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 (UAF) on binary code. The approach combines a static analyzer and a dynamic symbolic execution engine. We also introduce several original heuristics for the dynamic symbolic execution part, speeding up the exploration and making this combination effective in practice . 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 to the CVE-2015-5221.

CCS Concepts

•**Security and privacy** → **Formal security models;** *Software security engineering;* Vulnerability scanners;

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 such as static analysis [29, 11], dynamic analysis (fuzzing) [53] or Dynamic Symbolic Execution (DSE) [19, 17] have demonstrated their ability to detect these vulnerabilities, each with their own strengths and limitations. While *static analysis* allows to evaluate all possible program executions to detect complex patterns, it also requires to over-approximate the program behavior, leading to numerous *false positives* (i.e. misjudging safe parts of the code as vulnerable), which is a serious issue in security where *feasible* vulnerable paths are required. Note that this problem usually aggravates 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. *Dynamic Symbolic Execution* is a trade-off between static and dynamic analysis. This technique can trigger complex paths, but it comes with a significant execution overhead and scalability issues.

Challenge and goal. In this paper, we focus on a particular class of vulnerabilities named *Use-After-Free* [30]. A 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: on the one hand UAF are related to heap memory and require reasoning about possible aliases, which is known to be difficult in static analysis [41]; on the other hand, paths triggering UAF are hard to find, requiring to go through several events (allocation, release, use) possibly distant in the code, penalizing both dynamic analysis and DSE [36].

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, avoids false positives, and generates* proofs-of-concepts *as a set of inputs allowing to effectively trigger these vulnerabilities.*

Contribution.

- The main contribution of this paper is *a novel combination* of static analysis and dynamic symbolic execution *geared at detecting Use-After-Free* on binary code. This combination consists in computing a *weighted slice* from the static analysis and using it to *guide dynamic symbolic execution* (WS-Guided DSE).
- The second contribution is a running example of a *real vulnerability found by our approach* on the `JasPer` application (CVE-2015-5221), and practical comparison with standard DSE and fuzzing.
- Finally, we detail in depth aspects of our implementation – especially a novel *correct and complete* monitor

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

for UAF detection as well as library-driven tuning of search heuristics for DSE, and provide all tools and examples for the sake of reproducibility. By sharing tools and experiments[1], we hope to participate actively in the opening of the vulnerability detection activity and promoting better access to such techniques.

Discussion. Our work shows that static analysis can be fruitfully coupled with DSE for complex security purposes such as UAF detection, while most industrial approaches focus only on fuzzing techniques. In particular, the key issues which make our proposed approach effective are:

- a lightweight and dedicated static analysis, unsound and incomplete but *scalable* and *precise enough* to highlight a reasonable number of suspicious program slices with potential vulnerabilities.
- a dedicated DSE approach geared at exploring these suspicious slices in order to report only real vulnerabilities, together with original guidance heuristics for drastically reducing path exploration.

Our first experiments (Section 8) show that the approach already improves over basic DSE (which is not really adapted to single-goal coverage or safety properties), and that it may also beat fuzzing in some situations, typically when no good initial seed is available.

Yet, the presented work is still at an early stage. Especially, more experiments should be conducted, and the number of suspicious slices should be reduced. Possible mitigation measures are discussed in Section 8.4.

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 present the `Jasper` case-study in Section 8. Finally, we discuss related work and future improvements.

2. MOTIVATING EXAMPLE

A motivating example is presented in Figures 1 and 2, respectively showing the source code of the example and its graph representation. Line 11 represents potentially large 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 allocated memory block 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: line 6 is `false` → no UAF;
- P_2: line 6 is `true`, and line 14 is `true` → UAF in 23;
- P_3: line 6 is `true`, and line 14 is `false` → 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 paths of type P_2 contain 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 standard approaches. This example illustrates the limitation of standard vulnerability detection methods for UAF detection:

- static analysis can be used to detect UAF, however, as we discussed previously, reasoning with heap and aliases makes such analysis less precise [41] and dramatically increase the number of false positives;
- standard fuzzing techniques have difficulties 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, such as line 11; moreover it also brings a significant overhead, and can take a long time to trigger the UAF.

