FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

BSc Thesis

Søren Lund Jensen

Fuzzing with American Fuzzy Lop

Assisting Fuzzing with Symbolic Execution

Academic advisor:	Michael Kirkedal Thomsen
	Troels Larsen
Submitted	June 11, 2017

Fuzzing with American Fuzzy Lop

Assisting Fuzzing with Symbolic Execution

Søren Lund Jensen | PWS412
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

June 11, 2017

Author:	Søren Lund Jensen
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark
Title:	Fuzzing with American Fuzzy Lop, Assisting Fuzzing with Symbolic Execution
Academic Advisor:	Michael Kirkedal Thomsen Troels Larsen
Submitted:	June 11, 2017

Contents

1 Abstract

The following paper analyses the program Driller, by the Shellphish team of UC, Santa Barbara. The program Driller is designed to be an expanded upon version of American Fuzzy Lop[6], which has been used in the first annual DARPA Cyber Grand Challenge[4] competition. The DARPA CGC had the goal of enhancing research in the field of autonomous binary analysis, which it reached by offering generous prices for not only the winners of the competition, but everyone who qualified for the event in the first place.

Shellphish' Driller was among these programs, and have after the competition proven to be a quite useful tool, as it is able to discover a broader variety of bugs, due to its increased code coverage. The additional code coverage, which separates Driller from ordinary fuzzers, is its ability to make use of angr, the concolic execution engine, which allows Driller to dig deeper into the analysed binaries, triggering bugs, hidden by several layers of complex checks, which a standard fuzzer would take ages to randomly discover.

Driller does definitely take the idea of fuzzing, and binary analysis in general one step in the right direction, as, as discussed in section 9.3, it is able to increase the amount of crashed programs, in the tested dataset by $7,59\%$, which, although, may not seem much, represents a section of extremely complex binaries, which would normally go uncrashed. By extend, the vulnerabilities in the crashed programs would have never been discovered, and therefore never been patched, had it not been for Driller.

2 Introduction and concept

An ever-present danger in today's society is memory corruption vulnerabilities in software, be they use of uninitialized memory, using dangling null-pointers, buffer overflow, memory leaks, or a fifth, sixth- or seventh vulnerability. An attacker could, did he know of these vulnerabilities, exploit them in order to access confidential information, create DOS-attacks or other and as computer processing and connecting continues to be on the rise, playing a major role in present day, patching these vulnerabilities has to be a priority. This, of course, cannot be done without first discovering said bugs.

Memory corruption bugs are often-case virtually untraceable, as only specific input combinations may trigger them, or the fact that they may appear under very unusual conditions, which makes it very hard to discover, or in some cases reproduce, even known bugs . Add thereto, the fact, that the memory corruption's effect may manifest itself far away from its source, it can also be hard to even correlate the two, once a bug has been discovered.

A variety of tools exists, with the purpose of bug-discovery, but as the bugs are often very specific, and/or wide-spread, creating a silver bullet is hard, if not impossible. Many vulnerabilities are discovered manually, however, this solution is not scalable, as software applications generally increase in size and complexity. A handful of tools exist, including fuzzers and symbolic execution engines. These do, however have, in the worst cases, deal-breaking flaws, working against them, and their usefulness.

In late 2015, early 2016, The Defence Advanced Research Projects Agency (DARPA) hosted the 2016 Cyber Grand Challenge, with the theme of promoting and advancing automated computer security techniques, ranging from bug-detection to bug-squashing to hacking - all without interference from the teams who've created the software. Among the top three teams, in the final event, taking place August 2016 was Team Shellphish, who had created the Driller project: an extended version of American Fuzzy Lop, augmented by the concolic execution-engine, known as angr.

The idea behind this technology is to, through combining AFL with angr, mitigate as many of each of their respective drawbacks, while simultaneously utilizing many of their advantages, and even create new advantages, arising from the symbioses of the two techniques.

2.1 Problem Statement

Can American Fuzzy Lop be augmented with symbolic execution?

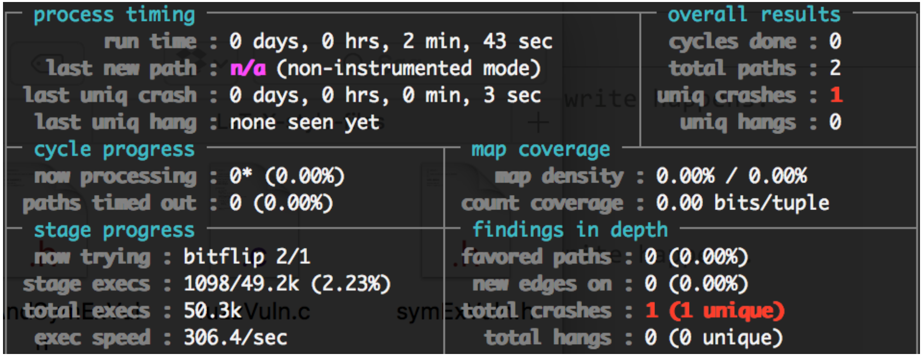
If yes, does the new tool pose an advantage, in the hands of the computer security, as well as the fuzzing society?

