SAARLAND UNIVERSITY

CISPA Helmholtz Center for Information Security
Department of Software Engineering
Bachelor's thesis

Effective Search Algorithms for Grey Box Fuzzing

Kai Greshake
Bachelor's Program in Cybersecurity
August 2019

Advisor:
Sascha Just, CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Supervisor:
Andreas Zeller, CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Submitted
16th August, 2019

Universität des Saarlandes
66123 Saarbrücken
Germany

Statement in Lieu of an Oath:
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, 16th August, 2019

Declaration of Consent:
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 16th August, 2019

Acknowledgements

I would first like to thank my thesis advisor Sascha Just as well as my good friend Jens Heyens for their support and counsel during the time of writing. I am also grateful to all the other people at CISPA who I've worked with over the past few years and who taught me how to do research.

I would also like to extend my gratitude to my parents who were always there for me in times of hardship.

Abstract

Fuzzing is a proven technology that is deployed all over the industry to find software bugs. In contrast to techniques like static analysis or handwritten unit tests, fuzzing is autonomous in its search for bugs. In 2018 a new approach to fuzzing was presented by Peng Chen and Hao Chen [3] with their software Angora, using an optimization algorithm to achieve remarkable performance on common fuzzing benchmarks. In this study I benchmark various different optimization algorithms on their fuzzing capabilities, to find that constraint solving with optimization might not have been the key to Angoras performance.
Nonetheless results show promising directions for future research and will expand the performance and use cases for fuzzing in the future.

Contents

List of Figures

1 Introduction

1.1 Fuzzing

Classical software testing methods in computer science require a lot of manual labor. Unit tests are the most commonly known example and have been a cornerstone of quality control in software projects for decades. Unfortunately, it can be very expensive to write tests for software manually, tests which are often incomplete and do not cover important edge cases. Especially when many components come together to create a large piece of software, keeping track of all the interactions and edge cases is very difficult for human operators. As such, software testing and validation can likely consume up to 60% of the development budget for large projects [8]. Fuzzing is an inexpensive way to augment manual testing. It involves providing the target software with inputs that trigger unexpected, buggy or novel behavior. The ultimate goal of a fuzzer is to find inputs which cause the target program to crash. These inputs and crashes can then be checked by human operators who can check if the program conforms to specifications or if faulty behavior has been triggered.

The simplest fuzzer is a so-called monkey tester that produces random inputs. Of course, this naïve approach does not yield the highest number of bugs per execution. Random testing is blind and unstructured, thus unable to explore the search space efficiently [9].
Besides random testing, many other fuzzing techniques emerged in recent years ([28], [22], [17], [11], [21], [27], [27]). These are categorized based on their knowledge of the target program and input structure [9]. Knowledge is the key to making fuzzers more efficient. Take, for example, an image decoding library. Simply knowing what the structure of accepted inputs (images) is, greatly constricts the search space and allows us to explore much more behavior with fewer executions by generating random, valid image files. Many fuzzers use formal

grammars to utilize and organize information about the input structure.
Unfortunately, input grammars can be difficult to create or obtain while fully capturing the parser implementation at hand. There are some approaches to automate this, but I am going to focus on a technique that does not require grammars. When grammar-based fuzzing is not possible, it becomes more important to focus on knowledge that can be obtained from the target program by executing it. Fuzzers that do not observe the behavior of the target are called black-box fuzzers.
All other fuzzers are called grey- or white-box depending on the amount and type of information collected while executing the target program. This distinction is not always clear, but the most white-box approach is considered symbolic tracing, where the fuzzer obtains a full representation of the logic and arithmetic data processing that occurs inside the program. This information allows more precise selection/generation of new inputs at the cost of speed, but it is subject to the limitations of modern constraint solvers. Complex processing in many programs prevents white-box fuzzers from reaching the deeper parts of a program, when the number of sequential constraints exceeds the solving capacity of state-of-the-art solvers [12].

1.2 Evolutionary Grey-Box Fuzzers

The middle ground is represented by grey-box fuzzers [10] which use some form of feedback mechanism, but still heavily rely on random chance to advance their exploration. One of the most influential grey-box fuzzers is American Fuzzy Lop (AFL) [29]. The core idea behind AFL is to start with any input and use random mutations to trigger new behavior. When new behavior is triggered, AFL measures it as a change in coverage. Inputs that trigger new execution paths are kept for the next round of mutations. This way complex input structures arise over time as mutations accumulate [10].
The only requirement to use this evolutionary approach is to have a coverage feedback mechanism. In the case of AFL, the target program is compiled with a special instrumentation pass which inserts instructions at every basic block that flip a bit in a shared-memory array when executed. This array is called the coverage map and does not always allow AFL to reconstruct exact execution paths. Instead, it is mostly used to determine a change or increase in coverage. For performance reasons, this array has a fixed size of (commonly) 64kB to fit in a Central Processing Unit (CPU)'s L2 cache. Hence there can be target programs which have more branches than the coverage map has bits. If two different branches flip the same bit twice it is called a collision. By observing the coverage map density (percentage of bits flipped) we can decide while running if the chosen map size is sufficient for the target program.
The mutations that AFL performs are carefully tailored by hand to trigger edge cases and bugs. Common mutations include arithmetic operations, min/max values and bitflips. Changing the length of the input by trimming, splitting and

```
1  int main() {
2      char buffer[4];
3      char* password = "ABCD";
4      read(STDIN_FILENO, buffer, 4);
5
6      if (!strcmp(buffer, password)) {
7          printf("Bug hit!");
8          abort();
9      }
10     return 0;
11 }
```

Figure 1.1: Simple program with a password check.

insertion are also common operations.
To understand the limitations of AFL, let us look at an example.

Figure 1.1 contains a simple program which reads four bytes and checks if they equal a fixed string. AFL has to rely on random mutations to produce the exact string required to pass the check. The limitation of AFL in this case is feedback granularity. Because AFL can only tell whether or not the second branch was triggered, it does not recognize an input that may be closer to the password than the previous iteration and throws any progress away. Such cases are called "magic bytes" because a specific, "magic" combination is needed to unlock the next branch. Magic bytes are not the only hurdle for coverage-only based fuzzers. There are many more complex, multi-byte calculations that require more granularity than coverage-based fuzzers can offer.
Important: Constraints. A constraint is a formal restriction that allows only inputs that satisfy it to trigger the branch. In the case of Figure 1.1, the constraint for reaching line 7 is: buffer[0:3] = 'ABCD'. Constraints can be arbitrarily complex and encapsulate the behavior of the program. While the goal of fuzzing is to find bugs, a surrogate goal is coverage. If we can generate inputs that reach more parts of the program, we have more surface area to find and trigger bugs. Solving these constraints efficiently thus becomes a main goal in fuzzing.

1.3 Angora

Angora is a fuzzer that shares a lot of ideas with AFL, but builds on them in important ways. First and foremost, Angora has much more granular feedback. Secondly, it utilizes that extra information very efficiently to guide the mutations in a promising direction. This allows the fuzzer to slowly work its way up to solving complex constraints, instead of solving them by chance in a single mutation.
Angora is implemented using Rust, a relatively new low-level language that uses the concept of data ownership to ensure memory safety while requiring no

Comparison	f	Constraint
$a < b$	$f = a - b$	$f < 0$
$a \leq b$	$f = a - b$	$f <= 0$
$a > b$	$f = b - a$	$f < 0$
$a \geq b$	$f = b - a$	$f <= 0$
$a == b$	$f = abs(a - b)$	$f == 0$
$a \neq b$	$f = -abs(a - b)$	$f < 0$

Table 1.1: Converting comparisons to black-box functions. From [3].

manual memory management or runtime garbage collection. [15]

1.3.1 Feedback Granularity

Angora has three distinct feedback mechanisms. First of all, the coverage feedback allows us to detect when a branch was executed in a different context than previous executions, which can imply new behavior. The coverage consists of the number of executions of a branch and a calling context, which is a hash of the call stack. This increases the amount of distinguishable branch executions by a factor of about 7 [3] and assists the evolutionary generation of new inputs.
The second mechanism is byte-level taint tracking. Angora compiles the target program twice: Once with lightweight coverage instrumentation ("fast" version) and once with taint tracking. Taint tracking allows Angora to reduce the number of bytes that need to be mutated to influence a single conditional (or branch). It assigns so-called shadow labels to all input bytes and tracks which conditionals are influenced by which bytes. This incurs a heavy performance penalty during execution, but the taint version of the target only needs to be executed when new branches are discovered. In this hybrid way, taint tracking can greatly improve the efficiency of solving constraints by reducing the number of bytes that have to be manipulated most of the time. Furthermore, the taint tracking algorithm groups bytes that are used in a single variable, say a 4-byte integer, thus reducing the number of input dimensions further.
Third, and most importantly, the fast version also includes feedback about the state of conditionals. This allows us to track our progression when trying to solve specific constraints. Almost every comparison operation in a program boils down to a comparison of two numbers. With some simple transformations Angora is able to calculate a numeric value that describes how close that comparison is to being true. Since Angora knows which bytes influenced the values in the comparison, we can view the comparison result as the output of a black-box function $f(\cdot)$ which takes a vector of input bytes x and outputs a value representing how close the comparison is. Doing this requires transforming the comparison into an arithmetic expression and a constraint on the output.

