Fitness Guided Vulnerability Detection with Greybox Fuzzing

Raveendra Kumar Medicherla
raveendra.kumar@tcs.com
Tata Consultancy Services
Bangalore, India

Raghavan Komondoor
raghavan@iisc.ac.in
Indian Institute of Science
Bangalore, India

Abhik Roychoudhury
abhik@comp.nus.edu.sg
National University of Singapore
Singapore

ABSTRACT

Greybox fuzzing is an automated test-input generation technique that aims to uncover program errors by searching for bug-inducing inputs using a fitness-guided search process. Existing fuzzing approaches are primarily coverage-based. That is, they regard a test input that covers a new region of code as being fit to be retained. However, a vulnerability at a program location may not get exhibited in every execution that happens to visit to this program location; only certain program executions that lead to the location may expose the vulnerability. In this paper, we introduce a unified fitness metric called *headroom*, which can be used within greybox fuzzers, and which is explicitly oriented towards searching for test inputs that come closer to exposing vulnerabilities.

We have implemented our approach by enhancing AFL, which is a production quality fuzzing tool. We have instantiated our approach to detecting buffer overrun as well as integer-overflow vulnerabilities. We have evaluated our approach on a suite of benchmark programs, and compared it with AFL, as well as a recent extension over AFL called AFLGo. Our approach could uncover more number of vulnerabilities in a given amount of fuzzing time and also uncover the vulnerabilities faster than these two tools.

CCS CONCEPTS

• **Security and privacy** → *Penetration testing*; • **Software and its engineering** → **Search-based software engineering**; **Empirical software validation**.

ACM Reference Format:
Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. 2020. Fitness Guided Vulnerability Detection with Greybox Fuzzing. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3391457

1 INTRODUCTION

Software vulnerabilities are bugs in code that can lead to unexpected behaviors of the software, which can potentially be exploited by the malicious attackers to gain control over the software during its execution. Automatically identifying test inputs that uncover such vulnerabilities in an existing software can help developers make software secure.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3391457

Search based software testing (SBST) [11, 18, 26] is a class of techniques that can be used to uncover run-time errors in programs by executing the target program with a large number of test inputs generated automatically. These techniques rely on a *fitness metric* for test inputs to guide the search towards test inputs that meet a *testing objective*, such as exposing an error. A test input with a higher fitness metric than other test inputs is closer to meeting the testing objective, and is hence used to generate newer inputs. In this way, a test input that reaches the test objective is likely to get generated eventually.

Greybox fuzzing is a specific kind of SBST approach that is employed in several practical tools and approaches [4, 9, 21, 27]. Here, the idea is to use lightweight instrumentation to collect a *profile* from a run of the program on each test input, and to use this profile to compute the fitness metric of the corresponding test input. Many of these approaches have the objective of finding test inputs as quickly as possible that cause execution to reach as many parts of the program as possible. Such approaches are known as *coverage-based* fuzzers. Another class of greybox fuzzing tools, known as *directed fuzzers* [4], use a fitness metric that gives preference to inputs that bring the program execution closer to a specified vulnerability location.

However, a vulnerability at a program point may not get revealed in every visit to the program point. Only certain executions that reach the vulnerability point may exhibit the vulnerable behavior. For instance, a buffer overrun vulnerability at a buffer access location will get exhibited only in test runs in which the buffer access pointer points outside the buffer when the location is reached. As another example, an integer overflow vulnerability at a program location will get exhibited only in test runs in which the variable being incremented at the location has a high enough value when the location is reached. A test input that causes a vulnerability location to be reached may need to be fuzzed further to produce an input that not only reaches the vulnerability location but also exposes the vulnerability. However, coverage-based fuzzers and directed fuzzers would consider the objective as being met when a test run reaches the vulnerability location, and would not fuzz the corresponding test input further.

This motivates our key hypothesis, which is that one needs *vulnerability specific* fitness metrics, which guide the search towards test inputs that not only reach the vulnerability locations, but reach them along execution paths that result in program states that cause the vulnerability to be exposed. We develop this intuition in the rest of this section.

1.1 Motivating example

The example program shown in Figure 1 serves as a motivating and running example. This program first reads its input into a buffer `inp` (Statement 2). It copies all occurrences of the character 'b' in