3 Team Shellphish

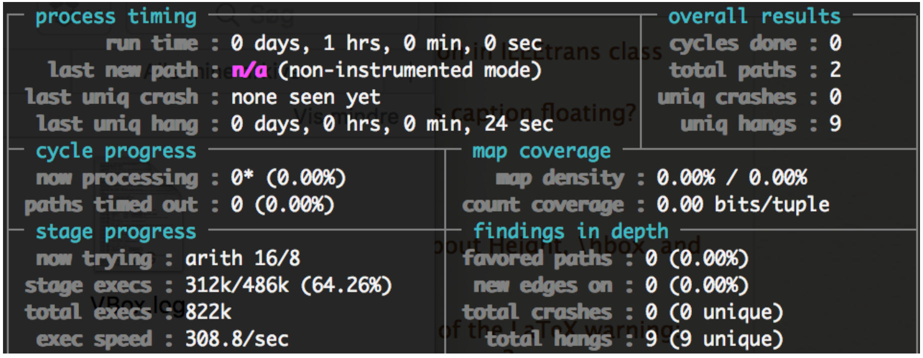
Team Shellphish, one of the top ranked teams in the Cyber Grand Challenge, is a self funded group of hackers, stemming from the secLab at University of California, Santa Barbara. The Team consists of Antonio Bianchi, Kevin Borgolte, Jacopo Corbetta, Francesco Disperati, Andrew Dutcher, Giovanni Vigna, Aravind Machiry, Chriss Salls, Yan Shoshitaishvili, Nick Stephens, Ruoyu Wang, John Grosen and Paul Grosen. In the DARPA Cyber Grand Chalenge, they have used a variety of tools, including angr and Driller, two applications, which team Shellphish have engineered themselves. After creating these tools, team Shellphish open-sourced all code and documentation they had gathered, in the name of computer security.

4 Background for the Problem Statement

This section will be dedicated to exemplifying the background of the problem statement. Fuzzing has worked well, since its origin, so why does it, all of a sudden, need to be reworked? To answer this question, I have written a small series of test programs, which demonstrate the problem. To cover a broader aspect, these programs, included both a fuzzable program as well as an unfuzzable program. The problem often comes to light in complex binaries, however, I have provoked the issue, only with 28 simple lines of code. The programs are shown in the appendix, and the result of the programs, being fuzzed by AFL can be seen underneath:



```
┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 2 min, 43 sec       │  cycles done : 0      │
│   last new path : n/a (non-instrumented mode)        │  total paths : 2      │
│ last uniq crash : 0 days, 0 hrs, 0 min, 3 sec        │ uniq crashes : 1      │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0* (0.00%)        │    map density : 0.00% / 0.00%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 0.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : bitflip 2/1           │ favored paths : 0 (0.00%)              │
│ stage execs : 1098/49.2k (2.23%)    │  new edges on : 0 (0.00%)              │
│ total execs : 50.3k                 │ total crashes : 1 (1 unique)           │
│  exec speed : 306.4/sec             │   total hangs : 0 (0 unique)           │
```



```
┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 1 hrs, 0 min, 0 sec        │  cycles done : 0      │
│   last new path : n/a (non-instrumented mode)        │  total paths : 2      │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 0 min, 24 sec       │   uniq hangs : 9      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0* (0.00%)        │    map density : 0.00% / 0.00%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 0.00 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 16/8            │ favored paths : 0 (0.00%)              │
│ stage execs : 312k/486k (64.26%)    │  new edges on : 0 (0.00%)              │
│ total execs : 822k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 308.8/sec             │   total hangs : 9 (9 unique)           │
```

5 American Fuzzy Lop

This program works by feeding random input to a program, at a very high rate, some of which will hit specific vulnerabilities in said input-program. Every input, fed is logged, and upon vulnerability-hit, AFL logs the input-ID. Hereafter information about where the vulnerability occurred, and which input triggered it is gatherable, based on input ID.

An advantage, as well as a drawback of most fuzzers, hereunder AFL is its execution method, which is to be as non-invasive as possible, as to prioritize speed before complexity handling. This means that AFL does not analyse a fuzzed application, but relies on instead directly executing the application with random input, which is immensely faster than mutating qualified input variables, based on an application analysis.

5.1 Features of AFL

AFL implements a variety of features, to enhance its efficiency. In this section, I will list some of the key features, offered.

Genetic Fuzzing

When stating that the AFL fuzzing engine relies on executing applications with inputs at absolute random, one is not totally correct. This is due to the technique known as 'Genetic Fuzzing'. Genetic fuzzing means that the engine generates - *unique* - input at total random. Simplified, this means that the current input, that AFL is generating cannot be the same as any previously generated input.

Stable Transition Tracking

AFL views the union of source and destination as a tuple of it's destination blocks. These tuples are prioritised, meaning that the tuples that cause the most different execution, compared to previous executions, are chosen first for future input generation.

Loop Bucketization

For symbolic execution engines and fuzzers alike, loops are complicated to handle, as looping potentially offers an added layer of complexity. The AFL fuzzer makes the following contortions, in order to avoid looping's added complexity, and path space requirements: When AFL detects that a triggered path contains a loop, it logs the executed loop iterations and compares this with previous inputs. The paths are grouped, based on the amount of iterations, and hereafter only *one* path in a group is selected for further fuzzing. Using this technique, the running time of executing slow loop-including paths is reduced to $O(log(N))$, as opposed to $O(N)$, if every single path should be explored.

