Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free *

Josselin Feist
Laurent Mounier
Marie-Laure Potet
Verimag / UGA
Grenoble, France
first.last@imag.fr

Sébastien Bardin
Robin David
CEA LIST,
Software Safety and Security Lab
Université Paris-Saclay, France
first.last@cea.fr

Abstract

This paper presents a fully automated technique to find and trigger Use-After-Free vulnerabilities on binary code. The approach proposed combines a static analyzer and a dynamic symbolic execution engine, and we introduced some heuristics to make this combination effective in practice for UaF. The tool we developed is open-source and it has successfully been applied on real world vulnerabilities. As an example, we detail a `proof-of-concept` exploit triggering a previously unknown vulnerability on `JasPer` leading later on to the CVE-2015-5221.

Keywords: binary code analysis; vulnerability detection; use-after-free; dynamic symbolic execution; automated exploit generation

1 Introduction

With sustained growth of software complexity, finding security vulnerabilities has become an important necessity. Both defenders and attackers are involved in the process of finding these holes in software, either to improve its security or to attack it. Nowadays, even large companies propose bug bounties rewarding people finding vulnerabilities in their systems. Source code is not necessarily available; thus, the need for binary analysis comes naturally. Well known techniques like static analysis [27, 10], dynamic analysis (fuzzing) [46] or dynamic symbolic execution (DSE) [18, 16] show their efficiency to detect these issues, each with their own strengths and limitations. While static analysis allows to analyze all possible program executions to detect complex patterns, it also requires to over-approximate the program behaviour. This leads to numerous false positives, which could be harmful for security purposes (where *feasible* vulnerable paths are required). Note that this problem usually increases when dealing with binary code. Dynamic analysis and fuzzing techniques give a small number of false positives (if not zero), but they can explore only a limited amount of program paths. In particular, vulnerabilities corresponding to complex patterns can be very hard to trigger without a deep knowledge of the target code. DSE is a trade-off between static and dynamic analysis. This technique can trigger complex paths, but bringing in the same way a significant execution overhead, making it difficult to scale.

Challenge and goal. In this paper, we focus on a particular vulnerability called Use-After-Free [28]. Use-After-Free (UaF) appears when a heap element is *used* after having been *freed*. This pattern is difficult to find and requires a thorough understanding of the program. Indeed, this vulnerability is related to the heap memory. This implies handling possible aliases, which is known to be difficult in static analysis [38]. Moreover, paths triggering this vulnerability are hard to find and require to go through events that can be distant in the code, which penalizes both dynamic analysis and DSE [33].

The goal of this paper is to show how the combination of two different techniques, static analysis and dynamic symbolic execution, can be used to detect UaF in a both efficient and precise manner, i.e. *the method detects real vulnerabilities (avoiding false positives), and generates* proof-of-concepts *as a set of inputs allowing to effectively trigger these vulnerabilities*.

Contributions.

- The main contribution of this paper is to combine static analysis with dynamic symbolic execution to detect Use-After-Free on binary code. This combination consists in computing a weighted slice as a result from the static analysis and used it as a guide for the dynamic symbolic execution.
- The second contribution is to detail heuristics and techniques needed to make the dynamic symbolic

** Work partially funded by ANR, under grant ANR-12-INSE-0002

execution effective. Without these heuristics, the DSE is either not able to find the desirable path or its performances degrade strongly.

- The third contribution is a running example of a real vulnerability found by our approach on the `Jasper` application (CVE-2015-5221).
- Finally, this work being entirely based on open-source tools, examples described in this paper are available and reproducible.

By sharing tools and experiments[1], we hope to participate actively in the opening of the vulnerability detection activity and to promote better access to such techniques.

Outline. The paper is organized as follows. First, we give a motivating example. Then we provide an overview of our approach and give some background on both static analysis and DSE used in this work. Section 5 details the core of our approach: the guided DSE. Afterward, we describe our oracle detecting UaF and explain some specificities of our symbolic engine. We show how our approach is effective through a real example in Section 8. Finally, we discuss related work and future improvement.

2 Motivating Example

The motivating example given in Figure 1 and 2 respectively shows the code and the graph representation of the example. Line 11 represents potentially significant part of the program not relevant for the UaF detection. At first, a memory block is allocated and put in `p` and `p_alias` (lines 1 and 2). Then a file is read at line 4, and its content is placed in the buffer `buf`. If the file starts with the string `"BAD\n"` the condition at line 6 is evaluated to `true`. In this branch, the pointer `p` is freed (line 7), however, there is a missing call to `exit` (line 8). This behavior simulates a part of the program reached in case of error but with a missing exit statement. Forgetting a call to exit or a return statement is, unfortunately, a common mistake (e.g.: CVE-2013-4232, CVE-2014-8714, CVE-2014-9296, CVE-2015-7199, etc.). In this case, `p` and `p_alias` become dangling pointers. Then another comparison is made at line 14, if it is evaluated to `true`, `p` points to a newly memory block allocated at line 15, but `p_alias` is still a dangling pointer. In the second case, both `p` and `p_alias` point to a newly allocated block. `p` and `p_alias` are used at line 22 and 23. While `p` is always pointing to a valid address, `p_alias` can be a dangling pointer, leading to a UaF. To summarize, we got three types of path:

- P_1: 6 is `false` $\rightarrow$ no UaF
- P_2: 6 is `true`, and 14 is `true` $\rightarrow$ UaF in 23
- P_3: 6 is `false`, and 14 is `false` $\rightarrow$ no UaF

The interesting point is that paths of type P_2 and P_3 contain the allocation, the free, and the use sites of the UaF, but only P_2 contains the vulnerability.

```
1  p=malloc(sizeof(int));
2  p_alias=p; // p and p_alias points
3             // to the same addr
4  read(f,buf,255); // buf is tainted
5
6  if(strncmp(buf,"BAD\n",4)==0){
7   free(p);
8   // exit() is missing
9  }
10 else{
11  .. // some computation
12 }
13
14 if(strncmp(&buf[4],"is a uaf\n",9)==0){
15   p=malloc(sizeof(int));
16 }
17 else{
18   p=malloc(sizeof(int));
19   p_alias=p;
20 }
21
22 *p=42 ; // not a uaf
23 *p_alias=43 ; // uaf if 6 and 14 = true
```

Figure 1: Motivating Example

Limitations of classical approaches. This example shows the limitation of classic vulnerability detection methods. Static analysis can be used to detect UaF, however, as we discussed previously, reasoning with heap and aliases makes such analysis less precise [38]. Thereby the number of false positives resulting from static analysis makes this technique not straightforward to be used all alone. Classical fuzzing techniques have trouble to find the right input needed, because of calls to `strncmp`. Some fuzzers may even not be able to trigger the desired path. DSE can find the right path; however, it can be lost during the exploration in irrelevant parts of the code, as line 11. Since it also brings a significant overhead, iti can take a long time to trigger the UaF.