Angora uses the reference table Table 1.1 to transform any comparison into a function value and a constraint. Minimizing the value of $f(x)$ becomes the way

to solve the constraint.

1.3.2 Constraint Solving with Optimization

Angora uses this generic representation of constraints and conditionals to help guide its mutations. Due to the increased granularity of feedback, Angora can distinguish when an input has brought it closer to solving a new constraint. This means that more small mutations can compound over time and eventually leads to solving complex conditionals.
How these black-box functions can be navigated efficiently will be the subject of this thesis. The algorithm used to solve constraints is called an optimization or search algorithm, since its task is to explore the input space and find the most optimal values to obtain the minimum output (optimum) of $f(x)$. Angora uses an algorithm called Gradient Descent to perform this optimization, more on that in section 2.2. Using these changes, Angora was able to outclass the performance of state-of-the-art fuzzing competitors like AFL, SES, Steelix and VUzzer [3].

2 Background

2.1 Optimization Algorithms

Optimization has been a topic of interest for mathematicians and computer scientists alike. For the former, the quest to efficiently solve optimization problems goes back centuries, to at least the times of Newton who developed an iterative method to approximate roots. In its most basic form, performing an optimization means searching for the best set of values that fulfill certain criteria in regard to an arbitrary objective function [1]. To capture the essence of optimization in the context of this work more precisely, I think the term search problem is more tangible, and I will refer to it as such in this thesis. The reason is that we are not truly looking for the absolute minimum values of the constraint functions, but instead we are interested in finding an input that is sufficient to satisfy the conditional, if it exists.

Optimization problems can be very diverse, either in the structure of their search space, constraints or the objective function itself. In the case of fuzzing with optimization, the objective function as well as the search space have multiple unfavorable properties that I will discuss in detail in section 2.3.

2.1.1 Problem Statement

Stated more formally in regard to the problems explored in this thesis (building on the definitions found in [1]):
Taint tracking yields a conditional tuple $C = (\mathcal{X}, f, x_0)$ where $\mathcal{X} = \mathbb{B}^{k_0} \times \ldots \times \mathbb{B}^{k_n}$ is the input or search space, $f : \mathcal{X} \to \mathbb{N}$ is the objective function and x_0 is the starting value which is guaranteed to fulfill any precondition and can reach the conditional being explored. Also, $\mathbb{B} := \{0, 1\}$ and n represents the number of input values that were found during taint tracking. Each value has its own k_i

which is the number of bits that comprise the value, e.g. $k_0 = 64$ in case the first value is a 64-bit integer.
When an input $x \in \mathcal{X}$ is selected that causes the program to prematurely exit or the execution trace does not include the target conditional for whatever reason, we define $f(x) := 2^{64} - 1$, representing the highest value the function can obtain with a 64 bit integer. This way any input that is not defined on the function will be considered worse than any input that reaches the conditional.

The goal is to find an input vector $x = (x_0, \dots, x_n)^T$ s.t. $f(x) = 0$.

Unfortunately, every input dimension can have a different size and assume different values. Furthermore, every value can be interpreted as signed or unsigned by the target program, which can cause us to misjudge the geometry of the search space. Take this example: $C = (\mathbb{B}^4, f, x_0 = (\text{0xF})), f(x_0) = 1$. Let the optimal value in this case be $x_{\text{opt}} = \text{0x1}$. If we had assumed the value is unsigned, we would assume at the start that the solution should be close to the starting point because the objective function is very close to being 0. But the distance, or difference in this one-dimensional case, between x_0 and x_{opt} is 15, the maximum distance possible for this search space. That is because the value is interpreted as signed, and the value of $x_0 = -1$. In a signed search space the values are geometrically close and easier to locate for a search algorithm. Luckily, we get the information about size and sign of the values from Angoras shape inference algorithm. For simplicity, we will ignore these issues when exploring the different algorithms and talk about an $\mathbb{R}^n$ search space, but the constraints on length and sign will be added to the calculations.
Since we are working with a grey-box fuzzer, we do not have access to an analytical description of f.

2.2 Angora: Gradient Descent

Angora used the Gradient Descent (GD) algorithm to solve these problems. GD is commonly used to optimize the weights of neural networks ([20], [1]). It is a simple algorithm that tries to minimize the objective function by traversing it along its inverse gradient $-\Delta f(x)$. This usually requires a continuous input and output space as well as a differentiable function, but Angora approximates a gradient numerically by calculating the gradient of each dimension separately: [3]

$$\frac{\partial f(x)}{\partial x_i} = \frac{f(x + \delta v_i) - f(x)}{\delta}$$

Since δ is either 1 or -1 in the implementation of Angora and v_i is the canonical basis vector with a 1 in the ith dimension, we can simplify this to:

$$\frac{\partial f(x)}{\partial x_i} \approx \delta(f(x \pm v_i) - f(x))$$

Angora first attempts to perturb the input vector in the positive direction to obtain the gradient, but if the input is not accepted by the target, it will attempt to obtain a gradient value by perturbing in the negative direction. Those values will not necessarily agree on the gradient, but both are equally valid approximations with the smallest possible measurement size in our search space. If both perturbations yield no usable value for f, the gradient in that dimension is set to 0 to instruct the algorithm not to move in that direction.

Algorithm 1 The gradient descent algorithm used by Angora with slight modification (omission of one-dimensional search heuristic)

procedure SOLVEGD($\mathcal{X}, f, x_0$)
  $x \leftarrow x_0$
  while $f(x) \neq 0$ **do**
    $\Delta f \leftarrow$ ApproxGradient(x)
    $\epsilon \leftarrow \frac{f(x)}{\sum_{k=1}^{n} \Delta f_k}$ ▷ The step size is chosen at the start based on a heuristic
    if $\sum_{k=1}^{n} \Delta f_k = 0$ **then**
      $x \leftarrow$ random()
      continue
    end if
    while $\epsilon > 1$ **do** ▷ Follow the gradient as long as f(x) improves
      stuck $\leftarrow$ False
      $x_0 \leftarrow x$
      $x \leftarrow x - \epsilon \Delta f(x)$
      if $f(x) < f(x_0)$ **then**
        $\epsilon \leftarrow 2\epsilon$ ▷ Step size is doubled when improving
      else
        $\epsilon \leftarrow \epsilon/2$ ▷ And halfed otherwise
      end if
    end while
  end while
  return x
end procedure

The reason this works can be explained intuitively and elegantly by thinking of a ball rolling down the slope of a hill. By following the curvature around it, the ball will always end up in the valley. But this brings some issues, too. What happens if the valley the ball ends up in is not the deepest point? In terms of fuzzing this means that GD might produce an input that is a local minimum but not sufficient to trigger the conditional. This is a very real problem and plenty such functions exist. In that case we need to recognize that the gradient is zero or that the next value is less optimal and select a new x_0 at random.
In fact, the objective functions that are encountered during fuzzing exhibit many very undesirable features that GD does not deal with adequately.

The efficiency of this algorithm is also heavily dependent upon the number of input dimensions. To estimate the gradient once, we need between n and $2n$ executions of the target program (function f), which is becoming significant when conditionals start having hundreds of input dimensions. I will later present an alternative algorithm that uses only 2 executions per gradient estimate.

2.3 Optimization Challenges

Let us review the fuzzing-specific problems with using optimization for these problems in full, since these heavily influenced the choices I made w.r.t. the selected algorithms.

- No analytic expression of the objective function.
- Expensive: We have to execute the target program every time we try to obtain a value for $f(x)$.
- Non-differentiability: We can only approximate a first-order gradient at any given point.
- Non-convex: The objective function may not be convex.
- Non-linear: The objective function may not be linear.
- Undefined: The objective function may not be defined on some inputs; For example when the input causes the program to take a different branch early on, so we can't get a value for $f(x)$.
- Discontinuous: The objective function may contain hard "edges" and plateaus in the surface, which causes gradient methods to fail.
- Local optima: The objective function may contain any number of local optima which causes local search algorithms to get stuck before finding a sufficient optimum.