De-randomization

When AFL meets a random number, mutating one input value might not yield the same result, as mutating the same input value immediately after. Because of this, AFL features de-randomization, which allows for it to force randomized variables, to consistently use a specific seed, which in turn allows for consistent input/output-relations.

5.2 Limitations of AFL

Because of the union of the above techniques and its general nature, AFL is able to quickly discover a wide selection of general vulnerabilities, meaning vulnerabilities, that are triggered by some *kind* of input. When vulnerability-triggers move past general input, and into the territory of general input AFL can potentially fall seriously behind.

```
int main(void)
{
    int x;
    read(0, &x, sizeof(x));

    if (x == 0x12345678){
        vulnerability();
    }else{
         ...
    }
}
```

Listing 1: A program that is difficult to fuzz

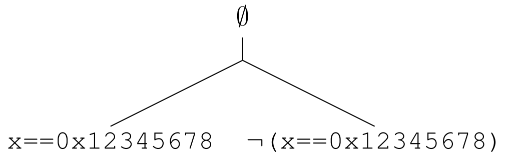
A generic example of this can be seen in Listing 1 above. This describes a program, that takes an input x from a user. If, and only if, x evaluates to `0x12345678` the program will fail, as a vulnerability will be triggered, and as so, at each command, executed by the fuzzer, the frequency, and by extension, the chance of discovering the bug, is 1 in 2^{32}. Furthermore, as AFL lacks the ability to produce new paths within this specific program, it lacks its instrumentation falls short, and AFL is reduced to dumb fuzzing, randomly mutating non-instrumented input.

6 Symbolic Execution

Symbolic execution, or symbolic evaluation, is a way of program analysis, designed in order to determine the different ways the program in question can be executed, and which type of input causes the different paths. Instead of executing actual values, to determine this, an interpreter assigns symbolic values to the input variables. The symbolic used to visualise how the program will execute, based on what *kind* of input it will be fed.

6.1 Features of Symbolic Execution

As symbolic execution relies on analysing input, instead of mindlessly executing input, it is able to detect specific inputs, which in this case, will cause the application to crash. See Listing 1 as an example. A symbolic execution engine will analyse it's functions, and generate the following tree:



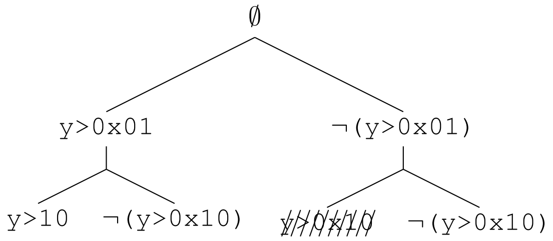
Another advantage of symbolic execution is the ability to invalidate sections of input.

```
int main(void)
{
    int x;
    read(0, &x, sizeof(x));

    if (x > 0x01){
        ...
    }else{
        ...
    }
    if x > 0x10){
        ...
    }else{
        ...
    }
}
```

Listing 2: Example of Symbolic Execution

In Listing 2 above, for example, the input `y` is evaluated. A symbolic execution engine will analyse Listing 2's formulae, to find that `y` will either assume a value greater than `0x01` or not greater than `0x01`. Furthermore `y` will either assume a value greater, or not greater than `0x10`, however greater than `0x10` cannot occur, if `y`, at the same time was not greater than `0x01`. This will produce the following tree:



Because of this trait, symbolic execution's relevance to the experiment is further heightened, as this allows for AFL to exclude certain value-ranges, when mutating input.

6.2 Limitations of Symbolic Execution

Program-Dependent Efficiency

The advantage of having a symbolic execution engine analyse paths, as opposed to input, is not present in all programs. If some inputs take the same paths in a program, the advantage in analysing the inputs instead of paths, is vanishingly small.

Environment Interactions

Often programs interact with their environment, such as executing system calls and/or receiving signals. This becomes a problem when these environment factors are not under the control of the symbolic execution tool, as the tool is not able to determine branching of a program, when it has insufficient data concerning the input it receives.

Path Explosion

The last, and possibly biggest drawback in symbolic execution engines is the path explosion problem. This problem is caused by a program that is containing loops, which causes the amount of paths to grow exponentially. In theory the path amount can even become infinite, because of unbound loops. This problem is near-not-existing in small programs, however the problem often scales faster than the symbolically executed program scales, rendering symbolic execution virtually useless for testing medium size to large applications.

7 angr: The Concolic Execution Engine

angr is a binary analysis framework, engineered modularly to be as versatile and composable as possible. Because of this, angr's use has potential to span wide, within the subject of binary analysis. Consider, for instance, just these few, but broad, uses of angr:

- angr can be used as a dynamic symbolic execution engine, which combined with value set analysis, allows to resolve bounds on symbolic variables.
- angr can be used as a static analysis engine, which also uses concolic tracing, in order to prove that detections are not false positives.
- angr can be used as a concolic tracer, along with a fuzzer, such as AFL, in order to allow for fully automatic bug discovery, such as what was done with Driller.

Driller would receive no benefit from invoking a regular symbolic execution engine, when stuck, as AFL cannot mutate input, based on symbolic values. Instead, it uses angr, an execution engine with the ability to mutate concrete values, based on the values found in the previous symbolic analysis-step.

angr, based on Mayhem [2], mutates concrete values via a four steps long algorithm. The steps can be broken down into the following:

1 The input (i.e. binary code) is translated into Valgrind's VEX [3] representation in order to determine the symbolic states of the binary code.