3 Overview of the Approach

Instead of seeing static analysis and dynamic symbolic execution as opposite, we propose to combine them directly:

- We use the strength of the static analysis to detect some interesting sections of the code and discard other parts;

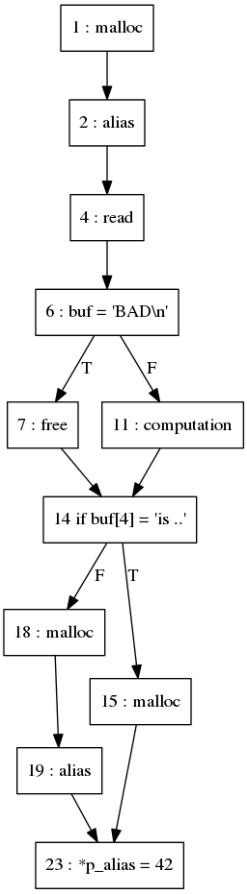


Figure 2: Motivating Example: Graph Representation

- Then, we focus the dynamic symbolic execution only on those interesting parts, reducing the path explosion issue.

Workflow. Figure 3 represents our tool-chain. The static analysis part relies on the recent GUEB [28, 34] tool. DSE is performed on the new platform BINSEC/SE [24]. From a binary, the static analysis detects a UaF and extract a slice containing it. We weight this slice and use it as a guide the DSE exploration. Then once a UaF is validated, a `proof-of-concept` is generated. The main contribution of this work is to show how the combination of static analysis and DSE can efficiently be used on UaF vulnerabilities. UaF found by our analysis are true positives, and we have both local (execution trace) and global (slice representation) information allowing a clear and deep understanding of it.

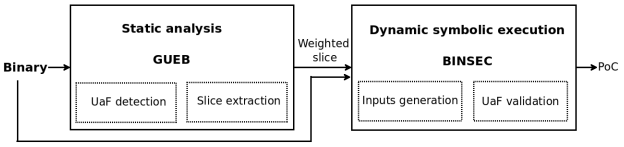


Figure 3: Combining static and DSE for UaF detection

4 Background

4.1 Static Analysis

GUEB performs a dedicated value-analysis [4] on x86 binary code to compute, at each program location, an abstraction of the current heap state as two address sets: the set of each **allocated** and **freed** memory chunks.

From this information, GUEB identifies static paths in the program control-flow graph (CFG) able to successively allocate, free, and use the *same* memory chunk, corresponding to a (potential) UaF. Then, for each of these UaF paths detected, the tool produces a program slice as a subgraph of the CFG, starting from the entry point of a root function and containing all the paths exercising this UaF. The slice obtained for the motivating example is depicted in Figure 4.

GUEB is built upon the BinNavi framework [51]. BinNavi offers an assembly level intermediate representation (REIL) [26], and a basic API to prototype data-flow static analysis. Translation from binary code to x86 assembly code can be provided by IDAPro [36] as a front-end. GUEB was at first applied on the CVE-2013-4232 (tiff2pdf) to validate the approach. Then it was used to find six previously unknown UaF vulnerabilities (in `JasPer`: **CVE-2015-5221**, in `openjpeg`: **CVE-2015-8871**, in `giflib`: **CVE-2016-3177**, inside a debugging tool of `bind`, in `accel-ppp` and in `gnome-nettool`).

However, as a binary-level static analysis tool, GUEB suffers from some limitations making it both unsound and incomplete. First, due to dynamic jumps in the code, the CFG taken as input is not the exact one. Moreover, GUEB is parameterized by a set of program entry points (e.g., the functions with no callers) and a set of default allocation/free functions (such as `malloc()` and `free()`). Inter-procedural analysis is achieved via function inlining, and allocation-site abstraction is used to identify points-to relations. Finally, loops are handled through (bounded) loop unrolling instead of fix-point computations. The paths leading to UaF in the extracted slice are not always feasible, and they may not even correspond to a real UaF. As a result, all these vulnerability had to be confirmed through a manual analysis of the program code. In many cases, we were not able to provide concrete inputs allowing to trigger these vulnerabilities, which is an important security requirement. These limi-

tations are the main motivation of this work, namely using a DSE to automatically confirm GUEB's results and produce a corresponding proof-of-concept.

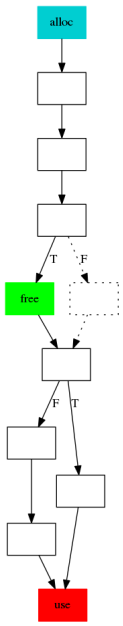


Figure 4: CFG slice produced by GUEB

4.2 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) [16, 32, 47, 44, 30, 15, 14] is a formal technique for exploring program paths in a systematic way. For each path p, it computes a symbolic *path predicate* Φ_p as a set of constraints on the program input leading to follow that path at runtime (path predicates are conjunctions of all the branching conditions c_i encountered along that path). Path exploration is achieved by iterating on (user-bounded) program paths as follows (a complete algorithm is given in Section 5):

1. the program is executed from an initial (concrete) program input i_0 to produce a first path p_0; its path predicate Φ_{p_0} is added to an (initially empty) working list WL;
2. a path predicate $\Phi_p = c_1 \wedge c_2 \wedge \ldots \wedge c_n$ is extracted from WL and, in order to try to explore an alternative branch, one branch condition c_i is negated to build a new path predicate Φ'_p;
3. Φ'_p is then fed to an to an off-the-shelf SMT solver: a solution to this predicate is a new test input i allowing to explore the targeted path p'; Φ'_p is added to WL and the algorithm resumes at step 2.

