Targeted Symbolic Execution for UAF Vulnerabilities

Zhen Huang
School of Computing
DePaul University
Chicago, USA
zhen.huang@depaul.edu

***Abstract*—Symbolic execution is a popular software testing technique that can systematically examine program code to find bugs. Owing to the prevalence of software vulnerabilities, symbolic execution has been extensively used to detect software vulnerabilities. A major challenge of using symbolic execution to find complicated vulnerabilities such as use-after-free is to not only direct symbolic execution to explore relevant program paths but also explore the paths in a specific order. In this paper, we describe a targeted symbolic execution, called UAFDetect, for finding use-after-free vulnerabilities efficiently. UAFDetect guides symbolic execution to focus on paths that are likely to cause a use-after-free by pruning paths that are unlikely or infeasible to cause that. It uses dynamic typestate analysis to identify unlikely paths and static control flow analysis to identify infeasible paths. UAFDetect performs typestate analysis to detect the occurrence of use-after-free vulnerabilities. Upon the detection of a vulnerability, UAFDetect generates an exploit to trigger the vulnerability. We develop the prototype of UAFDetect and evaluate it on real-world use-after-free vulnerabilities. The evaluation demonstrates that UAFDetect can find use-after-free vulnerabilities effectively and efficiently.**

***Index Terms*—Software testing, symbolic execution, vulnerability detection, use-after-free, typestate analysis, program analysis.**

I. INTRODUCTION

Symbolic execution [1] is a program analysis technique that systematically explores program paths and automatically produces test inputs. It aims to address the challenges of achieving high code coverage and generating comprehensive test inputs. For decades, it has been used in many areas such as test input generation [2], bug finding [3], and security testing [4].

By representing program input values as symbolic values that can denote any arbitrary values, symbolic execution runs a program and follows every branch that is feasible under the constraints of the symbolic input values. This exhaustive exploration of program paths enables symbolic execution to find complex bugs that are difficult to detect.

Use-after-free (UAF) is a complex software bug and is ranked as one of the top 25 most dangerous software weaknesses [5]. It can lead to severe security issues including program crash, data corruption, information leak, and code injection. UAF vulnerabilities exist in all kinds of software such as OS kernel [6], web browser [7], multimedia framework [8], and remote desktop client [9].

Three different program operations take part in a UAF: a malloc operation that allocates a memory region, a free operation that deallocates the memory region, and a use operation that references the deallocated memory region. A UAF happens when a program first performs a malloc operation, then a free operation on a pointer that points to the memory region allocated by the malloc, and finally a use operation via a pointer that points to already freed memory region. As we can see, it requires the three operations to occur in a specific order to trigger a UAF.

This makes it challenging for existing symbolic execution tools to detect UAFs, because they are typically designed to aim for higher code coverage and not designed to guide program execution to trigger UAFs. They spend most of the time exploring program paths irrelevant to UAFs and are unable to invoke the three operations in the specific order even when they happen to reach relevant program paths. To find UAFs efficiently, a symbolic execution tool must direct program executions not only toward program paths related to UAFs but also invoke the operations in the specific order.

In this work, we address the challenges of detecting UAFs efficiently by combining typestate analysis [10] with symbolic execution. Our work, called UAFDetect, uses a targeted symbolic execution that performs typestate analysis based path pruning to guide symbolic execution to find UAFs. The typestate analysis enables symbolic execution to keep track of the order of the operations involved in a UAF, while the path pruning forces symbolic execution to focus on program paths relevant to UAFs.

Similar to prior work [11], our typestate analysis models the lifecycle of a pointer as a finite-state machine, called *typestate*, which defines the set of operations, i.e. malloc, free, and use, that can be applied to each state of a pointer, and the state transitions that can be caused by these operations. A special error state denotes the occurrence of a UAF. Based on the typestate analysis, our symbolic execution finds a UAF when the typestate of a pointer ends in the error state.

UAFDetect uses two major techniques, a novel unlikely path pruning and an infeasible path pruning. The two techniques prevent symbolic execution from wasting time executing program paths that are unlikely or infeasible to trigger UAFs. When symbolic execution reaches such a program path, they terminate the execution of the path so that symbolic execution can quickly move on to explore other paths.

The unlikely path pruning leverages typestate analysis to identify program paths that are *unlikely* to trigger UAFs when they do *not* invoke UAF-related operations in the specific order that can trigger a UAF. We developed the unlikely path pruning based on the observation that a UAF is unlikely to occur when the operations does not follow the exact order of malloc, free, and use. We consider such a path as an unlikely path because there is no guarantee that this path can never trigger a UAF ultimately. Our symbolic execution performs typestate analysis to identify and prune unlikely paths.

The infeasible path pruning identifies and prunes paths that can never cause a UAF. It uses reachability analysis to identify program paths that can *never* reach the statements invoking one or more UAF-related operations. UAFDetect runs the reachability analysis statically on the program and feeds the analysis results to the symbolic execution, which use the results to identify and prune infeasible paths.