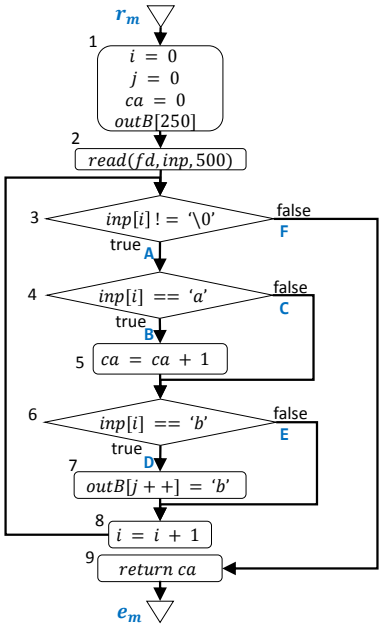

Figure 1: Example program with buffer overflow vulnerability at Statement 7

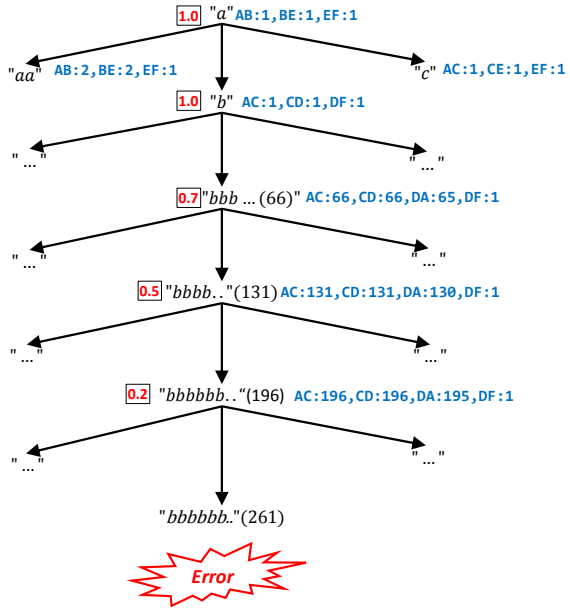

Figure 2: Generation of test inputs during fuzzing

`inp` into a another buffer `outB` (Statement 7). The branches in this program are labelled with letters **A** to **F**.

Note that this program is *vulnerable* to buffer overflow errors at Statement 7. Since the size of buffer `outB` is 250, if the input contains more than 250 occurrences of character 'b', then `outB` will overflow at this statement.

AFL [27] is a well-known coverage-based greybox fuzzing tool, which we refer to here in order to discuss why coverage-based fuzzing may not suffice to find vulnerabilities. A *branch pair* is a pair of consecutively executable branches in the program; e.g., in Figure 1, AB, AC, BD, BE, CE, and EF, are examples of branch pairs. A *branch pair profile* is a profile that records the number of visits to each branch pair in an execution. AFL considers a newly generated test input t to be *fit* and *retains* it for further fuzzing iff for at least one branch pair bp, the visit count of bp by the execution on t is *significantly different* from the visit count of the same branch pair by each execution on each other retained test input so far. Roughly speaking, two counts are significantly different if their logarithms to the base 2 are different. AFL uses the higher bar of significant difference as retaining too many test inputs dramatically slows down the efficiency and effectiveness of the search process.

Figure 2 shows (partially) a candidate tree of generated test inputs that may be generated by AFL for our running example. Due to randomization in the mutation operations that are used, the same tree may not be produced in every run of AFL. The root node is the given seed input "a". The edges represent the parent-child relationships between test inputs. The branch pair profile of each test input is shown in blue color to the right of the input. We may focus in particular on the path in the tree that runs vertically down the middle. The number that is indicated within round parentheses for each test input in this path depicts the number of 'b's in the input (there are no letters other than 'b' in the inputs in this path). We can ignore the red colored numbers within boxes for the moment, as they do not pertain to AFL. The first four inputs in this path will be retained, but the fifth input (with 196 'b's) will actually not be retained as its branch-pair profile is not significantly different from the profile of its parent (i.e., 131 'b's). Therefore, the sixth input (i.e., 261 'b's) would not actually be generated from the fifth input. Note that the sixth input would have been the first one in this path to cause a buffer overflow at Statement 7.

In other words, AFL would need to generate an input containing more than 250 'b's directly by mutating the fourth test input (i.e., 131 'b's). This can happen only via a mutation operation that inserts 129 or more 'b's in a single operation. The randomization in AFL is setup in a way that the probability of inserting a larger number of characters in a single mutation operation is directly proportional to the number of mutant test inputs generated by AFL so far. Therefore, a lot of time could elapse before such "large" mutations are tried.

A naive suggestion here would be to increase the probability of inserting larger number of bytes in mutation operations early in the run of AFL. However, this is a blunt idea, which would produce large input files early on, hence increasing the running time of each test input and hence slowing down the number of mutants that are generated by per unit time. Many of these large mutations may not useful also, e.g., ones that insert more number of 'a's than 'b's. Hence, the likelihood of finding vulnerabilities early may not be achieved.

Even directed fuzzing [4] is not necessarily effective in addressing this issue. In our example, a directed fuzzer would consider its objective as having been met the first time Statement 7 is reached. This would happen with the simple input "b" itself, which obviously does not expose the vulnerability.

1.2 Our test generation strategy

To address the issues mentioned above, we introduce a novel idea of using a vulnerability-oriented fitness metric, which we call *headroom*. A headroom is a number between zero and one, which indicates how *close* an execution using a test input came to exposing a potential vulnerability at a given vulnerability location. The value zero indicates that the vulnerability was actually exposed at the location, while the value one indicates that run did not come close at all to exposing the vulnerability. How to calculate the headroom depends on the type of vulnerability being targeted, which we will discuss subsequently in this paper. The remainder of our proposal, in a nutshell, is to retain a newly generated test input not only if its coverage is significant different from the coverage attained by previously generated test inputs, but also if achieves *significantly lower* headroom at any of the given potential vulnerability locations in the program than other test inputs retained so far.

