

Software Security & Secure Programming (L. Mounier)

Written Exam - Tuesday January the 23rd, 2018

Duration: 2 hours – Answers can be written either in English or in French.
All documents allowed apart books – Calculators are forbidden.

This exam contains two distinct parts:

1. One exercise, supposed to be solved in about 30 minutes;
2. Some questions on a research paper, allow 1h30 to read the paper and answer these questions.

Exercise. (~ 6 pts)

We consider the following C program which contains a *buffer-overflow* vulnerability at line 13:

```
1 int x, y ;
2 int T[10] ;
3
4 scanf("%d", &x) ;
5 y = x+1 ;
6
7 while (y <= 5) {
8     if (x > 0) {
9         y = y*3 ;
10        x = x+2 ;
11    } else {
12        y = y*2 ;
13        T[y] = x ;
14    } ;
15 }
```

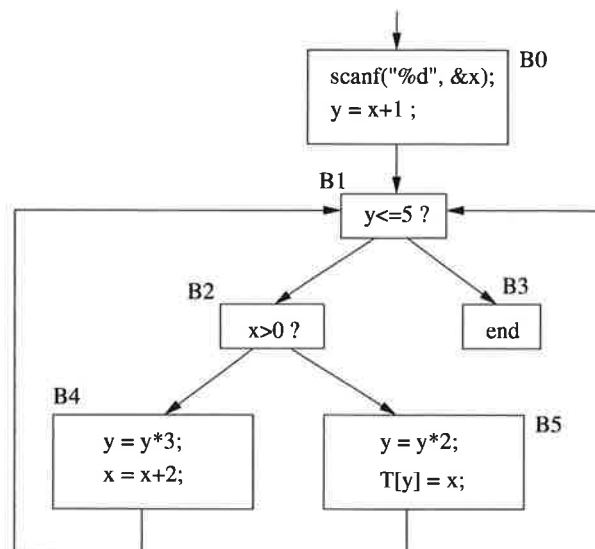


Figure 1: A vulnerable program and its Control-Flow Graph (CFG)

First we try to detect this vulnerability using **symbolic execution**, for instance with a tool like PathCrawler.

Q1. We consider the CFG path B0-B1-B2-B5.

1. Give its path predicate.
2. Give a solution (i.e., an input value of x) for this path predicate (making this path executable).
3. Give a necessary condition to follow this path **and activate the vulnerability**.
Is there a solution (i.e., an input value of x) which satisfies this condition ?

Q2. We consider the CFG path B0-B1-B2-B4-B1-B2-B5.

1. Give its path predicate.
2. Give a necessary condition to follow this path and activate the vulnerability. Is there a solution (i.e., an input value of x) which satisfies this condition ? You may consider two situations:
 - the C type `int` is interpreted as set \mathbb{Z}
 - the C type `int` is interpreted as **fixed-size** unsigned integers (e.g. 32 bits long)

Q3. We now consider a static verification approach based on **abstract interpretation**, for instance with a tool like Frama-C “value analysis”.

1. Add the assertions in the code allowing to detect runtime errors related to *arithmetic overflows* and *out-of-bound buffer accesses*.
2. We would like to know which of these assertions could be discharged using the *interval abstract domain* (i.e., the initial input value of x is in $[-\infty, +\infty]$) Explain your answer by giving the abstract values obtained for x and y at entry and exit points of each basic block of the CFG (after one iteration, and, if necessary, after several iterations).

Questions on a research paper. (~ 14 pts)

Read the paper given in appendix, from the beginning to the end of section 5, answering the following questions. The objective of this part of the exam is to evaluate your ability to understand a description of a vulnerability detection tool (its strengths and limitations, with respect to other approaches you know). When answering the following questions you should not copy entire sentences from the paper but rather illustrate your point with examples from your own.

Q1 [sections 1 and 2].

Explain the objective of *Dowser* and the workflow it follows. You should answer in about 10 lines, including picture of the tool workflow, indicating what is produced by each intermediate step.

Q2. Step 1 [section 3].

1. What are the input/outputs of this step ?
2. Is it based on a static or dynamic code analysis technique ?
3. Explain in a few lines how the “scores” are chosen and what is the motivation behind [section 3.3]
4. Is there some example of buffer overflow patterns not caught by this heuristic (i.e., with a score equal to 0) ?

Q3. Step 2 [section 4].

1. What are the input/outputs of this step ?
2. Is it based on a static or dynamic code analysis technique ?
3. Explain in a few lines how *implicit* (or control) dependencies are dealt with, giving some examples [section 4.1]

EXERCISE 1.

Q1

B0 - B1 - B2 - B5

Path predicate: $y_0 = x_0 + 1$, $y_0 \leq 5$, $x_0 > 0$, $y_1 = y_0 * 2$

equivalent to $y_1 > 9$

Example of value of $x \rightarrow x = 3$

Path to execute the vuln: $y_0 = x_0 + 1$, $y_0 \leq 5$, $x_0 > 0$, $y_1 = y_0 * 2$, ($y_1 < 0$ or $y_1 \geq 10$).

this executable path may be activated with: $x_0 = 4$

$$y_0 = 5$$

$$y_0 \leq 5$$

$$x_0 > 0$$

$$y_1 = 10 \rightarrow \text{vuln activated!}$$

Q2

B0 - B1 - B2 - B4 - B1 - B2 - B5

Path predicate:

$y_0 = x_0 + 1$, $y_0 \leq 5$, $x_0 > 0$, ($y_1 = y_0 * 3$ and $x_1 = x_0 + 2$),

$y_1 \leq 5$, $x_1 \leq 0$, $y_2 = y_1 * 2$

WATCH OUT FOR THIS \rightarrow The statement has to be FALSE in order for you to go to B5

To activate the vuln we would also need ($y_2 < 0$ or $y_2 > 9$)

But there is no solution in \mathbb{Z} to activate this, One possibility is

to consider wrap around: $x_0 = \text{MAX_INT}$

$$x_1 = \text{MAX_INT} + 1 = -1 \quad \checkmark$$

Q3.

```

1  int x, y ;
2  int T[10] ;
3
4  scanf("%d", &x) ;
5  y = x+1 ;
6
7  while (y <= 5) {
8      if (x > 0) {
9          y = y*3 ;
10         x = x+2 ;
11     } else {
12         y = y*2 ;
13         T[y] = x ;
14     } ;
15 } ;

