

DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries (Extended Abstract)

Konstantin Böttinger^(✉) and Claudia Eckert

Fraunhofer Institute for Applied and Integrated Security,
85748 Garching (near Munich), Germany
`konstantin.boettinger@aisec.fraunhofer.de`

Abstract. We introduce a new method for triggering vulnerabilities in deep layers of binary executables and facilitate their exploitation. In our approach we combine dynamic symbolic execution with fuzzing techniques. To maximize both the execution path depth and the degree of freedom in input parameters for exploitation, we define a novel method to assign probabilities to program paths. Based on this probability distribution we apply new path exploration strategies. This facilitates payload generation and therefore vulnerability exploitation.

Keywords: Concolic execution · Fuzzing · Random testing

1 Introduction

As ubiquitous software is ever increasing in size and complexity, we face the severe challenge to validate and maintain the systems that surround us. Software testing has come a long way from its origins to the recent developments of sophisticated validation techniques. In this paper we introduce a new method combining symbolic execution and random testing. Our goals are (1) code coverage in deep layers of targeted binaries which are unreachable by current technologies and (2) maximal degree of freedom in the input variables when discovering a program error.

Before we present the main idea of our approach and the summary of our contributions, we give some background on concolic execution and fuzzing. We especially highlight limitations of concolic execution and fuzzing when applied isolated and motivate a combination of both as a promising new strategy.

Concolic Execution. The main idea of symbolic execution is to assign symbolic representations to input variables of a program and generate formulas over the symbols according to the transformations in the program execution. Reasoning about a program on the bases of such symbolic representations of execution paths can provide new insight into the behavior of the program. Besides program

verification, symbolic execution nowadays has its biggest impact in program testing. The original idea was extended over the years and developed into concrete symbolic (concolic) execution. The program is initially executed with arbitrary concrete input values and symbolic constraints over the symbols are generated along the program execution path. Next, one of the collected branch conditions is negated and together with the remaining constraints given to an SMT solver. The solution (also called *model*) generated by the SMT solver is injected as new input into the program, which now takes the branch alternative when executed. This is because the SMT solver just calculated the solution of the negation of the former branch constraint so that the newly generated input follows the alternative path. This procedure is iteratively repeated until a halt condition is reached. In the best case the reached halt condition resembles full path coverage of all alternative paths of the program, in the worst case the halt condition is caused by an overloaded SMT solver. The latter is a natural consequence of the exponential growth of the number of paths we have to deal with, which we refer to as the *path explosion* problem. Concolic execution is advantageous in code regions where pure symbolic reasoning is ineffective or even infeasible. This is often the case for complex arithmetic operations, pointer manipulations, calls to external library functions, or system calls.

Pure concolic execution, however, has strong limitations. Current SMT solvers are very limited in the number of variables and constraints they can handle efficiently so that concolic execution gets stuck in very early stages of the program. Despite huge advances in the field of SMT solvers, concolic execution of large programs is infeasible and in practice will only cover limited parts of the execution graph. The major part of graph coverage must therefore be done with fuzzing.

Fuzzing. Existing fuzzing tools generate random input values for the targeted program in order to drive it to an unexpected state. Fuzzing has generated a long list of vulnerabilities over the years and is by now the most successful approach when it comes to program testing. However, it has severe limitations even in very simple situations. To illustrate this, consider the following code snippet:

```
#include <stdint.h>
...
int check( uint64_t num ){
    if( num == UINT64_C(0) )
        assert( false );
}
```

If we want to reach the assertion in the `check` function with a random choice of the integer `num`, we have a probability of 2^{-64} for each try to pass the `if` statement. The situation gets even worse if there are multiple such checks, e.g. in the calculation of a checksum or character match during input parsing. Such code areas are very hard to be passed by pure random input generation and code regions beyond such examples are most likely not covered by fuzzing. In the following we will refer to such cases as *fuzzing walls*. However, the `false`

assertion in the above code listing can easily be reached with concolic execution, as the comparison to zero directly translates to a simple expression for the SMT solver.