Basically, we can make no guarantees about the geometry of the objective function, which makes sense since it can encode the behavior of arbitrarily complex programs. Take, for example, a simple hash function. When the input bytes get hashed before they flow into the conditional it breaks our ability to infer any useful information from the gradient, since wherever we measure the gradient, we will get a pseudorandom result. Although we do not get any guarantees, we can make assumptions about **most** cases, where f may even be linear. In case of the hash function, the GD algorithm would simply behave like a random search.

2.4 Optimization: Local vs. Global

In optimization we need to consider trade-offs. Gradient descent performs local search and will always quickly find the optimum of any convex function. We can also deploy algorithms that are more robust against local optima and perform what is called a global search, but we will have to trade performance. Whether that trade-off is worth it depends on the target programs. If they are simple and most constraints contain only one optimum which is easily reachable, there is no point implementing a more complex algorithm. But we could also hit a coverage ceiling with such an algorithm which would require more sophisticated approaches to solve key constraints. So far there is no data to know whether local or global search approaches perform better on real-world programs.

3 Research

3.1 Research Goals

The goal of this thesis is to determine the influence of the chosen optimization algorithms on Angoras performance. In their original paper, the Angora authors did not mention comparative work to find out which part of their improvements was instrumental to the success of Angora. I expected that the choice of algorithm has a large influence on performance and that their choice (GD) was not optimal considering the many issues that may arise with local optima and efficiency. It may also be the case that different algorithms perform much better only on specific domains, for example many input dimensions. To find out if this is the case, we define the following key research questions:

1. Which algorithms are most likely to perform better than GD and should be considered for this study?
2. What performance metrics can be used to analyze and compare the performance of these algorithms?
3. How is Angoras performance influenced by the choice of algorithm?
4. Do the chosen algorithms perform better than GD?
5. How does the performance scale in relation to input dimensions?
6. Should the algorithm be chosen depending on the target program?

3.2 Contributions

This work may improve the performance of state-of-the-art fuzzers and allow others to determine if investigating this approach is worth it. It can also provide

a baseline and direction for future research. All of the algorithms will be released as open source software to the Angora GitHub project at some point.

3.3 Challenges

Fuzzing approaches are notoriously difficult to compare and evaluate due to the naturally high variability of results. The progression of any fuzzer relies on the discovery of breakthrough inputs that allow the fuzzer to access more parts of the target program. Some of these breakthroughs will only be made every few runs and distort the coverage data, sometimes inflating the coverage and sometimes being stuck. To deal with variability, I will increase the number of runs per target until we have reliable data.
There are also a lot of heuristics and parameters that could be tuned on each of the algorithms and even Angora itself. They make it harder to draw clear conclusions but are also essential to allow the algorithms to compete. To make the comparison as unbiased as possible, I will disable many unrelated features of Angora in order to compare the algorithms directly.

4 Algorithms

There are many different algorithms that are potential candidates for this study, but unfortunately there is only a limited amount of time to implement and test them. Since the goal of this study is not to determine an all-out best strategy/algorithm for this problem but instead a general overview of the relationship between optimization performance and fuzzing efficacy, I have decided on the following algorithms.
Each of these algorithms has some unique properties that could be useful for fuzzing based on the difficulties described in section 2.3. I will describe why I think the selection I made is particularly interesting in a fuzzing context, including the disadvantages and advantages of each approach.

4.1 Control: Random

To ensure that we are actually performing optimization, we need a random search algorithm as a baseline for performance. If the presented algorithms have any effect at all, their performance measurements need to significantly differ from this baseline. I first tried the random search that is already available in Angora, but I found out that it actually does not behave like the random search I was looking for. Given the problem statement in subsection 2.1.1, every algorithm should only have access to all the tainted input bytes; But Angora's random search swaps the entire input buffer with random bytes regularly, s.t. the given random implementation has a significant disadvantage. Assuming this was also the variant of the algorithm used in the Angora paper, this may also have had some effect on the evaluation and comparison to GD there, but more on that in chapter 6. Also, the random implementation used random mutations instead of uniformly distributed random values. The revised algorithm works like this:

Algorithm 2 Random search

```
procedure SOLVERANDOM(𝒳, f, x_0)
    x ← x_0
    while f(x) ≠ 0 do
        x ← random()
    end while
    return x
end procedure
```

4.2 Simultaneous Perturbation Stochastic Approximation

Simultaneous Perturbation Stochastic Approximation (SPSA) was first presented in Spalls 1992 paper "Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation" [24] and it is very similar to Gradient Descent. The only difference between Gradient Descent and SPSA is the way the gradient is measured. SPSA does not rely on a derivative or precise gradient, but instead uses only the objective function to estimate the gradient in constant-time, while the gradient estimation in Angora takes linear time (in relation to the number of input dimensions of the conditional). To achieve this remarkable property, SPSA randomly varies all the parameters of the input vector at once.
Over many iterations, SPSA turns out to be following a similar path as GD would have. Let us recall what GD uses to estimate the gradient of a function. For every input dimension, the partial derivative is calculated by:

$$\frac{\partial f(x)}{\partial x_i} = \frac{f(x + \delta v_i) - f(x)}{\delta}$$

Since we need to execute $f(x)$ once for all dimensions, and we attempt +1 and -1 as values for δ, we need between n and $2n$ executions. SPSA makes two executions per estimate. To obtain a gradient estimate with SPSA, we first generate a random perturbation vector Δ_k of length n. Each component of the vector is plus or minus 1 with an equal probability. To obtain a gradient using simultaneous approximation, we can use this function:

$$\frac{df}{dx} \approx \Delta_k \cdot \frac{f(x + \Delta_k) - f(x - \Delta_k)}{2}$$

Note that this is not a partial derivative, and this function outputs a vector of length n that can be used as a replacement in algorithm 1 to obtain the SPSA algortihm used. Two executions of f are needed to obtain an estimate of the gradient that contains as much information *for optimization* as individual, per-dimension measurements ([24], [25]). However, my implementation is missing gain sequences which are coefficients that get smaller as the number of iterations increases, to 'hone in' on the right value. Despite that we use an adaptive step size for descend that scales up exponentially as long as the traversal of the gradient yields better results. This method does not have the backing of Spall's theoretical

explorations of SPSAs properties, but it makes SPSA and GD in this case easier to compare and gives us less adjustable parameters that could give SPSA an unjustified disadvantage if chosen incorrectly. While the other algorithms have plenty of arbitrary parameters, my goal was to make sensible choices that would reduce these as much as possible.

Advantages

- Built for use without direct gradient measurements.
- Constant-time gradient estimation.
- Built with tolerance for noise and inconsistency.

Disadvantages

- Can get stuck in local minima.

4.3 Simulated Annealing

Simulated Annealing (SA) is a classic optimization technique that uses randomness and probability to gain robustness against local optima ([14], [2]). By sometimes accepting less optimal solutions, it is able to escape local optima early in the search progression. Assuming that in fuzzing we may encounter numerous local optima and plateaus, it makes SA a logical subject for this study. Unfortunately, SA uses numerous parameters that can be adjusted arbitrarily and impact the performance of the algorithm. SA requires us to first define a neighborhood function which, given an element $x \in \mathcal{X}$, returns "neighbors" of that point. In our case, the neighbors are random mutations of x. By sequentially applying a random number of various mutations, we can search the neighborhood of points similar to x in some way. The following mutations may be applied:

- Bitflip: Flips a random bit in x.
- Arithmetic Add: Add an integer to any individual value in x.
- Arithmetic Sub: Subtract an integer from any value in x.
- Interest: Replace a random value in x with a common "interesting" value, such as U64 max, 0, -1, etc.
- Random: Replace a random value in x with a random value.

Each round in SA, we evaluate a fixed number of neighbors of the current input. We then pass the results of each evaluation to a function which decides if we move to that input or not. The tolerance of this acceptance function to deterioration of the objective function value decreases with time and iterations of SA. To control this decrease, a property called temperature is introduced. In the beginning, the temperature is high and the input fluctuates a lot. Every iteration, the temperature decreases and so does the tolerance of the acceptance function. As the temperature closes in on zero, the behavior of this search strategy approaches that of a classic Hill climbing algorithm. By adjusting the rate of cooling and starting temperature, we can adjust the balance between global and local optimization. The cooling function used is:

$$T_{n+1} = (1 - \alpha)T_n$$

The cooling rate α is set to 22%. The starting temperature T_0 is set to 80. A minimum temperature may also be used, in this case it is $T_{min} = 0.02$. This allows us to define the acceptance function (returning the probability of accepting x):

$$P_{\text{accept}}(x_{new}, x_{old}, T_n) = exp\left[\frac{-\log_{1.2}\left(f(x_{new}) - f(x_{old})\right)}{T_n}\right]$$

This means, the higher the difference between the old and new objective function values, the lower the probability of accepting the new point is. The logarithmic

scaling is used to allow us to visit inputs that do not reach the conditional (at high temperature). The reasoning for this is that there may be "islands" of reachability in the objective function surface, which may not be directly connected. This allows us to potentially visit any part of the input space in the beginning, because the difference between values may be very large but it can be an acceptable tradeoff. The geometry of the acceptance function can be seen in Figure 4.1.