2 Symbolic values are set in place of all non-constant variables, such as user-defined input, randomized input, environment-dependant input, etc. Constant values are represented with the same concrete values, to which they are assigned in the program.

3 The values are, by execution, given constraints set up by the binary environment, as well as the concrete values described in step 2.

4 When the program reaches a new path or state, input values, which drive the program to that state is generated, based on the constraints described step 2.

At any point after the algorithm has run, the gathered concrete values can be fed to the binary code, which will trigger the binary to produce an output similar to the one, that a corresponding symbolic value has generated.

7.1 Diversity

One of the design goals for angr has been to tailor it to be as diverse as possible. This was done, as angr was not only designed for its use in Driller, but was meant to be reusable, by team Shellphish, as well as any other security enthusiast out there. Because of this, angr was aimed towards support for ARM and MIPS processors, as well s 64-bit processors, instead of the 32-bit processors of yesteryear.

In the same spirit as the above mentioned, regarding cross-architecture support, angr also has to be able to support input, stemming from multiple platforms. This functionality is provided by the binary loader, called CLE, which is described in further detail in section 7.2.

Furthermore, when creating angr, team Shellphish did so, not only to further their odds in the DARPA CGC, but also to advance the bug-discovering community, and as a result of this, the tool is available as a fully editable open sourced program, completely free to use for any non-commercial use.

7.2 The Binary Load Sub-Module

When angr loads binaries into the application, it is handled by the binary load module, by the name of CLE. CLE is named, jokingly and recursively as an abbreviation for the sentence 'CLE Loads Everything'.

CLE is, as the point of departure, designed to work in the same way as GNU LD does with ELF binaries. This means that CLE does not necessarily load all the information in a binary, as some of this may be stripped or corrupted. When this happens, the ignored information is encompassed in a Project class. The rest of the information is stored in a Loader class, an in turn the CLE loader is representing a conglomerate of binaries, loaded by CLE, which, after being loaded, is mapped onto a single memory space. CLE has a subclass/ loader for each file type.

Currently, CLE offers support through its backends for PE, CGC, and both ordinary and core dump ELF files. Furthermore, CLE is able to load binaries with IDA, as well as loading files into flat address spaces. Which backend a binary file should use is auto-detectable for all, except the binaries, being loaded into flat images, which do need to be specified.

8 Driller: Concolic Execution-Assisted Fuzzing

The core assumption, when team Shellphish designed Driller, was that 99+ % of input could be divided into two categories: *general* input, spanning over a broad pallet of values and *specific* input, which is only able to take on a select few forms. Further chasing this assumption, the Driller approach emerges, which sees binaries as a collection of compartments. A representation of this can be seen in Figure 1 below:

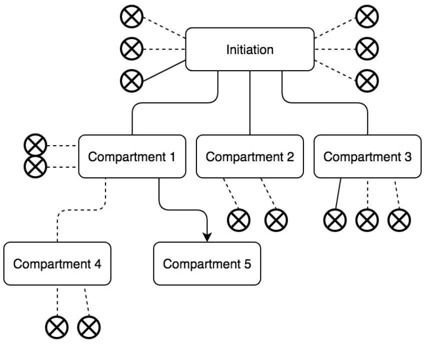


Figure 1: A graphic representation of binary compartments

In Figure 1, a path found by executing concolically is represented by a solid line, and a path found by means of fuzzing is represented by a dashed line. A bug is represented by ⊗.

The vast majority of bugs found in Figure 1 is found by the fuzzing component, because of its wide-spread nature. A few bugs are still discovered by the concolic execution engine, but as this technique is very slow, the task of bug-discovery is primarily done by AFL.

The task of discovering new components (i.e. paths) is primarily handled by angr. This doesn't mean that the fuzzer is unable to discover new paths, but the, sometimes very specific, input, required in order to move a program from one state to another is most efficiently handled by angr's concolic execution.

8.1 Input Preconstraining

The union of the AFL fuzzer and angr's concolic execution, offers more advantages than just the union of the two's functionalities. The possibility of preconstraining input, by the concolic execution engine is introduced, as the two components are passing along information to one another, before doing their respective tasks. Consider Listing 3 below:

```
1  int checker(bool *x, int limit)
2  {
3      if (depth >= 100){
4          return 0;
5      }
6      if (*x == 'TRUE'){
7          counter = counter + checker(x + 1, limit + 1)
8      } else{
9          counter = counter + checker(x + 1, limit)
10     }
11     return counter;
12 }
13 int main(void)
14 {
15     bool x;
16     int y;
17
18     read(0, &x, sizeof(x));
19     read(0, &y, sizeof(y));
20
21     if (checker(x, 0 ) == 50){
22         if (y == 0x12345678){
23             vulnerability();
24         }
25     }
26     return 0;
27 }
```