Figure 2 also illustrates the functioning of our proposed approach on our running example. When targeting buffer overrun vulnerabilities, we define the headroom of a run at a buffer access location (such as Statement 7) as the minimum distance between the location pointed to by the buffer access pointer and the end of the buffer across all visits to this location in the run, divided by the size of the buffer. The red numbers within boxes indicate the headroom achieved by each test input for the buffer access at Statement 7. With our approach, the fifth input in the central path in the tree (i.e., the input with 196 'b's) will be retained, because its headroom of 0.2 is significantly lower than the headrooms achieved by other test inputs generated so far (we will discuss the exact definition of significantly lower headroom in Section 2). Now, the test input with 261 'b's, which overflows the buffer, is likely to be produced sooner than if the fifth input was not retained, because a mutation operation that inserts 65 'b's (65 is 261-196) is much more likely to be tried than one that inserts 130 'b's (which would be required to directly generated the sixth input from the fourth input).

To summarize, our key idea is to retain and then further fuzz test inputs that may not be achieving new coverage, but that are effectively getting closer to exposing complex vulnerabilities, with closeness defined in a vulnerability-specific manner. This obviates the need to wait for undirected, low-probability events of generating large mutations that happen to find the same vulnerabilities.

In the context of grey-box fuzzing, our work is the first to the best of our knowledge that proposes the use of vulnerability-specific fitness metrics to detect complex vulnerabilities. The closest related approaches are optimization formulations in SBST to expose exception conditions [12, 25], and extending fuzzers with domain-specific objectives [20].

Our approach being a generic one, not aimed at a single kind of vulnerability, we describe instantiations of our approach to two different kinds of vulnerabilities, namely, buffer overruns and integer overflows. We describe an implementation of our approach, which is an extension over the popular greybox fuzzing tool AFL. We evaluate our approach using standard benchmarks – namely, the MIT Benchmarks [28], and the SV-COMP benchmarks. Our evaluation reveals that our approach finds 1.8 times more buffer overrun vulnerabilities than two baseline tools, namely, AFL [27] and AFLGo [4], in a given time budget. Our approach finds integer overflow vulnerabilities many times faster than the two baseline tools.

The rest of this paper is organized as follows. Section 2 describes our approach. Section 3 discusses how to instantiate the headroom notion to a few different types of vulnerabilities. Section 4 discusses our implementation, while Section 5 summarizes the evaluation of our approach. Section 6 discusses related work, while Section 7 concludes the paper.

2 VULNERABILITY DETECTION APPROACH

In this section we present our approach in detail. Before we present our approach, we introduce the terminology that is central to our approach.

2.1 Terminology

A *vulnerability* is a bug at a *vulnerability location* v_l in a program P such that along certain executions paths that reach v_l an error occurs, such that the error leads to unexpected behaviors that may cause malicious attackers to exploit the execution for their purposes. Vulnerabilities can be of different types, e.g., buffer overruns, integer overflows, use after free, etc. Let V_L be the set of all such vulnerability locations in a given program P. (Our approach allows for different vulnerability locations to exhibit different types of vulnerabilities.) A *headroom* $h \in [0, 1]$ is a measure of how close a test run came to exposing a vulnerability at a vulnerability location v_l. A lower value of h represents more *closeness* to exposing the vulnerability. The value $h = 0$ means that in the run the vulnerability got manifested at v_l. The *vulnerability profile* $I_h : V_L \rightarrow [0, 1]$ of a test run is a map from all vulnerability locations V_L to their corresponding head rooms as attained in the run.

In order to determine the fitness of a test input t, the vulnerability profile I_h from the test run on t is compared against the vulnerability profiles of the test runs on previously generated inputs so far. *Least headrooms* $I_L : V_L \rightarrow [0, 1]$ is a global map that contains the least headroom at each vulnerability location witnessed across all test runs retained so far. We define an operation isCloser that takes a vulnerability profile I_h from a run as argument, compares it with I_L and returns a boolean answer, and also updates I_L.

```
bool isCloser(I_h) {
    ret = ∃v_l ∈ V_L . I_h(v_l) ≺ I_L(v_l)
    I_L = min(I_L, I_h)
    return ret
}
```

The operation $\prec$ is a comparison in the log_2 scale; i.e., if its second argument is above 0.5 it returns true iff the first argument is below 0.5; if the second argument is between 0.25 and 0.5, it returns true iff the first argument is below 0.25, and so on.