```

assert $x+1 < \text{MAX_INT}$

assert $y+1 < \text{MAX_INT}$

$y \times 3 < \text{MAX_INT}$

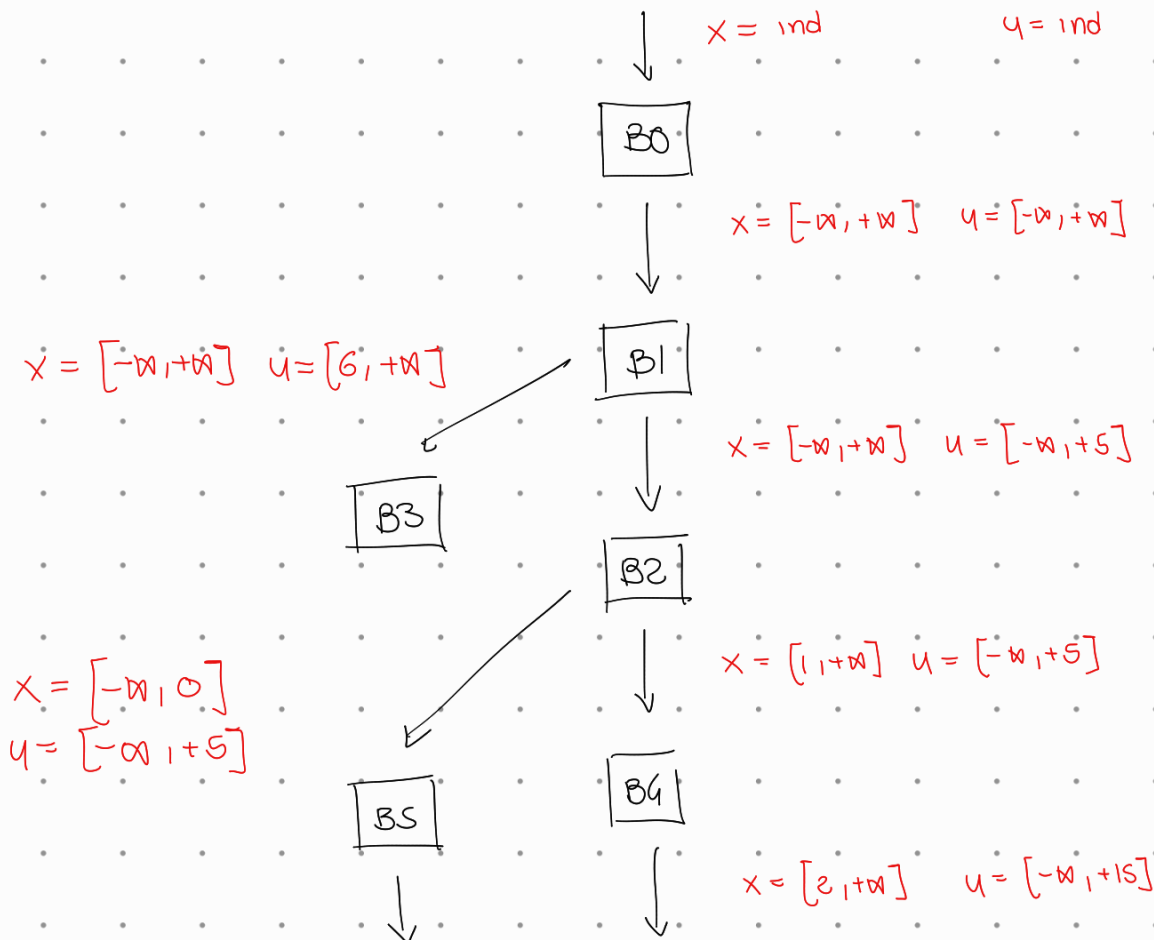
$x+2 < \text{MAX_INT}$

$y \times 2 < \text{MAX_INT}$

$y \geq 0$ and $y < 10$

We first select a path which may be: B0 - B1 - B2 - B4 - B5

$x \in [-\infty, +\infty]$



4. Where do come “false dependencies”, how are they weed out. What is the main requirement for this step ? [section 4.2]

Q4. Step 3 [section 5.]

1. What are the input/outputs of this step ?
2. Is it based on a static or dynamic code analysis technique ?
3. Explain why *loops* are a problem for symbolic execution ? How is this problem solved in Dowser ?
Is it a *sound* approach ? (why, illustrate your answer with some example ?)

Q5. To summarize ...

1. Is it possible to get false positives with *Dowser* ? And false negatives ?
2. Do you think a similar approach could be applied to other kinds of vulnerabilities (e.g., *use-after-free*) ?
Explain your answer in a few lines
3. Is it mandatory to have the source code of the target program to run *Dowser* ? Why, or why not ?
4. Do you think Dowser would be able to detect the vulnerability illustrated in Exercise 1 ?

Dowsing for overflows: A guided fuzzer to find buffer boundary violations

Istvan Haller
VU University Amsterdam

Asia Slowinska
VU University Amsterdam

Matthias Neugschwandtner
Vienna University of Technology

Herbert Bos
VU University Amsterdam

Abstract

Dowser is a ‘guided’ fuzzer that combines taint tracking, program analysis and symbolic execution to find buffer overflow and underflow vulnerabilities buried deep in a program’s logic. The key idea is that analysis of a program lets us pinpoint the right areas in the program code to probe and the appropriate inputs to do so.

Intuitively, for typical buffer overflows, we need consider only the code that accesses an array in a loop, rather than all possible instructions in the program. After finding all such candidate sets of instructions, we rank them according to an estimation of how likely they are to contain interesting vulnerabilities. We then subject the most promising sets to further testing. Specifically, we first use taint analysis to determine which input bytes influence the array index and then execute the program symbolically, making only this set of inputs symbolic. By constantly steering the symbolic execution along branch outcomes most likely to lead to overflows, we were able to detect deep bugs in real programs (like the `nginx` webserver, the `inspired` IRC server, and the `ffmpeg` videoplayer). Two of the bugs we found were previously undocumented buffer overflows in `ffmpeg` and the `poppler` PDF rendering library.

1 Introduction

We discuss *Dowser*, a ‘guided’ fuzzer that combines taint tracking, program analysis and symbolic execution, to find buffer overflow bugs buried deep in the program’s logic.