Listing 3: A program in need of preconstraining

This program would not be automatically de-buggable, by a fuzzer alone, neither a concolic execution engine, nor even a basic union of the two. A concolic execution engine would concede to the path explosion problem, when running the `checker` function, and a fuzzer would no be able to pass the check on line 23. Driller is not a basic union of these techniques, however, but a sophisticated collaboration. The information gathered by AFL is passed along to angr, which is able to constrain its concolic execution, as to bypass the path explosion problem completely. What happens is as follows: AFL initially fuzzes the program, which leads it to, but not further than, line 23. From here, it quickly realizes that it is stuck, as the possibility of "guessing" the correct value for `y` is approximately $\frac{1}{2^{32}}$. Next, Driller invokes the concolic execution engine, and as this happens, Driller constrains the bytes in the symbolically executed input to match the input traced from AFL. Hereafter, only one branch is followed, as the tracing of AFL's data only will allow for this, and the path explosion problem has successfully been mitigated. When the concolic execution reaches line 23, however, it/Driller recognizes an alternate state transition, which has not previously been discovered by the fuzzing engine. At this point, as the path-exploding step has been overcome, Driller removes most of its constraints, not including the ones, leading through the line 23 check, where after `y` is constrained, forcing it to assume one concrete values, based on the three unique symbolic values:

- `y < 0x12345678`
- `y > 0x12345678`
- `y == 0x12345678`

At this point, Driller is quickly able to determine that two of these values will execute without any errors, but also that a third one, `y == 0x12345678` will result in a crash.

8.2 Re-randomization

Another issue, presented towards fuzzing, is the occurrence of random variables. Checks regarding these, have the ability to be as unfuzzable and difficult-to-hit as checks with normal specific input, but with the added challenge of being unstable, as it is virtually impossible to determine the value of a random variable, before utilizing it. Below, a listing, showcasing an example of this problem is presented.

```
int main(void)
{
    int x;
    int rand;

    read(0, &x, sizeof(x));
    rand = random();

    if (*x == *rand){
        vulnerability();
    }
}
```

Listing 4: A program featuring randomness

Listing 4 represents a small program, which is difficult to fuzz in the same way as Listing 1: regarding specific input. Furthermore, if a crash *is* discovered here, a user have to manually observe the program output, in order to find what exactly crashed the program.

Luckily, this issue is easily handled by Driller. First of all, Driller's AFL component is very unqualified to handle the problem, and so forwards this to angr's concolic execution engine. The concolic execution engine breaks the input-variables into concolic values, as described in Section 5. Accordingly, this includes random variables, and as so, only three possibilities emerge:

- `x < rand`
- `x > rand`
- `x == rand`

Hereafter, angr quickly realizes that what triggers the found vulnerability isn't a specific value, assumed by the input variable `x`, but a specific *relation* between `x` and `rand`. Driller is hereafter able to use this principle, as instances of random-variable-checks occurs in real non-test binaries and programs.

8.3 The Algorithm

Input Test Cases

Driller has no direct need for initial input test cases, and in such an instance, where non is provided, the initial fuzzing-step will work similarly to normal AFL-fuzzing. If initial test cases are provided, however, the initial fuzzing step can often times be sped up, as a tailored presence of these can guide AFL towards known compartments.

Fuzzing

When Driller is initiated, it begins by having AFL fuzz the initial compartment of the binary, which Driller has received as input. Driller discovers bugs in this compartment, by means of ordinary instrumented fuzzing, but will eventually reach a complex check, rendering AFL stuck, and therefore virtually useless.

Concolic Execution

At this point, Driller invokes its concolic execution engine angr. angr cross checks and pre-constrains the user-provided input with the input provided by AFL, in order to avoid path explosion, after which it calculates, via its constraint solving engine, which inputs would lead execution towards a new path, towards a new compartment. If AFL has discovered, and exhausted sub-compartments before the invoking of angr's concolic execution, these already-discovered paths would represent flows of execution into new compartments.

Repeat and eventual halt

When angr is eventually done with analysing the binary, having found new execution paths, the inputs, that trigger these are passed to the "testcases" folder. When AFL's fuzzing component is again invoked, the new input will be there for future mutation. From here, AFL's state transition tracking quickly determines that the new input will result in a radically different output, which is why they are chosen first for mutation. This simple back-and-forth between angr and AFL is what, when it comes down to it, Driller is made up of, and the ping-pong repeats itself until either an input resulting in a crash is discovered, or the execution is aborted by the user.

8.4 Additional Advantages of Driller

The advantage of the combination of the technologies is vast. First of all the union of these two technologies are able to mitigate each of their biggest shortcomings (i.e. path explosion and specific input), by simply passing along the task, when a problem arises. Secondly, the information, gathered in angr can be utilized by AFL and vice versa. As such, angr does not only assist AFL in progressing deeper into the layers of the input binaries, but AFL further assists angr, with data, concerning executions, in order to avoid path explosion. Furthermore, angr's path explosion problem is additionally lowered, as angr does not necessarily have to analyse entire programs. Instead, because concolic execution with Driller only has the purpose to discover new paths, angr can analyse only a fraction of the program. This way, angr discovers only the compartments leading away from the current compartment, instead of every single state of the program at once, which will lead to a more smooth and uninterrupted fuzzing experience.

Because of these augmentations, Driller is able to work as not only an improved version of a concolic execution engine, but also like an improved version of AFL.

9 Testing

When testing Driller, Team Shellphish set two primary goals of achievement, as well as one secondary goal of documentation. The first achievement goal was to prove, primarily, that Driller is able to expand on the code coverage, offered by fuzzers, instrumented as described in Section 5.1, but not augmented by symbolic execution. The second achievement goal was, in continuation of the first one, that the increased code coverage would lead to a greater number of discovered vulnerabilities, than that of both traditional symbolic executers as well as fuzzers, such as American Fuzzy Lop.