2.2 Our approach

Our approach is a *co-evolutionary* computation model [14], where two (or more) populations of test inputs evolve simultaneously in a co-operative manner using their own fitness functions. We retain two populations of generated test inputs. *Population 1:* Test inputs that reach new regions of code compared to previous test inputs. This notion is inherited from coverage-guided fuzzing, and is meant to ensure that runs that progressively reach points closer

Algorithm 1 Test generation for vulnerability detection

Require: Program P, and a initial input s.
Ensure: A tree of test inputs T_G for P.

```
1:  Create an empty tree T_G of test inputs.
2:  (e_s, h_s) = Run(P,s.data)
3:  t_r.data = s                                  ▷ t_r is a new tree node
4:  t_r.fitness = computeFitness(e_s)
5:  t_r.reasonToRetain = "default"
6:  T_G.setRoot(t_r).
7:  I_L = λv_l ∈ V_L . 1
8:  repeat
9:      Let t = selectNextH(T_G)
10:     Let N = getFuzzPotentialH(t)
11:     Let T_n = generateOffspring(t,N)
12:     for all t_g in T_n do
13:         Let (e_g, h_g) = run(P,t_g.data)
14:         Let I_g = computeFitness(e_g)
15:         Let I_h = getVProfile(h_g)
16:         if isFit(I_g) ∨ (b = isCloser(I_h)) then
17:             t_g.fitness = I_g
18:             t_g.reasonToRetain = "default"
19:             if b then
20:                 t_g.closeness = I_h
21:                 t_g.reasonToRetain = "headroom"
22:             end if
23:             addChild(T_G,t,t_g)
24:         end if
25:     end for
26: until user terminates the run
27: return T_G
```

and closer to vulnerability locations are generated. *Population 2*: Test inputs that don't reach new code regions, but reduce headroom at vulnerability locations more than other previously generated test inputs.

Algorithm 1 describes our approach. The approach is basically an enhancement to the widely used greybox fuzzing tool AFL.

Given a program P and an initial test input s, the algorithm iteratively generates test inputs, and *retains* some of them in a tree T_G. Each node of the tree corresponds to a test input, and has four fields. The "*data*" field contains the actual test input (a sequence of bytes). The field "*reasonToRetain*" indicates which of the two populations the test input belongs to, with value "*default*" indicating the first population and value "*headroom*" indicating the second population. The "*fitness*" field contains the branch-pair coverage profile corresponding to this test input. The field "*closeness*" is used only for test inputs in the second population, and stores the vulnerability profile.

Lines 2-7 create the root of this tree using the given seed test input s. We assume that the program has been instrumented in a way that each run returns a pair of data structures (e_s, h_s) such that the branch-pair profile and the vulnerability profile can be computed from these data structures, respectively. Specifically, the routine computeFitness computes the branch-pair profile.

The algorithm then works iteratively and continually using the loop that begins at Line 8, generating one or more child test inputs from a selected test input in each iteration. The *selection operation* selectNextH (called in Line 9) works as follows. In its even-numbered invocations, it selects a Population 1 test input from T_G using AFL's underlying selection algorithm. In its odd-numbered invocations, it selects one of the Population 2 test-inputs from T_G randomly in a way that that the probability of a test input getting selected is high if it has never been fuzzed before (i.e., is a leaf of T_G) or has low headroom for some vulnerability location, is moderate if it was fuzzed many iterations back in the run of the algorithm, and is low if it was fuzzed in a recent iteration. The exact values of these thresholds is a tunable parameter.

The *fuzzing potential* of a test case t is a number, denoted as N, and is obtained by invoking the getFuzzPotentialH operation in Line 10. This operation invokes AFL's underlying fuzzing potential operation in case t is a Population 1 test input, and returns a number that is inversely proportional to the least entry in the vulnerability profile of t in case t belongs to Population 2. Again, the proportionality factor is tunable.

In Line 11, N test inputs are generated as offspring of t (by fuzzing $t.data$), using AFL's approach as-is. AFL uses different *genetic operators* such as *flipping bits* at specific locations in the $t.data$, *copying bytes* from one location in $t.data$ and writing them to some other location, etc. For simplicity, we do not depict cross-over operations that generate a single child test input from multiple parent test inputs.

Each newly generated test input t_g is executed in Line 13. The routine getVProfile creates the the vulnerability profile of the test input (see Line 15). The instrumentation that is required to generate a vulnerability profile depends on the type of vulnerability. We discuss this in further detail in Sections 3 and 4.

Lines 16-23 of Algorithm 1 add the newly generated test input t_g if either t_g achieves a different branch-pair profile than other inputs in T_G, as tested by AFL's built-in routine isFit, or has attained less headroom than previously generated test inputs for some vulnerability location, as tested by the routine isCloser, which we had already presented in Section 2.1.

After Algorithm 1 is terminated, the vulnerability exposing test inputs are obtained by picking up the test inputs in T_G whose vulnerability profiles have a zero entry for any vulnerability location.

3 INSTANTIATING HEADROOM TO SPECIFIC VULNERABILITY TYPES

In this section we demonstrate the generality of the notion of headroom by instantiating it to two types of real-life vulnerabilities.

3.1 Buffer overrun vulnerabilities

A buffer overrun occurs when a buffer (or array) is written to beyond its limits. It is a very prevalent vulnerability in real world software; for instance, it ranks third in the top 10 most dangerous software errors in the CVE database [8].

Consider a buffer access location v_l of the form "`*ptr = ...`", where ptr is a pointer, and consider a specific visit to this location during a run. Let A_c be the value of ptr during this visit, and let A_h and s be the starting address and size of the allocated buffer

within which ptr is supposed to point to during this visit (how to determine this is discussed in further detail in Section 4). We define the headroom h_l during this visit as follows:

$$h_l = \begin{cases} 0, & \text{if } A_c \geq A_h + s \\ \frac{(A_h + s - A_c)}{s}, & \text{if } A_h \leq A_c < A_h + s \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

Finally, the headroom for the entire run at v_l, i.e., $I_h(v_l)$, is defined as the least value of h_l as defined above across all visits to v_l during the run (it is taken as 1 if v_l was not visited at all during the run).