This paper makes the following main contributions:

- We propose an approach called UAFDetect to find UAF vulnerabilities efficiently. Based on typestate analysis and static reachability analysis, the approach uses symbolic execution to execute target programs and guide program executions to trigger UAFs efficiently by pruning program paths that are unlikely or infeasible to cause UAFs.
- We develop two program path pruning technique, unlikely paths pruning and infeasible path pruning, to improve the efficiency of symbolic execution in finding UAFs.
- We have implemented the approach in a prototype and evaluated it on real world UAF vulnerabilities. We describe our design and evaluation in the paper.
- Our evaluation on UAFDetect shows that it can effectively and efficiently discover UAF vulnerabilities in real world programs.

II. Background and Related Work

A. Typestate Analysis

A typestate analysis [10] is a program analysis commonly used for identifying complex bugs that involve multiple program operations and multiple program states. These operations can be valid for some specific program states, but can be invalid for some other program states. A typestate analysis models these operations and states as a typestate, and identifies a bug by finding operations that are invalid for a typestate.

We define a typestate as a finite-state machine $(\sum, S, s, \delta, F)$, where $\sum$ is the set of operations, S is the set of states, s is the initial state, δ is a state transition function, and F is the set of final states. A typestate is initialized to the initial state s and transits to other states, based on the invoked operations, the current state, and the state transition function δ, which maps a state $m \in S$ and an operation $op \in \sum$ on p to a state $n \in S$, meaning that when op is applied to a typestate in state m, the state will be changed to n.

B. Typestate Analysis for UAF

A use-after-free (UAF) vulnerability is a complex bug that occurs when a program attempts to use a freed pointer. It can cause unexpected program behavior and allow an attacker to compromise computer systems. Due to its complexity, a UAF can be manifested only by invoking multiple operations on a pointer in a specific order.

To trigger a UAF, a malloc operation needs to be first invoked to set a pointer to a valid memory region, then a free operation is invoked on the pointer to deallocate the memory region, and lastly a use operation is invoked on the pointer.

These operations involved in a UAF and the consequences of these operations can be modeled as a typestate for a pointer, as shown in Figure 1. The typestate of a pointer starts in the `Init` state. It can transit to the `Allocated` state upon a malloc operation. When a typestate is in the `Allocated` state, any use operation will not change the state. The typestate transits to the `Deallocated` state upon a free operation. And a free operation on the `Deallocated` state causes the typestate to end in the `Error` state.

C. Symbolic Execution

Symbolic execution runs target programs with symbolic input values to explore all possible program paths [12]. During an execution, the symbolic values constitute the program path constraints that determine whether a program path is feasible or not. Symbolic execution automatically transforms symbolic input values into concrete input values, which can be used as a test case, when a program path terminates.

Path explosion and constraint solving are two major challenges of applying symbolic execution to complex real-world programs [13]. Branches in a target program will cause the number of all possible paths to grow exponentially, i.e. path explosion, and substantially limit symbolic execution to scale to large and complex target programs. Constraint solving is used by symbolic execution to check whether path constraints are solvable and can incur dramatic performance overhead.

Techniques have been proposed to address the challenges by optimizing symbolic execution in several directions: to avoid executing uninteresting paths [14]–[17], to execute only individual functions instead of a whole program [18], to merge execution states for multiple paths [19], [20], to combine symbolic execution with fuzzing [21]–[23], or to execute a program in the backward order [24], [25].

In principle, our work guides symbolic execution to UAFs by avoiding executing uninteresting paths. As far as we know, we are the first to use typestate analysis during symbolic execution.

D. UAF Detection

Many techniques have been proposed to detect and address vulnerabilities [3], [4], [11], [16], [23], [26]–[38], particularly UAF vulnerabilities. On the one hand, some techniques detect exploits aiming to trigger UAFs at runtime and then mitigate the exploits [30]–[32]. Unlike our work, these techniques do not generate exploits, the inputs to trigger UAFs. On the

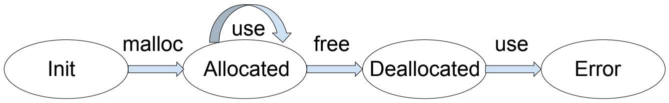

Fig. 1. Typestate for detecting a UAF: each oval represents a state; each arrow denotes a state transition and is labeled with the operation that caused the transition.

other hand, some techniques find UAFs and generate exploits to trigger the detected UAFs [11], [29]. We focus on these techniques as our work serves the same purpose.

Some UAF detection tools are based on fuzzing that finds bugs by executing a target program with some seed inputs, and mutating these inputs into new inputs based on execution results. UAFuzz [29] uses a fuzzing strategy that favors inputs that can guide program execution toward operations relevant to UAFs. UAFuzz sends the inputs it generated to a profiling tool such as Valgrind to determine whether the inputs cause UAFs. Unlike our work, UAFuzz does not leverage typestate analysis and focuses on binary code.