The exploration terminates either when WL is empty (the set of paths being made finite through limiting the size of the considered paths), or when a path satisfying a given condition is reached (e.g., a vulnerable path has been found). The main advantages of the approach are:

correctness: there is no false positive: a bug reported is a bug found;

flexibility: thanks to concretization [30, 44, 23] it is possible to decide which data should be considered as **symbolic** *vs.* **concrete** when building a path predicate; as a result the approach can handle unsupported features of the program under analysis without losing correctness;

easier adaptation to binary code analysis, compared to other formal methods; especially, many binary-level DSE tools have been developed [8, 31, 21, 19, 2]

The very main drawback of DSE is the so-called *path explosion problem*, leading DSE to crawl a giant set of paths blindly in the hope of finding a buggy path.

The BINSEC **platform.** BINSEC [25] is a recent platform for the formal analysis of binary codes. The platform currently proposes a front-end from `x86` (32bits) to a generic intermediate representation called DBA [9] (including decoding, disassembling, simplifications), and several semantic analysis, including the BINSEC/SE DSE engine [24]. BINSEC/SE features a strongly optimized path predicate generation as well as highly configurable search heuristics [24, 5] and C/S policies [23], and a stub mechanism for specifying the behavior of missing parts of the code (cf. Section 7).

5 Guided DSE

Designing a good search heuristic is a major concern in DSE. Search heuristics do not matter for path coverage, however, they can make a huge difference for instruction (or branch) coverage. Many heuristics have been proposed in the literature, starting for example from [15, 41, 31]. Unfortunately, it appears that relying on a single heuristics is often not sufficient and that different programs or different goals require different heuristics. Typically, such engines rely on a selection mechanims returning the "best" path candidate w.r.t. an user-defined `score` function. Scores are built from several (predefined or dynamically computed) path characteristics, such as length, instruction call-depth, distance to a target, etc.

Whereas most DSE strategies focus on exploring as many paths as possible in a minimum of time, our approach tries to trigger one particular path as soon as possible. This path needs to respect a specific property: containing a UaF. To find such path, we guide our exploration with a *slice* of the program extracted from the results of static analysis, i.e. a restriction of the program to

a set of paths possibly leading to the UaF vulnerability. Doing so, we explore only a small portion of the program and do not suffer from the `path explosion` problem as much as classical DSE strategies. Actually, we go a step further and use a *weighted slice*, i.e. a slice enhanced with score information used for guiding the DSE during the slice exploration.

5.1 Weighted Slice

Thanks to the static analysis (Section 4.1) we have the set of nodes representing the slice leading to a UaF. Three of them have a particular role: the allocation (n_{alloc}), the free (n_{free}) and the use (n_{use}) node. So, we need to find an input leading a path to go through these three nodes. As we discussed, a key feature of DSE exploration is to prioritize which branches will be inverted at first. We guide this selection using the slice, more precisely we compute for each node a score to the destinations.

Distance score. We call the score to the destinations *DS* (*distance score*). In our experimentation, *DS* is computed using shortest paths. It is computed as preprocessing of the exploration. As UaF requires three targeted nodes, we compute three scores for each node. To illustrate the need for three different scores, let us consider the example on Figure 5. Two executions gave two paths: $P0$

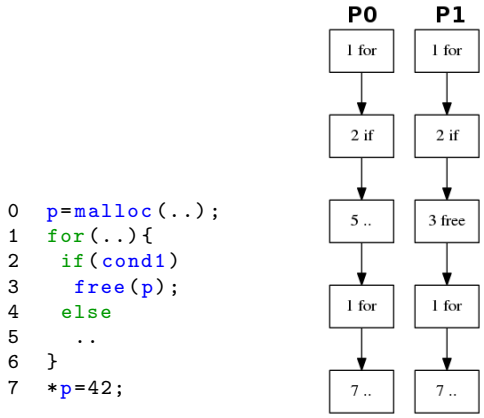


Figure 5: Score selection example

and $P1$, both unrolling the loop one time. $P0$ evaluated the condition at line 2 to `false` while $P1$ evaluated it as `true`, and so $P1$ contains the call to `free`. We focus in this example on the score used to know if it is interesting to unroll a second time the loop (the second node `"1 for"` in both cases). In $P0$, n_{free} was not reached; thus, we should try to unroll a second time the loop. In $P1$, n_{free} was reached; we do not need to unroll a second time the loop. Thereby, the node `"1 for"` do not take the same score depending on if n_{free} was reached previously in the path or not. During the exploration, the appropriate score is so selected according to previous events in the path (e.g.: if the path contains n_{free} then the score to n_{use} is used). Notice that since the static analysis performs inlining of functions, we associate nodes with their call-stack to distinguish the same instruction in different calling context. *The slice is thus weighted by DS to guide the exploration.*

Paths exploration. Algorithm 1 describes a high-level view of how our algorithm is working. P represents a deterministic program[2], where $P(i) = p$. Here `compute_predicate` gives a path predicate, from a path p and a conditional node c_i. `solve` computes this path predicate and gives new input i if the *verdict* of the path predicate is `SAT`. If a new path does not contain a UaF, checked by the oracle σ, all conditions on this paths that are inside the slice are extracted. Then the appropriate score is picked, using *DS* and according to the position of the condition in the path. Finally, the path and the conditional node are added to the working list WL. The algorithm stops either when a UaF is found or when there is no more path to explore.