3. OVERVIEW OF OUR APPROACH

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

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

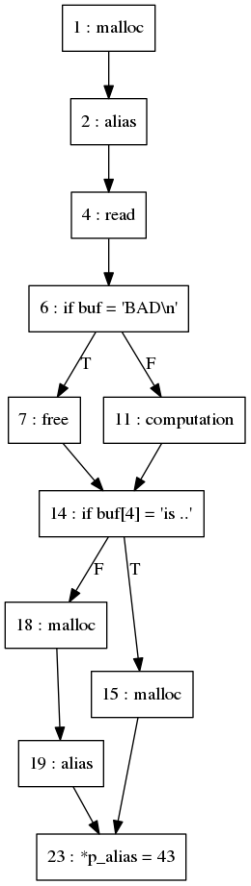


Figure 2: Motivating example: control-flow graph

- We use the strength of static analysis to detect some interesting sections of the code and discard other parts, thanks to a dedicated approach, scalable and precise enough, yet incomplete and unsound;
- Then, we focus the Dynamic Symbolic Execution (DSE) only on those interesting parts, with specific search heuristics reducing the path explosion. Note that DSE *enforces correctness*: a vulnerability reported is a real UAF, and we even get a *proof-of-concept* as an input able to trigger it.

Workflow. Figure 3 represents our tool-chain. The static analysis part relies on the GUEB [30, 37] tool. DSE is performed on the BINSEC/SE platform [26]. From a binary, the static analysis detects a UAF and extracts a slice containing it. We weight this slice and use it to guide the DSE exploration. Then, once a UAF is validated, a `proof-of-concept` (PoC) is generated.

Novelty. 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 both local (execution trace) and global (slice representation) information give a clear and deep understanding of root cause of the defect.

4. BACKGROUND

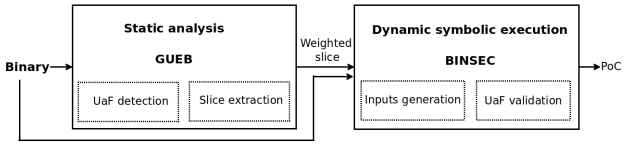


Figure 3: Combining static analysis and DSE

We detail in this section the static analysis step of our approach, and recall some basis about dynamic symbolic execution. While these techniques are not a novel contribution, they are needed to understand the remainder of this paper thoroughly.

4.1 Static Analysis

First we recall the main principle and design choices of GUEB, the static analysis tool we use for UAF detection. Then, we give some information about its implementation and some vulnerability examples we got using this tool.

Principle and design choices. GUEB performs a dedicated *value-analysis* [4] on binary code to provide, at each program location, an approximation of the set of values contained in each register and memory address. Knowing heap allocation and release functions (e.g., `malloc` and `free`), this analysis allows to retrieve an abstraction of the current heap state as two sets containing respectively the addresses of the **allocated** and **freed** heap 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.

Static analysis on binary code raises specific issues. In particular the Call Graph and CFG produced by disassemblers are usually both incorrect and incomplete, due either to dynamic jumps or difficulties to retrieve function bounds. Since GUEB targets potentially large codes, several design choices are required to make it *scalable* and *precise enough*, at the price of *soundness* and *completeness*.

- First, since correct and precise recovery of both CFG and Call Graph is a big challenge of binary-level static analysis [10], we rely on scalable syntactic recovery methods, as provided for example by IDAPro [39]. Despite being incorrect in principle, these techniques are the *de facto* standard in disassembly and reverse activities, and they perform reasonably well on non-protected code. We then consider as program entry points each functions without any predecessor in the Call Graph.
- Second, loop behaviors are under-approximated through bounded *loop unrolling*. Indeed, using fix-point computations to get precise enough over-approximations of a loop behaviors at the binary level is currently not tractable on large binary codes, while on the other hand, according to our experiments, a UAF does not depend that much on loop behaviors. Unrolling loops a few time happens to be sufficient in practice.
- Finally, functions are inlined with a bound on the depth of the call stack, in order to get a precise but limited (context-sensitive) analysis. Indeed, precise inter-procedural reasoning is required since *allocation*, *release* and *use* operations of a same memory chunk are usually distant from each other in the code.

The output produced by GUEB is a set of *CFG slices*, one per UAF found. Each slice is a sub-graph of the initial program CFG, starting from the entry point of a root function, and containing all the paths exercising a given UAF. More formally, $S = (V, E, n_{alloc}, n_{free}, n_{use})$ represent the slice S, where V denotes its nodes, E its edges, and $n_{alloc}, n_{free}, n_{use} \in V$ and are the allocation / free / use node representing the UAF. Each node refers to a basic block. As GUEB works on the inlined representation of the program, we represent a node by a pair $(addr * cs)$ where $addr$ is the instruction address and cs the call string. The slice obtained for the motivating example is depicted in Figure 4.