The secondary goal was to, through these tests, showcase the findings, which would serve to prove the advantage of Driller, advertising the tool, and thereby promoting the scientific community with their discoveries.

9.1 Test Cases

As such, the test cases were not binaries, potentially riddled with previously undiscovered bugs, but rather a set of binaries with both know bugs, and an assured challenging variety of these. Thus, the set of binaries of the qualifying event of the DARPA CGC was chosen, as these were "designed to test the abilities of a new generation of fully automated cyber defence systems"[4].

The full set consists of 131 binaries, five of which, however, involve cross-binary communication, why these five binaries were excluded from this dataset. The remaining 126 binaries featured bugs, hidden well by complex protocols, specific inputs and data structures, and large input spaces. The individual binaries were specifically designed with the purpose to stress the programs, analysing them, as well as pose a serious challenge to the techniques used, regardless of the strength of the computer running the analysis. The reason, such a challenging set of binaries was chosen was, that if every vulnerability was to be discovered, the dataset would either have showcased Driller's capabilities precisely, not have shown all of Driller's capabilities, or worst case, not shown Driller's flaws, which would provide no basis for further expanding on the techniques, used by Driller.

Additionally, the results of the other contenders in the CGC is available online, which gave Team Shellphish the possibility of comparing their test results to those of the top contenders in the CGC.

9.2 Basis

There was run total of three experiments, which were chosen in order to provide the best basis comparing the techniques, used in Driller, and all of them were used to try to find bugs in every single one of the CGC Qualifying event binaries. The techniques were the following:

Basic Fuzzing

In this test case, a slightly modified version of AFL was used, and assigned four cores, intended for fuzzing. AFL was modified, but only its QEMU backend, and this was only altered as to reduce inefficiency, when fuzzing CGC binaries.

Symbolic Execution

In this test case, an existing symbolic execution engine was used, based on the thoughts described in the Mayhem paper[2], which were also the basis for angr's concolic execution engine. all of the techniques, described in section 6.1 were used, in order to reduce the path explosion problem. This problem was the only major solvable threat to the symbolic execution engine, seeing as the environment interaction problem, described in section 6.2 was of no concern because the challenge binaries each were at their own respective, right places.

Driller

In this test case, four cores were assigned to the AFL component, and 64 were assigned to the concolic execution component. The concolic execution compartment analysed jobs in a first-in-first-out queue, and a job was assigned to the component, when Driller decided that the AFL fuzzing nodes had become stuck. The symbolic execution traces were restricted, disallowing them to exceed both a one-hour period, and a four giga-byte memory size. This was done to avoid letting large traces to exhaust resources, required by the other components of Driller.

9.2.1 Hardware and limits

The experiments above were run on a computer cluster of AMD64 processors. Each of the binaries had four nodes assigned for fuzzing only that specific binary. Furthermore, 64 nodes were assigned for symbolically executing binaries in need of assisting when the fuzzing component became stuck. The 64 concolic execution nodes were shared across all binaries, and limited to the memory use of 4 gigabytes RAM, as well as a running time of maximum one hour.

The experiments ran for 24 hours. If a crash was detected in a binary, that binary was considered done, and if not, it would continually be analysed by Driller until the 24 hours had ran out.

This time frame was made, in order to better be able to compare the test results with the test results from the Cyber Grand Challenge.

9.3 Results

Of the three techniques, Driller proved to present a clear advantage over both symbolic execution and basic fuzzing. The ranking was the following:

1: Driller, with crashes found in 77 of the 126 binaries.

2: Fuzzing, with crashes found in 68 of the 126 binaries.

3: Symbolic execution, with crashes found in 16 of the 126 binaries.

Out of the 16 crashes, found by symbolic execution and 68 crashes found by fuzzing, every single one was found by Driller. The relationship between the crashes found by the three techniques can be seen below, in table 1.

Driller	77
Fuzzing	68
Symbolic Execution	16
Driller ∩ Fuzzing	68
Driller ∩ Symbolic Execution	16
Fuzzing ∩ Symbolic Execution	13
Driller - (Fuzzing ∪ Symbolic Execution)	9

Table 1: Crashes, distributed between symbolic execution, fuzzing and Driller

Table 1 above shows the distribution of crashes, between symbolic execution, fuzzing and Driller. It states that the total number of crashes, discovered by fuzzing and symbolic execution, even when combined, cannot reach the total number of crashes found By Driller. The remaining crashes found by Driller sums to a total of 9 crashes, which equates an improvement at about $7,59\%$, when compared to the other 68 crashes. These additional 9 crashes, and the fact that neither fuzzing, nor symbolic execution discovered them, must lead to the conclusion that the additional advantages, described in section 8.4 had the significant effect of increasing the code coverage to the point of being able to discover these new kinds of bugs.

Driller might have been able to discover more bugs, had it not been for the restrictions, described in section 9.2, however, it was decided that 1. the resources were better spent elsewhere, and 2. because of this, the techniques were not deemed sufficient for finding these bugs, with the standard of today's computing power.