UAFL [11] incorporates typestate analysis into fuzzing to detect UAFs. It uses static analysis to identify the sequence of operations that can cause UAFs, and instruments code into target programs to record which operations has been executed. The recording is sent back to UAFL as a feedback to guide its input generation. Our work uses typestate analysis in symbolic execution to detect UAFs and to terminate paths unlikely or infeasible to cause UAFs.

Like our work, Feist et al. propose to combine static analysis with symbolic execution to find UAFs [28]. They focus on binary code and perform static program slicing to retain only the program slices that may lead to UAFs and then run symbolic execution on these retained paths. Due to the approximations used by the program slicing, some retained program slices can be infeasible.

III. PROBLEM DEFINITION

A use-after-free (UAF) vulnerability is one of the most common and dangerous software weaknesses. It occurs when a previously freed pointer is inadvertently dereferenced, which can cause detrimental consequences such as program crash, data corruption, or code injection.

Triggering of UAF. It involves a sequence of three operations to trigger a UAF.

1) A *malloc* operation allocates a memory region and returns the address of the memory region to a pointer
2) A *free* operation deallocates the memory region pointed to by the pointer
3) A *use* operation that dereferences the freed pointer

The three operations must occur in the above order to trigger a UAF. We use a vulnerability in `cxxfilt` to illustrate that.

As listed in Figure 2, two functions are relevant to this UAF. Function `register_Btype` can allocate a memory region and use the memory region via a pointer. Function `squangle_mop_up` deallocates the memory region pointed to by a pointer and then sets the pointer to NULL. The involved pointer `btypevec` is a member field of a structure `work`, whose memory address is passed to these two functions via function arguments.

To trigger the UAF, function `register_Btype` is first called to allocate a memory region and set pointer `btypevec` in a `work` structure to point to the memory region at line 20. Then function `squangle_mop_up` is called and it deallocates the memory region by freeing pointer `btypevec` at line 7 and assigns NULL to the pointer at line 10. Lastly, function `register_Btype` is called again and it attempts to use the already freed pointer `btypevec` at line 29, at which point a null-pointer dereference happens.

```
1  void squangle_mop_up(struct work_stuff *work)
2  {
3    // free
4    free((char *)work->btypevec);
5    work->btypevec = NULL;
6  }
7
8  int register_Btype(struct work_stuff *work)
9  {
10   int ret;
11   if (work->numb >= work->bsize)
12   {
13     work->bsize = 5;
14     // malloc
15     work->btypevec = XNEWVEC(char *, work->bsize);
16   }
17   ret = work->numb++;
18   // use
19   work->btypevec[ret] = NULL;
20   return(ret);
21 }
22
23 int demangle_signature(struct work_stuff *work)
24 {
25   ....
26   switch (...)
27   {
28     case 'Q':
29       break;
30     case '0': case '1': case '2':
31       // indirectly calls register_Btype
32       success = demangle_class (work);
33       ....
34   }
35 }
```

Fig. 2. A use-after-free (UAF) vulnerability in `cxxfilt`, a utility that demangles C++ symbols, adapted from CVE-2016-4487.

Consequences of UAF. After a pointer is freed, a program often sets the pointer to NULL and the memory region pointed to by the pointer may be re-allocated for other uses. If the pointer is set to NULL, the use operation (afterward dereference of the pointer) is a null-pointer dereference and will cause the program to crash, as illustrated in Figure 2. If the pointer is *not* set to NULL, the dereference of the pointer can corrupt data or lead to code injection.

Detection of UAF. To be able to detect a UAF, the three operations involved in the UAF need to be triggered orderly. This is particularly challenging when the three operations occur in different functions. An offline detection mechanism such as symbolic execution or fuzzing must not only find the path constraints to reach these operations in different functions, but also executes these operations in a specific order.

For the example UAF listed in Figure 2, its malloc, free, and, use operation all have different calling contexts, i.e. call traces. We omit the code of some functions for brevity.

The malloc operation in function `register_Btype` is called through the call chain: `demangle_qualified` → `register_Btype`.

The use operation in function `squangle_mop_up`, which contains the free operation, is called through the call chain: `delete_work_stuff` → `squangle_mop_up`.

The free operation in function `register_Btype` is called through the call chain: `demangle_signature` → `demangle_class` → `register_Btype` .

IV. UAFDetect

UAFDetect detects use-after-free (UAF) vulnerabilities and generates the inputs to trigger these vulnerabilities. In conjunction with static program analysis, UAFDetect uses symbolic execution to run the code of a target program to detect vulnerabilities and generate inputs. It is specifically designed to detect UAFs effectively and efficiently. This section demonstrates a typical usage of UAFDetect in detecting UAFs, using a C++ symbol demangling tool `cxxfilt` as the target program.