Buffer overflows are perennially in the top 3 most dangerous software errors [12] and recent studies suggest this will not change any time soon [41, 38]. There are two ways to handle them. Either we harden the software with memory protectors that terminate the program when an overflow occurs (at runtime), or we track down the vulnerabilities before releasing the software (e.g., in the testing phase).

Memory protectors include common solutions like shadow stacks and canaries [11], and more elaborate compiler extensions like WIT [3]. They are effective in preventing programs from being exploited, but they do not remove the overflow bugs themselves. Although it is better to crash than to allow exploitation, crashes are undesirable too!

Thus, vendors prefer to squash bugs beforehand and typically try to find as many as they can by means of fuzz testing. Fuzzers feed programs invalid, unexpected, or random data to see if they crash or exhibit unexpected behavior¹. As an example, Microsoft made fuzzing mandatory for every untrusted interface for every product, and their fuzzing solution has been running 24/7 since 2008 for a total of over 400 machine years [18].

Unfortunately, the effectiveness of most fuzzers is poor and the results rarely extend beyond shallow bugs. Most fuzzers take a ‘blackbox’ approach that focuses on the input format and ignores the tested software target. Blackbox fuzzing is popular and fast, but misses many relevant code paths and thus many bugs. Blackbox fuzzing is a bit like shooting in the dark: you have to be lucky to hit anything interesting.

Whitebox fuzzing, as implemented in [18, 7, 10], is more principled. By means of symbolic execution, it exercises all possible execution paths through the program and thus uncovers all possible bugs – although it may take years to do. Since full symbolic execution is slow and does not scale to large programs, it is hard to use it to find complex bugs in large programs [7, 10]. In practice, the aim is therefore to first cover as much unique code as possible. As a result, bugs that require a program to execute the same code many times (like buffer overflows) are hard to trigger except in very simple cases.

Eventual completeness, as provided by symbolic execution, is both a strength and a weakness, and in this paper, we evaluate the exact opposite strategy. Rather

¹See <http://www.fuzzing.org/> for a collection of available fuzzers

than testing all possible execution paths, we perform *spot checks* on a small number of code areas that look likely candidates for buffer overflow bugs and test each in turn.

The drawback of our approach is that we execute a symbolic run for each candidate code area—in an iterative fashion. Moreover, we can discover buffer overflows only in the loops that we can exercise. On the other hand, by homing in on promising code areas directly, we speed up the search considerably, and manage to find complicated bugs in real programs that would be hard to find with most existing fuzzers.

Contributions The goal we set ourselves was to develop an efficient fuzzer that actively *searches* for buffer overflows *directly*. The key insight is that careful analysis of a program lets us pinpoint the right places to probe and the appropriate inputs to do so. The main contribution is that our fuzzer directly zooms in on these buffer overflow candidates and explores a novel ‘spot-check’ approach in symbolic execution.

To make the approach work, we need to address two main challenges. The first challenge is *where* to steer the execution of a program to increase the chances of finding a vulnerability. Whitebox fuzzers ‘blindly’ try to execute as much of the program as possible, in the hope of hitting a bug eventually. Instead, *Dowser* uses information about the target program to identify code that is most likely to be vulnerable to a buffer overflow.

For instance, buffer overflows occur (mostly) in code that accesses an array in a loop. Thus, we look for such code and ignore most of the remaining instructions in the program. Furthermore, *Dowser* performs static analysis of the program to *rank* such accesses. We will evaluate different ranking functions, but the best one so far ranks the array accesses according to complexity. The intuition is that code with convoluted pointer arithmetic and/or complex control flow is more prone to memory errors than straightforward array accesses. Moreover, by focusing on such code, *Dowser* prioritizes bugs that are complicated—typically, the kind of vulnerabilities that static analysis or random fuzzing cannot find. The aim is to reduce the time wasted on shallow bugs that could also have been found using existing methods. Still, other rankings are possible also, and *Dowser* is entirely agnostic to the ranking function used.

The second challenge we address is *how* to steer the execution of a program to these “interesting” code areas. As a baseline, we use *concolic* execution [43]: a combination of concrete and symbolic execution, where the concrete (fixed) input starts off the symbolic execution. In *Dowser*, we enhance concolic execution with two optimizations.

First, we propose a new path selection algorithm. As we saw earlier, traditional symbolic execution aims

at code coverage—maximizing the fraction of individual branches executed [7, 18]. In contrast, we aim for *pointer value* coverage of *selected* code fragments. When *Dowser* examines an interesting pointer dereference, it steers the symbolic execution along branches that are likely to alter the value of the pointer.

Second, we reduce the amount of symbolic input as much as we can. Specifically, *Dowser* uses dynamic taint analysis to determine which input bytes influence the pointers used for array accesses. Later, it treats only these inputs as symbolic. While taint analysis itself is not new, we introduce novel optimizations to arrive at a set of symbolic inputs that is as *accurate* as possible (with neither too few, nor too many symbolic bytes).

In summary, *Dowser* is a new fuzzer targeted at vendors who want to test their code for buffer overflows and underflows. We implemented the analyses of *Dowser* as LLVM [23] passes, while the symbolic execution step employs S2E [10]. Finally, *Dowser* is a *practical* solution. Rather than aiming for all possible security bugs, it specifically targets the class of buffer overflows (one of the most, if not the most, important class of attack vectors for code injection). So far, *Dowser* found several real bugs in complex programs like *nginx*, *ffmpeg*, and *inspired*. Most of them are extremely difficult to find with existing symbolic execution tools.

Assumptions and outline Throughout this paper, we assume that we have a test suite that allows us to reach the array accesses. Instructions that we cannot reach, we cannot test. In the remainder, we start with a big picture and the running example (Section 2). Then, we discuss the three main components of *Dowser* in turn: the selection of interesting code fragments (Section 3), the use of dynamic taint analysis to determine which inputs influence the candidate instructions (Section 4), and our approach to nudge the program to trigger a bug during symbolic execution (Section 5). We evaluate the system in Section 6, discuss the related projects in Section 7. We conclude in Section 8.

2 Big picture

The main goal of *Dowser* is to manipulate the pointers that instructions use to access an array in a loop, in the hope of forcing a buffer overrun or underrun.

