This publication has been published first by IEEE under
https://doi.org/10.1109/QRS-C55045.2021.00023

Attack-based Automation of Security Testing for IoT Applications with Genetic Algorithms and Fuzzing

1st Steffen Lüdtke
Business Unit Quality Engineering (SQC)
Fraunhofer Institute for Open Communication Systems
Berlin, Germany
steffen.luedtke@fokus.fraunhofer.de

2nd Roman Kraus
Business Unit Quality Engineering (SQC)
Fraunhofer Institute for Open Communication Systems
Berlin, Germany
roman.kraus@fokus.fraunhofer.de

3rd Ramon Barakat
Business Unit Quality Engineering (SQC)
Fraunhofer Institute for Open Communication Systems
Berlin, Germany
ramon.barakat@fokus.fraunhofer.de

4th Martin A. Schneider
Business Unit Quality Engineering (SQC)
Fraunhofer Institute for Open Communication Systems
Berlin, Germany
martin.schneider@fokus.fraunhofer.de

***Abstract*—The Internet of Things (IoT) has increased the connectivity of systems and fueled the digitalization. However, IoT systems are prone to various security vulnerabilities that may lead to serious incidents that can cause physical harm and financial damage. The security of IoT systems is a major challenge for the their success. If vulnerabilities remain undetected, an attacker can exploit them remotely and thus, gain full access to confidential data, control the connected IoT devices with magnitudes of consequences.**

This paper presents an approach that exploits information on attacks detected and recorded at runtime to facilitate the development of security patches and uses the information to detect further vulnerabilities. This paper presents work in progress from an ongoing research project.

***Index Terms*—Internet of Things, IoT, fuzzing, genetic algorithms**

I. Introduction

The global interconnectivity of the Internet of Things (IoT) devices is steadily increasing, creating new, innovative products and applications such as predictive maintenance, asset tracking and others. At the same time, increasing networking raises the associated risks and complexity. With growing complexity, however, bugs and vulnerabilities are almost inevitable. The research project "Attack-based Automation of Security Testing for IoT Applications"[1] addresses this issue with two goals:

- A complete automation of security testing from test design to test evaluation for efficient detection of unknown vulnerabilities (0-day vulnerabilities) using genetic algorithms, fuzzing, and data analysis techniques.
- Support developing effective security patches for vulnerabilities that cannot be bypassed by small variations of the original attack. For this, attack data from operations will be used to increase the quality assurance of security patches to effectively secure IoT applications.

Genetic algorithms (GA) are heuristic search and optimization methods that are inspired by biological evolution. In particular, the concepts of inheritance, selection, crossover and mutation are used to find a solution for a given problem. There are various research papers, books, implementations, and frameworks to solve problems with GA. In order to use a GA framework, the given problem has to be encoded in a generic representation on which the GA framework applies its genetic operators. For complex problems, finding an encoding that provides a problem representation so that the GA efficiently finds a solution can be a major challenge. Another approach therefore is to bypass encoding with an existing framework and to implement the necessary genetic operators for the problem domain. This paper investigates both possibilities.

In addition to GA, the second topic examined is fuzzing. Fuzzing has long been established as a topic in the field of security testing, but unlike GA, it does usually not draw conclusions from previous test cases about how to make the next test cases more effective, in the sense of generating, semi-invalid input data. How to improve the data generation in fuzzers with the help of GA, and thus potentially find vulnerabilities in a system faster and more effectively, is part of this work.

The paper is organized as follows: It begins with an introduction to the field of genetic algorithms including an overview of the current state of the art. Afterwards, our approach to employ GA for security testing is explained and compared to other works. The presented approach is also considered from the implementation point of view. A first evaluation on a prototypical application is presented. The

[1]funded by the German Federal Ministry for Economic Affairs and Energy

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

paper closes with a conclusion and an outlook.

II. Related Work

A. Genetic Algorithms

Genetic algorithms are heuristic optimization methods inspired by biological evolution. To find a solution for a problem, they maintain a set of candidate solutions. These candidates are evaluated with respect to their ability to solve the problem. Based on this evaluation, the most suitable candidates are selected, and their characteristics are recombined and mutated to create a new set of solutions. This process is repeated in an iterative manner. The goal is to generate better candidate solutions with each iteration.