Due to the approximations made in the analysis, UAF paths in the extracted slice are not always feasible (false positives). As a result, all the potential vulnerabilities found by GUEB *have to be confirmed by providing some concrete inputs. This work objective is precisely to show how a guided dynamic symbolic execution can be used for this purpose.*

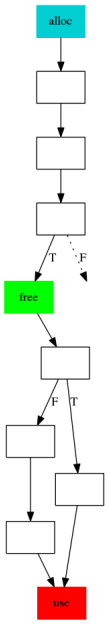


Figure 4: CFG slice produced by GUEB

Implementation and results. GUEB is built upon the BinNavi framework [58], which offers an assembly level intermediate representation (REIL) [28] and a basic API to prototype data-flow static analysis. Translation from binary code to assembly code can be provided by IDAPro [39].

GUEB has been first applied on (the already known) CVE-2013-4232 (`tiff2pdf`) to validate the approach. It has also been used to find six previously unknown UAF vulnerabilities (in `JasPer`: **CVE-2015-5221**, `openjpeg`: **CVE-2015-8871**, `giflib`: **CVE-2016-3177**, inside a debugging tool of `bind`, `accel-ppp` and `gnome-nettool`). *However, each of these new vulnerabilities has required a manual inspection of the results of* GUEB*, to confirm the feasibility of the issue.*

4.2 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) [17, 35, 54, 50, 33, 16, 15] 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) and assignations. 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 $\Phi_p = c_1 \wedge c_2 \wedge \cdots \wedge c_n$ 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 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: due to concretization [33, 50, 25], 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 – with or without losing correctness [32];

Easier adaptation to binary code analysis, compared to other formal methods; many binary-level DSE tools have been developed [8, 34, 22, 20, 2].

The 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 one.

The BINSEC **platform.** BINSEC [27] 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 named DBA [9] (including decoding, disassembling, simplifications), and several semantic analyses, including the BINSEC/SE DSE engine [26]. BINSEC/SE features strongly optimized path predicate generation as well as highly configurable search heuristics [26, 5] and C/S policies [25], and a stub mechanism for specifying the behavior of missing parts of the code (cf. Section 7.2).

5. CORE TECHNIQUE: WS-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 [16, 44, 34]. Unfortunately, it appears that relying on a single heuristic is often not sufficient and that different programs or different goals require different heuristics. Typically, DSE engines rely on a selection mechanism returning the *best* path candidate

w.r.t. a 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 quickly as possible. This path needs to respect a specific property: containing a UAF. To find such a 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 standard 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.

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}) nodes. We need to find an input leading through these three nodes. As already discussed, a key feature of DSE exploration is to prioritize which branches will be inverted first. We guide this selection using the slice, more precisely, we compute for each node a score to the destinations.

Distance Score. We denote DS the score to the destinations (*Distance Score*). In our experiments, DS is computed using shortest paths, in a preprocessing step before the exploration. As UAF requires three targeted nodes, we compute three scores for each destination, to: n_{alloc}, n_{free} and n_{use}. The prototype of DS is: $DS : n_{src} * n_{dst} \rightarrow score$

Remind that, since the static analysis performs function inlining, we associate nodes with their call-stack to distinguish the same instruction in different calling context (thus the node n is in fact represented as a pair, $addr * cs$).

To illustrate the need for three different scores, let us consider the example in Figure 5[2]. Two executions give two

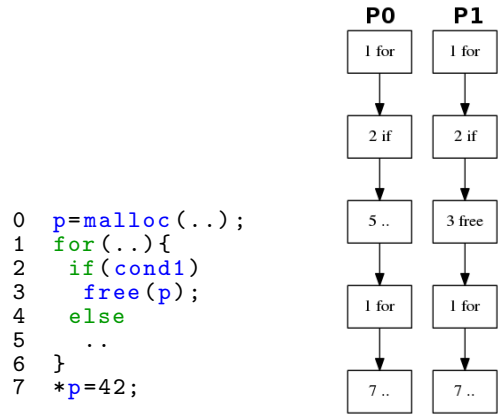


Figure 5: Score selection example

paths: $P0$ and $P1$, both unrolling the loop one time. $P0$ evaluates the condition at line 2 to `false` while $P1$ evaluates it as `true`, and so $P1$ contains the call to `free`. We focus in this example on the score used to determine 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 and n_{free} is the destination node used as parameter for DS. In $P1$, n_{free} was reached; we do not need to unroll a second time the loop and n_{use} is the destination node. Thereby, the node `"1 for"` does not take the same destination and therefore the same score, depending on if n_{free} was reached previously in the path or not.

The slice is thus weighted by DS to guide the exploration.