Algorithm 1: Guided DSE algorithm

```
Input: Program P, DS, slice S, oracle σ, seed i_0
Output: Input i, with P(i) respecting oracle σ
WL = init(i_0);
while WL ≠ ∅ do
    select (t, c_i) ∈ WL;
    WL := WL \ {(t, c_i)};
    Φ_p = compute_predicate(t, c_i);
    verdict, i = solve(Φ_p);
    if verdict = SAT then
        p = P(i);
        if p respects σ then
            return i;
        end
        extract c_0..c_n conditions from p, c_0..c_n ∈ S;
        score (p, c_0)..(p, c_n) with DS;
        WL = WL ∪ (t, c_0)..(t, c_n);
    end
end
return Not Found;
```

Property 1 (Correctness) *If $P(i)$ respects σ, input i is a proof-of-concept triggering the UaF.*

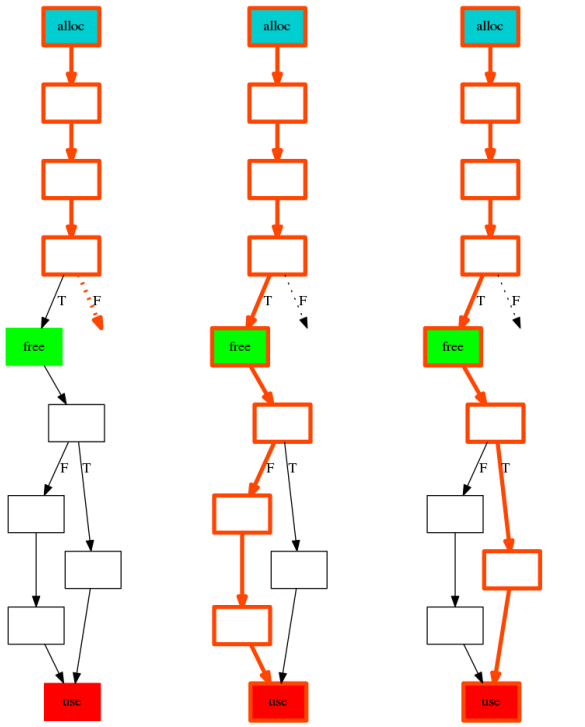
Property 2 (Completeness) *If DSE explored all paths of S without generating an input triggering a UaF described by n_{alloc}, n_{free}, and n_{use}, there is no such UaF.*

Exploring all paths of S is not always possible (there could be an infinite number of paths), we can therefore

bound the exploration with a timeout or a fixed number of paths, losing completeness.

Example. In our example, if we try at first to explore the program with a file containing only 'A'[3] as input, the first condition is evaluated as `true`, and we go out of the slice (see Figure 6a). Since the path goes out of the subgraph at the first condition, this one is selected. All the other possible conditions on this path are outside the slice and thus are not explored. DSE is able to invert the condition and creates a new input starting with `"BAD\n"` (Figure 6b). However, condition 14 is still evaluated to `false` with this new input and so, in this case, there is no UaF (since the path belongs to P_3 in Section 2). DSE creates then a third input, by inverting 14: `'BAD\nis a uaf\n'` (Figure 6c). As there is a UaF in the path generated with this input, our oracle (see Section 6) detects it.

The exploration stops, and we now have a `proof-of-concept` triggering the UaF.



(a) First path, input filled with 'A'

(b) Second path, input: 'BAD\nAAAA..'

(c) Third path, input: 'BAD\nis a uaf\n'

Figure 6: DSE: Paths generation

5.2 More Guiding Heuristics

Using *DS* to guide allows DSE to find the UaF, though the exploration can still be stuck in some part of the program. To be applied in real world programs, a set of heuristics is needed. Thanks to the modularity of BINSEC/SE (see Section 8.1), adding heuristics to the guidance is straightforward. In the current implementation, these heuristics are based on information coming from the knowledge of common libraries, such as *libc*. There are not meant to guide the exploration directly, but we use them when the exploration deviates from its goal. We describe in the following two heuristics needed during the exploration of two real world test cases.

5.2.1 String Length Based Heuristic

The first heuristic helps when there is a comparison on a string, those length depends on the number of iteration of a previous loop. Let us consider this example:

```
1  read(f,tmp,255);
2  for(i=0;i<255;i++){
3    buf[i] = tmp[i];
4    if(tmp[i]=='\0') break;
5  )
6  buf[i]='\0';
7  if(strcmp(buf,"this is really bad") ==
       0)
8    ..
```

`strcmp` checks if the two strings passed as parameters are exactly similar. Every time the loop in taken, a constraint is added to the `path predicate` forcing `tmp[i]` (and so `buf[i]`) to be different from `'\0'`. If the seed input is an array of 255 `'A'`, the first path will unroll this loop 255 times. To evaluate the comparison line 7, we need to unroll the loop at line 2 exactly 19 times (the size of the string more the character `'\0'` which ends the string). This is a classical example when DSE can take time to explore a path that seems easy to trigger. Indeed, in this case, DSE needs to explore all the 18 first iterations of the loop, before being able to invert the line 7.

Our solution. We propose to use the size s_i of constant strings passed to `strcmp` (and equivalent functions, as `strncmp`,..) to find this specific iteration. We use this only to explore conditions located after such calls that we were not able to invert. In the previous example, the condition line 7: .. == 0 follows a call to `strcmp`. We use so the size of the string `"this is really bad"`: 19, to prioritize the invertion of conditions located at the 19^{nth} iteration of the loop, saving us the exploration of conditions located in the 18 previous iterations of this loop. In the current implementation, all loops containing conditions in the iteration s_i are inverted. We planned to combine this heuristic with a backward data-dependency analysis, to locate which loops have an impact on the parameter of the call to `strcmp`. Nevertheless, even with this `naive` implementation, this heuristic showed a real impact on DSE performance during the exploration of `JasPer` (see Section 8).

5.2.2 Allocator Functions Behavior Based Heuristic

A second heuristic result in the fact that some paths need specific results of allocator function to be reached. This is illustrated in the following code:

```
1  p=malloc(size);
2  if(p==NULL)
3  {
4          // path to trigger
5  }
```

Although simple, standard DSE tools are not able to trigger the path, since to be reached, the size given to `malloc` need to exceed the available size of the heap.