3.2 Integer overflow vulnerabilities

Integer overflow (or wraparound) occurs when the result of an operation that involves integer variables goes beyond the range of values that can be represented by the integer type [10]. These operations could be arithmetic operations, value-losing type castings, or bit shift operations. An integer overflow can lead to other vulnerabilities such as buffer overflows or non-termination of programs.

Integer overflows can be of two types. The first type is *maximum value wrap around*, where a value greater than the maximum representable integer I_{max} is attempted to be computed, hence causing the result to erroneously wrap around to the negative side. Let i_o be the value that is sought to be computed at a vulnerability location (that is, a location where one of the operations named above occurs) during a specific visit to this location during a run. Then, the headroom corresponding to this visit can be defined as:

$$h_l = \begin{cases} 0, & \text{if } i_o > I_{max} \\ \frac{(I_{max} + 1 - i_o)}{I_{max}}, & \text{if } 0 < i_o \leq I_{max} \\ 1, & \text{otherwise.} \end{cases} \quad (2)$$

Finally, the headroom for the entire run at v_l, i.e., $I_h(v_l)$, is defined as the least value of h_l as defined above across all visits to v_l during the run (it is taken as 1 if v_l was not visited at all during the run). Intuitively, the headroom is the difference between the highest value sought to be computed at the location v_l during the run and I_{max}, expressed as a ratio of I_{max} itself.

The headroom can be defined for *minimum value wrap around* analogously. We provide more details about how the formula above can be computed by actual instrumentation in Section 4.

4 IMPLEMENTATION

In this section we describe the implementation of our approach, along with instantiation of this approach for finding buffer overrun and integer overflow vulnerabilities as described in Section 3. We have implemented our approach as an extension of AFL. AFL has been a popular platform for researchers to implement and evaluate extensions to fuzzing [4, 17, 24]. AFL is a fuzzer for C programs. Our extensions to AFL include changes to maintain two populations of test inputs, to select test inputs from Population 2 for fuzzing, to calculate the fuzzing potential for a Population 2 test input, and logic to retain a Population 2 test input if it achieved a significantly better than vulnerability profile than other Population 2 test inputs

```
1  void *glb_obuf;int glb_size_obuf;
2  void main(){
3    char inp[100],obuf[50], char c; int i = 0;
4    glb_obuf=(void *) obuf;
5    glb_size_obuf = 50;
6    fgets(inp,100,stdin);
7    while((c = in[i++]) != NULL){
8      _hdr_calc_bov(1,glb_obuf,(void *)out,glb_size_obuf);
9      *out++ = c;
10   }
11 }
```

Figure 3: Example instrumented program for buff. overrun

in the tree T_G. We had described all these changes conceptually in Section 2.

AFL comes with a compiler that instruments the given program to compute the branch-pair profile at run-time. Using the CIL [19] program transformation framework, we have implemented a source-to-source pre-processor that adds extra instrumentation to the program to-be-tested to populate the vulnerability profile at run-time. Our instrumentor runs first, and our instrumented program is supplied to AFL's compiler for AFL's instrumentation to be added. Our instrumentor adds a separate table in shared memory for communicating the vulnerability profile from the test run. Similar to AFL's visit-count table, each entry of our table (which is a headroom at a vulnerability location) is a 8-bit value.

4.1 Buffer overrun instrumentation

Here we describe how we have implemented buffer overrun instrumentation. CIL's simplifier converts all buffer-writing statements into the form "`*ptr = exp`". Hence, we only need to instrument statements of this form.

We illustrate the buffer-writing instrumentation using the example program in Figure 3. The code shown in blue colour is not part of the given program, and is instrumentation added by our approach. This program reads a NULL delimited string into an input buffer 'inp' in Line 6 and copies this string into an output buffer 'obuf' in Line 9. In this program there is a potential buffer overrun at Line 9, which will be exposed by any input string with length more than 50 characters.

To instrument a buffer-writing statement, our instrumentor uses CIL's built-in pointer analysis. The pointer analysis identifies the statement(s) where the buffer(s) that ptr may potentially point to are declared/allocated. The instrumentor adds actual headroom instrumentation for a buffer-writing statement only in cases where the pointer analysis returns a unique buffer declaration/allocation site.