2.1 Running example

Throughout the paper, we will use the function in Figure 1 to illustrate how *Dowser* works. The example is a simplified version of a buffer underrun vulnerability in the *nginx*-0.6.32 web server [1]. A specially crafted

A buffer underrun vulnerability in nginx

```

11 int ngx_http_parse_complex_uri(ngx_http_request_t *r)
12 {
13     state = sw_usual;
14     u_char* p = r->uri_start; // user input
15     u_char* u = r->uri.data; // store normalized uri here
16     u_char ch = *p++; // the current character
17
18     while (p <= r->uri_end) {
19         switch (state) {
20             case sw_usual:
21                 if (ch == '/')
22                     state = sw_slash; *u++ = ch;
23                 else if /* many more options here */
24                     ch == *p++; break;
25             case sw_slash:
26                 if (ch == '/')
27                     *u++ = ch;
28                 else if (ch == '.')
29                     state = sw_dot; *u++ = ch;
30                 else if /* many more options here */
31                     ch == *p++; break;
32             case sw_dot:
33                 if (ch == '.')
34                     state = sw_dot_dot; *u++ = ch;
35                 else if /* many more options here */
36                     ch == *p++; break;
37             case sw_dot_dot:
38                 if (ch == '/')
39                     state = sw_slash; u -= 4;
40                 while (*(u-1) != '/') u--;
41                 else if /* many more options here */
42                     ch == *p++; break;
43         }
44     }
45 }

```

Nginx is a web server—in terms of market share across the million busiest sites, it ranks third in the world. At the time of writing, it hosts about 22 million domains worldwide. Versions prior to 0.6.38 had a particularly nasty vulnerability [1].

When nginx receives an HTTP request, the parsing function `ngx_http_parse_complex_uri`, first normalizes a uri path in `p=r->uri_start` (line 4), storing the result in a heap buffer pointed to by `u=r->uri.data` (line 5). The while-switch implements a state machine that consumes the input one character at a time, and transform it into a canonical form in `u`.

The source of the vulnerability is in the `sw_dot_dot` state. When provided with a carefully crafted path, nginx wrongly sets the beginning of `u` to a location somewhere below `r->uri.data`. Suppose the uri is `"../../../../foo"`. When `p` reaches `"/foo"`, `u` points to `(r->uri.data+4)`, and state is `sw_dot_dot` (line 30). The routine now decreases `u` by 4 (line 32), so that it points to `r->uri.data`. As long as the memory below `r->uri.data` does not contain the character `"/"`, `u` is further decreased (line 33), even though it crosses buffer boundaries. Finally, the user provided input (`"foo"`) is copied to the location pointed to by `u`.

In this case, the overwritten buffer contains a pointer to a function, which will be eventually called by nginx. Thus the vulnerability allows attackers to modify a function pointer, and execute an arbitrary program on the system.

It is a complex bug that is hard to find with existing solutions. The many conditional statements that depend on symbolic input are problematic for symbolic execution, while input-dependent indirect jumps are also a bad match for static analysis.

Fig. 1: A simplified version of a buffer underrun vulnerability in nginx.

input tricks the program into setting the `u` pointer to a location outside its buffer boundaries. When this pointer is later used to access memory, it allows attackers to overwrite a function pointer, and execute arbitrary programs on the system.

Figure 1 presents only an excerpt from the original function, which in reality spans approximately 400 lines of C code. It contains a number of additional options in the `switch` statement, and a few nested conditional `if` statements. This complexity severely impedes detecting the bug by both static analysis tools and symbolic execution engines. For instance, when we steered S2E [10] all the way down to the vulnerable function, and made solely the seven byte long uri path of the HTTP message symbolic, it took over 60 minutes to track down the problematic scenario. A more scalable solution is necessary in practice. Without these hints, S2E did not find the bug at all during an eight hour long execution.² In contrast, *Dowser* finds it in less than 5 minutes.

The primary reason for the high cost of the analysis in S2E is the large number of conditional branches which depend on (symbolic) input. For each of the branches, symbolic execution first checks whether either the condition or its negation is satisfiable. When both branches are feasible, the default behavior is to examine both. This

procedure results in an exponentially growing number of paths.

This real world example shows the need for (1) focusing the powerful yet expensive symbolic execution on the most interesting cases, (2) making informed branch choices, and (3) minimizing the amount of symbolic data.

2.2 High-level overview

Figure 2 illustrates the overall *Dowser* architecture.

First, it performs a data flow analysis of the target program, and ranks all instructions that access buffers in loops ①. While we can rank them in different ways and *Dowser* is agnostic as to the ranking function we use, our experience so far is that an estimation of complexity works best. Specifically, we rank calculations and conditions that are more complex higher than simple ones. In Figure 1, `u` is involved in three different operations, i.e., `u++`, `u--`, and `u-=4`, in multiple instructions inside a loop. As we shall see, these intricate computations place the dereferences of `u` in the top 3% of the most complex pointer accesses across nginx.

In the second step ②, *Dowser* repeatedly picks high-ranking accesses, and selects test inputs which exercise them. Then, it uses dynamic taint analysis to determine which input bytes influence pointers dereferenced in the candidate instructions. The idea is that, given the for-

²All measurements in the paper use the same environment as in Section 6.

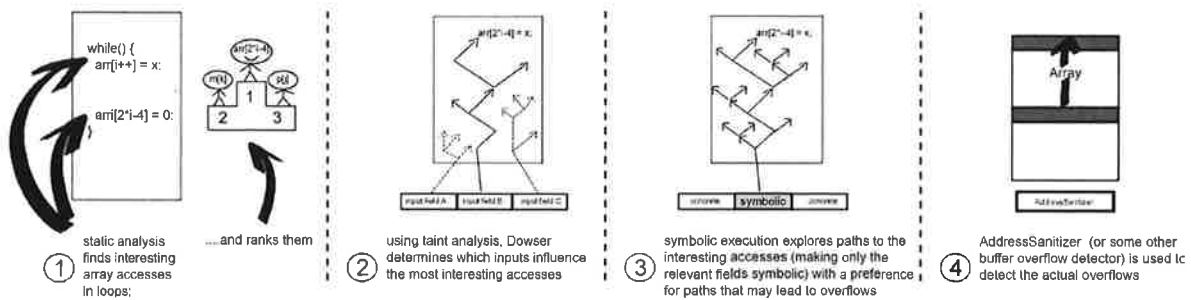


Fig. 2: Dowser—high-level overview.

mat of the input, *Dowser* fuzzes (i.e., treats as symbolic), only those fields that affect the potentially vulnerable memory accesses, and keeps the remaining ones unchanged. In Figure 1, we learn that it is sufficient to treat the uri path in the HTTP request as symbolic. Indeed, the computations inside the vulnerable function are independent of the remaining part of the input message.

