HDR-Fuzz: Detecting Buffer Overruns using AddressSanitizer Instrumentation and Fuzzing

Raveendra Kumar Medicherla[1], Malathy Nagalakshmi[2], Tanya Sharma[2], and Raghavan Komondoor[3]

[1] TCS Research, Tata Consultancy Services, Bangalore, India, raveendra.kumar@tcs.com
[2] Bangalore,India,{malathy124,tanya.sharma0217}@gmail.com
[3] Indian Institute of Science, Bangalore, India, raghavan@iisc.ac.in

Abstract. Buffer-overruns are a prevalent vulnerability in software libraries and applications. Fuzz testing is one of the effective techniques to detect vulnerabilities in general. Greybox fuzzers such as AFL automatically generate a sequence of test inputs for a given program using a fitness-guided search process. A recently proposed approach in the literature introduced a buffer-overrun specific fitness metric called "headroom", which tracks how close each generated test input comes to exposing the vulnerabilities. That approach showed good initial promise, but is somewhat imprecise and expensive due to its reliance on conservative points-to analysis. Inspired by the approach above, in this paper we propose a new ground-up approach for detecting buffer-overrun vulnerabilities. This approach uses an extended version of ASAN (Address Sanitizer) that runs in parallel with the fuzzer, and reports back to the fuzzer test inputs that happen to come closer to exposing buffer-overrun vulnerabilities. The ASAN-style instrumentation is precise as it has no dependence on points-to analysis. We describe in this paper our approach, as well as an implementation and evaluation of the approach.

1 Introduction

Software vulnerabilities are flaws in software that can be exploited by an attacker to gain control over the system. Buffer overruns are a very prevalent "implementation vulnerability" in real world software. Buffer overrun is the third most prevalent type of vulnerability as per the CVE database [4]. A buffer-overrun occurs when a program erroneously writes to a buffer (or array) beyond the limits of the allocated buffer (or array). An attacker can exploit this vulnerability to take control of the program, for example by overwriting the return address of a function on the stack.

Detecting vulnerabilities such as buffer overruns is a key requirement for building a secure software [12]. *Fuzz testing* [8] is an automated technique that aims to uncover vulnerabilities by generating new test inputs by *fuzzing* (i.e., mutating) already generated test inputs, and by looking for bug-inducing inputs among this generated sequence of test inputs. *Greybox fuzzing* is a specific kind of

fuzz testing that is employed in several practical tools and approaches [1, 10, 14]. It searches for bug-inducing inputs using a fitness-guided search process, and using lightweight instrumentation in the program to compute the fitness of each generated test input from the (instrumented) run on this input.

Most existing greybox fuzzers, such as the industry-standard tool AFL [14], retain a newly generated test-input if a run using it *covers* a new region of code. However, better coverage alone need not result in better detection of specific kinds of vulnerabilities. Recognizing this, an approach and tool called AFL-HR [7] was described recently, which introduced the novel notion of *headroom* as a fitness metric for buffer overruns (and other related vulnerabilities). This metric assigns higher fitness (and hence causes the retention of) test inputs that come *closer* to causing a buffer overrun than other test inputs generated so far at any buffer write location in the program. The approach mentioned above was implemented as an extension over AFL, and showed significant improvement over baseline AFL in its ability to identify real buffer overrun vulnerabilities. However, this approach has two key limitations. The first is its reliance on static points-to analysis to instrument buffer write locations, which can be impractical to use on large applications, and imprecise as well. The second limitation is that the headroom instrumentation is expensive, and thus the rate at which AFL executes the (headroom) instrumented program slows down. In this paper we propose an entirely new greybox fuzzing approach, which we call *HDR-Fuzz*, which still uses the headroom metric, but eliminates both these limitations.

