Dowser: a guided fuzzer to find buffer overflow vulnerabilities

Istvan Haller[1], Asia Slowinska[1], Matthias Neugschwandtner[2], Herbert Bos[1]
[1]VU University Amsterdam, [2]Vienna University of Technology
i.haller@vu.nl, asia@few.vu.nl, mneug@iseclab.org, herbertb@few.vu.nl

ABSTRACT

Dowser is a 'guided' fuzzer that combines taint tracking, program analysis and symbolic execution to find buffer overflow vulnerabilities buried deep in the program's logic. Intuitively, a piece of code with convoluted pointer arithmetic instructions may be more prone to memory errors than straightforward array accesses. More importantly, the more complex the bugs and the more convoluted the pointer arithmetic, the harder it will be to find using existing techniques like random fuzzing, and static analysis.

Dowser ranks pointer dereference instructions according to their complexity, and then uses symbolic execution to zoom in on the most interesting operations. Zooming in on individual operations allows *Dowser* to severely reduce the search space necessary to cover the application. Instead of traditional code coverage, the symbolic execution phase employs a novel search algorithm which aims to maximize pointer coverage. We steer the execution along branches that show more potential to manipulate the value of a pointer. As a result, *Dowser* finds deep bugs in real programs. Moreover, it achieves it significantly faster than other tools.

1. INTRODUCTION

Buffer overflows are perennially in the top 3 most dangerous software errors [8] and recent studies suggest this will not change any time soon [24]. 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) [7, 2], or we track down the vulnerabilities before releasing the software (e.g., in the testing phase). Vendors prefer the latter option and typically try to find as many bugs as they can by means of fuzz testing. Blackbox fuzzers feed programs invalid or random data to see if they crash or exhibit unexpected behavior. This method is popular and fast, but misses many relevant code paths and thus many bugs. The effectiveness is poor and the results rarely extend beyond shallow bugs.

Whitebox fuzzing [4, 12] on the other hand, is more principled. By means of symbolic execution, it tries to find all execution paths through a binary. Unfortunately, symbolic execution is slow and hard to scale. Therefore, usually fuzzers strive for a high code coverage, which reflects the fraction of branches that were executed. In principle, this strategy does not aim for complex paths which are likely to be bug prone, but for single individual basic blocks.

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 key contribution is that our fuzzer directly zooms in on these buffer overflow candidates.

The first problem we address 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 some information about the target program to identify, and later focus on, instructions which are more likely to expose non-trivial buffer overflow bugs. It performs a static analysis of the program to rank pointer dereference instructions according to their complexity. The intuition is that a piece of code with convoluted pointer arithmetic instructions is more prone to memory errors than straightforward array accesses. After the ranking, *Dowser* probes the most interesting pointer dereferences. The key idea is that we prioritize bugs that are more complicated—typically, the vulnerabilities that static analysis or random fuzzing cannot find—and waste less time on shallow bugs that could be found using existing methods.

The second problem we explore is *how* to steer the execution of a program to the "interesting" locations identified by *Dowser*. As a baseline, we use a combination of concrete and symbolic execution (also known as *concolic* execution [26]). The concrete (fixed) input essentially starts off the symbolic execution. In *Dowser*, we introduce two optimizations. First, we propose a new search algorithm. Instead of aiming at the traditional code coverage, i.e., maximizing the fraction of individual branches executed [12], we focus on a thorough analysis, but only of selected code fragments, and we aim at pointer value coverage. When *Dowser* examines an "interesting" pointer dereference instruction, it steers the symbolic execution along branches that show most potential to alternate the value of the pointer. Second, we reduce the amount of symbolic input. *Dowser* performs dynamic taint analysis to figure out which parts of the input influence memory accesses in the target location. Later, it treats solely these fragments as symbolic.

As a result, *Dowser* is an entirely new fuzzer targeted at vendors who want to test their code for buffer overflows. We implemented the analyses of *Dowser* as LLVM [14] passes, while the symbolic execution step employs S2E [6].