Our solution. When meeting this pattern, our engine will try to create an input with a very large value for `size`, in the hope of making `malloc` returning a null pointer. This heuristic targets all functions with a similar behavior (e.g.: `realloc`, `calloc`, ..). CVE-2013-4232 (tiff2pdf) is an example where this heuristic is needed to trigger the UaF.

6 Use-After-Free Detection

Runtime UaF detection is not a trivial task. In particular, executing in sequence the three operations *alloc*, *free* and *use* on the same address is not a sufficient condition to get a UaF. As seen in Section 2, paths in P_3 fulfill this condition, but they correspond to false positives. We need *a way to validate if a path contains the UaF*.

Issue. The main problem when defining a correct and complete runtime checker for UaF detection on a given execution path is *aliasing due to re-allocations*, as shown in our motivating example. Allocation calls at lines 15 and 1 and may return the *same* address when condition 6 is `true`. Therefore, pointers `p` and `p_alias` would be implicitly alias, but without referring to the *same* memory block (returned by the same allocation call). In particular, only `p_alias` is a dangling pointer and leads to a UaF if it is accessed. Tracking the points-to addresses relations only is therefore not sufficient, and the current allocation site of each memory block has to be taken into account.

Our solution. A UaF occurs on a path p if and only if the following property Φ holds:

(i) $p = (\ldots, n_{alloc}(size_{alloc}), \ldots, n_{free}(a_f), \ldots, n_{use}(a_u))$

(ii) a_f is a reaching definition of the block returned by n_{alloc}

(iii) a_u is a reaching definition of an address in the block returned by n_{alloc}

Property Φ could be verified using dataflow analysis. However, the execution traces we consider during the DSE are "incomplete" in the sense that the behavior of some library functions is summarized by *stubs* to build the path predicates (see Section 7). Data-dependency computations should then take into account the side-effects of these functions. On the other hand, `path predicates` implicitly keep the data-dependencies of symbolic values. Therefore, we propose to use another approach, directly based on the symbolic reasoning performed by our DSE engine. The idea is to build a `path predicate` φ_t for the path p with only one *unconstrained* symbolic variable S_{alloc} corresponding to the value returned during the target allocation site n_{alloc}.

Then, Φ holds on a path p if and only if the SMT formula $\varphi_t \wedge \Phi'$ is not `SAT`, where

$$\Phi' = \exists S_{alloc}.\ (a_f \neq S_{alloc}) \vee (a_u \notin [S_{alloc}, S_{alloc} + size_{alloc}])$$

Examples. In the motivating example (Figure 1), we have two distinct paths containing n_{alloc}, n_{free} and n_{use}. For paths in P_2[4]:

$$\begin{aligned} \varphi_t &= (p_0 = S_{alloc} \wedge p\_alias_0 = p_0 \wedge p_1 = 0x8040000) \\ \Phi' &= (p_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_alias_0 \leq S_{alloc} + 4) \end{aligned}$$

$\varphi_t \wedge \Phi'$ is `UNSAT`, which confirms the presence of a UaF at line 23.

For paths in P_3,

$$\begin{aligned} \varphi_t &= (p_0 = S_{alloc} \wedge p\_alias_0 = p_0 \wedge p_1 = 0x8040000 \\ &\quad \wedge p\_alias_1 = p_1) \\ \Phi' &= (p_0 \neq S_{alloc}) \vee \neg(S_{alloc} \leq p\_alias_1 \leq S_{alloc} + 4) \end{aligned}$$

In this case $\varphi_t \wedge \Phi'$ is `SAT` (e.g., with $S_{alloc} = 0x0$), which confirms that there is no UaF in this path.

7 Stub

The stub mechanism implemented in BINSEC/SE allows to over-approximate or simulate logical effects of an untraced library call. This allows us to preserve symbolic execution soundness without having to execute the library code symbolically. Stubs are required to ensure a relative compactness of the trace. Furthermore, library calls generally contain system calls that would require some over-approximation, thus doing the stub at the library call level is a fair balance.

The implemented mechanism allows us to parameterize actions to be performed on the parameters and the result of a given function. Possible actions are

- `CONC`: concretize the given value with its runtime value
- `LOGIC`: apply a given logical operation on the value, if one is implemented in the stub

- SYMB: symbolize the value creating a new input

realloc is an example of library code where the stub is not straightforward, although necessary. Tracing this call add complex constraints on the *path predicate* (especially on the heap state). Skipping this function without taking into account its logical effect induces error. Indeed, if the pointer returned by realloc is different from the pointer gave as the parameter, the values present in the original buffer are copied into the fresh buffer. By not keeping this side-effect, results of the analysis are incorrect. The stub mechanism allows to handle this behavior by skipping the code of this function while keeping its logical effect.

In our experimentation, we use logical stubs on significant functions (15 in the current implementation) while we only concretize the return value of some functions (such as open, printf,...).

8 Experimental Evaluation

In this section, we describe our implementation, and we detail the successful automated proof-of-concept creation generated by the guided DSE. To illustrate its efficiency, we compare our approach with and without the guiding heuristics, and we use two well-known fuzzer (AFL and radamsa) as comparison with fuzzing.

8.1 Implementation

Algorithm 1 is primarily implemented in BINSEC/SE using OCaml functors. We thus allow a modular usage of the guided DSE. For example, the criteria σ is a functor that can be easily changed. The current implementation includes the UaF detection and also a buffer overflow detection. *DS* and the select function are also part of a functor. Thereby we can choose from traditional *DFS*, *BFS*, shortest paths or using shortest paths enhanced by the guiding heuristics (Section 5.2). It is then straightforward to combine different criteria with different guiding strategies.

8.2 JasPer

We demonstrate the efficiency of our approach through the study of a UaF in JasPer[5]. JasPer is used to convert an image from one format to another. The vulnerability was found by GUEB, and it is located in the function mif_process_cmpt (**CVE-2015-5221**). Unfortunately, GUEB does not give any input exhibiting the reality of the vulnerability. The entry point of the slice is the function mif_hdr_get. The first step of the analysis is therefore to find a way to trigger this function starting from the main function. Fortunately, this is directly done by forcing JasPer to take as input an image in the format MIF (which is a format specific to JasPer). The following command triggers mif_hdr_get:

```
jasper --input input --input-format mif
    --output output --output-format jpg
```

The file input is then converted from the format MIF to the format JPEG into the file *output*.