The Hybrid Approach. As we just showed, critical limitations of fuzzing can be overcome with concolic execution, and in turn fuzzing scales much better to path explosion than SMT solvers do. As a natural next step we combine both methods. The idea is to apply concolic execution whenever fuzzing saturates (i.e. stops exploration at a fuzzing wall), and in turn switch back to fuzzing whenever the fuzzing walls are passed by concolic execution.

However, we still have to deal with the problem of path explosion and therefore still may end up covering only the first execution layers of a program. In the following, we refer to *path depth* as the number of branches along that path, which directly corresponds to the number of basic blocks. Even in the combined approach we are confronted with two challenges. First, if we want to fuzz deep areas of a program, we have to find a way to construct execution paths into such areas and somehow delay path explosion until we have found such a tunnel. Second, to generate a payload and exploit a detected vulnerability in the program under test, we not only have to reach the bug with a single suitable input, but we have to reach it with maximal degree of freedom in the input values. To be more precise, if we reach a vulnerability with exactly one constellation of the input variables, we most probably would not be able to exploit it in a meaningful way because any attempt to generate a payload (and thereby change the input variables) would lead the input to take a different path in the execution graph. Therefore, we propose a way to maximize the degree of freedom regarding input variables. This yields both alleviation of vulnerability exploitation and execution paths that reach into deep layers of the program.

In summary, we make the following contributions:

- We propose a new search heuristic that delays path explosion effectively into deeper layers of the tested binary.
- We define a novel technique to assign probabilities to execution paths.
- We introduce DeepFuzz, an algorithm combining initial seed generation, concolic execution, distribution of path probabilities, path selection, and constrained fuzzing.

2 Related Work

Symbolic execution has experienced significant development since its beginnings in the seventies to the advanced modern variants invented for program testing in recent years. Especially the last decade has seen a renewed research interest due to powerful Satisfiability Modulo Theory (SMT) solvers and computation capabilities that have led to advanced tools for dynamic software testing. Cadar et al. [2] give an overview of the current status of dynamic symbolic execution. In concolic execution [5, 10] symbolic constraints are generated along program execution paths of concrete input values.

Research in random test generation established powerful fuzzing tools such as AFL, Radamsa, the Peach Fuzzer, and many more. We refer to [12] for a comprehensive account.

Both concolic execution and fuzzing have severe limitations when aiming for code coverage (see Sect. 1). Since those limitations are partly complementary to each other, a fusion of concolic execution and fuzzing emerges as natural approach. Majumdar et al. [8] made a first inspiring step into this direction by proposing hybrid concolic testing: by interleaving random testing with concolic execution the authors of [8] increase code coverage significantly. However, it is still an open question how to efficiently generate restricted inputs for random testing. We propose a solution for high frequency test case generation that scales to large sets of constraints. Further, we specify the rather general test goals of [8] by focusing on maximization of the degree of freedom regarding input variables to achieve both, alleviation of vulnerability exploitation and execution paths that reach into deep layers of the program.

Closely related to our approach is Driller by Stephens et al. [11] who also combine fuzzing with selective concolic execution in order to reach deep execution paths. Driller switches from pure fuzzing to concolic execution whenever random testing saturates, i.e. gets stuck at a fuzzing wall. To keep the load for symbolic execution low while simultaneously maximizing the chance to pass fuzzing walls with concolic execution, Driller also selects inputs. This selection privileges paths that first trigger state transitions or first reach loops which are similarly iterated by other paths. In contrast, we systematically assign probabilities to paths based on SMT solving performance and select paths according to this probability distribution. This assignment of probabilities to execution paths has no direct counterpart in related work. Although the authors of [4] also propose assertion of probability weights to paths in the execution graph, they differ significantly in their proposed methods which are based on path condition slicing and computing volumes of convex polytopes.