Finally, *Dowser* is a *practical* solution. Rather than aiming for all possible security bugs, it specifically targets the class of buffer overflows. *Dowser* is more likely to find deep bugs in real programs. For example, it has no problems finding a bug in a complex

A buffer underrun vulnerability in `nginx`

```
[1] int ngx_http_parse_complex_uri(ngx_http_request_t *r)
[2] {
[3]    state = sw_usual;
[4]    u_char* p = r->uri_start;   // user input
[5]    u_char* u = r->uri.data; // store normalized uri here
[6]    u_char ch = *p++;           // the current character
[7]
[8]    while (p <= r->uri_end) {
[9]       switch (state) {
[10]          case sw_usual:
[11]             if (ch == '/')
[12]                state = sw_slash; *u++ = ch;
[13]             else if /* many more options here */
[14]             ch = *p++; break;
[15]          case sw_slash:
[16]             if (ch == '/')
[17]                *u++ = ch;
[18]             else if (ch == '.')
[19]                state = sw_dot; *u++ = ch;
[20]             else if /* many more options here */
[21]             ch = *p++; break;
[22]          case sw_dot:
[23]             if (ch == '.')
[24]                state = sw_dot_dot; *u++ = ch;
[25]             else if /* many more options here */
[26]             ch = *p++; break;
[27]          case sw_dot_dot:
[28]             if (ch == '/')
[29]                state = sw_slash; u -=4;
[30]                while (*(u-1) != '/') u--;
[31]             else if /* many more options here */
[32]             ch = *p++; break;
[33]       } }
[34] }
```

`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 `nginx_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 transforms 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 27). The routine now decreases `u` by 4 (line 29), 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 30), 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.

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

program like nginx [1], which is unachievable for existing tools.

2. 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 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.

Figure 1 presents only an excerpt from the original function, which in reality spans approx. 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 example, when we steered S2E all the way down to the vulnerable function, and made solely the five byte long uri path of the whole HTTP message symbolic, it took over 60 minutes to track down the problematic scenario. Without these hints, S2E did not identify the bug at all during an eight hour long execution.

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

This real world example shows the need for (1) focusing the powerful yet expensive symbolic execution on the toughest cases, (2) harnessing information about the execution to make informed branch choices, (3) minimizing the amount of symbolic data.

3. HIGH-LEVEL OVERVIEW

Rather than fuzzing randomly or exhaustively, *Dowser guides* the execution to interesting sets of instructions and tries to fuzz only those fields of the input that play a role in triggering the bug. Since buffer overflows occur when a pointer to a buffer also reads from or writes to memory beyond the buffer, *Dowser* tries to manipulate these pointers by means of the program inputs to see if it can trigger an overflow. Figure 2 illustrates the overall *Dowser* architecture.

First, *Dowser* performs a data flow analysis of the target program, and ranks all instructions that access buffers in loops ①. Calculations that are more error prone rank higher. In Figure 1, `u` is involved in three different operations, i.e., `u++`, `u-`, and `u-=4`, in multiple instructions inside a loop. These intricate computations place the dereferences of `u` in the top 2% most complex pointer accesses across `nginx` (Section 6).

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 format 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, it is sufficient to treat the uri path as symbolic.

Next ③, for each candidate instruction and the input bytes involved in calculating the array pointer, *Dowser* uses symbolic execution to try and nudge the program toward overflowing the buffer. Specifically, we execute symbolically the loop that contains the candidate instructions treating only the relevant bytes as symbolic.

Symbolic execution of loops is very costly, as the number of states grows exponentially, so finding the *interesting* paths to explore (and ignoring the rest) is essential. For this reason, *Dowser* analyses the loop *a priori* to find the branch outcomes that are most likely to lead to new arithmetic on the relevant pointers, and uses the outcome of this analysis to select paths in the actual symbolic execution. Thus, unlike traditional symbolic execution, *Dowser* aims not so much for code coverage, but rather for *pointer value coverage*. In the running example, it would prioritize all `true` outcomes for the branches in lines 11, 23, and 28, and it would prefer a jump to `sw_dot_dot` over a jump to any of the other states.

Finally, we detect any overflow that may occur. Just like whitebox fuzzers, we can use any technique to do so (e.g., Purify, Valgrind [17], or BinArmor [23]). In our work, we use Google's AddressSanitizer [21] ④. It detects the underflow when the `u` pointer reads memory outside its buffer boundaries (line 33).

We explain the static analysis phase ① in Section 4, and we present our approach to guiding the execution to the *interesting* instructions, step ③, in Section 5.