PoC. Our DSE engine takes as input the weighted slice computed from GUEB, the command line and a first file *input* as seed. In our experimentation, we give a file filled with 'A'. It successfully generates a test-case triggering the UaF (a double-free in this case), given in Figure 7.

```
MIF
component
```

Figure 7: PoC of CVE-2015-5221 generated by DSE

At the time of writing this article, no official patch is available for this vulnerability, the PoC generated is still working on JasPer. It was tested with success on the last version 1.900.1, on the version available on Ubuntu 16.04 (package libjasper-runtime, version 1.900.1-debian1-2.4ubuntu1) and Debian 8.04 (version 1.900.1-debian1-2+deb8u1). Since the vulnerable code is inside the library of JasPer itself (libjasper1), it affects all software using the library and allowing the manipulation of MIF files (such as utilities provided within the library: imginfo,..). Fortunately, to the knowledge of the authors, most of the tools based on this library do not allow this format.

Exploration. We can separate the PoC in two parts: the first line "MIF\n" and the second line "component". The first line is the result of a comparison byte per byte of the first four characters of the file, as showed in the following simplification of the code of JasPer:

```
if (m[0] != "M" || m[1] != "I" || m[2] !=
    "F" || m[3] != "\n")
```

This is naturally easy for a DSE to solve this condition and the creation of this first line took only a few inversions. However, the second line is harder to create. The following code represents a simplification of the necessary conditions to create this line:

```
bufptr = buf;
while(i > 4096){
        if ((c=get_char()) == EOF) break ;
        *bufptr++=c;
        i--;
        if(c=='\n') break;
}
if (!(bufptr = strchr(buf,'\n'))) exit(0);
*bufptr = '\0'
..
p = malloc();
```

```
strcpy(p,buf);
..
if(!(strcmp(p,"component"))) exit(0);
```

The loop copies the input file to a buffer until it reaches the size of the buffer (4096), the end of the file or the character `'\n'`. Then the function `strchr` and the next assignement replace the character `'\n'` in this buffer by `'\0'`. Finally, the buffer is copied to a new buffer and compared to the string `"component"`. Similar to the example at Section 5.2.1, every time the loop is iterated, the constraint `buf[i] != '\n'` is added. Thereby to successfully find an input validating the comparison made in `strcmp`, the comparison `c=='\n'` needs to be evaluated as true at the tenth iteration and false during the nine preceding iterations. In the code of `JasPer`, the loop and this comparison are distant in the code, so triggering this specific path is not straightforward.

Efficiency. Figure 8 gives details of the experimentation. The creation of the PoC takes 30 minutes on a standard laptop (`i7-2670QM`). We use Boolector [12] as SMT solver. 19 test cases are correctly generated while 153 path predicates are `UNSAT`. The C/S policy [23] used during our exploration kept as logical load operations and concretized those on store. We limit the exploration of loops up to the hundredth iteration. Notice that by removing the guided heuristic (GH) (see Section 5.2), the DSE was not able to find the UaF after running for 5 hours. This shows the importance of these heuristics

Name	Time	MIF line	UaF found	# Paths
SP + GH	40 *m*	3*min*	**Yes** : 30*min*	9
SP	*todo h*	3*min*	*No*	*todo*
DFS	6 *h*	3*min*	*No*	354
AFL	7 *h*	$< 1m$	*No*	172[6]
Radamsa	*Todo*			

SP: Shortest Path, GH: Guiding Heuristic, DFS: classical Depth-First Search

Figure 8: `JasPer` evalution

Comparison with fuzzing. As a comparison with the state of the art technique, we used `AFL` [1] (American Fuzzy Lop) and `radamsa` [43] to try to reproduce the crash. Starting from the same state: an input file filled with `'A'` and the same command line, we run `AFL` and `radamsa` on `JasPer` for 7 hours. We used both the simple and the `quick & dirty` mode on `AFL`, with same results. *No crashes have been found*. There fuzzers were built in a perspective of coverage, not to find a particular path, thereby finding this specific bug is harder, as expected. We should mention that `AFL` successfully manages to create inputs starting with `MIF` in less than one minute. However, the second line being more complicated, the fuzzer was not able to reproduce it. Notice that by using as seed a correct `MIF` file, provided within the `JasPer` library, both fuzzers were able to generate a PoC of the UaF in less than one minute. Since all proper `MIF` files contain the line `MIF` as header and at least one line starting with `component`, few mutations on these files generate a PoC of the vulnerability. However, vulnerabilities are not necessary trigger in such easy ways and test-case provided are not good seeds in these cases. Our approach is therefore well adapted where classical fuzzing techniques are not.

Discussion. GUEB gives hundreds of possible UaF on `JasPer`. We used as validation only the slice that we knew to be a true positive. We plan to improve GUEB to reduce the number of results, using more accurate analyses. For example, GUEB analyzes the same binary from several entry points; we analyze same part of the code multiple times and several results refer in fact to the same issue. We are working to improve this limitation on GUEB. Nevertheless, the DSE can be launched in parallel in different slices. Exploring hundreds of slices for few hours is still realistic.

Our method, combining static analysis with dynamic symbolic execution is, therefore, robust enough to find UaF vulnerabilities that are not found by standard fuzzing methods in a short amount of time.

9 Related Work

Vulnerability detection is now an active research area. The techniques proposed are essentially based on static and/or dynamic analysis.

UaF detection.

In the specific case of UaF detection, static techniques exist [17, 33] but they are not available; no fair comparison can so be made. Moreover, they are not always able to provide precise enough results on large examples [39]. This is particularly the case when they are applied from the binary code. Therefore, most existing detection techniques are based on dynamic runtime analysis.