9.3.1 Shown strengths of Driller

Concolic invocations

Driller's key advantage, when compared to non-hybrid bug-discovery techniques is the ability to make concolic invocations. The vast majority of the bugs, discovered by Driller is made by the AFL fuzzing component, and so, the only real job of the concolic execution engine is to escort the fuzzer to new, and previously "undiscovered" components.

Despite the concolic execution engine, angr's, very minimal role in the actual process of bug-discovery, it is the very core module, in regards to what separates Driller from most of other bug discovery tools out there. As seen in Fig 7 [1, p. 11], a lot of the discovered crashes were actually found without even needing the invocation of concolic execution. The fact that the biggest amount of crashes discovered by Driller were done so, unassisted by angr, does not mean, however, that the intervention of angr is insignificant, as some otherwise would assume. Rather, angr is what allows for Driller to receive the benefit of what is described underneath:

Increased Code Coverage

Increased code coverage is the bi-product of two ingredients:

State Transition Coverage

Firstly, the ability to actively seek out transitioning to other program states (or compartments, as referred to in figure 1) is far superior to basic fuzzing, where this is not an option. The ability to *determine* that AFL is stuck, and probably won't find any more bugs, at the block it is currently, and invoking the concolic execution engine, which, according to Table I in the Driller Paper [1, p. 12] helped discover new blocks in 57 % of the cases of when it was used.

Application Component Coverage

Secondly, what this leads to an the increase in application component coverage. Driller, having excellent application component coverage basically means, that it is able to efficiently cover a bug portion of the compartments, possessed by binaries. Fig. 8. in the Driller paper [1, p. 11] describes a small selection of binaries, being analysed by Driller, each of these featured crashes, which Driller was able to discover, due to its concolic execution. This figure describes how, when executing concolically, new block transitions were found each time. Furthermore the percentage of the transitions, distributed between the two techniques is shown. By looking at the figure, it is clear to see that fuzzing makes out the greatest number of block transitions. This was what was expected, before testing Driller, as the few concolic execution invocations only served to open up into new compartments, which would allow for the fuzzing component to make new discoveries elsewhere.

Combined, these two factors allow for a wider code coverage, as mentioned a couple of times already, which means that the fuzzing component isn't fuzzing alone any more, fuzzing randomly, until, if lucky, a new compartment is discovered. As so, when Driller is covering more code, it is able to discover a significantly greater number of bugs, than both unassisted fuzzing, as well as unassisted symbolic execution is able to on their respective own.

9.3.2 Shown weaknesses of Driller

Though most of what is discussed in this report states the contrary, Driller is not without its fair share of limitations. One of these comes in the form of a bi-product of one of its greatest advantages. The strength in question is the one, concerning preconstraining input, described in section 8.1. In most cases this is greatly advantageous, as it allows for the AFL component of Driller to interact with the symbolic execution component of Driller, allowing the lastly mentioned to skip, otherwise slow and tedious symbolic invocations, if Driller had previously deemed it unworthy of further analysis, based on AFL's discoveries. Where this technique falls short, however, is when AFL, given its broad and general nature of testing, fails to determine that a given path is actually indeed interesting, and denies the symbolic execution component of this.

```
 1 int strcmp(char *a, char *b){
 2     for ( ; *a; a++; b++){
 3         if (*a != *b){
 4             break();
 5         }
 6     }
 7     return 0;
 8 }
 9
10 int main(void)
11 {
12     char x;
13     char y = "fine"
14     char z = "notFine"
15
16     read(0, &x, sizeof(x));
17
18     if (strcmp(*x, *y) == 0){
19         return 0;
20     } else if (strcmp(*x, *z) == 0){
21         vulnerability();
22     }
23     return 0;
24 }
```

Listing 5: An example of preconstraining being a disadvantage

An example of this is given in listing 5 [1, p.14]. The check at line 18 will in the far majority of cases be satisfied by Driller. As each branch, derived from the check on line 18, relies on `strcmp`, the AFL component of Driller isn't able to evaluate which state transitions `strcmp` will lead to. This is calculated before the check, and so, the check will always be met, which in turn deprives Driller from discovering the crash at line 21.

Another example where Driller fall short is when the tool encounters a combination of both specific and general input, by the same input value.

```
 1
 2  int main(void)
 3  {
 4      int x;
 5      int y = 0;
 6      int z = 0;
 7
 8      read(0, &x, sizeof(x));
 9
10      if (*x == 0x12345678){y = 1;}
11      if (*x != 0x87654321 AND pathExplosion()){z = 1;}
12      if (*y + *z == 2){
13          vulnerability();
14      }
15      return 0;
16  }
```

Listing 6: An example of Driller being outmanoeuvred by specific and general input

An example of this can be seen in the small program, listing 6. This program takes an input of x, which is exposed to a check at line 10. This check requires that x is *exactly* 0x12345678, and so, as Driller is here dealing in specific input, its symbolic execution engine is invoked. Therefore, this check is quickly, and painlessly dealt with. On line 11, however, x is subjected to yet another check, however, and this check makes it harder for the symbolic execution engine to progress, as this will create a path explosion, and thus, Driller is unable to discover the crash at line 15.

As such, the majority of Driller's issues show themselves, when the fuzzing component is being "disabled" by the program, being analysed. The mitigation of this issue will be further discussed in section 10.2.