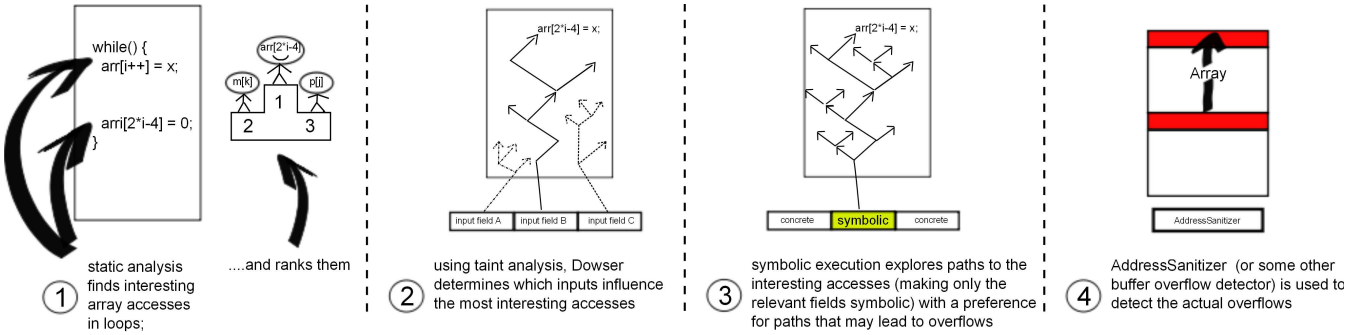

Figure 2: ***Dowser*– high-level overview.**

4. DOWSING FOR CANDIDATES

Previous research has shown that software complexity metrics collected from software artifacts are helpful in finding vulnerable code components [10, 27, 22, 19]. 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 [27], there is a need for metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns.

To boost the chance of detecting overflows, *Dowser* introduces a new way to select bug prone code fragments. It leverages the primary pragmatic reason behind overflows, i.e., convoluted pointer computations are hard to follow by a programmer. *Dowser* focuses on 'complex' array accesses realized inside loops. Further, it limits the analysis to pointers that evolve together with loop induction variables, i.e., are repeatedly updated to access array elements.

Dowser ranks buffer accesses by evaluating the complexity of the array index (pointer) calculations. For each loop in the program, it first statically determines the set of all instructions involved in modifying an array pointer (we will call this a pointer's *analysis group*), and then labels each such set with a score reflecting its complexity. We explain these two steps in Sections 4.1 and 4.2.

4.1 Building analysis groups

Say that a pointer `p` is involved in an "interesting" memory access instruction $\texttt{acc}_\texttt{p}$, which reads or writes elements of an array in a loop. The *analysis group* associated with $\texttt{acc}_\texttt{p}$, $\texttt{AG}(\texttt{acc}_\texttt{p})$, collects all instructions that influence the value of the dereferenced pointer during the execution of the loop.

To determine $\texttt{AG}(\texttt{acc}_\texttt{p})$, we compute an intraprocedural data flow graph representing operations in the loop that compute the value of `p` dereferenced in $\texttt{acc}_\texttt{p}$. 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. Refer to Figure 3 for an example.

4.2 Scoring analysis groups

The crux of *Dowser* comes in evaluating the complexity of array accesses. To rate a pointer dereference instruction, the scoring algorithm examines its analysis group, and assigns each component penalty points. The more points an analysis group scores, the more complex it is. Table 1 introduces all types of instructions present in analysis groups, and their impact on the final score.

Observe that the algorithm should provide roughly the same results for semantically identical code. To do so, we implement two mechanisms. First, we enforce the optimizations present in the LLVM compiler (e.g., to eliminate common subexpressions). This

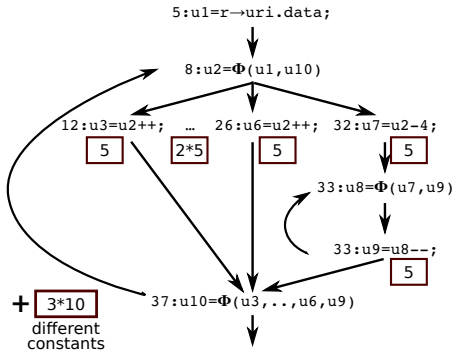

Figure 3: Data flow graph (the SSA form) and analysis group associated with the pointer `u` from Figure 1. For the sake of clarity, the figure presents pointer arithmetic instructions in a pseudo code. In reality, LLVM compiles them to an equivalent `GetElemPtr` instruction. For example, `u7=u2-4` gets encoded as `u7=GetElemPtr(u2,-4)`. The numbers in the boxes represent points assigned by *Dowser*.

way, we minimize the differences in (the amount of) instructions arisen from the compiler options. Second, we extensively analyzed the LLVM code generation strategies, and devised a number of *equivalence rules*, which minimize the variation in the scores assigned to syntactically different but semantically equivalent code. We highlight them in Table 1.