To detect UAFs in `cxxfilt`, UAFDetect performs two major steps. First, UAFDetect statically identifies potential operations that can lead to UAFs. It examines the code of the program to find relevant malloc operations, free operations, and use operations. Second, it uses symbolic execution to run the program. Based on the information provided by the static analysis, the symbolic execution performs typestate analysis, and prunes program paths that will unlikely or never reach the operations or trigger UAFs. For each detected UAF, it produces a concrete input for triggering the UAF.

For the code illustrated in Figure 2, UAFDetect identifies that line 15 has a malloc operation and sets the pointer `work->btypevec` to point to the allocated memory region. It then finds a free operation for `work->btypevec` at line 4 and a use of `work->btypevec` at line 19.

With such information, UAFDetect uses symbolic execution to run the program and conducts typestate analysis when the execution reaches the malloc, the free, and the use.

The typestate for the pointer can be in one of three possible states: `Init`, `Allocated`, and `Deallocated`. The typestate for a pointer is initialized to `Init`. When the malloc at line 15 is invoked, UAFDetect sets the state of the typestate to `Allocated`. When the free at line 4 is invoked, it sets the state to `Deallocated`. When the symbolic execution reaches the use, it checks if the typestate is `Deallocated`. If so, it deems that a UAF occurs and reports the UAF.

Particularly it checks whether the malloc occurs before the free by examining the state of the typestate when the execution arrives at the free. If the state is *not* `Allocated`, it deems that no malloc happens before the free and the execution path is unlikely to cause a UAF. As a result, it prunes the execution path. As an example of the infeasible path, it prune the execution path when it reaches line 28 because execution path can never call function `demangle_class` to reach function `register_Btype` to invoke the malloc or the use.

V. Design

A. Overview

UAFDetect takes the source code of a target program as input, locates memory operations in the program, uses symbolic execution to execute the program, and produces an input to trigger a UAF when the UAF is found. Its symbolic execution engine performs typestate analysis, identifies and prunes the paths that are unlikely or infeasible to trigger UAFs, and to detect UAFs. Figure 3 shows UAFDetect's workflow.

The symbolic execution used by UAFDetect creates symbolic input values for the target program, and systematically executes the program paths. When the special UAF report function is invoked by the target program, it produces concrete input values that satisfy the constraints on the symbolic input values. By pruning the unlikely paths and infeasible paths, UAFDetect enables the symbolic execution to focus on the paths that are more likely to cause UAFs so that it can detect UAFs efficiently. When a UAF is detected, UAFDetect generates a concrete input (exploit) to trigger the UAF.

B. Locating Memory Operations

UAFDetect uses static analysis to locate pointers and the malloc, use, and free operations, which are collectively called UAF-operations, performed on these pointers. For each pointer, the static analysis starts from a malloc operation and follows data dependency to find use and free operations. The static analysis is a flow-sensitive and field-sensitive intra-procedural analysis, which gathers relevant information for each function. The static analysis takes into account pointer aliasing and aggregate operations based on pointer aliases.

The intra-procedural analysis is invoked for all the function in a target program. Algorithm 1 shows the algorithm for locating memory operations. For each function, the intra-procedural analysis searches for malloc operations to find UAF-relevant pointers. It treats function calls to dynamic memory allocation functions, such as `malloc`, `calloc`, and `new`, as malloc operations, and locates the pointers involved in the malloc operations.

For each pointer located in a function, the intra-procedural analysis follows the data dependency graph of the function to find the use and free operations on the pointer. Any access via the pointer is deemed as a use operation. This includes the case when a pointer is dereferenced or used as an array. Any function call to dynamic memory deallocation functions, such as `free` and `delete`, is deemed as a free operation.

The intra-procedural analysis takes a function and the dependency graph for the function as input, and outputs a mapping from the pointers that it located in a function to a list of malloc, use, and free operations on each pointer. Each operation is denoted by a tuple of the instruction performing the operation and the type of the operation.

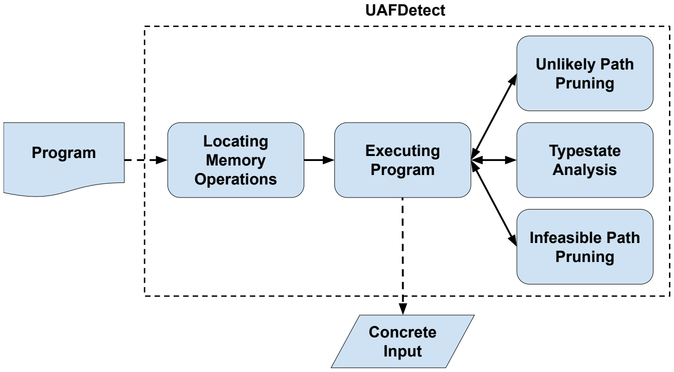

Fig. 3. Workflow of UAFDetect.