10 Discussion

Another basis this paper could have taken as its point of entry was, could be any of the other tools, by the qualifying teams in the DARPA Cyber Grand Challenge, specifically the Mayhem system, by ForAllSecurity of Pittsburgh, Pennsylvania. One of the main reason for choosing Driller was that team Shellphish were the first team to open-source their tool, in order to advance the computer scientific community. Furthermore, this allowed for a more in-depth analysis of the tool, and more importantly, the techniques, advantages, and synergies, featured by the tool.

Moreover, Mayhem did not defeat Driller, the program, as such, despite the fact that ForAllSecurity placed first, while Shellphish was ranked third, in the CGC Final Event (CFE). This is because, that the CFE was a match of capture the flag, as opposed to bug discovery. Driller did, however, with some modifications after the Cyber Grand Challenge, manage to crash 77 of the binaries, which amounts to the exact same score as the, eventual, winning team, ForAllSecurity.

10.1 Conclusion

In this report, Driller the symbolic execution-assisted fuzzing engine was further analysed and discussed, based on team Shellphish' findings. What can be concluded is, that assisting fuzzing with symbolic execution does indeed present a significant advantage for the fuzzing community, this advantage being the beginning of the journey towards automating fuzzing. Normally, when using a fuzzer, you, the user, is able to determine when it gets stuck, and from there, decide which way, if any, you want to take the fuzzing tool to next.

Though Driller has a relatively high requirement, in terms of time and computing power, Driller is able to, once set in motion, run autonomously, freeing potential manpower to do other work, while Driller, itself discovers, not just shallow bugs, residing in the top-few compartments of the analysed binaries, but deep and complex bugs, hidden by up to several specific checks on input.

This doesn't mean that this current state-of-the-art technique, which has been developed no more than dating a year back is already flawless, however. First of all, as stated in section 9.3.2 hindering it from being a universally "perfect"[1]bug-discovery tool. Second of all, there were still a significant number of binaries in the dataset, in which Driller was not able to discover any bugs - 49 to be exact, and though extending any tool to capable of this is by no means any small feat, doing at least this is one of the necessary steps in order to create the "perfect" bug discovery tool.

10.2 Future Work

As team Shellphish has open-sourced the application, the potential future work is available to be done by everyone with the know-how, resourcefulness and not least, the gumption. As computer security is still increasing in importance, regarding today's society, the relevancy of this topic is immense, and the more people working on advancing the "good" non-invasive side, the better.

Both regarding Driller, and the topic in general, there is room for improvement. One example of future development of Driller is the problem, described in section 9.3.2, regarding issues arising, when either angr's concolic execution engine, or the fuzzing had been set out of motion. A potential fix to this problem could be to, instead of a symbolic execution engine (which in turn allows for concolic execution), to create input generators. An input generator is a semi-symbolic execution engine, creating constraints to the input, which the fuzzing component mutates upon. This would happen, based on the same principle as Driller operates on now, but at a much lower level. Instead of having concolic execution invocations, AFL would work concurrently with angr, creating *one* bug-discovery technique, as opposed to two symbiotic techniques.

Second of all, Driller expands upon the discussion of enhancing fuzzing with all sorts of techniques. angr, the heart to the project itself, is open sourced, and as described in section 7, it is composed to be as modular and diverse as possible, making its broad uses virtually endless.

[1] Identifying if a program will *crash* is inherently the same as identifying if a program will *stop*, which as proved by the halting problem is logically impossible to do in **every** case

References

[1] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishivili, C. Kruegel, G. Vigna, Driller: Augmenting Fuzzing Through Selective Symbolic Execution, UC Santa Barbara, 2016.

[2] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, Unleashing Mayhem on binary code, In Proceedings of the IEEE Symposium on Security and Privacy, 2012.

[3] N. Nethercote and J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Volume 42, pages 89–100, ACM, 2007.

[4] DARPA Cyber Grand Challenge. http://cybergrandchallenge.com.

[5] Y. Shoshitaishivili, R Wang, C Hauser, C Kruegel, G Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware, 2015

[6] American Fuzzy Lop, http://lcamtuf.coredump.cx/afl/

[7] Infosec Institute, `http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation`

[8] angr, N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishivili, C. Kruegel, G. Vigna, Driller: Augmenting Fuzzing Through Selective Symbolic Execution, `https://www.gitbook.com/book/angr/angr/details`

11 Appendix

```c
//
//  fuzzVuln.c
//
//
//  Created by Soeren Lund Jensen on 17/05/2017.
//
//  Purpose: A program, to test fuzzing.

#include "fuzzVuln.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int int1[32];
    div_t d;
    //Take input of an integer

    scanf("%i", int1);
    //Every 5000th input will result in a crash
    d = div(*int1, 20000);

    if (d.rem == 0) {
        //Crash here
        abort();
        return 0;
    }

    //Don’t crash here
    return 0;
}
```

Listing 7: fuzzVuln.c

```c
//
//  symExVuln.c
//
//
//  Created by Soeren Lund Jensen on 15/05/2017.
//
//  Purpose: A, uncrashable by fuzzing.

#include "symExVuln.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int int1[32];
    //Take input of second integer
    scanf("%i", int1);

    //Every 2^32th input will result in a crash.
    if (*int1 == 123456789){

        //Crash here
        abort();
        return 0;
    }

    //Don’t crash here
    return 0;
}
```

Listing 8: symExVuln.c