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. As we show in Section 6, our scheme places the complex parser from Figure 1 in the top 2% most complex operations across the whole `nginx` (5th position).

5. EXPLORING CANDIDATES

Once *Dowser* has ranked array accesses in loops in order of complexity, we examine the analysis groups in turn. For each of them, we try to find a program input that proves it vulnerable to a memory corruption attack. We assume that we can obtain a test input `I` that exercises the potentially vulnerable analysis group.

First, we perform dynamic taint analysis (DTA) [18] on the input `I` to learn which part of it influences $\texttt{AG}(\texttt{acc}_\texttt{p})$. We taint each input byte with a unique color, propagate the colors on data movement and arithmetic operations, and we log all colors and input bytes involved in $\texttt{AG}(\texttt{acc}_\texttt{p})$. Given the format of the input, *Dowser* maps

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 that 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 indicates retrieving the base pointer of an object, or using memory allocators. We treat all *remote* 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.

these bytes to individual fields. In Figure 3, it is the `uri` field.

Once we know which fields of the input influence $\texttt{AG}(\texttt{acc}_\texttt{p})$, 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 $\texttt{AG}(\texttt{acc}_\texttt{p})$ symbolically. Since in principle the cost of a complete loop traversal is exponential, loops present one of the hardest problems for symbolic execution [13]. 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. We discuss them in turn.

5.1 Baseline: concrete + symbolic execution

Like DART and SAGE [11, 12], *Dowser* generates new test inputs by combining concrete and symbolic execution. This technique is also known as *concolic* execution [20]. It runs the program on a concrete input, while gathering symbolic constraints from conditional statements encountered along the way. In order 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. As we have mentioned already, a challenge is how to select the paths to explore first. To bootstrap the procedure, *Dowser* takes an input which exercises the analysis group $\texttt{AG}(\texttt{acc}_\texttt{p})$.

5.2 Phase 1: learning

In the *learning phase*, we rate the `true` and `false` directions of all conditional branches that depend on the symbolic input in the loop `L`. The idea is to evaluate the chances that the execution path contains unique pointer dereferences, i.e., not expected in the other direction. Thus, this indicates the expected gain when we follow this path, rather than the alternative one.

We encode this information into weights that 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 $\texttt{u}_0$, then the input `"//../"` triggers the following access pattern of the pointer `u`: $(\texttt{u}_0,\ \texttt{u}_0\texttt{+1},\ \texttt{u}_0 + 2,\ \texttt{u}_0\texttt{-2,...})$.

To compute the weights, we need to learn about the effects of individual outcomes. In principle, each branch may (a) directly affect the value of a pointer, or (b) be a precondition for another important branch statement, or (c) be irrelevant from the computation's standpoint. Since every execution path contains a combination of these branches, it is necessary for *Dowser* to exercise all possibilities for a given symbolic input. Only a complete picture allows us to compare the sets of possible access patterns when both directions of a branch are taken, and detect the irrelevant ones.

Dowser exhaustively executes `L` on a short symbolic input, and it records how the decisions taken at conditional branch statements influence pointer dereferences. 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. Intuitively, the final weight of each direction is the fraction of the access patterns that were unique, i.e., were not observed when the opposite one was taken.

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 to steer its symbolic execution toward the maximum range of pointer values. 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 ob-

serve 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 weights.

This concludes our description of *Dowser*'s architecture. In the next section, we will evaluate our system.

6. EVALUATION

In this section, we zoom in on the vulnerability in `nginx` from Figure 1 to evaluate individual components of the system in detail. An extensive evaluation of the system is still in progress.

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. Since symbolic execution of a single loop is highly optimized in *Dowser*, our preliminary results suggest that it is enough to execute each run for a maximum of a few minutes.

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.

6.1 Dowsing efficiency

To evaluate our dowsing scheme, 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 very simple. Given the scoring function from Section 4, 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.

`Nginx` has 517 outermost loops, and only 277 analysis groups that access arrays. Thus, we throw out almost 50% of the loops immediately[1]. Figure 4 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. 69.7% of the analysis groups score too low to be of interest. This means that *Dowser* needs to examine only the remaining 30.3%, i.e., 84 out of 277 analysis groups, or at most 17% of all loops.. Out of these, the buffer overflow in Figure 1 ranks 13th.