Next ③, for each candidate instruction and the input bytes involved in calculating the array pointer, *Dowser* uses symbolic execution to try to nudge the program toward overflowing the buffer. Specifically, we execute symbolically the loop that contains the candidate instructions (and thus should be tested for buffer overflows)—treating only the relevant bytes as symbolic. As we shall see, a new path selection algorithm helps to guide execution to a possible overflow quickly.

Finally, we detect any overflow that may occur. Just like in whitebox fuzzers, we can use any technique to do so (e.g., Purify, Valgrind [30], or BinArmor [37]). In our work, we use Google’s AddressSanitizer [34] ④. It instruments the protected program to ensure that memory access instructions never read or write so called, “poisoned” red zones. Red zones are small regions of memory inserted inbetween any two stack, heap or global objects. Since they should never be addressed by the program, an access to them indicates an illegal behavior. This policy detects sequential buffer over- and underflows, and some of the more sophisticated pointer corruption bugs. This technique is beneficial when searching for new bugs since it will also trigger on silent failures, not just application crashes. In the case of *nginx*, AddressSanitizer detects the underflow when the *u* pointer reads memory outside its buffer boundaries (line 33).

We explain step ① (static analysis) in Section 3, step ② (taint analysis) in Section 4, and step ③ (guided execution) in Section 5.

3 Dowsing for candidate instructions

Previous research has shown that software complexity metrics collected from software artifacts are helpful in finding vulnerable code components [16, 44, 35, 32]. However, even though complexity metrics serve as useful indicators, they also suffer from low precision or recall values. Moreover, most of the current approaches operate at the granularity of modules or files, which is too coarse for the directed symbolic execution in *Dowser*.

As observed by Zimmermann et al. [44], we need metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. In principle, *Dowser* can work with any metric capable of ranking groups of instructions that access buffers in a loop. So, the question is how to design a good metric for complexity that satisfies this criterion? In the remainder of this section, we introduce one such metric: a heuristics-based approach that we specifically designed for the detection of potential buffer overflow vulnerabilities.

We leverage a primary pragmatic reason behind complex buffer overflows: convoluted pointer computations are hard to follow by a programmer. Thus, we focus on ‘complex’ array accesses realized inside loops. Further, we limit the analysis to pointers which evolve together with loop induction variables, i.e., are repeatedly updated to access (various) elements of an array.

Using this metric, *Dowser* ranks buffer accesses by evaluating the complexity of data- and control-flows involved with the array index (pointer) calculations. For each loop in the program, it first statically determines (1) the set of all instructions involved in modifying an array pointer (we will call this a pointer’s *analysis group*), and (2) the conditions that guard this analysis group, e.g., the condition of an *if* or *while* statement containing the array index calculations. Next, it labels all such sets with scores reflecting their complexity. We explain these steps in detail in Sections 3.1, 3.2, and 3.3.

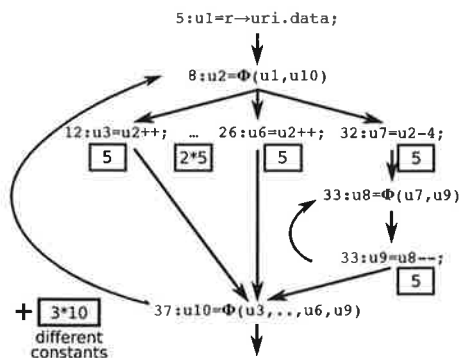


Fig. 3: Data flow graph and analysis group associated with the pointer u from Figure 1. For the sake of clarity, the figure presents pointer arithmetic instructions in pseudo code. The PHI nodes represent locations where data is merged from different control-flows. The numbers in the boxes represent points assigned by *Dowser*.

3.1 Building analysis groups

Suppose a pointer p is involved in an “interesting” array access instruction acc_p in a loop. The *analysis group* associated with acc_p , $AG(acc_p)$, collects all instructions that influence the value of the dereferenced pointer during the execution of the loop.

To determine $AG(acc_p)$, we compute an intraprocedural data flow graph representing operations in the loop that compute the value of p dereferenced in acc_p . Then, we check if the graph contains cycles. A cycle indicates that the value of p in a previous loop iteration affects its value in the current one, so p depends on the loop induction variable.

As mentioned before, this part of our work is built on top of the LLVM [23] compiler infrastructure. The static single assignment (SSA) form provided by LLVM translates directly to data flow graphs. Figure 3 shows an example. Observe that, since all dereferences of pointer u share their data flow graph, they also form a single analysis group. Thus, when *Dowser* later tries to find an illegal array access within this analysis group, it tests all the dereferences at the same time—there is no need to consider them separately.

3.2 Conditions guarding analysis groups

It may happen that the data flow associated with an array pointer is simple, but the value of the pointer is hard to follow due to some complex control changes. For this reason, *Dowser* ranks also control flows: the conditions that influence an analysis group.

Say that an instruction manipulating the array pointer p is guarded by a condition on a variable var , e.g., `if (var < 10) { *p++ = 0; }`. If the value of var is diffi-

cult to keep track of, so is the value of p . To assess the complexity of var , *Dowser* analyzes its data flow, and determines the analysis group, $AG(var)$ (as discussed in Section 3.1). Moreover, we recursively analyze the analysis groups of other variables influencing var and p inside the loop. Thus, we obtain a number of analysis groups which we rank in the next step (Section 3.3).

3.3 Scoring array accesses

For each array access realized in a loop, *Dowser* assesses the complexity of the analysis groups constructed in Sections 3.1 and 3.2. For each analysis group, it considers all instructions, and assigns them points. The more points an AG cumulatively scores, the more complex it is. The overall rank of the array access is determined by the maximum of the scores. Intuitively, it reflects the most complex component.