First, general "memory error detectors" like Valgrind [42] or AddressSanitizer [45] can be used to detect UaF at runtime. These tools face the problem of false positives due to re-allocations of heap memory blocks (see Section 6). To address it, AddressSanitizer replaces the system memory allocation function by a custom one, using a quarantine zone for the liberated blocks. However, memory reallocations can still occur when the heap is full, leading to undetected UaF. An example is given in Appendix 1 and in [39]. Note also that this solution is not well suited for custom allocators.

A second category contains more specific tools like Undangle [13] or DANGNULL [39]. Their principle is to track at runtime the memory allocation and free operations, in order to maintain some metadata allowing to

detect dangling pointer dereferences.

Although these tools happen to be rather effective in finding (and even preventing) UaF errors in large applications, they still require either to permanently run instrumented code (with the associated overhead) or to correctly "guess" the relevant inputs during a fuzzing campaign. The approach we propose in this paper fulfills a different need: potentially dangerous execution paths are identified statically, and they are subsequently confirmed/invalidated using a dynamic symbolic execution (DSE). The expected benefits are twofold: first, the target application is analyzed "once for all", trying to find as much as possible UaF vulnerabilities, and second, concrete inputs are provided to exercise each vulnerability found. Moreover, since we focus on *a few numbers* of suspicious UaF, we can afford a more expensive checker (based on an SMT solver).

Guided DSE. Using score functions to guide the path explorarion during a DSE has been introduced in several tools, either in dedicated [31, 15, 14] or generic ways [48, 5], either to improve the instruction coverage or to reach a specific statement, or, as explained above, to confirm some potential vulnerabilities. More recently, specific guiding techniques have been proposed [22, 50] to cover sequential execution patterns (e.g., safety properties). In these latter works, scores are associated with branch conditions, essentially based both on control-flow and data-flow properties. We currently rely only on control-flow information. Using data-flow as well, or other control-flow metrics (such as random walks [29]) would be an interesting work direction.

Combination of static analysis and DSE. Despite its many successes [16, 32], DSE suffers from the path explosion issue. Hence, a few teams have tempted to mitigate this problem through combining DSE with a first static analysis geared at reducing the search space to a subset of interesting paths. For example, Kosmatov *et al.* [20] use first a static analysis to check common classes of runtime errors in a C program, then, they try to trigger all remaining potential errors through DSE restricted to a slice of the original program. In [49], authors combine static analysis with DSE, using *proximity heuristic* computed statically to guide the exploration to generate, from a bug, similar traces leading towards it. In [1], data-flow analysis is combined with shortest path in the Visible Pushdown Automaton (VPA) representation of the program A similar approach is in [3], where a data-flow analysis is combined with shortest to find vulnerabilities identified by static analysis.

In coverage-oriented testing, Bardin *et al.* [7, 6] designs a similar approach for proving the infeasibility of some white-box testing objectives, before launching DSE in order to cover them all. In vulnerability detection, we can mention for instance the work performed with Dowser [35], to find buffer-overflows by guiding the DSE tool S2E [21] in order to focus on execution paths containing "complex" array access patterns identified by a (lightweight) source-level static analysis. A similar approach is proposed in [40], to confirm memory leaks found in C++ programs by the static analyzer HP Fortify [37].

If our work follows a similar approach, it differs in several respects: it fully operates on binary code (both on the static and on the dynamic side), the DSE is here guided by a weighted slice containing a set of (potentially) vulnerable paths. Moreover, these paths need to contain several targets in a specific order. We are also looking for safety-like properties rather than invariant-like properties. Finally, our combination is not purely black-box, in the sense that search heuristics is more deeply coupled with the static analysis.

10 Conclusion

In this paper we detailed a novel approach combining static analysis with dynamic symbolic execution to detect UaF. Our approach relies on the use of a weighted slice and we showed that it is efficient enought to find complex vulnerabilities on real world applications. Our platform is not mature enough, and a lot of improvement are ongoing.

Future work. Guiding heuristics proposed in Section 5.2 are just a first step to a larger set of heuristics. Our platform is still in the stage `driven by examples`, where we define new heuristics when needed. By doing so, we hope to be able to build a platform robust and diversified enough to explore classical programming patterns. Naturally we need to apply our methodology to a larger dataset, and we intend to continue to use our platform to detect new vulnerabilities. We shared our platform as an open-source project to go to this direction. More complex software (like multithreading applications, browser,..) are for now out-of-scope of our analysis, yet its use on parts of them (such are libraries) is naturally a path to explore.

Other applications. The approach described in this paper focuses on UaF detection. However, the combination of static analysis and DSE can be applied to other kinds of vulnerability. It only requires to give as input a targeted instruction and a score on the graph. We also planned to apply our approach to other types of subgraph. For example, from a patched binary and its original version, binary comparison tools (such as BinDiff) allow us to know the difference between both version. It could be interesting to extract a subgraph from this difference, and explore it with our approach. This would lead to the automated creation of `1day vulnerability` [11].

References

[1] AFL. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[2] AVGERINOS, T., REBERT, A., CHA, S. K., AND BRUMLEY, D. Enhancing symbolic execution with VeriTesting. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ICSE '14, ACM Press.

[3] BABIC, D., MARTIGNONI, L., MCCAMANT, S., AND SONG, D. Statically-directed dynamic automated test generation. In *ISSTA* (2011), M. B. Dwyer and F. Tip, Eds., ACM, pp. 12–22.

[4] BALAKRISHNAN, G., AND REPS, T. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst. 32*, 6 (Aug. 2010), 23:1–23:84.

[5] BARDIN, S., BAUFRETON, P., CORNUET, N., HERRMANN, P., AND LABBÉ, S. Binary-level testing of embedded programs. In *13th International Conference on Quality Software, QRS'13* (2013), pp. 11–20.

[6] BARDIN, S., CHEBARO, O., DELAHAYE, M., AND KOSMATOV, N. An all-in-one toolkit for automated white-box testing. In *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings* (2014), Springer, pp. 53–60.

[7] BARDIN, S., DELAHAYE, M., DAVID, R., KOSMATOV, N., PAPADAKIS, M., TRAON, Y. L., AND MARION, J. Sound and quasi-complete detection of infeasible test requirements. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015* (2015), IEEE.