6.2 Importance of guiding symbolic execution

We now use the `nginx` example to assess the importance of guiding symbolic execution to a vulnerability condition.

For `nginx`, the input message is a generic HTTP request. Since it exercises the vulnerable loop for this analysis group, its URI starts with "//". Taint analysis allows us to detect that only the URI field is important, so we mark only this field as symbolic. As we shall see, without guidance, symbolic execution does not scale beyond very short URI fields (5-6 byte long). In contrast, *Dowser* successfully executes 50-byte-long symbolic URIs.

When S2E [6] executes a loop, it can follow one of the two search strategies: depth-first search, or maximizing code coverage (as proposed in SAGE [12]). The first one aims at complete path coverage, and the second at executing basic blocks that were not seen before. However, none can be applied in practice to examine the complex loop in `nginx`. The search is so costly that we measured the runtime for only 5-6 byte long symbolic URI fields. The DFS strategy handled the 5-byte-long input in 139 seconds, the 6-byte-long in 824 seconds. A 7-byte input requires more than 1 hour to finish. Likewise, the code coverage strategy required 159, and 882 seconds, respectively. These results show that, in practice classic symbolic execution is limited to very short inputs. Observe also that the code coverage heuristic is not a great fit for the search for buffer overflows. Indeed, besides executing specific instructions from the loop, memory corruptions require a very particular execution context. Even if 100% code coverage is reached, buffer overflows may stay undetected.

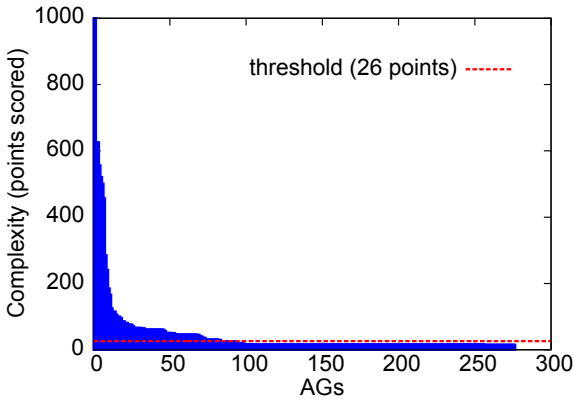

Figure 4: Scores of the analysis groups in `nginx`.

As we explained in Section 5, the strategy employed by *Dowser* does not aim at full coverage. Instead, it actively searches for paths which involve new pointer dereferences. The learning phase uses a 4-byte-long symbolic input to observe access patterns in the loop. It follows a simple depth first search strategy. As the bug clearly cannot be triggered with this input size, the search continues in the second, hunting bugs, phase. The result of the learning phase disables 66% of the conditional branches significantly reducing the exponentially of the subsequent symbolic execution. Because of this heuristic, *Dowser* easily scales up to 50 symbolic bytes and finds the bug after just a few minutes. A 5-byte-long symbolic input is handled in 20 seconds, 10 bytes in 42 seconds, 20 bytes in 63 seconds, 30 in 146 seconds, 40 in 174 seconds, and 50 in 253 seconds. These numbers maintain an exponential growth of 1.1 for each added character. Even though *Dowser* still exhibits the exponential behavior, the growth rate is fairly low. Even in the presence of 50 symbolic bytes, it quickly finds the complex bug.

In practice, symbolic execution has problems dealing with real world applications and input sizes. The number of execution paths quickly overwhelms these systems. Since triggering buffer overflows not only requires a vulnerable basic block, but also a special context, traditional symbolic execution tools are ill suited. *Dowser*, instead, requires the application to be executed symbolically for only a very short input, and then it deals with real-world input sizes instead of being limited to a few input bytes. Combined with the ability to extract the relevant parts of the original input, this enables searching for bugs in applications like web servers where input sizes were considered until now to be well beyond the scalability of symbolic execution tools.

[1]In principle, if a loop accesses multiple arrays, it also contains multiple access groups. Thus, these 277 analysis groups are located in fewer than 277 loops.

7. RELATED WORK

Software complexity metrics. Many studies have shown that software complexity metrics are positively correlated with defect density and vulnerabilities [16, 22, 10, 27, 22, 19]. All these approaches consider a generic set of measurements, e.g., the number of basic blocks in a function's control flow graph. the number of global or local variables read or written, the maximum nesting level of `if` or `while` statements and so on. *Dowser* is very different in this respect, and to the best of our knowledge, the first of its kind. We focus on a narrow group of security vulnerabilities, i.e., buffer overflows, so our scoring function is tailored to reflect the complexity of pointer manipulation instructions.