Paths exploration. Algorithm 1 describes a high-level view of how our method works. P represents a deterministic program[3], where $P(i)$ denotes the *(execution) trace* followed at runtime on input i [4]. At first, we use an input seed, i_0, to generate a set of initial paths and to initiate the working list WL, using `get_initial_paths`. Here, `select` chooses from the WL a triplet: a path p, a condition to invert c and its score. `compute_predicate` gives a path predicate from a path p with respect to a conditional node c_i (see Section 4.2). Function `solve` evaluates this path predicate and returns either an `UNSAT` *verdict*, or a `SAT` verdict with a corresponding input i. If the *trace* $P(i)$ does not contain a UAF, checked by the oracle σ (see Section 6), a set of pairs (c_i, dst_i) are *extracted* from p and S such that:

- c_i are conditions of p corresponding to nodes of S;
- dst_i is the destination node (n_{alloc}, n_{free} or n_{use}) associated to (p, c_i).

Then each (c_i, dst_i) is ranked with a score s_i, according to DS. Finally, the path p with these branch nodes and their scores are added to the working list WL. The algorithm stops either when σ holds on the current path, or when there is no more path to explore. Therefore, WS-Guided DSE ensures correctness of its output, assuming the oracle σ is correct. Such an oracle will be described in Section 6.

Algorithm 1: WS-Guided DSE

Input: Program P, DS, slice S, oracle σ, input seed i_0
Output: Input i, with P(i) respecting oracle σ
$WL := get\_initial\_paths(i_0)$;
while $WL \neq \emptyset$ **do**
  $select\ (p, c_i, s_i) \in WL$;
  $WL := WL \backslash \{(p, c_i, s_i)\}$;
  $\Phi_p := compute\_predicate(p, c_i)$;
  $verdict, i := solve(\Phi_p)$;
  if $verdict = SAT$ **then**
    $t := P(i)$;
    if t *respects* σ **then**
      return i;
    end
    $(c_0, dst_0)..(c_n, dst_n) := extract(p, S)$;
    compute $s_0..s_n$ where $s_i = DS(c_i, dst_i)$;
    $WL := WL \cup \{(p, c_0, s_0)..(p, c_n, s_n)\}$;
  end
end
return *Not Found*;

[2]Here the call stack is empty, and a node is only represented by its line number in the code

[3]Two runs of $P(i)$ give the exact same internal computations.

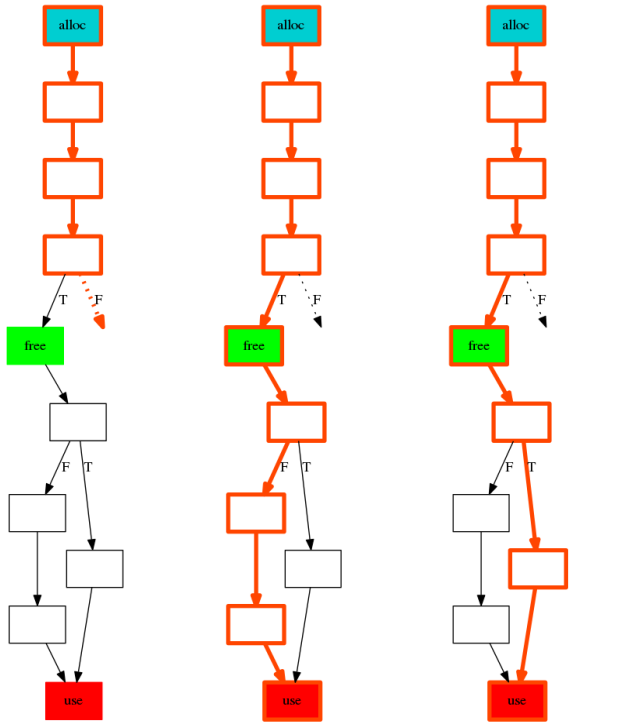
[4]A path is defined statically from the CFG, while a trace is defined at runtime.

PROPERTY 1 (CORRECTNESS). *If algorithm 1 terminates and returns an input value i, then the execution trace $P(i)$ respects σ. Hence, if σ is correct then i is a proof-of-concept triggering an UAF on program P.*

On the other hand, completeness is not ensured, since the oracle is checked against a single trace and not against all input following the same path. Moreover, since exploring all paths of S is not always possible (there could be an infinite number of paths), we must bound the exploration with a timeout or a fixed number of paths, losing completeness – but this is a standard and acceptable tradeoff with DSE.

Example. In our example, if we first try to explore the program with a file containing only 'A'[5] as input, the first condition is evaluated to `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 are thus 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 then creates a third input, by inverting line 14: `'BAD\nis a uaf\n'` (Figure 6c). As there is a UAF in the trace generated from this input, our oracle detects it (see Section 6).

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]This is a standard seed for dynamic analysis.