3 The DeepFuzz Algorithm

In this section we present the DeepFuzz algorithm in detail. The main idea is interleaving concolic execution with constrained fuzzing in a way that allows us to explore paths providing maximal input generation frequency. We achieve this by assigning weights (corresponding to fuzzing performance) to the explored paths after each concolic execution step in order to select the ones with highest probability. In the following, we first describe the individual building blocks, namely initial seed generation, concolic execution, distribution of path probabilities, path selection, and constrained fuzzing. Next, we combine these parts in the overall DeepFuzz algorithm.

3.1 Initial Seed Generation

Initially we start with a short period of concrete input generation for the subsequent concolic execution. If the inputs belong to a predefined data format, we

generate inputs according to the format definition (as in generational fuzzing). If there is no format specified or available we just generate random input seeds. We denote the set of all possible concrete input values as X and the initial seeds generated in this initial step as $X_0 \subset X$.

3.2 Concolic Execution

The concolic execution step receives a set of concrete program inputs $X_{seed} \subset X$ and outputs a set of symbolic constraints collected along the paths belonging to these inputs. At the beginning, directly after the initial seed generation step, we set $X_{seed} = X_0$. The symbolic expressions are basically generated as described in Sect. 1. However, we adapt the path search heuristics to our approach in a similar way as introduced in [6]. We conduct concolic execution of the program with each input $x_i \in X_{seed}$ until one of the following two halt conditions occur: either the program reaches the predefined goal, which in our case is basically an unexpected error condition, or the number of newly discovered branches taken exceeds a fixed maximum $b_{max} \in \mathbb{N}$.

To keep the notation as clear as possible, in the following we assume without loss of generality that the halting conditions are reached after exactly b_{max} branches. Let c'_i denote the execution path belonging to input x_i and $n' = |X_{seed}|$ denote the number of inputs in X_{seed} . For each branch $j \in \{1, \dots, b_{max}\}$ there is a sub-path c'_{ij} which equals c'_i until branch number j is reached. Clearly, the c'_{ij} are sub-paths of c'_i . For each $i = 1, \dots, n'$ and $j = 1, \dots, b_{max}$ we store the logical conjunction of the negated branch condition λ_{ij} (corresponding to branch number j of execution path c'_i) and the path condition ρ_{ij} of the sub-path c'_{ij} leading to this branch, which yields the $n' * b_{max}$ expression sets $\phi_{ij} := \neg \lambda_{ij} \wedge \rho_{ij}$. With this notation, concolic execution of the input set X_{seed} yields the total set of constraints $\Phi := \{\phi_{ij} \mid i = 1, \dots, n', j = 1, \dots, b_{max}\}$. For each element in Φ the SMT solver checks if the symbolic constraints are satisfiable and in that case computes a new input x_{ij} for each element $\phi_{ij} \in \Phi$. These newly generated inputs x_{ij} drive the program execution along the original paths c'_i until branch number j is reached and then takes the alternative. We denote these new explored paths as c_{ij} . In the next step we assign probabilities to these paths. To maintain a clear notation and avoid too many indices we work with the union set

$$C := \{c_1, \dots, c_n\} := \bigcup_{i,j} c'_{ij}. \quad (1)$$

3.3 Distribution of Path Probabilities

Next, we describe our approach to assign probabilities to program paths. This step takes as input a set of paths C and outputs a probability distribution on this set.

One possible strategy is to calculate the cardinality $|I_i|$ of the set of solutions I_i for the path constraint $\phi_i \in \Phi$ corresponding to c_i and then define weights on the paths according to number of inputs that travel through it. This strategy