The novelty of HDR-Fuzz is in its architecture. The architecture consists of two parallel processes. The first process is a modified version of AFL that uses only AFL's original, lightweight coverage instrumentation, and relies on the second process to compute the headroom of any test input. The first process passes a selected subset of test inputs that it generates to the second process via a queue. The second process is based on a custom extension of Address Sanitizer (ASAN) [11] that we have devised. ASAN is an industry-standard tool to *detect* if a buffer-overrun occurs in a given run. We have extended ASAN's instrumentation regime to calculate the headroom of the run at each buffer write location that was visited but not overflowed in the run. This ASAN-based process needs no static pointer analysis, and is hence applicable practically on a large set of real programs. The second process calculates the headroom at each buffer write location due to each test input it receives, and communicates back test inputs that achieve good headroom-based fitness to the first process. The first process then retains these test inputs (in addition to test inputs that achieve better coverage) for further fuzzing.

We describe in this paper our approach as well a prototype tool based on the approach. We also describe initial experimental results from our tool on three standard suites of benchmarks. The results indicate substantial performance gains of our tool against the baseline AFL as well as the recent tool AFL-HR [7].

2 Background

In this section, we provide brief background on the existing tools ASAN and AFL that is key to understanding our approach.

2.1 AddressSanitizer

AddressSanitizer (ASAN) [11] is an instrumentation based memory-error detector that detects various types of memory-related errors. We restrict our attention to its buffer-overrun detection mechanism. The ASAN instrumentation appends fixed-size extra regions of memory, called *red zones*, around both sides of every buffer in the *main* (i.e., actual) memory (whether the buffer is a global, or is on stack or in heap). ASAN also maintains a separate *shadow memory*, wherein there is a single byte corresponding to every eight bytes in the main memory. Shadow memory locations corresponding to actual in-use memory locations contain value 0, whereas shadow memory locations corresponding to red zones in the main memory contain negative values. When any memory lookup happens, ASAN uses an efficient address translation scheme to translate the main-memory address to its corresponding shadow-memory address, and reports an overrun if there is a negative value at the shadow memory location.

2.2 AFL

AFL is a fitness-guided greybox fuzzer geared to discover test inputs that maximize code coverage. Algorithm 1 gives a high-level and simplified overview of AFL. For the current discussion, ignore Lines 2, 6–9, 17, and 18, as they are not part of the original AFL. Given an instrumented program P_a that carries AFL's coverage-based instrumentation, and an initial test input s, the approach 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 its "*data*" field contains the actual test input (a sequence of bytes). The algorithm then works iteratively and continually using the loop that begins at Line 5, by selecting a test input t from T_G for fuzzing in each iteration (Line 10).

The *fuzzing potential* of a test input t is a number N, computed heuristically in Line 11 by invoking a sub-routine getFuzzPotentialH. In Line 12, AFL generates N test inputs as offspring of t (by fuzzing $t.data$). AFL uses different *genetic operators* such as *flipping bits* at specific locations in $t.data$, *copying bytes* from one location in $t.data$ and writing them to some other location, etc.

Each test input generated in Line 12 is executed using AFL's coverage-based instrumentation in Line 14. The result of the run is a *coverage profile*, which is stored in the structure I_g. Line 16 of Algorithm 1 adds the newly generated test input t_g to T_G as a child of t if t_g has *significantly different* coverage profile than the profiles of test inputs that are already in T_G. This "significant difference" check is done by routine isFit, which is called in Line 15. Due to space limitations we are compelled to omit many details from the discussion above.

Algorithm 1 Enhanced AFL Test generation

Require: Given program P_a with AFL's coverage instrum., and a "seed" input s.
Ensure: A tree of test inputs T_G for P_a

```
 1: Create an empty tree T_G of test inputs
 2: Initialize queues Q_A, Q_R to empty
 3: t_r.data = s                                   ▷ t_r is a new tree node
 4: T_G.setRoot(t_r)
 5: repeat
 6:     ⟨t_h, S_h⟩ = DEQUEUE(Q_R)
 7:     for all t_a in S_h do
 8:         if t_a ∉ T_G then
 9:             ADDCHILD(T_G,t_h,t_a)
10:     Let t = SELECTNEXTH(T_G)
11:     Let N = GETFUZZPOTENTIALH(t)
12:     Let T_n = GENERATEOFFSPRING(t,N)
13:     for all t_g in T_n do
14:         Let I_g = RUN(P_a, t_g.data)
15:         if ISFIT(I_g) then
16:             ADDCHILD(T_G,t,t_g)
17:     Let S_n = SAMPLEINPUTS(T_n)
18:     ENQUEUE(Q_A,⟨t, S_n⟩)
19: until user terminates the run
20: return T_G
```