The scoring algorithm should provide roughly the same results for semantically identical code. For this reason, we enforce the optimizations present in the LLVM compiler (e.g., to eliminate common subexpressions). This way, we minimize the differences in (the amount of) instructions arising from the compiler options. Moreover, we analyzed the LLVM code generation strategies, and defined a powerful set of *equivalence rules*, which minimize the variation in the scores assigned to syntactically different but semantically equivalent code. We highlight them below.

Table 1 introduces all types of instructions, and discusses their impact on the final score. In principle, all common instructions involved in array index calculations are of the order of 10 points, except for the two instructions that we consider risky: pointer casts and functions that return non-pointer values used in pointer calculation.

The absolute penalty for each type of instruction is not very important. However, we ensure that the points reflect the difference in complexity between various code fragments, instead of giving all array accesses the same score. That is, instructions that complicate the array index contribute to the score, and instructions that complicate the index a lot also score very high, relative to other instructions. In Section 6, we compare our complexity ranking to alternatives.

4 Using tainting to find inputs that matter

Once *Dowser* has ranked array accesses in loops in order of complexity, we examine them in turn. Typically, only a small segment of the input affects the execution of a particular analysis group, so we want to search for a bug by modifying solely this part of the input, while keeping the rest constant (refer to Section 5). In the current section, we explain how *Dowser* identifies the link

Instructions	Rationale/Equivalence rules	Points
Array index manipulations		
Basic index arithmetic instr., i.e., addition and subtraction	GetElemPtr, that increases or decreases a pointer by an index, scores the same. Thus, operations on pointers are equivalent to operations on offsets. An instruction scores 1 if it modifies a value which is not passed to the next loop iteration.	1 or 5
Other index arithmetic instr. e.g., division, shift, or xor	These instructions involve more complex pointer calculations than the standard add or sub. Thus, we penalize them more.	10
Different constant values	Multiple constants used to modify a pointer make its value hard to follow. It is easier to keep track of a pointer that always increases by the same value.	10 per value
Constants used to access fields of structures	We assume that compilers handle accesses to structures correctly. We only consider constants used to compute the index of an array, and not the address of a field.	0
Numerical values determined outside the loop	Though in the context of the loop they are just constants, the compiler cannot predict their values. Thus they are difficult to reason about and more error prone.	30
Non-inlined functions returning non-pointer values	Since decoupling the computation of a pointer from its use might easily lead to mistakes, we heavily penalize this operation.	500
Data movement instructions	Moving (scalar or pointer) data does not add to the complexity of computations.	0
Pointer manipulations		
Load a pointer calculated outside the loop	It denotes retrieving the base pointer of an object, or using memory allocators. We treat all <i>remote</i> pointers in the same way - all score 0.	0
GetElemPtr	An LLVM instruction that computes a pointer from a base and offset(s). (See add.)	1 or 5
Pointer cast operations	Since the casting instructions often indicate operations that are not equivalent to the standard pointer manipulations (listed above), they are worth a close inspection.	100

Table 1: Overview of the instructions involved in pointer arithmetic operations, and their penalty points.

between the components of the program input and the different analysis groups. Observe that this result also benefits other bug finding tools based on fuzzing, not just *Dowser* and concolic execution.

We focus our discussion on an analysis group $AG(acc_p)$ associated with an array pointer dereference acc_p . We assume that we can obtain a test input \mathcal{I} that exercises the potentially vulnerable analysis group. While this may not always be true, we believe it is a reasonable assumption. Most vendors have test suites to test their software and they often contain at least one input which exercises each *complex* loop.

4.1 Baseline: dynamic taint analysis

As a basic approach, *Dowser* performs dynamic taint analysis (DTA) [31] on the input \mathcal{I} (tainting each input byte with a unique color, and propagating the colors on data movement and arithmetic operations). Then, it logs all colors and input bytes involved in the instructions in $AG(acc_p)$. Given the format of the input, *Dowser* maps these bytes to individual fields. In Figure 1, *Dowser* finds out that it is sufficient to treat `uri` as symbolic.

The problem with DTA, as sketched above, is that it misses *implicit flows* (also called *control dependencies*) entirely [14, 21]. Such flows have no direct assignment of a tainted value to a variable—which would be propagated by DTA. Instead, the value of a variable is completely determined by the value of a tainted variable in a condition. In Figure 1, even though the value of `u` in

line 12 is dependent on the tainted character `ch` in line 11, the taint does not flow directly to `u`, so DTA would not report the dependency. Implicit flows are notoriously hard to track [36, 9], but ignoring them completely reduces our accuracy. *Dowser* therefore employs a solution that builds on the work by Bao et al. [6], but with a novel optimization to increase the accuracy of the analysis (Section 4.2).

Like Bao et al. [6], *Dowser* implements *strict control dependencies*. Intuitively, we propagate colors only on the most informative (or, information preserving) dependencies. Specifically, we require a direct comparison between a tainted variable and a compile time constant. For example, in Figure 1, we propagate the color of `ch` in line 11 to the variables `state` and `u` in line 12. However, we would keep `state` and `u` untainted if the condition in line 11 for instance had been either `"if(ch != '/')"` or `"if(ch < '/')"`. As implicit flows are not the focus of this paper we refer interested readers to [6] for details.

4.2 Field shifting to weed out false dependencies

Improving on the handling of strict control dependencies by Bao et al. [6], described above, *Dowser* adds a novel technique to prevent overtainting due to false dependencies. The problems arise when the order of fields in an input format is not fixed, e.g., as in HTTP, SMTP (and the commandline for most programs). The approach from [6] may falsely suggest that a field is dependent on all fields that were extracted so far.

Input:	Colors in handlers:					Excluded colors:
	A_hand()	B_hand()	C_hand()	D_hand()	E_hand()	
ABCDE	A	AB	ABC	ABCD	ABCDE	
DABCE	AD	ABD	ABCD	D	ABCDE	E
BDACE	ABD	B	ABCD	BD	ABCDE	CE
						ACE

Fig. 4: The figure shows how *Dowser* shuffles an input to determine which fields really influence an analysis group. Suppose a parser extracts fields of the input one by one, and the analysis group depends on the fields B and D (with colors B and D, respectively). *Colors in handlers* show on which fields the subsequent handlers are strictly dependent [6], and the shaded rectangle indicates the colors propagated to the analysis group. *Excluded colors* are left out of our analysis.