Algorithm 1 Locating memory operations.

```
Input:
$F$: a function $F$
DEPENDENCY: data dependency graph for $F$
Output: $M$: a mapping from pointers to operations
  procedure LOCATE_MEMORY_OPERATIONS
      $M \leftarrow \varnothing$
      $Worklist \leftarrow \varnothing$
      for instruction $I \in F$ do
          if IS_MALLOC_OPERATION($I$) then
              $P \leftarrow$ TARGET($I$)
              $Worklist \leftarrow Worklist \cup \{(I, P)\}$ ▷ Add malloc operation
              $M[P] \leftarrow M[P] \cup \{(I, Malloc)\}$
          end if
      end for
      while $Worklist \neq \varnothing$ do
          $(J, Q) \leftarrow$ HEAD($Worklist$)
          $Worklist \leftarrow Worklist - \{(J, Q)\}$
          for instruction $K \in$ DEPENDENCY($J$) do
              if IS_USE_OPERATION($K$) then ▷ Add use operation
                  $M[Q] \leftarrow M[Q] \cup \{(K, Use)\}$
              else if IS_FREE_OPERATION($K$) then ▷ Add free operation
                  $M[Q] \leftarrow M[Q] \cup \{(K, Free)\}$
              else if IS_ASSIGNMENT($K$) then ▷ Handle pointer alias
                  $Worklist \leftarrow Worklist \cup \{(K, Q)\}$
              end if
          end for
      end while
  end procedure
```

C. Dynamic Typestate Analysis

The typestate analysis on pointers is the foundation for UAFDetect to detect UAFs. It keeps track of the state of the typestates for pointers, based on the operations applied on the pointers. As described in Section II, we can detect UAFs by checking the state of a typestate and the operation to be performed on the typestate.

UAFDetect performs dynamic typestate analysis in its symbolic execution engine. The typestate analysis is based on the typestate defined in Figure 1. For the purpose of detecting UAFs, the typestate has four states: `Init`, `Allocated`, `Deallocated`, and `Error`. It also defines three operations: malloc, use, and free. These operations can cause the typestate to transit from one state to another state. For example, applying a free operation when the typestate is in the `Allocated` state will cause the state to transit to the `Deallocated` state.

For each pointer identified in the step of "Locating Memory Operations", UAFDetect associates a typestate variable with the pointer. It modifies the value of the typestate variable when executing each UAF-related operation involving the pointer. The modifications follow the state transit function defined for the typestate. Based on the typestate analysis, UAFDetect identifies and prunes unlikely paths.

D. Unlikely Paths Pruning

We define a path as an unlikely path if the path invokes UAF-related operations in an *unusual* order. Our observations show that such a path is unlikely to cause UAFs. We deem an order of UAF-related operations as unusual if it is neither the normal order of malloc-use-free, nor the UAF-triggering order of malloc-use-free-use. To find unlikely paths, UAFDetect identifies and prunes unlikely paths dynamically during the symbolic execution of the target program. It identifies unlikely paths by checking whether the path performs UAF-related operations in an unusual order. When an unlikely path is identified, it prunes the path.

One approach to find whether the operations follow an unusual order is to check whether each operation to be performed is defined for the current state of the typestate. If the operation is defined for the state, we consider the operation does not cause an unusual order. Otherwise, we consider the operation causes an unusual order.

However, this approach would also consider the orders of operations that can lead to some other types of errors, such as the use of uninitialized pointer, as unusual orders. Because UAFDetect prunes the paths leading to unusual orders, these types of errors will not manifest when their paths are pruned. To avoid pruning paths to other types of errors, we choose to refine our way to find unusual paths to not identify those kind of paths. As a result, it simplifies the check and leaves the check on only the free operation. Because the free operation is defined only for the `Allocated` state, we consider the cases when a free operation is to be performed on a pointer that is not in the `Allocated` state as causing an unusual order.

Our symbolic execution engine performs the check on unlikely paths into a target program. One check is performed before each free operation identified by the step of locating memory operations. Based on the typestate analysis, the check examines the state of the typestate associated with the pointer on which a free operation is applied. If the state is not `Allocated`, it prunes the execution path by calling the `assert` function to terminate the path.

E. Infeasible Paths Pruning

An infeasible path is a path that can never cause a UAF. We consider a path as an infeasible path if the path will never invoke one or more of the UAF-related operations. UAFDetect identifies infeasible paths statically and prunes infeasible paths during the symbolic execution.

To identify infeasible paths, UAFDetect performs context-sensitive and flow-sensitive inter-procedural reachability analysis on a target program. The analysis finds paths that cannot reach UAF-related functions. It consists of two steps: identifying UAF-related functions and labeling infeasible paths.