In our example program, the pointer 'out' at Line 9 is determined by pointer analysis to point to the buffer 'obuf' that is declared in Line 3. Our instrumentor introduces and initializes two global instrumentation variables right after each such buffer declaration/allocation site. Lines 4-5 show this code. The first variable records the starting address of the buffer, while the second records the size of the buffer (see symbols A_h and s in Section 3.1).

```
1  /* v_prof is a global array.
      It is the vuln profile. */
2  void _hdr_calc_bov(int v_id,
3      void *buf_head,
4      void *cur_pos,
5      int buf_size){
6    double hr = 1.0;int tmp;
7    if(cur_pos < buf_head)
8      return;
9    tmp = buf_head + buf_size;
10   if (cur_pos >= tmp){
11     v_prof[v_id] = 0;
12     return;
13   }
14   tmp -= cur_pos
15   hr = (double)tmp/buf_size;
16   if(hr < v_prof[v_id])
17     v_prof[v_id] = hr;
18   return;
19 }
```

(a)

```
1   /* v_prof is a global array */
2   void _hdr_calc_iov(int v_id,
     long expVal){
3     double hr = 1.0;int tmp;
4     if(expVal <= 0)
5       return;
6     if(expVal > INT_MAX ){
7       v_prof[v_id] = 0;
8        return;
9     }
10    tmp = INT_MAX - expVal + 1;
11    hr = (double)tmp/INT_MAX;
12    if(hr < v_prof[v_id])
13       v_prof[v_id] = hr;
14    return;
15  }
16
```

(b)

Figure 4: Instrumentation functions (a) buffer overflow (b) integer overflow

The actual headroom calculation is implemented in a separate function `_hdr_calc_bov`. The instrumentor adds a call to this function just before each vulnerability location (see Line 8). The parameters to this function are: a constant representing the vulnerability location, the starting address of the (unique) buffer to which the access-pointer points to, the current value of the access pointer, and the size of the buffer.

Figure 4(a) shows the pseudo code for the function `_hdr_calc_bov`. This function encodes the calculation that was described in Section 3.1. This function first checks for underrun condition, and simply returns if so (Lines 7-8). It then checks for overrun condition. If an overrun has already occurred, it sets the headroom in the vulnerability profile to zero (Line 11). Otherwise, it calculates and sets the headroom in the vulnerability profile in Line 17.

Our current implementation has some limitations, and does not handle the following scenarios: (a) An access pointer pointing to more than one buffer as per pointer analysis, (b) Buffer underflows, (c) Buffers allocated within loops, and (d) Buffer over reads. We believe it would be possible to alleviate most of these limitations via appropriate extensions as part of future work.

4.2 Integer overflow instrumentation

For integer overflow checking, our automated approach instruments arithmetic operation statements of the form `v1 = v2 + v3`, `v1 = v2 * v3`, `v1 = v2 - v3`. In computing the headroom, we have adopted *width extension test* [10] approach. In this approach, the operands of each integer expression are first converted into wider bit-width numbers, and the expression is computed using the higher bit-width. The result (which is in the higher bit-width) is checked to see whether it is within the range of values represented by the original bit-width. In our implementation, we target expressions whose original operands are 32-bit integers, and we use 64-bit "long" integers as the extended bit-width.

Figure 4(b) shows pseudo code for the function `_hdr_calc_iov`, a call to which is inserted by our instrumentor just before each vulnerability location. The arguments to the function are the ID of the vulnerability location, and the result from the evaluation of the expression at the higher bit-width. This function encodes the calculation that was described in Section 3.2 for maximum value wraparound checking. (We use an analogous function for minimum value wraparound checking, which we use only for "minus" statements.) This function first checks whether `expVal` is a negative number, in which case maximum value wraparound has not happened. It then checks if the maximum value wraparound overflow has already happened (Lines 6-10). Otherwise, it calculates and stores the headroom in Line 14.

Currently, our implementation does not check overflows that occur due to type casting and bit-shift operations. Also, it does not handle integer types with bit-widths other than standard 32-bit integers.

5 EVALUATION

In this section we provide evaluation of our approach in the context of buffer overrun vulnerabilities (BOV) as well as integer overflow vulnerabilities (IOV).

5.1 Benchmark programs

For our evaluations we have used a set of eight "MIT Benchmarks"[1] [28] programs, as well as the "SV-COMP benchmarks". The MIT Benchmark suite has been used by previous researchers to evaluate symbolic-execution based techniques that detect buffer overflow vulnerabilities [22]. The SV-COMP benchmarks are used in premier competitions such as SV-COMP [1] and Test-comp [2], to compare and evaluate state-of-the-art tools that are based on static analysis [3], symbolic execution [6], or fuzz testing [7, 16], and that identify vulnerabilities such as integer overflows, errors such as assertion violations, etc.

We compare the performance of our tool, which we call AFL-HR, with two baselines: AFL, and the directed fuzzer AFLGo [4]. We used two machines in our evaluations: one with an Intel i7-6700 processor and 32 GB RAM, and one with an Intel i7-8700 processor and 64 GB RAM.

5.2 Buffer overrun vulnerabilities

The names of the MIT Benchmarks that we use are s1, s3, s4, s5, b1, b3, b4, and f1. These eight MIT Benchmarks, between them, have 46 vulnerable buffer overrun locations. These locations have been identified by the suite designers, and serve as a ground truth for our evaluations (although this information is not made available to the tools in our evaluation). For each benchmark we made a single small seed test input, and gave it to all three tools. Each tool was given a budget of 3 hours for each run on each benchmark, and was run 5 times on each benchmark in order to mitigate the effects of randomness that is present in each tool. Note that this evaluation is more rigorous than tool evaluations in competitions like SV-COMP and Test-Comp, as they evaluate each tool based on only one run on each benchmark. We have used Mann-Witney U-Test [15] for validating the performance of our tool with respect to the two baseline tools.

