcasesupport -D
     OMITGOOD -D INCLUDEMAIN -emit-llvm -
     D__SEAHORN__ -fdeclspec -O1 -Xclang -disable-
     llvm-optzns -fgnu89-inline -m64 -o $fname.bc
     $fname.c
done
output=$(timeout $1 sea uaf -m64 --replaceMemset
   --doubleFreeOnly --uafVersion=$uafVersion --
   horn-use-euf-gen=false $flist)
if echo $output | grep "[[:space:]]sat$" >/dev/
   null; then
  echo $testCase >> $oPath/version$uafVersion/
     truePositives.txt
elif echo $output | grep "[[:space:]]unsat$" >/dev
   /null; then
  echo $testCase >> $oPath/version$uafVersion/
     falseNegatives.txt
else
  echo $testCase >> $oPath/version$uafVersion/
     positiveUnknown.txt
fi
echo "running good test for $testCase"
flist=""
for fname in $testCase?([[:alpha:]]).c; do
  fname="${fname%.*}"
  flist+=" "$fname".bc"
  clang-10 -c -I Juliet/C/testcasesupport -D
     OMITBAD -D INCLUDEMAIN -emit-llvm -
     D__SEAHORN__ -fdeclspec -O1 -Xclang -disable-
     llvm-optzns -fgnu89-inline -m64 -o $fname.bc
     $fname.c
done
output2=$(timeout $1 sea uaf -m64 --replaceMemset
   --doubleFreeOnly --uafVersion=$uafVersion --
   horn-use-euf-gen=false $flist)
if echo $output2 | grep "[[:space:]]sat$" >/dev/
   null; then
  echo $testCase >> $oPath/version$uafVersion/
     falsePositives.txt
elif echo $output2 | grep "[[:space:]]unsat$" >/
```

```
          dev/null; then
        echo $testCase >> $oPath/version$uafVersion/
           trueNegatives.txt
      else
        echo $testCase >> $oPath/version$uafVersion/
           negativeUnknown.txt
      fi
    done
done
```

C.2 Use-after-free Evaluation Script

```
#!/bin/bash

shopt -s extglob

oPath=evaluation/UAF$1

mkdir $oPath

for uafVersion in {2..4}; do
  mkdir $oPath/version$uafVersion
  cat /dev/null > $oPath/version$uafVersion/
     truePositives.txt
  cat /dev/null > $oPath/version$uafVersion/
     falsePositives.txt
  cat /dev/null > $oPath/version$uafVersion/
     trueNegatives.txt
  cat /dev/null > $oPath/version$uafVersion/
     falseNegatives.txt
  cat /dev/null > $oPath/version$uafVersion/
     positiveUnknown.txt
  cat /dev/null > $oPath/version$uafVersion/
     negativeUnknown.txt
  for testCase in Juliet/C/testcases/
     CWE416_Use_After_Free/CWE416_*[[:digit:]]?(a).c;
     do
    testCase="${testCase%.c}"
    if testcase=~*a; then
      testCase="${testCase%a}"
    fi
    echo "running bad test for $testCase"
    flist=""
    for fname in $testCase?([[:alpha:]]).c; do #
       Combine test cases with the same number
      fname="${fname%.*}"
  flist+=" "$fname".bc"
  clang-10 -c -I Juliet/C/testcasesupport -D
     OMITGOOD -D INCLUDEMAIN -emit-llvm -
     D__SEAHORN__ -fdeclspec -O1 -Xclang -disable-
     llvm-optzns -fgnu89-inline -m64 -o $fname.bc
     $fname.c
done
output=$(timeout $1 sea uaf -m64 --replaceMemset
   --uafVersion=$uafVersion --horn-use-euf-gen=
   false $flist)
if echo $output | grep "[[:space:]]sat$" >/dev/
   null; then
  echo $testCase >> $oPath/version$uafVersion/
     truePositives.txt
elif echo $output | grep "[[:space:]]unsat$" >/dev
   /null; then
  echo $testCase >> $oPath/version$uafVersion/
     falseNegatives.txt
else
  echo $testCase >> $oPath/version$uafVersion/
     positiveUnknown.txt
fi
echo "running good test for $testCase"
flist=""
for fname in $testCase?([[:alpha:]]).c; do
  fname="${fname%.*}"
  flist+=" "$fname".bc"
  clang-10 -c -I Juliet/C/testcasesupport -D
     OMITBAD -D INCLUDEMAIN -emit-llvm -
     D__SEAHORN__ -fdeclspec -O1 -Xclang -disable-
     llvm-optzns -fgnu89-inline -m64 -o $fname.bc
     $fname.c
done
output2=$(timeout $1 sea uaf -m64 --replaceMemset
   --uafVersion=$uafVersion --horn-use-euf-gen=
   false $flist)
if echo $output2 | grep "[[:space:]]sat$" >/dev/
   null; then
  echo $testCase >> $oPath/version$uafVersion/
     falsePositives.txt
elif echo $output2 | grep "[[:space:]]unsat$" >/
   dev/null; then
  echo $testCase >> $oPath/version$uafVersion/
     trueNegatives.txt
else
  echo $testCase >> $oPath/version$uafVersion/
```

```
          negativeUnknown.txt
    fi
  done
done
```