The first step builds a call-graph of the target program, and then identifies functions that can directly or indirectly invoke functions contain UAF-related operations. These functions are called UAF-related functions. The building of the call graph takes into account function calls via function pointers. It conservatively considers any function whose address has been directly used in a call, or stored to variables as potential callees via function pointers. It then matches callers and callees by matching function prototypes.

The second step works on each function of a target program. It follows the function's control flow graph (CFG) to find program paths that can never reach UAF-related operations directly nor indirectly via calls to UAF-related functions. For such paths, it identifies the earliest branch on the path that diverts the path away from invoking UAF operations. We call this kind of branches as infeasible branches.

VI. EVALUATION

We evaluate UAFDetect on seven real world UAF vulnerabilities in C/C++ programs. First, we describe the environment of our experiments. Second, we measure the effectiveness and performance of UAFDetect on detecting the vulnerabilities. Last, we use case studies to describe the details on how our targeted symbolic execution works on each vulnerability.

A. Experimental Setup

We implement the prototype of UAFDetect in LLVM [39], a compiler framework, and on top of KLEE [40], a symbolic execution engine. We also incorporate the input preconditions developed by SPDetect [17] in UAFDetect. All our evaluations were performed on a 32-core 2.2GHz workstation with 128 GB memory. The workstation runs 64-bit Ubuntu 20.04.

B. Vulnerability Detection

We found real world vulnerabilities from popular online vulnerability databases and exploit databases including CVE - MITRE [41], BugZilla [42], and benchmarks used by prior work [29]. We were able to reproduce seven UAF vulnerabilities. They are from a myriad of programs, which serve different purposes including archive repairing, image processing, data conversion, and programming. Their sizes range from 544 to 115,862 lines of source code. Table I shows the information on these programs. For each program, the column "#SLOC" is its number of C/C++ source code lines and the column "#Bitcode" is its number of LLVM bitcode instructions.

TABLE I
LIST OF EVALUATED PROGRAMS.

Program	Type	Version	#SLOC	#Bitcode
autotrace	Image Processing	0.31.1	19,264	100,027
bzip2recover	Archive Repairing	1.0.6	544	2,085
cxxfilt	Programming	2.26	1,234	722,537
gifsicle	Image Processing	1.90	15,156	95,989
nasm	Programming	2.14	115,862	186,477
patch	Programming	2.7.6	8,309	4,527
rec2csv	Data Conversion	1.8	1,279	134,316

We run UAFDetect to detect vulnerabilities in each of these programs. We use the DFS search strategy and the Z3 constraint solver. Table II lists the nine vulnerabilities that are successfully detected by UAFDetect. The column "Type" describes whether the vulnerability is a UAF (use-after-free) or DF (double-free), a special case of UAF. The column "Input" shows the type of input to each program. The column "Time" presents the time when UAFDetect detects the first occurrence of each vulnerability, measured in seconds.

TABLE II
LIST OF DETECTED UAF VULNERABILITIES.

Vulnerability	Type	Program	Input	Time
CVE-2017-9182	UAF	autotrace	BMP image	66
CVE-2016-3189	UAF	bzip2recover	bzip2 archive	22
CVE-2016-4487	UAF	cxxfilt	C++ symbol	105
gifsicle-issue-122	DF	gifsicle	GIF image	30
CVE-2017-10685	UAF	nasm	assembly program	N/A
CVE-2019-20633	DF	patch	patch file	77
CVE-2019-6455	DF	rec2csv	text records	24

Similar to prior work [14], [17], we limit UAFDetect to run on each program for a time period of 10,000 seconds. UAFDetect successfully detects all but one vulnerability within the time period and generates the inputs to reproduce these vulnerabilities. Overall it takes UAFDetect from 22 seconds to 105 seconds to find a UAF vulnerability.

C. Case Studies

CVE-2017-9182. The UAF vulnerability is in `autotrace`, a program that converts images to vector graphics. `autotrace` takes an image file as the input, and outputs a vector description of the image in various formats.

For a BMP image file, function `ReadImage` in `autotrace` reads the image data from the file. It stores the image data in dynamically allocated memory regions, but for some image data it inadvertently deallocates the memory regions due to pointer aliases. This causes a UAF later when function `find_outline_pixels` tries to access the freed memory regions.

`autotrace` supports multiple image file formats and determines the image file format of an input file by examining the content of the input file. Because the UAF only occurs for a BMP file, UAFDetect uses the prefix input condition to make sure that it generates a BMP file.

CVE-2016-3189. `bzip2recover` recovers data from broken bzip2 archives. It reads a bzip2 archive file to extract compressed data blocks, and then writes each successfully extracted block to a different bzip2 archive file. It uses a dynamically allocated structure to encapsulate miscellaneous information for each file it opens for reading or for writing.

Function `bsOpenWriteStream` is responsible for allocating the structure for each file to be opened for writing. After closing a file, function `bsClose` frees the structure. The UAF vulnerability occurs when another function `bsPutBit` tries to read some miscellaneous information in the freed structure.