6. RUNTIME UAF 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 characterize 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 trace contains the UAF.*

Issue. The main problem when defining a *correct and complete runtime checker* for UAF detection is *aliasing due to re-allocations*, as shown in our motivating example. Allocation calls at lines 1 and 15 may return the *same* address when condition 6 is `true`. Therefore, pointers `p` and `p_alias` would be *implicit aliases*, but without referring to the *same* memory block (i.e., 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 an execution trace t if and only if the following property Φ holds:

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

(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 data-flow 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 7.2). Data-dependency computations should then take into account the side-effects of these functions. On the other hand, by using the *stubs*, `path predicates` implicitly keep the data-dependencies of symbolic values when pieces of code are missing. Therefore, we propose to use another approach, directly based on the symbolic reasoning performed by our DSE engine. The idea is to build a `trace predicate` φ_t for trace t, similar to a path predicate except that all input are fixed but only one *unconstrained* symbolic variable S_{alloc} corresponding to the value returned by the target allocation site n_{alloc}. Using a symbolic variable for S_{alloc} allows to detect the absence of data-flow relations between this address and the ones used by n_{free} and n_{use} and avoids the problem of implicit aliases occurring with concrete values. Indeed, Φ holds on a trace t if and only if the SMT formula $\varphi_t \wedge \Phi'$ is not `SAT`, where

$$\Phi' = (a_f \neq S_{alloc}) \vee (a_u \notin [S_{alloc}, S_{alloc} + size_{alloc} - 1])$$

Φ' is the negation of the properties (ii) and (iii) of Φ:

- $a_f \neq S_{alloc}$: the pointer given as the parameter for `free` is not the one allocated at n_{alloc} (negation of property (ii))
- $a_u \notin [S_{alloc}, S_{alloc} + size_{alloc} - 1]$: the pointer used is not a reaching definition of the pointer allocated at n_{alloc} (negation of property (iii))

Thus, if $\varphi_t \wedge \Phi'$ is `UNSAT`, these two conditions are false and thus (ii) and (iii) are respected: a UAF is present.

We use $\varphi_t \wedge \Phi'$ as oracle σ in Algorithm 1.

PROPERTY 2 (CORRECTNESS). *As (ii) and (iii) are respected, UAF detected by our Oracle are true positives.*

PROPERTY 3 (COMPLETENESS). *If a UAF is present in the trace, it is detected by our Oracle; there is no false negative.*

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[6]:

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

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

- For paths in P_3,

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

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. DSE APPLIED IN THE REAL WORLD

While DSE shows good ability to explore paths of a program, it requires some adaptations in order to be used effectively in real world examples. One problem is to explore part of the program related to standard libraries. When dealing with such cases, we use two different approaches:

- either we locally improve the search heuristic by using a set of known code programming behaviors;
- Or we use stubs of libraries to model their effects without tracing them.

Our goal here is not to replace the main search heuristics, but rather to improve it locally in order to avoid getting stuck in uninteresting parts of the path space.

7.1 Library-Driven Search Tuning

Libraries are generally used respecting some specific code programming behaviors. By knowing such patterns, we can improve the exploration and drives it with this knowledge. In the current implementation, these heuristics are based on information of common libraries, such as *libc*. We called them Libraries Driven Heuristics (LDH). As we guide the DSE by using a score provided by *DS*, we can detect that the exploration diverges from the destination (e.g. the score decreases). The idea is to use LDH to generate a set of new paths to explore. Then, thanks to *DS*, we can know if these new paths are interesting to reach the destination, or not. We describe in the following two heuristics needed during the exploration of realistic codes.

7.1.1 String Length Based Heuristic

The first heuristic helps when there is a comparison on a string whose 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 the same. Every time the loop is iterated, 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 at line 7, we need to unroll the loop at line 2 exactly 19 times (the size of the string plus the character `'\0'` which ends the string). This is a standard 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 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 at line 7: .. == 0 follows a call to `strcmp`. We use the size of the string `"this is really bad"`: 19, to prioritize the inversion of conditions located at the 19^{th} 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 plan 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).

7.1.2 Allocator Functions Behavior Based Heuristic

A second heuristic is imposed by 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 this example appears simple, standard DSE tools are not able to trigger the path, since to be reached, the size given to `malloc` needs 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` return 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.

7.2 Stubs

The new stub mechanism implemented in BINSEC/SE allows to over-approximate or simulate logical effects of an

[6]In this example, malloc returned `0x8040000` as concrete value, and `sizeof(int)=4`. p_0 and p_1 and p_alias_0 are the SSA variables created during the path predicate computation.

untraced library call. This makes possible 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 themselves system calls that would require some over-approximation, thus specifying the stub at library-level is a fair balance.