is chosen and comprehensively described in [4], where the purpose of assigning probabilities to paths is to provide estimates of likelihood of executing portions of a program in the setting of general software evaluation. In contrast to this we are interested in deep fuzzing and therefore must guarantee maximal possible sample generation in a fixed amount of time. To illustrate this more clearly, consider two sets of constraints Φ_A and Φ_B with (non-empty) solution sets A and B . If we are given only the constraints Φ_A and Φ_B and are interested in *some* solutions in A or B , we simply feed an SMT solver with the constraints and receive solutions. However, computing the cardinality $|A|$ and $|B|$ of *all* solutions corresponding to Φ_A and Φ_B (also called the model counting problem) can be significantly more expensive than the decision problem (asking if there is a single solution of the constraints at all). The authors of [4] rely on expensive algorithms for computing volumes of convex polytopes and integrating functions defined upon them. This would yield a theoretical sound distribution of path probabilities, with the disadvantage of extremely low fuzzing performance in our setting. Further, even if cardinality $|A|$ is significantly greater than $|B|$, meaning that Φ_A has much more solutions than Φ_B , computation of B may take much longer than computation of A . In other words $(|A| > |B|) \not\Rightarrow (T(\Phi_A) > T(\Phi_B))$, where $T(\Phi_i)$ is the time it takes an SMT solver to compute *all* solutions corresponding to the constraints Φ_i . To guarantee high frequency of model generation for effective deep fuzzing we have to build our strategy around a time constraint. Therefore, in order to assign probabilities to the paths c_1, \dots, c_n we apply another strategy.

For a fixed time interval T_0 let $k_i(\phi_i, T_0)$ denote the number of solutions for constraints ϕ_i that the applied SMT solver finds in the amount of time T_0 . Among the paths c_1, \dots, c_n we choose the one whose constraints yield - when given to the SMT solver - the maximal number of satisfying solutions in the fixed amount of time T_0 . Therefore, we distribute the probabilities $p(c_i)$ belonging to path c_i according to

$$p(c_i) := k_i(\phi_i, T_0) \left(\sum_{j=1}^n k_j(\phi_j, T_0) \right)^{-1} \quad (2)$$

for $i = 1, \dots, n$. With $\sum_{i=1}^n p(c_i) = 1$ this probability distribution is well defined.

3.4 Path Selection

Now that we have n explored paths $C = \{c_1, \dots, c_n\}$ weighted with probabilities according to Eq. (2) in the execution graph, our goal in this step is to select the paths that provide us maximal model generation frequency. Such a set of paths will guarantee us efficient fuzzing and maximal degree of freedom for subsequent payload generation in case we detect a vulnerability.

The defined probabilities $p(c_i)$ in Eq. (2) directly correspond to the performance in computing inputs for subsequent fuzzing. Practical calculation of those probabilities is efficient: we simply let the SMT solver compute solutions for the path constraints $\Phi_i (i = 1, \dots, n)$ in a round-robin schedule and count the number

of solutions for each path, which directly yields the probabilities $p(c_i)$. A sufficiently small choice of the computing time T_0 will result in fast path selection. To gain maximal input generation frequency, we could simply choose the single path whose assigned probability is maximal. However, some paths are dead ends and if we would restrict the algorithm to select only a single path for subsequent fuzzing, path exploration might stop too early in some binaries.

Therefore, we select the $m < n$ different paths \tilde{c}_j ($j = 1, \dots, m$) with highest probability. In order to make sure that the following path choice is well defined, we prepend a short side note first: it almost never happens in practice that there are two paths assigned with exactly the same probability. If this unlikely situation occurs in practice, we could just randomly choose one among these equiprobable paths and proceed without much changes in the subsequent algorithm. For simplicity of notation we assume without loss of generality that the set $\{p(c_i) \mid i = 1, \dots, n\}$ is strictly ordered. We initially choose the path with highest probability

$$\tilde{c}_1 = \arg \max_{c_i \in C} p(c_i) \quad (3)$$

and then proceed in the same way

$$\tilde{c}_j = \arg \max_{c_i \in C \setminus \{\tilde{c}_1, \dots, \tilde{c}_{j-1}\}} p(c_i) \quad (4)$$

until we obtain the path set $C_{high} = \{\tilde{c}_j \mid j = 1, \dots, m\}$ including the m paths with highest probability. On the one hand, setting the parameter m close to n will result in fast path explosion. On the other hand, setting $m = 1$ might be too restrictive for some binaries. Therefore, we initially set m to a small integer and then run parameter optimization to adapt to the specific binaries in testing experiments.