For instance, `lighttpd` reads new header fields in a loop and compares them to various options, roughly as follows:

```
while () {
    if(cmp(field, "Content") == 0)
        ***
    else if(cmp(field, "Range") == 0)
        ***
    else exit (-1);
    field = extract_new_header_field();
}
```

As the parser tests for equivalence, the implicit flow will propagate from one field to the next one, even if there is no real dependency at all! Eventually, the last field appears to depend on the whole header.

Dowser determines which options really matter for the instructions in an analysis group by *shifting* the fields whose order is not fixed. Refer to Figure 4, and suppose we have run the program with options A, B, C, D, and E, and our analysis group really depends on B and D. Once the message gets processed, we see that the AG does not depend on E, so E can be excluded from further analysis. Since the last observed color, D, has a direct influence on the AG, it is a true dependence. By performing a circular shift and re-trying with the order D, A, B, C, E, *Dowser* finds only the colors corresponding to A, B, D. Thus, we can leave C out of our analysis. After the next circular shift, *Dowser* reduces the colors to B and D only.

The optimization is based on two observations: (1) the last field propagated to the AG has a direct influence on the AG, so it needs to be kept, (2) all fields beyond this one are guaranteed to have no impact on the AG. By performing circular shifts, and running DTA on the updated input, *Dowser* drops the undue dependencies.

Even though this optimization requires some minimal knowledge of the input, we do not need full understanding of the input grammar, like the contents or effects of fields. It is sufficient to identify the fields whose order is not fixed. Fortunately, such information is available for many applications—especially when vendors test their own code.

5 Exploring candidate instructions

Once we have learnt which part of the program input influences the analysis group $AG(acc_p)$, we fuzz this part, and we try to nudge the program toward using the pointer p in an illegal way. More technically, we treat the interesting component of the input as symbolic, the remaining part as fixed (concrete), and we execute the loop associated with $AG(acc_p)$ symbolically.

However, since in principle the cost of a complete loop traversal is exponential, loops present one of the hardest problems for symbolic execution [19]. Therefore, when analyzing a loop, we try to select those paths that are most promising in our context. Specifically, *Dowser* prioritizes paths that show a potential for knotty pointer arithmetic. As we show in Section 6, our technique significantly optimizes the search for an overflow.

Dowser's loop exploration procedure has two main phases: learning, and bug finding. In the *learning phase*, *Dowser* assigns each branch in the loop a weight approximating the probability that a path following this direction contains new pointer dereferences. The weights are based on statistics on the variety of pointer values observed during an execution of a short symbolic input.

Next, in the *bug finding phase*, *Dowser* uses the weights determined in the first step to filter our uninteresting parts of the loop, and prioritize the important paths. Whenever the weight associated with a certain branch is 0, *Dowser* does not even try to explore it further. In the vulnerable `nginx` parsing loop from which Figure 1 shows an excerpt, only 19 out of 60 branches scored a non-zero value, so were considered for the execution. In this phase, the symbolic input represents a real world scenario, so it is relatively long. Therefore, it would be prohibitively expensive to be analyzed using a popular symbolic execution tool.

In Section 5.1, we briefly review the general concept of concolic execution, and then we discuss the two phases in Sections 5.2 and 5.3, respectively.

5.1 Baseline: concrete + symbolic execution

Like DART and SAGE [17, 18], *Dowser* generates new test inputs by combining concrete and symbolic execution. This technique is known as *concolic* execution [33]. It runs the program on a concrete input, while gathering symbolic constraints from conditional statements encountered along the way. To test alternative paths, it systematically negates the collected constraints, and checks whether the new set is satisfiable. If so, it yields a new input. To bootstrap the procedure, *Dowser* takes a test input which exercises the analysis group $AG(acc_p)$.

As mentioned already, a challenge in applying this approach is how to select the paths to explore first. The

classic solution is to use depth first exploration of the paths by backtracking [22]. However, since doing so results in an exponentially growing number of paths to be tested, the research community has proposed various heuristics to steer the execution toward unexplored regions. We discuss these techniques in Section 7.

5.2 Phase 1: learning

The aim of the learning phase is to rate the `true` and `false` directions of all conditional branches that depend on the symbolic input in the loop `L`. For each branch, we evaluate the likelihood that a particular outcome will lead to unique pointer dereferences (i.e., dereferences that we do not expect to find in the alternative outcome). Thus, we answer the question of how much we expect to gain when we follow this path, rather than the alternative. We encode this information into *weights*.

Specifically, the weights represent the likelihood of unique *access patterns*. An access pattern of the pointer `p` is the sequence of all values of `p` dereferenced during the execution of the loop. In Figure 1, when we denote the initial value of `u` by `u0`, then the input `"//.../"` triggers the following access pattern of the pointer `u`: (`u0`, `u0+1`, `u0+2`, `u0-2`, ...).

To compute the weights, we learn about the effects of individual branches. In principle, each of them may (a) directly affect the value of a pointer, (b) be a precondition for another important branch, or (c) be irrelevant from the computation's standpoint. To distinguish between these cases, *Dowser* analyzes all possible executions of a *short* symbolic input. By comparing the sets of `p`'s access patterns observed for both outcomes of a branch, it discovers which branches do not influence the diversity of pointer dereferences (i.e., are irrelevant).

Symbolic input In Section 4, we identified which part of the test input `I` we need to make symbolic. We denote this by `IS`. In the learning phase, *Dowser* executes the loop `L` exhaustively. For performance reasons, we therefore further limit the amount of symbolic data and make only a short fragment of `IS` symbolic. For instance, for Figure 1, the learning phase makes only the first 4 bytes of `uri` symbolic (not enough to trigger the bug), while scaling up to 50 symbolic bytes in the bug finding phase.

Algorithm *Dowser* exhaustively executes `L` on a short symbolic input, and records how the decisions taken at conditional branch statements influence pointer dereference instructions. For each branch `b` along the execution path, we retain the access pattern of `p` realized during this execution, `AP(p)`. We informally interpret it as "if you choose the `true` (respectively, `false`) direction of the branch `b`, expect access pattern `AP(p)` (respectively, `AP'(p)`)". This procedure results in two sets of access patterns for each branch statement, for the taken

and non-taken branch, respectively. The final weight of each direction is the fraction of the access patterns that were unique for the direction in question, i.e., were not observed when the opposite one was taken.