Table 1 and Figure 5 summarizes the results of the evaluation. In Table 1, Column 2 shows the benchmark name. Column 3 shows the

[1] http://bitblaze.cs.berkeley.edu/lese/lese_benchmark_100208.tar.gz

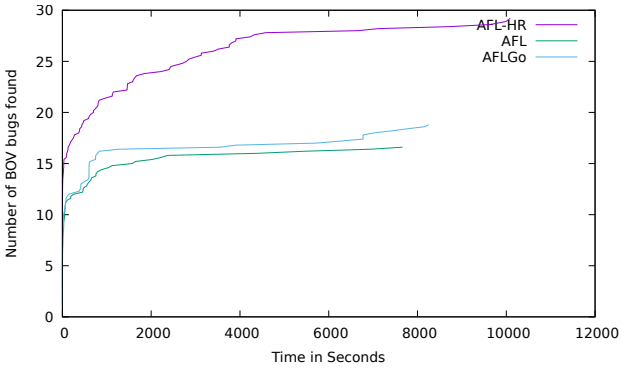

Figure 5: Results for buffer overrun detection

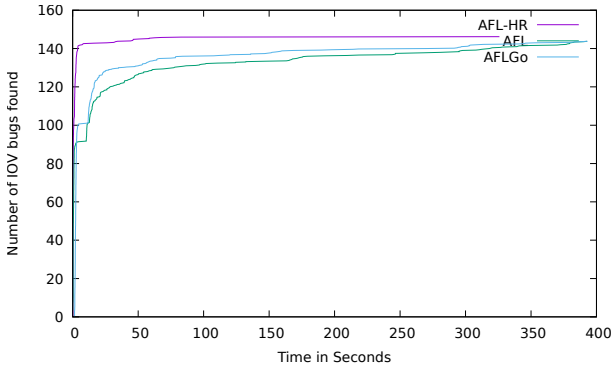

Figure 6: Results for integer overflow detection

S.No	Bench-mark	Number vulnerable locations	# vuln. locations detected			Statistically Significant?	
			AFL	AFLGo	AFL-HR		
			(A)	(B)	(C)	(D)	(E)
1	s1	28	8	10	17	Y	Y
2	s3	3	2	2	3	Y	Y
3	s4	4	3	3	4	Y	Y
4	s5	3	0	0	1	Y	Y
5	b1	1	1	1	1	N	N
6	b3	1	0	1	1	Y	N
7	b4	2	1	1	1	N	N
8	f1	4	1	1	1	N	N

Table 1: Average number of buffer overflow violations detected by each tool and results of statistical significance tests.

number of known vulnerability locations. Columns A to C show the mean number of buffer overflows detected by AFL-baseline, AFLGo, and AFL-HR modes across five runs. Column D shows whether the number of buffer violations detected by AFL-HR are statistically significantly different when compared to AFL or not. Column E shows the same evaluation with respect to AFLGo.

In Figure 5, each plot shows the cumulative number of vulnerabilities found by a single tool over the three hour duration across all the benchmarks, by starting all the runs simultaneously and letting them run concurrently. For each benchmark, at any point of time in the three hour window, the average number of vulnerabilities found in this benchmark across the five runs corresponding to this benchmark up to this point is considered.

In summary, our approach AFL-HR finds a total of 29 (from the total of 46) vulnerabilities within 3 hours, while AFLGo and AFL find 19 and 16 vulnerabilities, respectively. All the vulnerabilities found by AFLGo and AFL also found by AFL-HR. Furthermore, as the plots reveal, AFL-HR finds the vulnerabilities much faster. For instance, by 1 hour, AFL-HR has found 26 vulnerabilities, while in the same time AFL has found only 16 vulnerabilities and AFLGo has found 17 vulnerabilities. The statistical significance test shows that AFL-HR detected significantly more buffer overflow violations than AFL and AFLGo in 50% of the programs.

5.3 Integer overflow vulnerabilities

Our primary benchmark suite for this evaluation is the suite of SV-COMP 2019 benchmarks. We use 177 integer overflow benchmarks, with each benchmark known to have one vulnerable integer operation. These benchmarks being smaller in size, each tool was given a budget of 900 seconds per benchmark (this timeout is a standard in the SV-COMP and Test-comp competitions). Each tool was given three seeds, which contained zero, the smallest negative number and the largest positive number, respectively. Each tool was run 5 times on each benchmark.

Figure 6 summarizes the results, in the same format as in Figure 5. We show the plots only until the first 400 seconds, as after that point all tools have identified almost the same number of vulnerabilities. It is notable that in the first 15 seconds itself, AFL-HR found 143 vulnerabilities, while AFLGo found 118 vulnerabilities and AFL found 110 vulnerabilities. Further, when we consider all vulnerabilities in all benchmarks, the increased number of vulnerabilities found by AFL-HR compared with both AFL and AFLGo in the first 15 seconds is statistically significant. At the end of the entire 900 seconds, AFL-HR found 148 vulnerabilities, AFLGo found 146 vulnerabilities, and AFL found 147 vulnerabilities. On average, AFL-HR needed 7 seconds to find each vulnerability, while AFLGo needed 27 seconds and AFL needed 42 seconds.