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

- `CONC`: concretize the given value (take runtime value);
- `LOGIC`: apply the logical operation specified in the stub;
- `SYMB`: symbolize the value, creating a new input.

`realloc` is an example of library code where the stub is not straightforward, but necessary. Tracing this call adds complex constraints on the *path predicate* (especially on the heap state). Skipping this function without taking into account its logical effect induces errors. Indeed, if the pointer returned by `realloc` is different from the pointer given as parameter, the values present in the original buffer are copied into the fresh buffer. By not keeping this side-effect, results of the analysis become 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 fifteen significant functions (such as `strcpy`, `strchr`) while we simply concretize the return value of the other functions (such as `open`, `printf`).

8. EXPERIMENTAL EVALUATION

We now describe our implementation and we detail the successful automated `proof-of-concept` creation achieved by WS-Guided DSE on a real CVE. To illustrate its efficiency, we compare our approach with and without WS-guidance or LDH, and we also compare the approach with two well-known fuzzers (`AFL` [1] and `radamsa` [49]).

8.1 Implementation

WS-Guided DSE (Algorithm 1) is primarily implemented in Binsec/se using OCaml functors. We thus allow a modular usage of the guided DSE. For example, the criterion 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 a DFS (depth-first search) exploring the slice, shortest paths or using shortest paths enhanced by the libraries driven heuristics (Section 7.1). As the LDH requires a score to know if generated paths are interesting, we could not combine them with DFS. We compute shortest paths using the `igraph`[7] library.

8.2 JasPer: case study

We demonstrate the efficiency of our approach through the study of a UAF in `JasPer`[8]. `JasPer` is used to convert an image from one format to another. GUEB returns several suspicious slices as result of the analysis of `JasPer`, *we detail in the following the validation of a particular selected slice, which was strongly suspected to contain a UAF*. The issue of slice selection is discussed in Section 8.4.

The vulnerability was found by GUEB and 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), shown in Figure 7.

```
MIF
component
```

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

At the time when this document is written, no official patch is available for this vulnerability, so the generated PoC 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")
```

It 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, `"component"` 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;
}
```