We briefly describe here the main terms of GA that are used in the rest of this paper (therefore, the list is not complete):

- **Individual:** A single candidate solution for a given problem. In security testing, a candidate could be a SQL injection string.
- **Gene:** An atomic characteristic of an individual. An individual usually consists of several genes. In the case of a string, a gene may refer to one of its letters.
- **Genotype:** The data structure of an individual. In security testing, the genotype refers to the test data and the series of test steps to interact with the system under test.
- **Chromosome:** A genotype in the form of a fixed length vector [11]; rarely also used as a synonym for gene.
- **Generation:** The current set of individuals during one iteration. They are subject to a fitness assessment to subsequently produce a new generation from the best individuals. In testing, a generation may be the recently generated test suite.
- **Population:** All the individuals from all generations.
- **Selection:** Describes the process of selecting suitable individuals for the production of the next generation.
- **Parent:** Refers to those individuals that haven been selected for the production of the next generation.
- **Child:** An individual of a new generation derived from one or more parents through mutation or recombination.
- **Fitness:** A value which is calculated for the selection of candidates for the next generation. The fitness makes a statement about how well an individual or generation solves a given problem. It is highly problem specific and determines how efficiently a GA is able to find a solution.

B. Phases of Genetic Algorithms

Each GA is characterized by a loop that is exercised for each generation. It comprises the following steps:

Selection of suitable individuals (fitness and selection): During this step, the individuals of the current generation are examined with regards to their suitability for solving the given problem. In security testing, individuals often represent test cases (e.g., a SQL injection string). To assess their suitability, they are executed against the system under test (SUT). Based on the observed behavior of the SUT, each test case is assigned a fitness value (see Section II-A), which provides information about the quality of the candidate and its ability to find a vulnerability.

From the pool of individuals from the current generation, suitable individuals are selected for the next phases (for instance, the 10 % of individuals with the highest fitness). The selected individuals are called parents.

Recombination: In this phase, the traits (genes) of the parents are recombined to generate new individuals, see Section II-B2. In recombination, only existing traits are recombined, but no new traits are generated.

Mutation: Mutation alters one or more genes of an individual to change its characteristics. New traits are needed to explore the search space beyond the initial generation. Typically, the frequency and type of mutation are chosen randomly within predefined thresholds, cf. Section VI.

Selection, recombination and mutation are so-called genetic operators. Evolutionary optimization (i.e., GA) executes these operators iteratively and uses the results of the final iteration to find the solution for the investigated optimization problem.

Figure 1 shows an iteration step and the operators. The step "execution/measurement" takes a specific role: it is not an operator in the sense of a genetic algorithm. However, it belongs to each iteration and examines the individuals of the current generation with respect to their suitability to solve the given problem. This is done by determining their fitness.

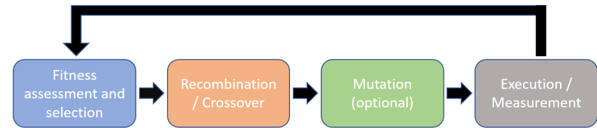

Fig. 1. Cycle of optimization through the different phases of a genetic algorithm

An important aspect when applying GA is the termination criterion. This criterion determines when a found solution is considered sufficient. Trivially, this is the case when we reach an optimal solution or when we pass a predefined fitness threshold. It is more complex if this fitness threshold cannot be determined a priori. There are three basic approaches that can be used as termination criterion:

- Terminate after a fixed number of generations.
- Terminate if the fitness value deteriorates in one generation compared to the previous generation(s).
- Terminate if the fitness value remains largely the same over a number of generations.

The selection of one or several of these termination criteria is problem-dependent and subject of current research.

We will now discuss the concepts of fitness, recombination and mutation in more detail.

1) Fitness: The fitness of an individual is typically calculated with a dedicated function. This "fitness function" is a core concept of every genetic algorithm. The obtained fitness values are used to select a set of individuals from which a

new generation is derived. Two basic approaches can be distinguished: single-objective and multi-objective optimization. In *single-objective optimization*, we have *one* fitness value per individual and thus, exactly one optimization goal. The fitness values of all the individuals are calculated in the 4th iteration step (execution/measurement). They may be based on a comparison to an ideal solution or on measurements (like time lags and resource consumption). A simple example of single-objective optimization would be a normalized sum of the different measurements. Other combinations are also possible, e.g., by means of weighting. The aim is always to generate a single, cumulative value from all measurements that contribute to the fitness.