3.5 Constrained Fuzzing

Now that we have selected the paths C_{high} with highest probability, we continue with fuzzing deeper layers of the program. Remember we denoted the set of all possible concrete input values as X and the set of inputs belonging to path c_i as $I_i \subset X$ ($i = 1, \dots, n$). To start fuzzing into the program from an endpoint of a selected path $c_i \in C_{high}$, the generated fuzzing inputs have to fulfill the respective path constraints ϕ_i , otherwise they would result in a different execution path. There are basically three possible strategies to generate inputs (i.e. subsets of I_i) that satisfy the respective constraints:

Random Generation of Inputs with Successive Constraint Filtering. This strategy would initially generate a random input set $X_{rand} \subset X$, which would be given to an SMT solver in order to check if a concrete input $x \in X_{rand}$ satisfies the constraint ϕ_i and therefore belongs to I_i . However, filtering the generated inputs in X_{rand} by checking for satisfiability of respective path constraints would

most unlikely leave any input over, i.e. $X_{rand} \cap I_i = \emptyset$ with high probability. This is obvious due to the fact that the path constraints in ϕ_i symbolically represent all branch conditions along the path c_i , in particular fuzz-walls (as introduced in Sect. 1). Randomly generating input values that satisfy such a fuzz-wall constraint in ϕ_i is therefore clearly as unlikely as passing such a wall with pure fuzzing.

Pure SMT Solver-Based Input Generation. With this strategy we would inject all the constraints in ϕ_i into an SMT solver, that in turn computes a set of possible solutions. The problem with this strategy is that an SMT solver is sometimes slow and inefficient in computing solutions and the fuzzing input generation rate would drop significantly. This is due to the fact that an SMT solver cannot effectively handle large amounts of variables constrained in large amounts of equations. For example, consider a situation where the input consists of a large file F and the targeted program only checks a small part F' of it during initial parsing. Using an SMT solver to generate both the constrained part F' and the unconstrained part of F would be inefficient. This motivates the third strategy.

Random Generation of Independent Input Variables with Subsequent Constraint Solving. Here, we randomly generate input values for all variables that are independent (also called *free*) in ϕ_i . An SMT solver subsequently generates a model for the remaining dependent variable constraints.

In summary, the first strategy is infeasible, whereas strategies two and three are more similar to each other for small input sizes. However, if we deal with larger inputs where only a small minority of input variables are constrained by the current path constraint ϕ_i there is no need to feed a huge amount of path constraints for independent input variables into an SMT solver. We proceed with the third approach as it guarantees us maximal input generation frequency and scales better to large inputs.

In the following, we refer to the frequency of input generation for path c_i as $f(\phi_i)$. The above reasoning yields

$$f(\phi_i) \geq \frac{k_i(\phi_i, T_0)}{T_0}, \quad (5)$$

i.e. the number of models for ϕ_i found by the SMT solver in time T_0 is less or equal than the number of inputs generated with strategy three in time T_0 .

3.6 Joining the Pieces

Now that we have described all individual parts we can combine them for the overall DeepFuzz algorithm, as depicted in Fig. 1. After the initial seed generation (SG) is completed we run concolic execution (CE), distribution of path probabilities (DP), path selection (PS), and constrained fuzzing (CF) in a loop, where CF is run for a fixed amount of time T_1 . This loop is executed until a halt condition is reached. A halt condition is given either if a predefined goal (e.g. a


```

Input: Program  $P$ , Parameters  $m, k_{min}, T_0, T_1, T_2, b_{max}$ 

 $X_{seed} \leftarrow \text{SG}(P)$ 
do:
   $\Phi = \emptyset$ 
   $C = \emptyset$ 
  for each  $x$  in  $X_{seed}$  do:
     $c, \phi \leftarrow \text{CE}(x, b_{max})$ 
    append  $\phi$  to  $\Phi$ 
    append  $c$  to  $C$ 
   $Prob \leftarrow \text{DP}(\Phi, C, T_0)$ 
   $C_{high} \leftarrow \text{PS}(Prob, C)$ 
   $X_{seed} \leftarrow \text{CF}(C_{high}, \Phi, T_1)$ 
while  $\neg$  condition (11) ; i.e.  $(\sum_{i=1}^m k_i(\phi_i, T_0) \geq k_{min})$ 

 $\text{CF}(C_{high}, \Phi, T_2)$ 