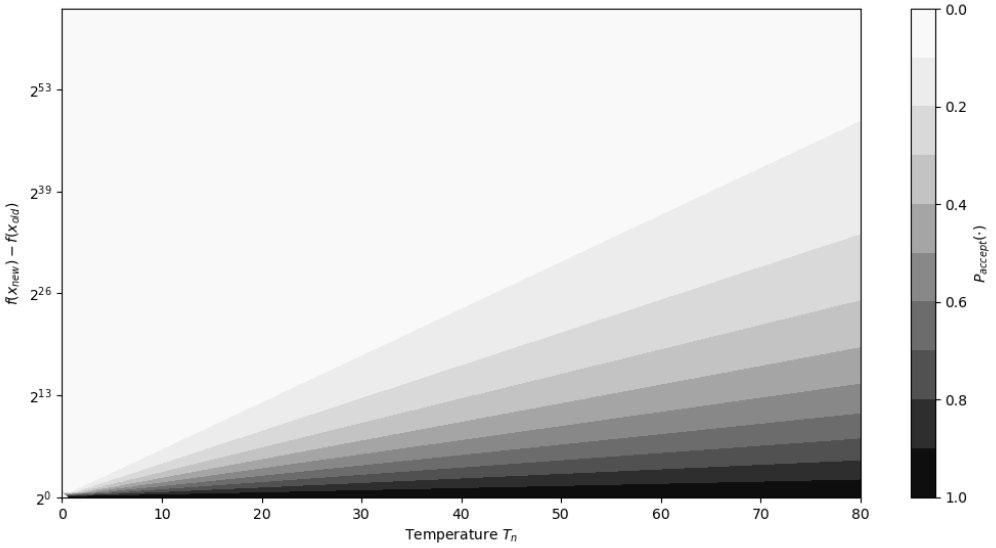

Figure 4.1: Probability distribution of $P_{\text{accept}}(\cdot)$ (contour plot)

Advantages

- Global search possible.
- Robustness against plateaus with zero gradient.

Disadvantages

- If the cooling schedule is not good, it can converge too late or too early.
- Can be slow to move through the search space if the neighborhood function is poorly chosen.

4.4 Pattern Search

Pattern search is a variant of direct search, described in the 1961 paper by Hooke and Jeeve ([13]). In particular, I implemented an early version of this algorithm as described in [5]. The authors attributed the simplified algorithm to early work at Los Alamos National Laboratory. It was used for optimization tasks and was included in this selection for its simplicity, determinism and because the implementation is straightforward. Furthermore, it serves as a slightly advanced Hill climbing algorithm replacement for the comparisons.
When starting with a point x_0, pattern search uses a adaptive pattern to determine the next points to explore and evaluate. It then progresses to a better point (lower $f(x)$) or changes its pattern if none were found. In the case of this implementation, I chose to use the simplest pattern possible, with only one adjustable parameter called the magnitude or step size. In a two-dimensional input space, this pattern forms a cross. Every dimension of the starting point is changed individually (by +/- magnitude) and evaluated. If no better point was found, the step size is reduced according to a fixed-value table (see Table 4.1). In this way, the algorithm is the same as a Hill climbing algorithm with variable step size.

Start →								End
65533	512	256	32	8	4	3	2	1

Table 4.1: The magnitude table of the Pattern search algorithm.

Advantages

- Deterministic and predictable behavior (when x_0 is not random).
- Very simple.
- Can escape some plateaus smaller than the largest magnitude.

Disadvantages

- Can get stuck in local minima.

Algorithm 3 Pattern search

```
procedure SolvePattern(𝒳, f, x_0)
    x ← x_0
    m ← 0
    while f(x) ≠ 0 do
        improved ← False
        for all i ∈ [0, n[ do                    ▷ Go through all dimensions
            x_i ← x_i ± magnitude-table[m]
            if f(x) > f(x_0) then
                improved ← True
                break
            end if
            reset x_i
        end for
        if !improved then
            m ← m + 1                            ▷ Use the next smaller magnitude
        end if
    end while
    return x
end procedure
```

4.5 Bees Algorithm

All of the algorithms presented so far are single-candidate based, meaning they start with a single potential solution and attempt to refine that solution over the course of many iterations.
The following two algorithms are population-based approaches; They spawn multiple potential solutions and evaluate many different areas of the search space simultaneously.
Population-based algorithms usually have a greater focus on global rather than local optimization, because they spread their candidates over the search space. This means they might be much better at solving constraints with many local optima.
Many heuristic optimization algorithms, including Simulated Annealing, Bees algorithm and Particle Swarm Optimization (PSO) are inspired by natural phenomena. In the case of Bees algorithm, it is inspired by the foraging behavior of bees ([18], [19]).
Bee hives are able to explore vast landscapes for food by sending out multiple scout bees that sparsely probe the lands for promising flower patches. Once a promising food source has been found, the scout returns to the hive and recruits a number of worker bees to harvest the flower patch. The scouts convey information about the distance, direction and size of the food source that was found.
In more formal terms, we start out with a uniform randomly distributed population of points in the search space, and each iteration we select the k best points which will trigger the neighborhood search. The neighborhood or patch search is a round where additional bees (points in the search space) are dispatched to search the close proximity of the best points. After evaluating the neighborhoods, we again select the best k points as new neighborhood centers.
Because many evaluations have to be made for the initial exploration and subsequent search and the points are very spread out, we incur a high initial and continuous cost which can only be justified if the resilience against local optima outweighs it.
The algorithm is detailed in algorithm 4 analogous to the description in [18]. The algorithm has a few tweakable parameters. The following parameters and values have been selected for this implementation:

- Number of scouts: 8
- Number of best bees selected (k): 4, with the 3 best bees forming "elite" patches that are searched with higher focus.
- Number of bees per patch: 5 per normal patch, 8 per elite patch
- Patch sizes: Normal patches start with size 80 (forming a hypercube search space around the center point of the neighborhood with width 80), elite patches with size 20. Every iteration, the neighborhood cubes shrink by 15 percent.

Algorithm 4 Bees algorithm

procedure SOLVEBEES($\mathcal{X}, f, x_0$)
Initialize random population of size n_{bees} including x_0 as P
while $f(x_{best}) \neq 0$ **do**
Evaluate $\forall x \in P : f(x)$
Select k best points in P
Evaluate $k_{patch-size}$ points around the selected points
Select the best points from neighborhood search
Abandon neighborhoods that have been searched multiple times without improvement
Let remaining bees scout random points to form a new population with the best bees
end while
return x_{best}
end procedure

Advantages

- Focused on global optimization.
- Robustness against plateaus with zero gradient.
- No derivative or analytic form required.

Disadvantages

- A lot of parameters that can influence and distort performance.
- Can be slow to converge on an optimum.

4.6 Particle Swarm Optimization

At its core, PSO is a very simple algorithm that utilizes emergent swarm intelligence to perform optimization. Individual particles (independently moving points in the search space) only know the best location they visited, their current location and speed and the best known position the swarm has found so far. By introducing simple rules of movement for the particles, global optimization emerges. PSO was first introduced by Eberhart and Kennedy in 1995 [7] and has been studied ever since, with many variations and adaptations available.
PSO is a very robust algorithm, able to perform well on a large number of optimization benchmarks [23], but it is also very sensitive to parameter changes as well as the chosen swarm topology [26]. In this implementation, the focus is to keep it as simple as possible, using a global topology that shares the best position with the entire swarm.
There is a simple formula that governs the movement of particles in the search space, and in this case there exists a physical analog to these rules of movement. Essentially, each particle is pulled towards its own best known location p_k and global best known position g by jittery springs and moves in a fluid with a fixed friction coefficient ω. The version of the algorithm I will be implementing can also be found at [4].
There are three main parameters that can be changed in this variant of PSO beside the swarm size: First, the aforementioned friction coefficient ω, second and third are coefficients ϕ_p and ϕ_g which act as weights allowing us to adjust the influence of the local best position vs the global best on the next location. The algorithm starts out with randomly distributed particles with random velocities.
Given a particle's position $\vec{x_k}$ and velocity $\vec{v_k}$, its own best location $\vec{p_k}$, two uniform random numbers $r_p, r_g \in [0, 1] \subset \mathbb{R}$ and the global best $\vec{g}$, its next position $\vec{x_{k\_next}}$ is defined as:

$$\vec{x_{k\_next}} := \vec{x_k} + \omega\vec{v_k} + \phi_g r_g(\vec{g} - \vec{x_k}) + \phi_p r_p(\vec{p_k} - \vec{x_k})$$

The population size S can fortunately be determined with a fixed formula which is still debated (part of Standard PSO 2006) but returns sensible numbers for a large amount of different dimensionalities (n):

$$S = 10 + \lfloor 2\sqrt{n} \rfloor$$

I selected the other parameters based on recommendations in [16]. Unfortunately, selecting the best parameters is also dependent upon the dimensionality of the problem; since our algorithm will have to solve problems with different dimensionalities, having a fixed set of parameters is not great. But after some initial testing it already turned out that the problems encountered have very few dimensions (less than five covers almost all) so it should not unjustly influence my results. Meta-optimization or adaptive variants of PSO are also an option, but that requires a lot of implementation effort and goes beyond the scope of this work. The chosen parameters are: $\omega = 0.1, \phi_g = 1.33, \phi_p = 2.55$.