3 Our approach

In this section, we describe our approach in terms of our changes to AFL, our enhancements to ASAN to yield headroom-checking instrumentation, and finally, the *Driver* component of our tool, which uses headroom instrumentation to identify test inputs that have good fitness based on the headroom metric. Note that (our modified version of) AFL and the Driver are the two parallel processes that constitute our overall system.

3.1 AFL enhancements

Lines 2, 6–9, 17, and 18 in Algorithm 1 describe our enhancements to AFL to make it utilize headroom checking. Lines 17-18 sample a subset S_n of the set of newly generated test inputs T_n, and send them (along with their parent test input t) to the driver via a shared queue Q_A. Sampling is necessary, as running every generated test input with headroom instrumentation within the Driver would be prohibitively expensive. Our sampling selects all test inputs that are already in T_G, and additionally a random subset of test inputs from $T_n - T_G$ (i.e., test inputs that AFL chooses not to retain due to insufficient new coverage). The size of this subset (as a percentage) is a parameter to the approach. The intuition behind selecting inputs in $T_n - T_G$ is that though a test input may

be uninteresting from a coverage perspective, it may come closer to exposing a vulnerability.

We will discuss the working of the driver in more detail in Section 3.3. For each record $\langle t, S_n \rangle$ that the driver receives from the queue Q_A, it returns to Algorithm 1 the subset of S_n consisting of test inputs that are fit wrt the headroom metric, along with the parent test input t. Algorithm 1 adds the test inputs received from the driver to T_G (see Lines 6–9 in Algorithm 1). The dequeing operation in Line 6 is non-blocking, and returns an empty set S_h in case the queue is empty.

SELECTNEXTH (called in Line 10) works as follows. As it is invoked repeatedly throughout the run of the algorithm, it alternatively selects test inputs that were added to T_G in Lines 9 and 16 of the algorithm. The intuition is that we would like to give equal priority to fuzzing test inputs that attain new coverage (in order to cover all parts of the program well) and to fuzzing test inputs that reduce headroom (in order to come closer to exposing vulnerabilities in parts of the program that are already covered).

After Algorithm 1 is terminated, the vulnerability exposing test inputs are reported by picking up test inputs in T_G that cause crashes when the program is run using (normal) ASAN instrumentation.

3.2 Headroom-checking instrumentation

Our headroom-checking instrumentation piggy-backs on ASAN's instrumentation. The *raw headroom* [7] at a buffer write location due to a run measures how close (in terms of number of bytes) the buffer write pointer came to the end of the buffer across all visits to the location in the run. The *scaled headroom* is the ratio of raw headroom over the size of the buffer multiplied by 128. A low value of headroom means that the run came close to exposing a buffer overrun.

```
// Instrumented program has a global
// array called headroom_array,
//indexed by buffer-write locations.

calculate_headroom(uptr addr, int storeIdx) {
  // storeIdx is location of currently visited
  // buffer-write instruction.
  // addr is memory address used in this visit.
  writeAddr = getShadowAddress(addr);
  endAddr = begAddr = writeAddr;
  int rawHeadroom = leftMargin = 0;
  while (endAddr is not in right redzone) {
    // Move right
    endAddr = endAddr + 1;
    rawHeadroom = rawHeadroom + 1;
  }
  while (begAddr is not in left redzone) {
     // Move left
     begAddr = begAddr - 1;
     leftMargin = leftMargin + 1;
  }
  byte scaledHeadroom = (rawHeadroom * 128) /
              (leftMargin + rawHeadroom - 1);
  headroom_array[storeIdx] = scaledHeadroom;
}
```