One major challenge to trigger the UAF is that `bzip2recover` checks the format of the input bzip2 archive file which is expected to contain three magic numbers, a CRC code for each compressed data block, and a CRC code for the entire file. UAFDetect uses the prefix input condition to ensure the symbolic input file contains the magic numbers.

CVE-2016-4487. This is a UAF vulnerability in `cxxfilt`, a programming tool to demangle C++ symbols. Our example vulnerability is adopted from it. `cxxfilt` takes a mangled C++ symbol from the command line and demangles it to regular C++ symbol.

Function `register_Btype` in `cxxfilt` dynamically allocates an array of pointers in a structure to store the type information embedded in a mangled a C++ symbol. The array is associated with a size field in the same structure. Unfortunately the size field is not set to zero when the array is deallocated by function `squangle_mop_up`. This causes a data inconsistency and can result in a UAF.

A mangled C++ symbol does not have a complex format. As a result, UAFDetect does not need to use input precondition to find the UAF.

gifsicle-issue-122. `gifsicle` is a tool for creating and editing animated GIF image files. It takes multiple GIF files as input and merges them into an animated GIF file. For each GIF file, it reads the image data and store it in dynamically allocated memory regions.

Function `read_gif` allocates memory regions and reads image data into these regions. After finishing processing a GIF file, it deallocates the memory regions. However, function `blank_frameset` tries to deallocate the memory regions again after all the input GIF files have been processed. This causes a DF vulnerability.

Although `gifsicle` checks the format of GIF files, UAFDetect does not need to use any input precondition to find the DF. This is because a GIF file format itself does not use any CRC code.

CVE-2017-10686. `nasm` is a popular assembler for x86 assembly programs. It takes an assembly program file as input and translates the assembly program into an object code file. The assembly syntax used by `nasm` supports macros.

Function `pp_getline` processes tokenized lines of an assembly program. When it encounters a macro, it expands the macro by calling function `expand_mmacro`, which dynamically allocates memory regions to store the new tokens generated by the macro expansion. A UAF can occur when the assembly program includes a malformed macro definition.

Finding the UAF has two major challenges. First, the input to `nasm` is expected to be an assembly program and thus needs to comply to the assembly code syntax. Second, the UAF requires a macro defined in a specific way in the assembly program. None of the input preconditions supported by SPDetect [17] can help address these challenges. Consequently UAFDetect is unable to find the UAF within the time limit.

CVE-2019-20633. This is a DF vulnerability in `patch`, a common utility for applying source code patches. Its input is the code difference generated by the `diff` utility. `patch` applies the difference to source code to produce patched code.

The input to `patch` can consists of multiple changes that need to be applied. `patch` uses function `another_hunk` to process each change in the input. It allocates memory regions for each change and deallocates them after processing the change. A DF can occur for an input containing changes that do not conform to the appropriate format. Interestingly, function `another_hunk` has some code attempting to detect DFs but it still misses this one.

Because `patch` expects the difference to contain one ore more changes described in a specific format, UAFDetect uses input conditions to ensure the input includes specific characters indicating the occurrence and type of of each change.

CVE-2019-6455. This is a DF vulnerability in `rec2csv`, a utility to convert plain text data records into CSV format. It can read data records from a file or standard input, and write them in CSV format to standard output.

`rec2csv` allows the plain text data records to include comment lines. Function `rec_parse_rset` parses the data records and uses dynamically allocated memory regions to store the comments. When a malformed comment exists in the input, function `rec_parse_rset` can attempt to free an already freed memory region storing the comment.

As the input format is simple, `rec2csv` does not perform complicated format check. UAFDetect can find the DF without using any input precondition.

VII. Conclusion

We present our approach that fuses typestate analysis with symbolic execution for finding use-after-free (UAF) vulnerabilities. The approach, called UAFDetect, uses symbolic execution to run target programs and performs typetate analysis in symbolic execution to direct program executions toward triggering UAFs. We develop two techniques to boost its efficiency in finding UAFs: unlikely path pruning and infeasible path pruning. Before executing a program, UAFDetect uses static analysis on the program to locate the information on pointers and operations relevant to UAFs, and the paths that cannot reach these operations. During symbolic execution, the two techniques use this information and the results of typestate analysis to prune paths that are unlikely or infeasible to trigger UAFs. The evaluation on our prototype shows that our approach can find real world UAF vulnerabilities efficiently.

Acknowledgment

This work was supported in part by US National Science Foundation (NSF) grant CNS-2153474.

References

[1] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, p. 385–394, jul 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[2] C. S. Păsăreanu and N. Rungta, “Symbolic pathfinder: symbolic execution of java bytecode,” in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 179–180.