As a population-based global heuristic search algorithm like the Bees algorithm PSO shares a lot of its advantages and disadvantages in this context, so I will not recount them here.

Algorithm 5 Particle Swarm Optimization

procedure SOLVEPSO($\mathcal{X}, f, x_0$)
  Initialize random population of size S including x_0 as P
  while $f(x_{best}) \neq 0$ **do**
    Evaluate $\forall x \in P : f(x)$
    Update g and p_k for all $x \in P$
    Update particle positions
  end while
  return x_{best}
end procedure

5 Methodology

5.1 Performance Metrics

To compare the performance of these algorithms, we will measure additional performance metrics to those used in the original Angora paper. We will consider the overall fuzzing success (coverage, found bugs) as well as solved/unsolved constraints. In addition, I will implement precise tracking of constraint solving process. The following information will be collected for each encountered constraint:

- A unique Identifier (ID)
- Context ID: A hash of the call stack
- Time and executions needed to solve the conditional
- Convergence of the objective function values (regularly recording the best/-median value after n executions)
- Dimensionality (total bytes and dimensions)

5.2 Targets

The target programs will be the LAVA-M set and two real-world programs. LAVA-M is a set of four standard GNU utilities that have been artificially injected with bugs to benchmark fuzzers [6]. The LAVA-M dataset was also used in the original Angora evaluation, allowing us to draw comparisons to that. Unfortunately I was not able to get one of the programs (uniq) to work with Angora's taint tracking in time, but the other programs were evaluated.

The two real-world programs are jhead[1] and xmlwf[2], two programs that parse input files to extract information. The former is a jpeg metadata extraction program, the latter used to parse XML files.
While this is by no means a comprehensive test set, it is the amount of software I was able to evaluate in the given time frame, especially considering that each evaluation was repeated multiple times.
I have decided to fuzz the LAVA-M programs with a time limit of 10 minutes as it was done in the Angora paper, and 45 minutes each for jhead and xmlwf.
Per target and algorithm pair, 15 repetitions were executed for the LAVA-M programs and 10 repetitions for jhead and xmlwf.

5.2.1 Seeds

For the LAVA-M programs I used the same seeds that were used in the Angora paper by recreating them from the description on Angoras GitHub page. Base64 started with a single seed with one random byte, md5sum starts with a set of correct hashes and paths to corresponding files (since correct examples can not be produced by fuzzing itself) and the seed for who is created by writing a correct utmp struct to a file from a separate C program.
Jhead started with two examples of correct jpeg images of different size and xmlwf started with two small XML files.

5.3 Execution Framework

To coordinate and automate the evaluation I developed a small Python program that simultaneously runs an arbitrary number of evaluations and ensures that no faults occur during execution of Angora and finish correctly.
The evaluations were done on a Ryzen 2700X CPU with 8 cores and 16 threads, and up to 12 simultaneous Angora instances running.
Core binding was disabled in Angora to prevent some instances from obtaining an uneven amount of CPU time. While this can slow throughput due to memory being evacuated from the CPU cache, it levels the playing field while allowing us to execute more tests in the same amount of time.
All logging was done in-memory until a run finished and then dumped to disk in a single batch after the measurements to avoid spikes of slow throughput. The goal of all changes to the base Angora version and the environment was not to achieve maximum performance, but to create an equal and stable environment for all participating algorithms.

[1] Version 3.03
[2] From libexpat version 2.2.7

5.4 Angora: Features and Heuristics

I used the version of Angora released to GitHub at [`github.com/AngoraFuzzer/Angora`][3] by the original authors as a basis for my work.

Because the goal of this study is to compare the different optimization algorithms, I have disabled or changed some features that are included in Angora. They include different heuristics and modes that have nothing to do with constraint solving by optimization and exist to improve the bug finding capabilities. The first is the exploit mode, which is enabled by some heuristics to exploit known issues like out-of-bounds reads and writes to trigger a crash. It replaces input values with a pre-defined, fixed selection of "problematic" values, such as 0, 1, -1 and high or low values at the very limits of the search space. While this is very useful in a production environment, we are only going to use and measure constraint solving to expand the coverage of the program. This also means that we will likely underperform compared to the original version in terms of bugs found and potentially total coverage, but any gain over the gradient descent implementation in my version should translate to improvements over the original when the algorithms are backported into the main Angora. This is a summary of all the changes made to Angora for the first evaluation:

- Per-constraint execution budget: Angora used 376 maximum executions per cycle and added execution budget with heuristics. I fixed the execution budget to 400 per cycle. I could not determine the reason for choosing 376 executions in the original version.
- Disabled exploit and AFL fuzz modes (AFL mode is a built-in approximation of AFLs mutation strategy but disabled by default).
- Changed parts of input randomization routine: Original Angora randomizes the entire input buffer with a $\frac{1}{3}$ chance when, for example, searching for a new reachable start point for the search. This is inconsistent with the formalization of constraint solving in this work and gives algorithms that use the routine the possibility to reach points outside of the search space defined by taint tracking. It might also be bad for performance, as it is a very radical step that resets any progress. My implementation uses a uniform random replacement only for the input bytes tainted by the current conditional.
- Gradient Descent: The implementation contains heuristics that makes it hard to compare to traditional GD described in the original paper.
  - Remove "interesting point" routine: Angora regularly inserts fixed, known problematic values like 0, -1 etc. to probe for bugs.

[3] Commit `92fba70194879075ad9000eb329e23e62c81acf2`, June 21st 2019

  - Rewrite descend routine: The original implementation does not descend only by the calculated gradient directly, but instead applies each component of the gradient individually. My implementation uses the entire gradient to simplify the implementation and bring it closer to a minimal GD version. The rewritten version is also the one described in algorithm 1.

- Remove deterministic and one-byte fuzz modes. These were used to exhaustively search small or one-dimensional search spaces in a deterministic way. This is favorable in production, but it had to be disabled so we can observe the behavior of the algorithms in these small search spaces. By default, one-byte mode is used for any single-byte constraint and simply iterates over all possible 256 values. Deterministic mode is enabled when other methods did not solve the conditional and iterates over all bits in the input, flipping each one separately.

6 Evaluation

6.1 Results

To compare the efficacy of the different algorithms, I selected three main metrics. First is the total number of solved constraints. This is preferable to the total number of discovered constraints and the ratio of solved/unsolved constraints, since every conditional that is solved can yield many more new paths and constraints, thus potentially creating a disadvantage for algorithms that actually perform well. The other two main metrics are discovered crashes and paths. Crashes describes the number of inputs that have been found to trigger distinguishable traces and crashes. Paths is a measure of how many inputs were generated that trigger distinguishable behavior.
Figure 6.1 shows the results of the first experiment in light of these main metrics. Each bar represents the average value for the given algorithm and target over a set of 10 (jhead, xmlwf) to 15 (base64, who, md5sum) runs. The black bars represent the standard deviation.
The LAVA-M programs were fuzzed for 10 minutes per run each, jhead and xmlwf for 45min each to account for the bigger program size and complexity.

The performance of all algorithms on the LAVA-M test set is very similar, even indistinguishable for the base64 and md5sum programs. On who, GD has a measurable advantage, but the other algorithms show very similar results to random search. On jhead we can see some more pronounced differences, with GD, SA and Pattern search coming out measurably better than random search. The similarity of the results may also suggest that every algorithm hit the same coverage ceiling on base64 and md5sum, even completely random search. This suggests that GD was not the main contributor to Angoras stellar performance on the LAVA-M dataset.