[7] http://igraph.org/
[8] https://www.ece.uvic.ca/~frodo/jasper/

```
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 assignment replace the character `'\n'` in this buffer by `'\0'`. Finally, the buffer is copied to a new buffer and compared to the string `"component"`. Similarly to the example at Section 7.1.1, 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.

8.3 Evaluation

We compare several versions of DSE and two standard fuzzers on their ability to detect the UAF fault in the considered slice. Actually, we report for each tool the time required to produce an input file with `"MIF\n"` as a first line and the time required to trigger the vulnerability. Experiments are performed on a standard laptop (`i7-2670QM`), and we use Boolector [13] as SMT solver. Table 1 reports details of the experimentation.

Table 1: `JasPer` evaluation

Name	Time	MIF line	UAF found	# Paths
DSE (in BINSEC/SE)				
WS-Guided+LDH	$20m$	$3min$	**Yes**	9
WS-Guided	$6h$	$3min$	*No*	44
$DFS(slice)$	$6h$	$3min$	*No*	68
DFS	$6h$	$3min$	*No*	354
standard fuzzers (arbitrary seed)				
AFL	$7h$	$< 1min$	*No*	174[†]
Radamsa	$7h$	$> 1h$	*No*	[‡]
standard fuzzers (MIF seed)				
AFL (*MIF input*)	$< 1min$	$< 1min$	**Yes**	< 10
Radamsa (*MIF input*)	$< 1min$	$< 1min$	**Yes**	< 10

[†] *AFL generates more input, 174 is the number of unique paths.*
[‡] *For radamsa it is not trivial to count the number of unique path.*

WS-Guided DSE and variants. The creation of the PoC with WS-Guided DSE (+ LDH heuristic) takes 20 minutes. Nine test cases are correctly generated while 225 path predicates are `UNSAT`. The C/S policy [25] used during our exploration kept as logical load addresses and concretized store addresses. We limit the exploration of loops up to one hundred iterations.

We try other variants of DSE in order to assess the effectiveness of our approach. Especially, we consider: WS-Guided DSE without LDH, standard DFS-based DSE and DFS-based DSE restricted to the slice of interest – where the search is cut when exiting the slice, but no distance score information is available (getting a technique similar to [21], yet simpler). All these variants easily find the word `MIF`, but none of them is able to trigger the UAF, illustrating the importance of both weighted slices and LDH.

Comparison with fuzzing. As a comparison with the state of the art technique, we used `AFL` [1] (American Fuzzy Lop) and `radamsa` [49] 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 the same results. *No crashes have been found.* These fuzzers are built in a perspective of coverage, not to find a particular path, thereby finding this specific bug is hard, as expected. We should mention that `AFL` creates an input with the `MIF` header in less than one minute, `radamsa` generates the word `MIF` after 1 hour and not as header of the file. The second line of the PoC being more complicated, they were not able to reproduce it.

Notice that by using as seed a correct `MIF` file (from the `JasPer` library), both fuzzers were able to generate the PoC 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.

Therefore, our approach is complementary to standard fuzzing in certain situations, typically when no good initial seed is available. Moreover, recent tools such as Driller [52] or the cyber reasoning systems used during the Cyber Grand Challenge [24] showed that fuzzing can be efficiently combined with symbolic execution; such combinations are a natural evolution of our system.

8.4 Discussion: slice selection

GUEB identifies 450 suspicious slices on `JasPer`, and we choose as validation a slice strongly suspected to be a true positive (through a quick manual check). This slice selection issue can be mitigated in two ways:

- First, we plan to improve GUEB to reduce the number of suspicious slices. For example, GUEB analyzes the same binary from several entry points, so that several results refer in fact to the same issue. By simply merging together slices sharing the same root cause, we can already reduce the number of slices by $30-50\%$. For example, in the case of `JasPer` the number of slices falls to 250. We are confident that slightly more accurate analyses and more sophisticated slice merges should allow a significant reduction of suspicious slices;
- Second, considering that DSE can be launched in parallel on several slices and that additional cores go cheaper and cheaper with a Moore-like rate, exploring a few hundreds of slices for a few hours is already a realistic scenario, even for a small size organization.

As a concluding remark, note also that our approach is already well adapted in a context of semi-automated analysis when a user has already selected a set of interesting slices from GUEB and wants them to be automatically validated with DSE. Indeed, even with the knowledge that a slice contains effectively a UAF, an input triggering the vulnerability is generally mandatory to demonstrate its reality. From our experiences of vulnerability disclosure, developers are more willing to patch the vulnerability if a test-case is provided.

9. RELATED WORK

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

Static UAF detection. In the specific case of UAF detection, static techniques exist [18, 36] but unfortunately they are not available; so no fair comparison can be made. Moreover, they are not always able to provide precise enough results on large examples [42]. This is particularly the case when they are applied on binary code. Therefore, most existing detection techniques are based on dynamic runtime analysis.

Dynamic UAF detection. First, general "memory error detectors" like Valgrind [48] or AddressSanitizer [51] 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 8 and in [42]. Note also that this solution is not well suited for custom allocators. Another approach, adding metadata information, as it can be found in SoftBoundCETS [45] (a combination of SoftBound [46] and CETS [47]) allows a dynamic detection of UAF, yet it requires to recompile the program from the source code.

A second category contains more specific tools like Undangle [14] or DangNull [42]. 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 many UAF vulnerabilities as possible, 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 – but *correct* and *complete* – oracle based on SMT solvers.

Guided DSE. Using score functions to guide the path exploration during a DSE has been introduced in several tools, in dedicated [34, 16, 15] or generic ways [55, 5], either to improve 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 [23, 57] to cover sequential execution patterns (e.g., safety properties). In these latter work, scores are associated with branch conditions, essentially based both on control-flow and data-flow properties. We currently rely only on one control-flow information: the shortest path to the destination. Using data-flow as well, or other control-flow metrics is an interesting work direction. In [31], we studied how random walks can be used to better guide a DSE exploration inside a given subpart of the program. This is a natural extension of the approach proposed in this paper.