[3] T. Wang, T. Wei, Z. Lin, and W. Zou, “Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution.” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*. The Internet Society, 2009. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2009.html#WangWLZ09

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[5] “2023 CWE Top 25 Most Dangerous Software Weaknesses,” https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[6] “A use-after-free vulnerability in the linux kernel’s netfilter,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-5197.

[7] “Use after free in mediastream in google chrome,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-4572.

[8] “Gpac through 2.2.1 has a use-after-free vulnerability,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-41000.

[9] “Freerdp is subject to a use-after-free issue,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-40187.

[10] R. E. Strom and S. Yemini, “Typestate: A programming language concept for enhancing software reliability,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, 1986.

[11] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, “Typestate-guided fuzzer for discovering use-after-free vulnerabilities,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 999–1010. [Online]. Available: https://doi.org/10.1145/3377811.3380386

[12] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[13] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[14] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley, “Aeg: Automatic exploit generation,” in *Proceedings of the Network and Distributed System Security Symposium*, 2011, pp. 283–300.

[15] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, “Chopped symbolic execution,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 350–360. [Online]. Available: https://doi.org/10.1145/3180155.3180251

[16] F. Brown, D. Stefan, and D. Engler, “Sys: A Static/Symbolic tool for finding good bugs in good (browser) code,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/brown

[17] Z. Huang and M. White, “Semantic-aware vulnerability detection,” in *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2022, pp. 68–75.

[18] D. A. Ramos and D. Engler, “{Under-Constrained} symbolic execution: Correctness checking for real code,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.

[19] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, “Efficient state merging in symbolic execution,” in *Proceedings of the 2012 Acm Sigplan Conference on Programming Language Design and Implementation.* Association for Computing Machinery, 2013, p. 574.

[20] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, “Java ranger: Statically summarizing regions for efficient symbolic execution of java,” in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, 11 2020, pp. 123–134.

[21] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner, “Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach,” in *Proceedings of the ACM Symposium on Applied Computing*. Association for Computing Machinery, 4 2018, pp. 1475–1482.

[22] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, “Hfl: Hybrid fuzzing on the linux kernel,” in *Network and Distributed Systems Security (NDSS) Symposium 2020*. Internet Society, 2 2020.

[23] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, “FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities,” in *27th USENIX Security Symposium (USENIX Security 18).* Baltimore, MD: USENIX Association, Aug. 2018, pp. 781–797. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei

[24] P. Dinges and G. Agha, “Targeted test input generation using symbolic-concrete backward execution,” in *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, Inc, 2014, pp. 31–36.

[25] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*. Springer, 2011, pp. 95–111.

[26] J. Caballero, G. Grieco, M. Marron, and A. Nappa, “Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 133–143.

[27] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” *arXiv preprint arXiv:1801.01681*, 2018.

[28] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet, “Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free,” in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 2016, pp. 1–12.

[29] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, “Binary-level directed fuzzing for Use-After-Free vulnerabilities,” in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020).* San Sebastian: USENIX Association, Oct. 2020, pp. 47–62. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/nguyen

[30] Y. Younan, “Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers.” in *NDSS*, 2015.

[31] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, “Preventing Use-After-Free attacks with fast forward allocation,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2453–2470. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wickman

[32] D. Liu, M. Zhang, and H. Wang, “A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1635–1648.

[33] Z. Huang and X. Yu, “Integer overflow detection with delayed runtime test,” in *Proceedings of the 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, ser. ARES 2021. ACM, 2021, pp. 28:1–28:6. [Online]. Available: https://doi.org/10.1145/3465481.3465771

[34] A. Aumpansub and Z. Huang, “Detecting software vulnerabilities using neural networks,” in *Proceedings of the 13th International Conference on Machine Learning and Computing, Shenzhen China, 26 February, 2021- 1 March, 2021*, ser. ICMLC 2021. ACM, 2021, pp. 166–171. [Online]. Available: https://doi.org/10.1145/3457682.3457707

[35] ——, “Learning-based vulnerability detection in binary code,” in *2022 14th International Conference on Machine Learning and Computing (ICMLC)*, ser. ICMLC 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 266–271. [Online]. Available: https://doi.org/10.1145/3529836.3529926

[36] Z. Huang and G. Tan, “Rapid Vulnerability Mitigation with Security Workarounds,” in *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research*, ser. BAR ’19, February 2019.

[37] Z. Huang, T. Jaeger, and G. Tan, “Fine-grained program partitioning for security,” in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 21–26. [Online]. Available: https://doi.org/10.1145/3447852.3458717

[38] Z. Huang, “Runtime recovery for integer overflows,” in *6th International Conference on System Reliability and Safety (ICSRS).* IEEE, November 2022, pp. 324–330.

[39] “The LLVM Compiler Infrastructure,” http://llvm.org/, 2022.

[40] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[41] “CVE - MITRE,” http://cve.mitre.org, 2023, accessed: October, 2023.

[42] “BugZilla,” http://bugzilla.maptools.org/, 2023, accessed: October, 2023.