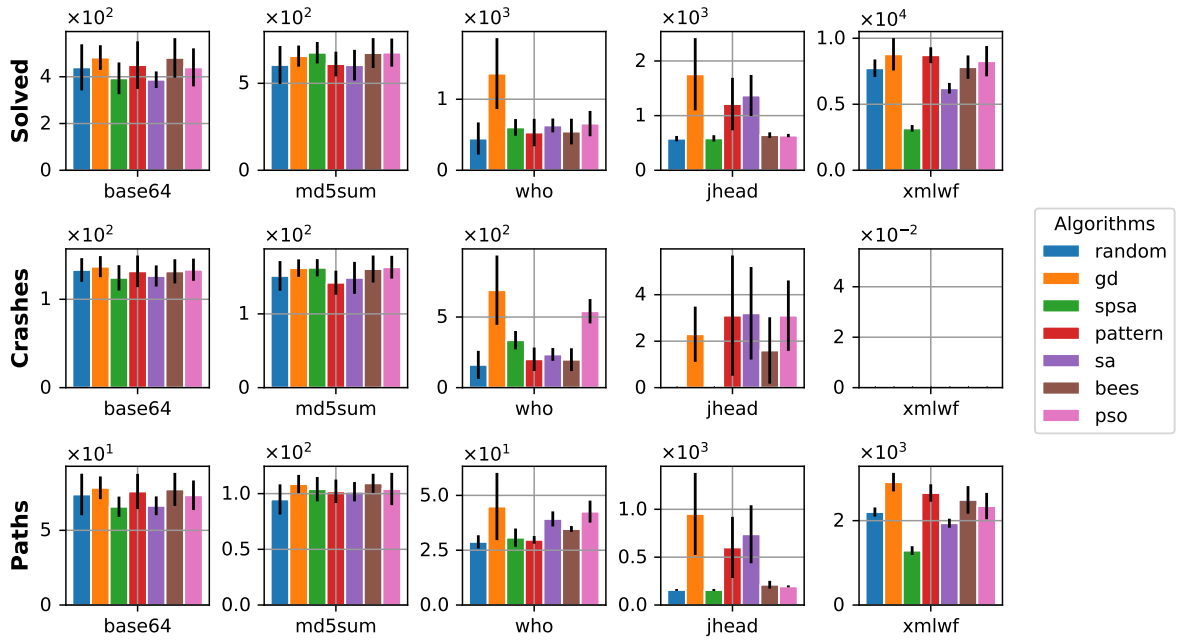

Figure 6.1: Results of the first run.

In the Angora paper [3] a random search algorithm and a random search with magic byte extraction were also directly compared to GD, but the methodology and results were different. Chen et al. used AFL to generate a fixed corpus of inputs that were used as seeds for fuzzing with Angora under the different strategies.
Angora was not allowed to use entirely new seeds and ran for two hours with each respective algorithm. Since the total number of discovered constraints must be the same for all strategies with the same inputs, they used the solve rate as a measure of performance, with GD reaching 97% for xmlwf and jhead and random/magic bytes reaching up to 87% each on the same programs. While there was a strong advantage for GD in every one of the tested programs, the usage of AFL to collect the seeds may have subjected the input set to a form of selection bias. Since AFL mainly uses random mutations to solve constraints, it would not be surprising if the constraints that can easily be solved with random strategies had already been solved by AFL in the input corpus to Angora, s.t. the constraints that are left are the ones that require targeted optimization to solve them. This means the percentages may not represent the correct total solve rate of conditionals. It is also why I decided against using a pre-computed, fixed input corpus.

But this prompted me to investigate the magic byte algorithm. It can be found in the codebase of Angora as “MbSearch”. While exploring the code, I discovered

that the magic byte extraction used in that algorithm was also used in the GD implementation, thus ensuring that GD will always perform *at least as* good as random search with magic byte extraction (which is not represented as an algorithm in my measurements). While implementing the other algorithms, I used code snippets from the GD implementation as a reference, and so the same behavior applies to all listed algorithms, they represent magic byte extraction + the algorithm. Further indication that this was the reason for the similarity can be found in the convergence histogram Figure 6.2 where I have aggregated the number of executions it took each algorithm to solve a conditional (each conditional is a separate datapoint, collectively displayed as a histogram). We can clearly see that most constraints are solved instantaneously (with the first execution of x_0). The results suggest that magic byte extraction is the dominant feature in constraint solving.
The reason that the data points for bees algorithms are so spaced out is that I only recorded the minimum objective function values for every completed iteration of the algorithms. Bees algorithm simply takes a lot of executions per iteration.

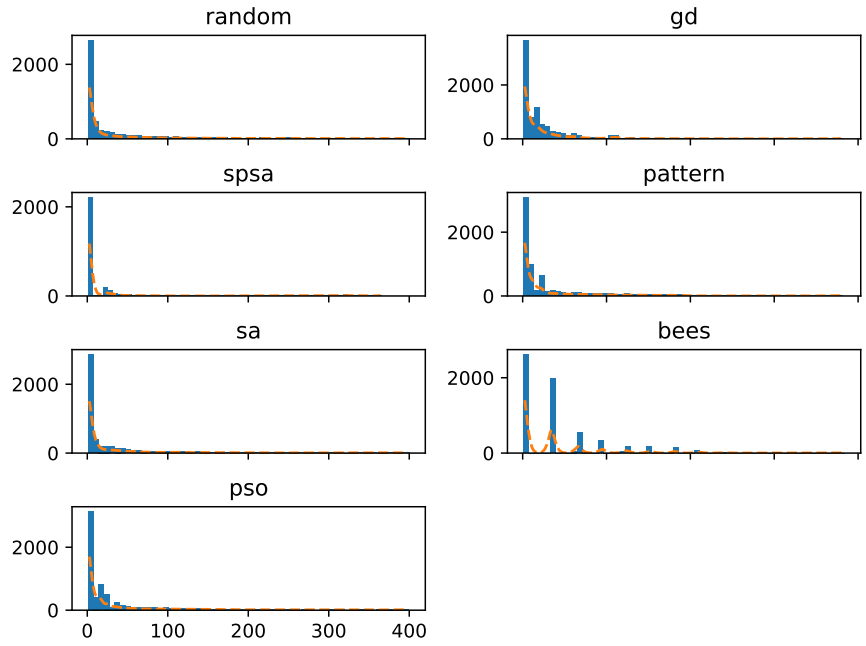

Figure 6.2: Histogram of executions needed to solve constraints (Run 1). X-Axis: Executions, Y-Axis: Incidence of constraints solved after x executions.

For further analysis, I removed the magic byte extraction, instead placing random bytes in x_0 and repeated the experiments. The results of the second run without the magic byte extraction can be seen in Figure 6.3.

The new distribution is also consistent with the hypothesis that magic byte extraction is dominating constraint solving, particularly when we compare Figure 6.2

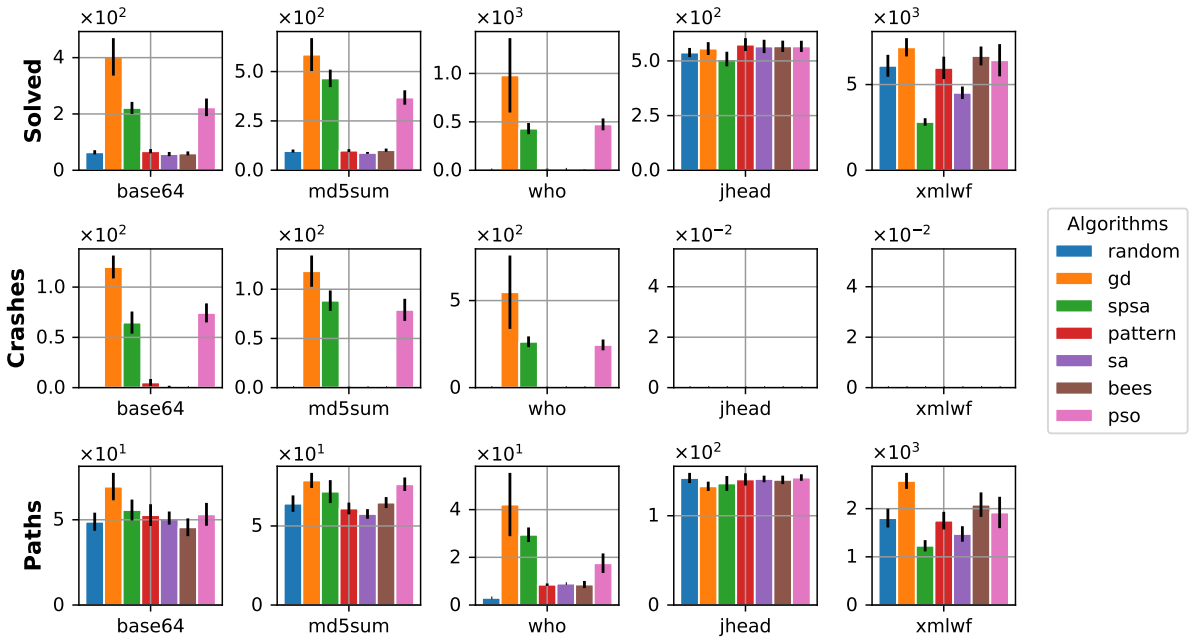

Figure 6.3: Results of the second run.

with Figure 6.4, where we can now see a much more even distribution of solved constraints. While a lot of constraints are still solved early, it is no longer concentrated in the first execution.