The above description explains the intuition behind the learning mechanism, but the full algorithm is more complicated. The problem is that a conditional branch `b` might be exercised multiple times in an execution path, and it is possible that all the instances of `b` influence the access pattern observed.

Intuitively, to allow for it, we do not associate access patterns with just a single decision taken on `b` (`true` or `false`). Rather, each time `b` is exercised, we also retain which directions were previously chosen for `b`. Thus, we still collect "expected" access patterns if the `true` (respectively, `false`) direction of `b` is followed, but we augment them with a precondition. This way, when we compare the `true` and `false` sets to determine the weights for `b`, we base the scores on a deeper understanding of how an access pattern was reached.

Discussion It is important for our algorithm to avoid false negatives: we should not incorrectly flag a branch as irrelevant—it would preclude it from being explored in the bug finding phase. Say that `instr` is an instruction that dereferences the pointer `p`. To learn that a branch directly influences `instr`, it suffices to execute it. Similarly, since branches retain full access patterns of `p`, the information about `instr` being executed is also "propagated" to all its preconditions. Thus, to completely avoid false negatives, the algorithm would require full coverage of the instructions in an analysis group. We stress that we need to exercise all instructions, and not all paths in a loop. As observed by [7], exhaustive executions of even short symbolic inputs provide excellent instruction coverage in practice.

While false positives are undesirable as well, they only cause *Dowser* to execute more paths in the second phase than absolutely necessary. Due to the limited path coverage, there are corner cases, when false positives can happen. Even so, in `nginx`, only 19 out of 60 branches scored a non-zero value, which let us execute the complex loop with a 50-byte-long symbolic input.

5.3 Phase 2: hunting bugs

In this step, *Dowser* executes symbolically a real-world sized input in the hope of finding a value that triggers a bug. *Dowser* uses the feedback from the learning phase (Section 5.2) to steer its symbolic execution toward new and interesting pointer dereferences. The goal of our heuristic is to avoid execution paths that do not bring any new pointer manipulation instructions. Thus, *Dowser* shifts the target of symbolic execution from traditional code coverage to *pointer value coverage*.

Dowser's strategy is explicitly dictated by the weights. As a baseline, the execution follows a depth-first exploration, and when *Dowser* is about to select the direction of a branch *b* that depends on the symbolic input, it adheres to the following rules:

- If both the *true* and *false* directions of *b* have weight 0, we do not expect *b* to influence the variety of access patterns. Thus, *Dowser* chooses the direction randomly, and does not intend to examine the other direction.
- If only one direction has a non-zero weight, we expect to observe unique access patterns only when the execution paths follows this direction, and *Dowser* favors it.
- If both of *b's* directions have non-zero weights, both the *true* and *false* options may bring unique access patterns. *Dowser* examines both directions, and schedules them in order of their weights.

Intuitively, *Dowser's* symbolic execution tries to select paths that are more likely to lead to overflows.

Guided fuzzing This concludes our description of *Dowser's* architecture. To summarize, *Dowser* helps fuzzing by: (1) finding “interesting” array accesses, (2) identifying the inputs that influence the accesses, and (3) fuzzing intelligently to cover the array. Moreover, the targeted selection procedure based on pointer value coverage and the small number of symbolic input values allow *Dowser* to find bugs quickly and scale to larger applications. In addition, the ranking of array accesses permits us to zoom in on more complicated array accesses.

6 Evaluation

In this section, we first zoom in on the running example of *nginx* from Figure 1 to evaluate individual components of the system in detail (Section 6.1). In Section 6.2, we consider seven real-world applications. Based on their vulnerabilities, we evaluate our dowsing mechanism. Finally, we present an overview of the attacks detected by *Dowser*.

Since *Dowser* uses a ‘spot-check’ rather than ‘code coverage’ approach to bug detection, it must analyze each complex analysis group separately, starting with the highest ranking one, followed by the second one, and so on. Each of them runs until it finds a bug or gets terminated. The question is when we should terminate a symbolic execution run. Since symbolic execution of a single loop is highly optimized in *Dowser*, we found each bug in less than 11 minutes, so we execute each symbolic run for a maximum of 15 minutes.

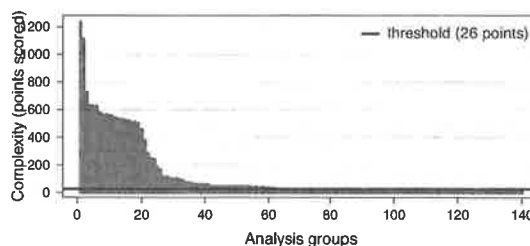


Fig. 5: Scores of the analysis groups in *nginx*.

Our test platform is a Linux 3.1 system with an Intel(R) Core(TM) i7 CPU clocked at 2.7GHz with 4096KB L2 cache. The system has 8GB of memory. For our experiments we used an OpenSUSE 12.1 install. We ran each test multiple times and present the median.

6.1 Case study: Nginx

In this section, we evaluate each of the main steps of our fuzzer by looking at our case study of *nginx* in detail.

6.1.1 Dowsing for candidate instructions

We measure how well *Dowser* highlights potentially faulty code and filters out the uninteresting fragments.

Our first question is whether we can filter out all the simple loops and focus on the more interesting ones. This turns out to be simple. Given the complexity scoring function from Section 3, we find that across all applications all analysis groups with a score less than 26 use just a single constant and at most two instructions modifying the offset of an array. Thus, in the remainder of our evaluation, we set our cut-off threshold to 26 points.

As shown in Table 2, *nginx* has 517 outermost loops, and only 140 analysis groups that access arrays. Thus, we throw out over 70% of the loops immediately³. Figure 5 presents the sorted weights of all the analysis groups in *nginx*. The distribution shows a quick drop after a few highly complex analysis groups. The long tail represents the numerous simple loops omnipresent in any code. 55.7% of the analysis groups score too low to be of interest. This means that *Dowser* needs to examine only the remaining 44.3%, i.e., 62 out of 140 analysis groups, or at most 12% of all loops. Out of these, the buffer overflow in Figure 1 ranks 4th.

6.1.2 Taint analysis in context of hunting for bugs

In Section 4 we mentioned that ‘traditional’ dynamic taint analysis misses implicit flows, i.e., flows that have

³In principle, if a loop accesses multiple arrays, it also contains multiple access groups. Thus, these 140 analysis groups are located in fewer than 140 loops.