```

Fig. 1. DeepFuzz main algorithm.

program crash) is reached, or if the constrained fuzzing performance collapses. In the latter case the total number of solutions that the applied SMT solver finds in the fixed amount of time $m * T_0$ drops below a predefined bound k_{min}

$$\sum_{i=1}^m k_i(\phi_i, T_0) < k_{min} \quad (6)$$

and we leave the loop to procede with solely constrained fuzzing for a long testing time T_2 .

4 Conclusion

We present an approach to trigger vulnerabilities in deep layers of binary executables. DeepFuzz constructs a tunnel into the program by applying concolic execution, distribution of path probabilities, path selection, and constrained fuzzing in a way that allows fuzzing deep areas of the program.

Instead of source code instrumentation, we only need compiled binaries for program testing. This is an advantage for the same reasons as stated in [7]. First, we are independent on the high level language and build processes. Second, we avoid any problems caused by compiler transformation after the build process, realized for example by obfuscation. Third, DeepFuzz is suited to fuzz closed source targets. Another important aspect of DeepFuzz is the ability to highly parallelize the proposed algorithm in Sect. 3. All intermediate steps can be modularized and distributed for parallel computing with a suitable framework. One disadvantage of DeepFuzz is that it is not directed towards a tagged point in the execution graph. It builds paths as deep as possible into the program, however

with no preferably direction. In order to address this issue we are currently considering how to combine our approach with previous work on driving execution of the input space towards a selected region. Such a directed exploration can be achieved by using fitness functions as described in [13]. For example, we could integrate fitness functions in the path selection step.

First tests targeting OpenSSL-based parsers of Base64-encoded X.509 certificates promise well. Here, we adapted the concolic execution framework Triton [9], which itself uses the Z3 SMT solver [3]. A comprehensive evaluation of our approach on a broad range of targets is subject to future work.

Finally, DeepFuzz may help to circumvent current bottlenecks related to automatic exploit generation as described by Avgerinos et al. in [1]. We expect that our proposed algorithm can be deployed for automatic exploitation of vulnerabilities deeply hidden in binaries.

References

1. Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. *Commun. ACM* **57**(2), 74–84 (2014)
2. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. *Commun. ACM* **56**(2), 82–90 (2013)
3. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 166–176. ACM (2012)
5. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: *ACM SIGPLAN Notices*, vol. 40, pp. 213–223. ACM (2005)
6. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. *Commun. ACM* **55**(3), 40–44 (2012)
7. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: *NDSS*, vol. 8, pp. 151–166 (2008)
8. Majumdar, R., Sen, K.: Hybrid concolic testing. In: *29th International Conference on Software Engineering, 2007, ICSE 2007*, pp. 416–426. IEEE (2007)
9. Sadel, F., Salwan, J.: Triton: a dynamic symbolic execution framework. In: *Symposium sur la sécurité des technologies de l’information et des communications, SSTIC, France, Rennes, 3–5 June 2015*, pp. 31–54. SSTIC (2015)
10. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: *European Software Engineering Conference*, pp. 263–272 (2005)
11. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2016)
12. Takanen, A., Demott, J.D., Miller, C.: *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Norwood (2008)
13. Xie, T., Tillmann, N., De Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: *IEEE/IFIP International Conference on Dependable Systems and Networks DSN 2009*, pp. 359–368. IEEE (2009)

Detection of Intrusions and Malware, and Vulnerability
Assessment

13th International Conference, DIMVA 2016, San
Sebastián, Spain, July 7-8, 2016, Proceedings
Caballero, J.; Zurutuza, U.; Rodríguez, R.J. (Eds.)
2016, XIV, 442 p. 103 illus., Softcover
ISBN: 978-3-319-40666-4