6 RELATED WORK

In this section we compare our work with related work across three different categories.

In recent years there has been a large body of reported work on greybox fuzz testing [11, 21, 27], as this approach has been found to be scalable and practical. Basic coverage-based greybox fuzzing approaches came first. Subsequently, researchers have proposed extensions such as to prioritize the coverage of low-frequency paths [5, 7, 21, 24], and to *direct* fuzzers to reach more quickly a given set of target program locations [4].

FuzzFactory [20] is a framework for instantiating a fuzzer with domain-specific testing objectives. Our approach cannot be seen as a possible instantiation of their approach, as they maintain a single population of test inputs, and use AFL's default selection logic to select test inputs to fuzz from this population. In other words, test inputs that are more fit as per the domain-specific objective may

not be fuzzed more, unlike in our approach. Also, their paper does not focus on detecting vulnerabilities.

None of the approaches mentioned above are targeted at identifying difficult vulnerabilities that get exhibited only in runs that reach vulnerability locations with certain specific vulnerability-inducing program states.

Whitebox techniques employ deep analysis of code using symbolic or concolic execution, which enumerates paths systematically and symbolically up to a timeout [6, 13]. These techniques should be able to identify vulnerability exposing test runs. However, their scalability is known to be low. Unlike greybox fuzzers, they are generally not good at finding vulnerabilities that are induced by execution paths that need to iterate loops numerous times.

A number of SBST approaches [12, 25] calculate how close a conditional came in a run to evaluating to a desired value (*true* or *false*) that would send control to an exception statement. They use this distance as a fitness metric to guide the test exploration effort towards the exception statement. Our headroom notion is similar to this closeness. The main novelty of our work is to bring this idea into a prototypical greybox fuzzer (AFL) with a set of practical design decisions. The main contrast between fuzzers and SBST is that fuzzers use lightweight and scalable heuristics to maintain and evolve a population of test cases, whereas SBST approaches typically use an optimization formulation to search for ideal test inputs or test suites.

7 CONCLUSIONS AND FUTURE WORK

In this paper we addressed the challenging problem of exposing vulnerabilities that are exposed by runs that reach vulnerability locations with specific vulnerability-inducing program states. We proposed an approach that for the first time uses vulnerability-specific fitness metrics to generate and retain test inputs that come closer to exposing vulnerabilities. An evaluation of our tool on commonly used benchmarks reveals that our approach either detects multiple times more vulnerabilities or detects them much quicker over the baseline AFL as well as over a recent directed fuzzer.

A limitation of this paper is that while we have evaluated the approach on benchmarks that are used by researchers or in competitions, we have not evaluated it on large programs. For this, we feel we need to migrate our instrumentation approach to use LLVM (rather than CIL), for example, by enhancing runtime memory checkers such as Address Sanitizer (ASAN) [23] to instrument the program to precisely track memory operations and their headroom during the program execution. These would constitute interesting directions for future work. It would also be interesting to instantiate our approach to other challenging types of vulnerabilities, like failing assertions, using memory after freeing, SQL injection attacks.

Acknowledgments. This work was partially supported by a grant from TCS Limited to Indian Institute of Science, Bangalore.

REFERENCES

[1] Dirk Beyer. 2019. Automatic verification of C and Java programs: SV-COMP 2019. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 133–155.
[2] Dirk Beyer. 2019. International competition on software testing (Test-Comp). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 167–175.
[3] Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*. Springer, 184–190.
[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2329–2344.
[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1032–1043.
[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.
[7] Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R Venkatesh. 2019. VeriFuzz: Program aware fuzzing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 244–249.
[8] CVE database. 2017. *CVE Details - Vulnerabilities by type*. Technical Report. https://www.cvedetails.com/vulnerabilities-by-types.php
[9] Jared DeMott, Richard Enbody, and William F Punch. 2007. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. *BlackHat and Defcon* (2007).
[10] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 2.
[11] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.
[12] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical software engineering* 20, 3 (2015), 611–639.
[13] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.
[14] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.
[15] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2123–2138.
[16] Hoang M. Le. 2019. KLUZZER: Whitebox Fuzzing on Top of LLVM. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. 246–252.
[17] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 254–265.
[18] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.
[19] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*. Springer, 213–228.
[20] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[21] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *USENIX security*. 1–14.
[22] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proc. Int. Symposium on Softw. Testing and Analysis (ISSTA)*. 225–236.
[23] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.
[24] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 1–16.
[25] Nigel Tracey, John Clark, Keith Mander, and John McDermid. 2000. Automated test-data generation for exception conditions. *Software: Practice and Experience* 30, 1 (2000), 61–79.
[26] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.
[27] Michael Zalewski. [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/
[28] Misha Zitser, Richard Lippmann, and Tim Leek. 2004. Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (FSE)* (Newport Beach, CA, USA). New York, NY, USA, 97–106.