[8] BARDIN, S., AND HERRMANN, P. Osmose: Automatic structural testing of executables. *Software Testing, Verification Reliability 21*, 1 (Mar. 2011), 29–54.

[9] BARDIN, S., HERRMANN, P., LEROUX, J., LY, O., TABARY, R., AND VINCENT, A. The Bincoa Framework for Binary Code Analysis. In *Computer Aided Verification - 23rd International Conference, CAV 2011, 2011. Proceedings* (2011).

[10] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM 53*, 2 (2010), 66–75.

[11] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications.

[12] BRUMMAYER, R., AND BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS* (2009), S. Kowalewski and A. Philippou, Eds., vol. 5505 of *Lecture Notes in Computer Science*, Springer, pp. 174–177.

[13] CABALLERO, J., GRIECO, G., MARRON, M., AND NAPPA, A. 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* (New York, NY, USA, 2012), ISSTA 2012, ACM, pp. 133–143.

[14] CADAR, C., DUNBAR, D., AND ENGLER, D. 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* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[15] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2006), CCS '06, ACM, pp. 322–335.

[16] CADAR, C., AND SEN, K. Symbolic execution for software testing: Three decades later. *Commun. ACM 56*, 2 (Feb. 2013), 82–90.

[17] CESARE, S. Bugalyze.com - detecting bugs using decompilation and data flow analysis. In *BlackHatUSA* (2013).

[18] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy* (2012), IEEE Computer Society, pp. 380–394.

[19] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), SP '12, IEEE Computer Society.

[20] CHEBARO, O., CUOQ, P., KOSMATOV, N., MARRE, B., PACALET, A., WILLIAMS, N., AND YAKOBOWSKI, B. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng. 21*, 1 (2014), 107–143.

[21] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 2:1–2:49.

[22] CUI, H., HU, G., WU, J., AND YANG, J. Verifying systems rules using rule-directed symbolic execution. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS* (2013), pp. 329–342.

[23] DAVID, R., BARDIN, S., FEIST, J., MARION, J.-Y., MOUNIER, L., POTET, M.-L., AND TA, T. D. Specification of concretization and symbolization policies in symbolic execution. In *Proceedings of ISSTA* (July 2016).

[24] DAVID, R., BARDIN, S., FEIST, J., MARION, J.-Y., POTET, M.-L., AND TA, T. D. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *Proceedings of SANER 2016* (Osaka, Japan, march 2016), p. (to appear).

[25] DJOUDI, A., AND BARDIN, S. Binsec: Binary code analysis with low-level regions. In *TACAS 2015* (2015), Springer.

[26] DULLIEN, T., AND PORST, S. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest* (2009).

[27] EMANUELSSON, P., AND NILSSON, U. A comparative study of industrial static analysis tools. *Electr. Notes Theor. Comput. Sci. 217* (2008), 5–21.

[28] FEIST, J., MOUNIER, L., AND POTET, M. Statically detecting use after free on binary code. *J. Computer Virology and Hacking Techniques 10*, 3 (2014), 211–217.

[29] FEIST, J., MOUNIER, L., AND POTET, M.-L. Guided dynamic symbolic execution using subgraph control-flow information. In *Proceedings of SEFM* (Vienna, Austria, July 2016 (to appear)).

[30] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: Directed automated random testing. *SIGPLAN Not. 40*, 6 (2005).

[31] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008* (2008), The Internet Society.

[32] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. SAGE: whitebox fuzzing for security testing. *Commun. ACM 55*, 3 (2012), 40–44.

[33] GOODMAN, P. Pointsto: Static use-after-free detector for c/c++. https://blog.trailofbits.com/2016/03/09/the-problem-with-dynamic-program-analysis/.

[34] GUEB. Static analyzer detecting use-after-free on binary. https://github.com/montyly/gueb.

[35] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 49–64.

[36] HEX-RAYS. Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/index.shtml.

[37] HP. Fortify static code analyzer. http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[38] LANDI, W. Undecidability of static analysis. *LOPLAS 1*, 4 (1992), 323–337.

[39] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015).

[40] LI, M., CHEN, Y., WANG, L., AND XU, G. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA 2013, ACM, pp. 112–122.

[41] MAJUMDAR, R., AND SEN, K. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007* (2007), IEEE Computer Society, pp. 416–426.

[42] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

[43] RADAMSA. A general purpose fuzzer. https://github.com/aoh/radamsa.

[44] SEN, K., MARINOV, D., AND AGHA, G. Cute: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes 30*, 5 (2005).

[45] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 28–28.

[46] SUTTON, M., GREENE, A., AND AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[47] WILLIAMS, N., MARRE, B., AND MOUY, P. On-the-fly generation of k-path tests for C functions. In *Automated Software Engineering, 2004.* (2004).

[48] XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009* (2009), IEEE Computer Society, pp. 359–368.

[49] ZAMFIR, C., AND CANDEA, G. Execution synthesis: a technique for automated software debugging. In *EuroSys* (2010), C. Morin and G. Muller, Eds., ACM, pp. 321–334.

[50] ZHANG, Y., CLIEN, Z., WANG, J., DONG, W., AND LIU, Z. Regular property guided dynamic symbolic execution. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 643–653.

[51] ZYNAMICS. BinNavi. http://www.zynamics.com/binnavi.html.

Notes

[1]Note for reviewers: all tools and detailed experimentation will be available for the conference

[2]$P(i) = p$ means that two runs of $P(i)$ gives the exact same path p

[3]This is a classical seed for dynamic analysis.

[4]In this example, malloc returned `0x8040000` as concrete value, and `sizeof(int)=4`.

[5]`https://www.ece.uvic.ca/~frodo/jasper/`

[6]AFL generates in reality several paths, 174 is the number of unique path generated.

Appendices

```
p1=malloc(sizeof(int));
*p1=0;
free(p1);
p2=malloc(sizeof(int));
while(p2!=p1)
{
        free(p2);
        p2=malloc(sizeof(int));
}
*p2=42;
printf("p1 %d\n",*p1);
```

Appendix 1: Example of UaF not detected by classical techniques that replace heap allocator functions