Fig. 1. Headroom calculation at a buffer-write event

Our instrumentation calculates the scaled headroom due to the run at every *buffer write* location, whether the write location is a global, stack, or heap buffer. Figure 1 depicts the pseudo code of routine *calculate_headroom*, which our

instrumentation invokes from each buffer-write location. The parameter *addr* is the address at which the current write is happening, while *storeIdx* is the Location ID (or program counter) of the current write instruction. The subroutine *getShadowAddress* is ASAN's address translation routine (see Section 2.1). `headroom_array` is a global array indexed by buffer-write locations, which, at the end of each run of the program, stores the scaled headrooms at the buffer write locations that were visited in the run. We refer to the contents of this array at the end of a run as the *headroom profile* of the run. The rest of the pseudo-code in Figure 1 is self-explanatory.

3.3 Driver

Algorithm 2 Coordinating driver

Require: Given program P_h with headroom instrumentation
Ensure: Set of test inputs S_h to be retained due to headroom.
1: Initialize array *minHProfile* with value 128 in all entries.
2: **repeat**
3: $\langle t, S_n \rangle$ = DEQUEUE(Q_A)
4: $S_h = \emptyset$
5: **for all** t_s **in** S_n **do**
6: Let I_h = RUN(P_h, $t_s.data$)
7: **if** ISLESS(I_h, *minHProfile*) **then**
8: Add t_s to S_h
9: *minHProfile* = MIN(*minHProfile*, I_h)
10: **if** $S_h \neq \emptyset$ **then**
11: ENQUEUE(Q_R, $\langle t, S_h \rangle$)
12: **until** *user terminates*
13: **return**

Algorithm 2 describes the driver, which runs in parallel with Algorithm 1. The given program P_h carries our headroom instrumentation as described in Section 3.2. The algorithm maintains a globally minimum headroom profile across all runs of the program in the global array *minHProfile*, indexed by the buffer-writing locations in the program. The algorithm waits for test inputs from Algorithm 1 in the queue Q_A (Line 3). On each received test input t_s, it runs the instrumented program P_h in Line 6, and picks up the headroom profile at the end of the run and puts it into the temporary array I_h. The function ISLESS in Line 7 checks whether the headroom in any entry of I_h is less than the corresponding entry of *minHProfile*; if yes, t_s is saved into a set S_h. In Line 11, all the saved test inputs are sent back to AFL via the queue Q_R.

4 Implementation and Evaluation

We have implemented our approach as a tool named HDR-Fuzz. Our implementation is built on top of AFL 2.52 and ASAN 10, and uses the C and C++ languages.

We evaluate our tool on three buffer-overrun benchmark suites. The first suite is a set of eight programs from the "MIT Benchmarks", which have been used by previous researchers to evaluate buffer overflow detection tools [15]. We use the following MIT benchmark programs: s1, s3, s4, s5, b1, b3, b4, and f1. The second suite is a set of 10 programs from the "Cyber Grand Challenge" (CGC) benchmarks [13], which have been used to evaluate fuzzers and symbolic execution tools. We use the following 10 CGC benchmarks: CROMU_00030, CROMU_00084, CROMU_00088, KPRCA_00060, TNETS_00002, KPRCA-00001, CROMU_00041, CROMU_00020, KPRCA_00041, and KPRCA_00045. The third suite contains nine programs from Google's "fuzzer-test-suite" (FTS) [6]. These benchmarks are much larger in size, and are derived from real applications or libraries. We use the following nine FTS benchmarks: openssl-1.0.1f, libxml2-v2.9.2, libarchive-2017-01-04, woff2-2016-05-06, openthread-2018-02-27-rev1, openthread-2018-02-27-rev2, openthread-2018-02-27-rev7, openthread-2018-02-27-rev8/9/10, and openthread-2018-02-27-rev11. We compared the performance of our tool with (standard) AFL. We used Google cloud "standard" machines, with 8 hardware threads and 32GB memory, for our evaluations. For each benchmark program, we ran both AFL and our tool HDR-Fuzz on the benchmark (starting from a small manually created "seed" test input). To mitigate the effects of randomness in the fuzzers, we ran both tools on each benchmark program three times, with a 3-hour time budget for each run.