Application of DTA to fuzzing. BuzzFuzz [9] uses DTA to locate regions of seed input files that influence values used at library calls. They specifically select library calls, as they are often developed by different people than the author of the calling program and often lack a perfect description of the API. Buzzfuzz does not use symbolic execution at all, but uses DTA only to ensure that they preserve the right input format. TaintScope [25] is similar in that it also uses DTA to select fields of the input seed which influence security-sensitive points (e.g., system/library calls). In addition, TaintScope is capable of identifying and bypassing checksum checks. Unlike BuzzFuzz, TaintScope operates at the binary level, rather than the source.

Symbolic-execution-based fuzzing. Recently, there has been much interest in whitebox fuzzing, symbolic execution, concolic execution, and constraint solving. Examples include EXE [5], KLEE [4], CUTE [20], DART [11], SAGE [12]. All of these systems substitute (some of the) program inputs with symbolic values, gather input constraints on a program trace, and generate new input that exercise different paths in the program. They are very powerful, and can analyze programs in detail, but in general they don't scale. The problem is that the number of paths grows very quickly.

Marinescu et al. [15] take a different approach, and execute existing regression tests symbolically. Intuitively, they check if by slightly modifying the input, they can trigger a vulnerable condition. While this technique scales better, the paths exercised are limited to the neighborhood of existing test suites.

Finally, Babić et al. [3] guide symbolic execution to potentially vulnerable program points detected with static analysis. However, the interprocedural context- and flow-sensitive static analysis proposed does not scale well to real world programs and the experimental results contain only short traces.

8. CONCLUSION

Dowser is a guided fuzzer that combines static analysis, dynamic taint analysis, and symbolic execution to find buffer overflow vulnerabilities deep in a program's logic. Each of its steps contain novel contributions in and of themselves (e.g., the ranking of array accesses, and the symbolic execution based on pointer value coverage), but the overall contribution is a new, practical and complete fuzzing approach that scales to real applications and complex bugs that would be hard or impossible to find with existing techniques.

9. REFERENCES

[1] CVE-2009-2629: Buffer underflow vulnerability in nginx. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629, 2009.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, S&P'08, 2008.

[3] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, 2011.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th Symposium on Operating Systems Design and Implementation*, OSDI'08, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[6] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in vivo multi-path analysis of software systems. In *Proc. of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, 2011.

[7] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, SSYM'98, 1998.

[8] CWE/SANS. CWE/SANS TOP 25 Most Dangerous Software Errors. `www.sans.org/top25-software-errors`, 2011.

[9] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE'09, 2009.

[10] M. Gegick, L. Williams, J. Osborne, and M. Vouk. Prioritizing software security fortification through code-level metrics. In *Proc. of the 4th ACM workshop on Quality of protection*, QoP'08. ACM Press, Oct. 2008.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. of the 2005 Conf. on Programming language design and implementation*, PLDI'05, 2005.

[12] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium*, NDSS'08, 2008.

[13] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, 2011.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization*, CGO'04, 2004.

[15] P. D. Marinescu and C. Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *Proc. of the 2012 International Conference on Software Engineering*, ICSE'12, pages 716–726, June 2012.

[16] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. of the 28th international conference on Software engineering*, ICSE'06, 2006.

[17] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the 3rd Intern. ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE'07, 2007.

[18] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proc. of the Network and Distributed Systems Security Symposium*, NDSS'05, 2005.

[19] V. H. Nguyen and L. M. S. Tran. Predicting vulnerable software components with dependency graphs. In *Proc. of the 6th International Workshop on Security Measurements and Metrics*, MetriSec'10. ACM Press, Sept. 2010.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of the 10th European software engineering conference*, ESEC/FSE-13, 2005.

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of USENIX Annual Technical Conference*, 2012.

[22] Y. Shin and L. Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proc. of the 7th Intern. Workshop on Software Engineering for Secure Systems*, SESS'11, 2011.

[23] A. Slowinska, T. Stancescu, and H. Bos. Body Armor for Binaries: preventing buffer overflows without recompilation. In *Proc. of USENIX Annual Technical Conference*, 2012.

[24] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Proc. of The 15th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID'12, 2012.

[25] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, SP'10, 2010.

[26] N. Williams, B. Marre, and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE'04, 2004.

[27] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proc. of the 3rd International Conference on Software Testing, Verification and Validation*, ICST'10, Apr. 2010.