After removing magic byte extraction, the performance of the random algorithm drops off, producing significantly fewer crashes and exploring overall less of the LAVA-M dataset, probably because the programs contain a lot of magic byte comparisons that random now has no way to solve easily.
This is also where GD shows that it is able to solve constraints most effectively, beating all other algorithms on every metric.
While GD shows that it is able to perform well on LAVA-M even without magic byte extraction, the case for GD on jhead and xmlwf is much less clear. Whereas in the first run the algorithms were producing similar results on LAVA-M, now the similarity is with jhead.
It is also not the case that the algorithms have explored all of jhead and xmlwf which would explain the similar results, since the first run showed that there are a lot more solvable constraints and paths to discover in these programs. GD with magic byte extraction solved on average 1757 constraints in jhead, while without magic byte extraction none of the algorithms achieved more than 570 solved constraints.
I began this study to determine the relative differences between optimization algorithms in this domain, but it seems we should also revisit the question whether

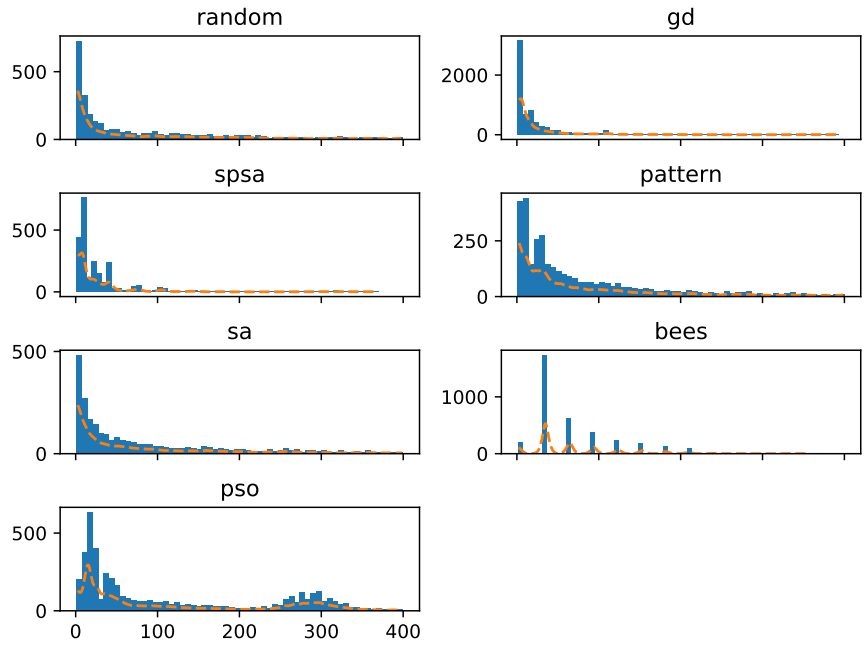

Figure 6.4: Histogram of executions needed to solve constraints (Run 2).

GD constraint solving is actually effective in real-world programs, and if not, how does Angora achieve such good performance on the benchmarks?
Unfortunately, my experimental setup does not allow me to answer this question. We will discuss potential experiments in section 6.3.
While the overall benefit of constraint solving through optimization is inconclusive, we can nevertheless explore some of the interesting differences between algorithms in this domain. First off, Figure 6.4 shows that most algorithms converge on a solution within the first 100 executions. For GD, we should be able to use an execution budget of 60 per cycle and still solve the same constraints. This is a huge optimization, since the remaining extra executions do not seem to provide any benefit. The only algorithm that does not have a single center of convergence is PSO, where we can observe a secondary center around 290 executions. This is expected to arise from the behavior of PSO, which converges rather slowly on a solution. The convergence rate of PSO is dependent upon the chosen parameters and problem at hand; an adaptive variant of PSO can probably determine the convergence progress and terminate itself when convergence has been reached (whether or not the constraint has been solved). PSO, beside SPSA, is also the best performing algorithm behind GD. Since my implementation of PSO is very minimal, we can expect even better results with an advanced PSO variant. GD and SPSA do not have many ways to improve, while PSO has the most unrealized potential.
It is possible that SA and Bees algorithm also have secondary centers of convergence outside the set execution budget of 400 executions per cycle, but the

parameters were specifically chosen to allow quick convergence. Even if that is the case it does not seem very promising to pursue algorithms with such slow convergence when the alternatives are more effective.
Lastly, I want to explore the relationship between constraint dimensionality and the ability to solve them. Figure 6.5 plots the percentage of constraints solved against the dimensionality of the constraints in bytes. To create this graph, I used logarithmically sized bins to sort the constraints based on the number of dimensions and used the bins with more than 50 constraints each to calculate a solve rate for the given algorithm and dimensionality.
We can see a progressive decline in solve rate for constraints with less than five dimensions, at which point the data becomes far more erratic. Since we are looking at a small sample of programs, we have relatively few examples of high-dimensionality constraints and even with the cutoff I introduced the data is very noisy. I assume a larger and more diverse set of target programs would smooth the curve. Despite that, we can see that GD and SPSA are particularly good at high-dimensionality constraints with PSO coming in second. There is also a large dip in solve rate at four dimensions. This is because lots of bugs inserted in LAVA-M are four-byte comparisons (with linear objective functions) that extremely difficult for random, non-targeted algorithms to solve. Only SPSA, GD and PSO are targeted enough to find the correct bytes fast enough.
It is also surprising that the algorithms achieve no more than 70% solve rate on single-byte constraints; This might be solved by switching to deterministic fuzzing for these constraints, but it could also be a limitation due to incomplete taint tracking.

6.2 Threats to Validity

The far greatest issue with this study is the small number of target programs. Unfortunately there is no set of programs used as a standard for fuzzing that covers all the aspects of modern fuzzers (LAVA-M has been exhausted as a benchmark by modern fuzzers) and my resource constraints did not allow me to test a larger set of programs. However there are some insights that will generalize beyond this work.
Another issue is snowballing; Many of the metrics used to test fuzzing performance do not scale in proportion to the efficacy of constraint solving. If one constraint solver is able to solve a few more difficult constraints than another algorithm, it might unlock a huge number of new, easier constraints, thus inflating the results. It still allows us to tell if there is a difference, but quantifying that difference correctly and reliably is very difficult right now.
The last point is the implementation and selection of algorithms. I used the best parameters I could determine in the given time frame, but it is highly unlikely that they represent an optimum for any of the tested algorithms. Future research with meta-optimizers would be needed. The chosen selection also does not represent the entirety of optimization algorithms available; I had to limit the selection

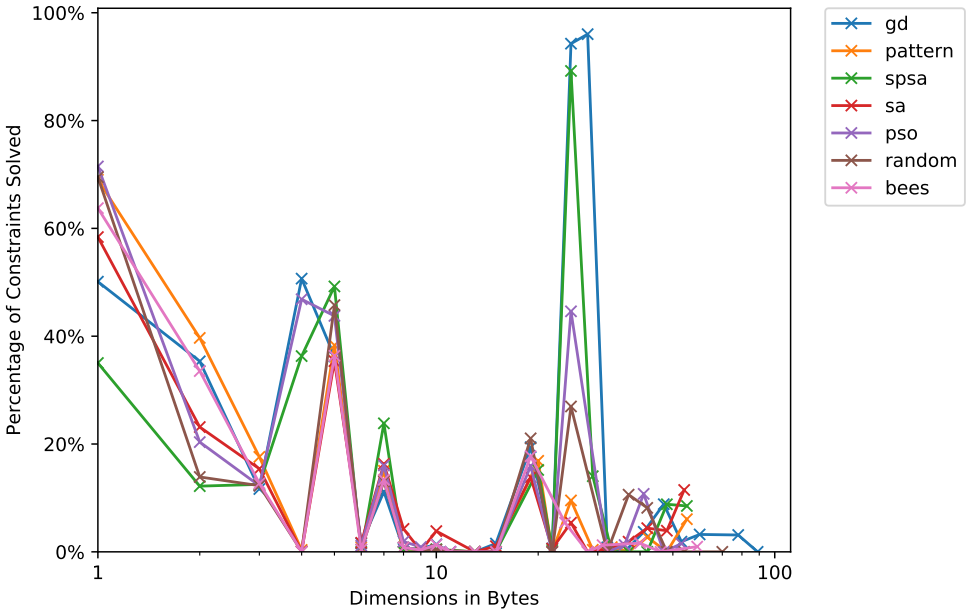

Figure 6.5: Dimensionality vs Solve Rate (Run 2).

and chose based on assumptions about the objective function.