4.1 Experimental results on MIT and CGC benchmarks

We present our experimental results on the MIT and CGC benchmarks together, as both these suites consist of small to medium size programs. The MIT and CGC benchmarks together have 61 known vulnerable buffer overrun locations (49 in MIT, and 12 in CGC). These locations are indicated by the suite designers, but this information is (obviously) not given by us to either tool.

Figure 2 shows the cumulative number of vulnerabilities found over time across all the eighteen benchmarks by AFL and by our tool. At any point of time t, for each benchmark, we have considered the average number of vulnerabilities exposed up to that point of time across the three runs, added up the averages corresponding to all eighteen benchmarks, and plotted this sum against point t in Figure 2. Overall, our tool detects an average of 40.5 (out of 61) vulnerabilities, which is more than twice compared to AFL, which detects only 18. From the graph, it is also clear that our tool finds vulnerabilities much more quickly; e.g., within the first 200 seconds our tool has found more than 20 vulnerabilities while in the same time AFL has found only about 7 vulnerabilities. Note, we cutoff the plots at 9000 seconds (without going all the way to 3 hours) because no new vulnerabilities were found by either tool beyond the cut-off point chosen.

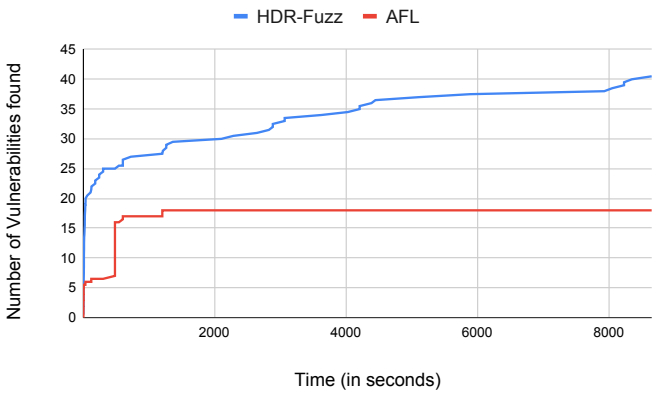

Fig. 2. Results from MIT and CGC Benchmarks

4.2 Experimental results on FTS benchmarks

We followed the same experimental procedure with the FTS benchmarks as with the MIT and CGC benchmarks. Figure 3 depicts the results (in the same manner as Figure 2). There are a total of 11 known vulnerability locations across the nine benchmarks. In summary, our approach detects an average of 8 vulnerabilities, while AFL detects an average of 6.5 vulnerabilities. That is, our tool finds 23% more vulnerabilities. Our tool is quicker here as well, as the gap in vulnerabilities found by the two tools arises very early in the runs itself. Although the gap between the two tools is not as large as with the MIT benchmarks, it is still quite significant.

4.3 Comparison with AFL-HR

We briefly compare here the performance of our approach with the recent approach AFL-HR, which originally introduced the notion of headroom. Our comparison is based on the results as reported in their paper [7]. They had applied their approach only on the MIT benchmarks. Their tool had detected 29 vulnerabilities on average (out of 49 total) within the same 3-hour time budget, whereas our tool detects approximately 38.5 vulnerabilities on average on these same benchmarks. This is a substantial improvement in performance. We believe the improvement is due to two reasons: Firstly, AFL executes more runs per unit time in our approach due to the absence of headroom instrumentation in the version of the program given to AFL. Secondly, our driver process (based on ASAN) calculates headroom directly and precisely at all buffer write locations using memory-instrumentation. Whereas AFL-HR's headroom computation instrumentation is based on (imprecise) static points-to analysis, and calculates headroom precisely only at buffer write locations where the buffer write pointer