Combination of static analysis and DSE. Despite its many successes [17, 35], 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.* [21] first use 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. Bardin *et al.* [7, 6] design a similar approach for proving the infeasibility of some white-box testing objectives, before launching DSE in order to cover them all. In [56], authors combine static analysis with DSE, using a *proximity heuristic* computed statically to guide the exploration to generate from a bug similar traces leading towards it. In [3], a data-flow analysis is combined with shortest path in the Visible Pushdown Automaton (VPA) representation of the program to find vulnerabilities. In vulnerability detection, we can also mention for instance Dowser [38], which finds buffer-overflows by guiding the DSE tool S2E [22] 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 [43], to confirm memory leaks found in C++ programs by the static analyzer HP Fortify [40].

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

10. CONCLUSION

In this paper, we have 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 enough to find complex vulnerabilities on real world applications. Our platform is still at a early stage, and many improvements are ongoing.

Future work. Guiding heuristics proposed in Section 7.1 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 standard programming patterns. Naturally we need to apply our methodology to a larger dataset, and we intend to continue using our platform to detect new vulnerabilities. We shared our platform as an open-source project to go in this direction. More complex software (like multithreaded applications, browser) are for now out of the 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 plan to apply our approach to other types of subgraphs. For example, from

a patched binary and its original version, binary comparison tools (such as BinDiff) allow us to know the difference between the two versions. 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` [12].

11. REFERENCES

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

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with VeriTesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14. ACM Press, 2014.

[3] D. Babic, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*. ACM, 2011.

[4] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.

[5] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbé. Binary-level testing of embedded programs. In *13th International Conference on Quality Software, QRS'13*, 2013.

[6] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov. 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*. Springer, 2014.

[7] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion. 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*. IEEE, 2015.

[8] S. Bardin and P. Herrmann. Osmose: Automatic structural testing of executables. *Software Testing, Verification Reliability*, 21(1), 2011.

[9] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The Bincoa Framework for Binary Code Analysis. In *Computer Aided Verification - 23rd International Conference, CAV 2011, 2011*. Springer, 2011.

[10] S. Bardin, P. Herrmann, and F. Védrine. Refinement-based CFG reconstruction from unstructured programs. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Springer, 2011.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 2010.

[12] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *SP 2008*. IEEE, 2008.

[13] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *Lecture Notes in Computer Science*. Springer, 2009.

[14] 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*, ISSTA 2012. ACM, 2012.

[15] 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*, OSDI'08. USENIX Association, 2008.

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06. ACM, 2006.

[17] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2), 2013.

[18] S. Cesare. Bugalyze.com - detecting bugs using decompilation and data flow analysis. In *BlackHatUSA*, 2013.

[19] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2012.

[20] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12. IEEE Computer Society, 2012.

[21] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.*, 21(1), 2014.

[22] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1), 2012.

[23] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2013.

[24] Darpa. Cyber grand challenge. https://www.cybergrandchallenge.com.

[25] R. David, S. Bardin, J. Feist, J.-Y. Marion, L. Mounier, M.-L. Potet, and T. D. Ta. Specification of concretization and symbolization policies in symbolic execution. In *Proceedings of ISSTA*. ACM, 2016.

[26] R. David, S. Bardin, J. Feist, J.-Y. Marion, M.-L. Potet, and T. D. Ta. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *Proceedings of SANER 2016*. IEEE, 2016.

[27] A. Djoudi and S. Bardin. Binsec: Binary code analysis with low-level regions. In *TACAS 2015*. Springer, 2015.

[28] T. Dullien and S. Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[29] P. Emanuelsson and U. Nilsson. A comparative study of industrial static analysis tools. *Electr. Notes Theor. Comput. Sci.*, 217, 2008.

[30] J. Feist, L. Mounier, and M. Potet. Statically detecting use after free on binary code. *J. Computer Virology and*

Hacking Techniques, 10(3), 2014.

[31] J. Feist, L. Mounier, and M.-L. Potet. Guided dynamic symbolic execution using subgraph control-flow information. In *Proceedings of SEFM*. Springer, 2016.

[32] P. Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011.

[33] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6), 2005.

[34] P. Godefroid, M. Y. Levin, and D. A. Molnar. 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*. The Internet Society, 2008.

[35] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012.

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

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

[38] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13. USENIX Association, 2013.

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

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

[41] W. Landi. Undecidability of static analysis. *LOPLAS*, 1(4), 1992.

[42] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[43] M. Li, Y. Chen, L. Wang, and G. Xu. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013. ACM, 2013.

[44] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007.

[45] S. Nagarakatte. Softboundcets. http://www.cs.rutgers.edu/~santosh.nagarakatte/softbound/.

[46] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In M. Hind and A. Diwan, editors, *PLDI*, pages 245–258. ACM, 2009.

[47] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *ISMM*, 2010.

[48] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6), 2007.

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

[50] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5), 2005.

[51] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12. USENIX Association, 2012.

[52] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*. The Internet Society, 2016.

[53] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[54] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for C functions. In *Automated Software Engineering, 2004*. IEEE, 2004.

[55] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. 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*. IEEE Computer Society, 2009.

[56] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*. ACM, 2010.

[57] Y. Zhang, Z. Clien, J. Wang, W. Dong, and Z. Liu. Regular property guided dynamic symbolic execution. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15. IEEE Press, 2015.

[58] Zynamics. BinNavi. http://www.zynamics.com/binnavi.html.

APPENDIX

```
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); // uaf in *p1
```

Figure 8: Example of UAF not detected by standard techniques that replace heap allocator functions