6.3 Future Work/Discussion

While the results of this study are consistent with the overall performance of Angora reported in [3], the contributions of constraint solving with GD are questionable. Additional research will be necessary to determine the effectiveness of this approach on a larger set of programs. Unfortunately, there is always a mix of algorithms and heuristics involved with any fuzzer, so figuring out the impact of individual components and improvements can be tricky. This issue is exacerbated since there is no commonly accepted evaluation standard that allows us to determine the efficacy or regressions of changes reliably on a large, representative set of programs.
There are important improvements to the state-of-art in Angora, but finding and integrating the right ones into new projects is impossible without component-wise analysis. I suspect that due to the increased feedback granularity and taint tracking the complexity of solving constraints is already reduced so much that a random algorithm with magic byte extraction can perform as good as the original Angora on LAVA-M. That is not to say that there are no targets with complex non-linear constraints which might only be solvable with optimization or symbolic constraint solving where we would see significant benefits, but the given data does not support this assumption right now. In fuzzing, fast-and-dumb

sometimes beats slow-and-smart approaches.
To this end, it is also worth investigating the properties of objective functions in this context further. Are local minima and plateaus really a big issue in real-world programs? My data does not allow me to assert this right now, since global search approaches did not yield overall higher coverage. An experimental setup where the most difficult constraints are collected from many programs and their topology and search space geometry mapped, analyzed and tallied with the amount of simple, linear constraints would be required for this.

In future work, I hope to see how an advanced implementation of PSO would fare in fuzzing. While not beating GD in this study on performance, it should be the most promising direction for future research investments due to its proven ability to be effective in diverse domains. Adaptive PSO implementations would also improve the performance on a wider range of constraint dimensionalities.
It is also probably worth implementing a meta-optimizer for these algorithms (when a larger test set is available) to obtain better parameters and performance. The fact that some of the implemented algorithms can keep up even without proper fine-tuning should be a testament to their robustness and applicability to this problem domain.

It would also be interesting to investigate the impact of length extension heuristics and taint tracking accuracy on the solvability of constraints.
The findings of this study will also be helpful to the approach by She et al [22], who used a neural network to smooth the objective function and perform gradient-guided optimization on the resulting transformed problem. By deploying and benchmarking PSO as optimization algorithm, we might be able to achieve higher accuracy than GD alone.

6.4 Conclusion

While the findings of this study surprised me in many ways, the results tell a much more interesting story than anticipated. Starting with things that can be learned to improve the state-of-the-art: It seems that for most optimization algorithms, convergence on a solution either occurs quickly or not at all (due to various reasons, incomplete taint tracking or type inference, many simple constraints etc.). Using this knowledge, we can already cut the execution budget of GD in Angora by 70% and retain similar accuracy with more throughput. That means we can spend more time fuzzing these unsolvable constraints with other strategies in hope of success.
We can also assert that in the current Angora implementation, the algorithm used for optimization has minimal impact on the performance on LAVA-M.
Furthermore we now have a promising lead for future improvements to constraint solving with optimization: PSO. The algorithm has demonstrated to be

robust and effective on a surprisingly large set of constraints without proper fine-tuning or other recent, state-of-the-art improvements. At the same time we know that GD was not a bad choice for this problem at all, outperforming all of my implementations (by a small margin at least).
It does not seem to be the case that switching of optimization algorithms would be very beneficial. I could not determine a common property of constraints that reliably predicted performance of a certain approach. Dimensionality seems to play a smaller role than expected, with many constraints with huge dimensionalities being solved easily and some low-dimensionality constraints proving to be very difficult to solve for all algorithms. In general, there is a downward trend in solve rate for larger input spaces, but more data is needed for a correct determination.

Acronyms and Abbreviations

AFL American Fuzzy Lop. 12, 13, 15, 42

CPU Central Processing Unit. 12, 38

GD Gradient Descent. 18–20, 23, 25–27, 41–43, 47–49

ID Identifier. 37

PSO Particle Swarm Optimization. 32, 34, 35, 45, 46, 48

SA Simulated Annealing. 28, 41, 45

SPSA Simultaneous Perturbation Stochastic Approximation. 26, 27

Bibliography

[1] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[2] Vladimir Cerny. "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm". In: *Journal of optimization theory and applications* 45.1 (1985), pp. 41–51.

[3] Peng Chen and Hao Chen. "Angora - Efficient Fuzzing by Principled Search." In: *CoRR* (2018).

[4] Maurice Clerc. "Standard Particle Swarm Optimisation". 15 pages. Sept. 2012.

[5] William C Davidon. "Variable metric method for minimization". In: *SIAM Journal on Optimization* 1.1 (1991), pp. 1–17.

[6] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[7] R. Eberhart and J. Kennedy. "A new optimizer using particle swarm theory". In: *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*. Oct. 1995, pp. 39–43.

[8] Avner Engel and Mark Last. "Modeling software testing costs and risks using fuzzy logic paradigm". In: *Journal of Systems and Software* 80.6 (2007), pp. 817–835. ISSN: 0164-1212.

[9] Patrice Godefroid. "Random Testing for Security: Blackbox vs. Whitebox Fuzzing". In: *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. RT '07. Atlanta, Georgia: ACM, 2007, pp. 1–1. ISBN: 978-1-59593-881-7.

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART - directed automated random testing." In: *PLDI* (2005), p. 213.

[11] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing". In: *Queue* 10.1 (Jan. 2012), 20:20–20:27. ISSN: 1542-7730.

[12] Patrice Godefroid, Michael Y Levin, and David A Molnar. "Automated Whitebox Fuzz Testing." In: *NDSS* (2008).

[13] Robert Hooke and Terry A Jeeves. ""Direct Search"Solution of Numerical and Statistical Problems". In: *Journal of the ACM (JACM)* 8.2 (1961), pp. 212–229.

[14] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[15] Nicholas D Matsakis and Felix S Klock II. "The rust language". In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[16] Magnus Erik Hvass Pedersen. "Good parameters for particle swarm optimization". In: *Hvass Lab., Copenhagen, Denmark, Tech. Rep. HL1001* (2010), pp. 1551–3203.

[17] H Peng, Y Shoshitaishvili, and M Payer. "T-Fuzz: fuzzing by program transformation". In: *nebelwelt.net* (2018).

[18] DT Pham et al. "The bees algorithm". In: *Technical Note, Manufacturing Engineering Centre, Cardiff University, UK* (2005).

[19] Duc Truong Pham et al. "-The Bees Algorithm—A Novel Tool for Complex Optimisation Problems". In: *Intelligent production machines and systems*. Elsevier, 2006, pp. 454–459.

[20] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *ArXiv* abs/1609.04747 (2016).

[21] Bhargava Shastry et al. "Static Program Analysis as a Fuzzing Aid." In: *RAID* 10453.8 (2017), pp. 26–47.

[22] Dongdong She et al. "NEUZZ: Efficient Fuzzing with Neural Program Smoothing". In: *arXiv.org* (July 2018).

[23] Yuhui Shi and Russell C Eberhart. "Empirical study of particle swarm optimization". In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. Vol. 3. IEEE. 1999, pp. 1945–1950.

[24] J. C. Spall. "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation". In: *IEEE Transactions on Automatic Control* 37.3 (Mar. 1992), pp. 332–341. ISSN: 0018-9286.

[25] James C. Spall. "AN OVERVIEW OF THE SIMULTANEOUS PERTURBATION METHOD FOR EFFICIENT OPTIMIZATION". In: 1998.

[26] Ioan Cristian Trelea. "The particle swarm optimization algorithm: convergence analysis and parameter selection". In: *Information processing letters* 85.6 (2003), pp. 317–325.

[27] Weiguang Wang, Hao Sun, and Qingkai Zeng. "SeededFuzz - Selecting and Generating Seeds for Directed Fuzzing." In: *TASE* (2016), pp. 49–56.

[28] Xiong Xie and Yuhang Chen. "Research on Fuzz Testing Framework based on Concolic Execution". In: *DEStech Transactions on Computer Science and Engineering* 0.csae (Jan. 2018).

[29] Michal Zalewski. *American fuzzy lop (AFL) fuzzer*. 2017.