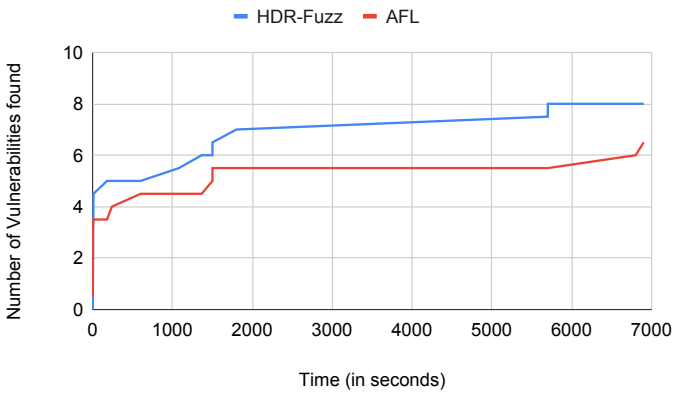

Fig. 3. Results from FTS benchmarks

is guaranteed to point to a unique source-level declared buffer across all visits to the location in all runs.

5 Related work

Due to space limitations, we compare our work only with very closely related work. We have taken the idea of the headroom metric from the recent approach AFL-HR [7]. However, our tool architecture is quite different from AFL-HR's, and is based on two communicating and collaborating processes as opposed to a single process. Also, our tool has no dependence on static points-to analysis. The improvement in effectiveness of our approach due to these factors was discussed in Section 4.3. Also, due to the independence from static points-to analysis, our approach is applicable on large benchmarks like FTS, on which it is not clear whether AFL-HR would scale.

In recent years there has been a large body of reported work on greybox fuzz testing, as this approach has been found to be scalable and practical. Basic coverage-based greybox fuzzing approaches were proposed first [5, 14]. Subsequently, researchers have proposed extensions such as to prioritize the coverage of low-frequency paths [2, 3, 10, 13], and to direct fuzzers to reach more quickly a given set of target program locations [1]. These approaches are not adept at exposing complex vulnerabilities such as buffer overruns that get exhibited only in runs that reach vulnerability locations with certain specific vulnerability-inducing program states. FuzzFactory [9] is a framework for instantiating a fuzzer with domain-specific testing objectives. However, their approach does not focus on detecting vulnerabilities, and does not have the notion of how close a test run comes to exposing a vulnerability.

6 Conclusions and Future Work

In this paper, we proposed an approach that combines AFL's fuzzing and headroom instrumentation based on ASAN's shadow memory to detect buffer overrun vulnerabilities effectively and efficiently. Our tool was robust and applicable to very large benchmarks, and detected many more vulnerabilities than baseline AFL as well as the recent tool AFL-HR.

In future work, we would like to extend our approach to identify other kinds of vulnerabilities that ASAN currently targets such as use-after-free errors, as well as other types of vulnerabilities such as integer overflows and assertion violations for which we might need to use runtime tools beyond ASAN.

References

1. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2329–2344. ACM (2017)
2. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 1032–1043. ACM (2016)
3. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 244–249. Springer (2019)
4. CVE Database: CVE details - vulnerabilities by type. Tech. rep. (2017), `https://www.cvedetails.com/vulnerabilities-by-types.php`
5. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2), 276–291 (2012)
6. Google: Fuzzer test suite, `https://github.com/google/fuzzer-test-suite`
7. Medicherla, R.K., Komondoor, R., Roychoudhury, A.: Fitness guided vulnerability detection with greybox fuzzing. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW '20). pp. 513–520 (2020)
8. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12), 32–44 (dec 1990). https://doi.org/10.1145/96267.96279
9. Padhye, R., Lemieux, C., Sen, K., Simon, L., Vijayakumar, H.: Fuzzfactory: domain-specific fuzzing with waypoints. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–29 (2019)
10. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: Application-aware evolutionary fuzzing. In: USENIX security (2017)
11. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference. pp. 309–318 (2012)
12. Shahriar, H., Zulkernine, M.: Mitigating program security vulnerabilities: Approaches and challenges. ACM Comput. Surv. **44**(3) (Jun 2012)
13. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2016)

14. Zalewski, M.: American fuzzy lop, `http://lcamtuf.coredump.cx/afl/`
15. Zitser, M., Lippmann, R., Leek, T.: 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). pp. 97–106. New York, NY, USA (2004)