In *multi-objective optimization*, there is no single fitness value per individual but rather several independent ones. A simple example would be an algorithm that should calculate as quickly as possible (fitness value 1) and as close as possible to the optimal solution (fitness value 2) at the same time. Both fitness values are independent in this example and presumably hinder each other. Solutions can be found that are very fast, but only sufficiently optimal. Or they can be close to the optimum, but take long to calculate. Multi-objective optimization is useful if not all goals can be reached at the same time and an optimal trade-off is needed. Depending on the given problem, there can be also more than two fitness values. A solution is always considered better than the last one, if at least one value is better than before, while the rest is not worse. This group of solutions with optimal trade-offs is called Pareto-optimal.

2) Recombination: Recombination refers to the combination or exchange of traits (or genes) from two or more individuals to generate new individuals. Usually, two parents are merged to form one or more children. A common recombination procedure is crossover. There are several crossover types. In one-point crossover, the chromosomes of two parent individuals are separated at a random location and one half each is exchanged between the two, cf. Figure 2. This results in two new individuals that have mixed traits from the parents. Two-point crossover and multi-point crossover are working analogously.

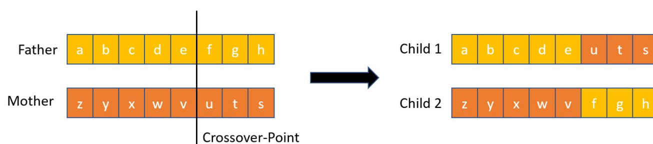

Fig. 2. Example of one-point-crossover for two parents comprising eight genes

The goal of recombination is to create individuals with even higher fitness values than those of the parents. During recombination, only existing traits of one generation are exchanged and no new traits are created. Besides simple exchange recombination, there are also other recombination methods. For example, line recombination and intermediate recombination [11] are useful when individuals are represented in a numerical encoding. Furthermore, Luke [11] shows that it is possible to perform recombination in tree-like individuals by exchanging subtrees between the individuals.

3) Mutation: Mutation is the process of randomly changing one or several genes of an individual. While recombination attempts to combine existing traits of two or more individuals into a new, fitter individual, mutation attempts to introduce new traits into the population. An important parameter is the mutation rate. It controls how many genes are changed per mutation operation.

If one were to dispense with mutation altogether, the genetic algorithm would only ever recombine the properties of the initial generation and would not be able to explore the search space further to find local or global maxima. This would be equal to a mutation rate of zero. However, is the mutation rate set too high, the heuristic search becomes more of a randomized search. Individuals would experience so many differences from one generation to the next one that they would rather skip one or more local optima rather than approaching them. Choosing the mutation rate should consider both the concrete implementation and the problem to be solved.

Several types of mutation operators have been proposed, e.g. insert mutation swap mutation and flip mutation [14]. In [17], [18] *adaptive mutation* is proposed. This refers to mutations with rates that are not fixed, but change adaptively during the course of an GA.

The selection of a suitable mutation type cannot be predetermined in a general way and is always problem-specific. The mutation rate can also vary depending on the problem. Mutation types continue to be the object of research. Suitable mutation operators and mutation rates can lead to better results faster when solving a search or optimization problem with GA.

C. Genetic Algorithms and Fuzzing

In order to apply genetic algorithms for the purpose of security testing, some basics of testing and security testing have to be considered. Whole test cases, single test messages or test data can be subject of the optimization through a GA.

1) Fuzzing: Fuzzing is a software testing technique in which a SUT is executed with unexpected, invalid test data and the response is observed and evaluated. It is a highly-automated approach to cover a high number of boundary cases [12] to identify potential vulnerabilities, e.g., resulting from missing or invalid input validation. Fuzzers can be categorized in different ways. Schneider et al. [13] (and in a similar way Knudsen [7]) propose a classification according to the degree of adaptation of the fuzzer to the SUT and according to how this adaptation is achieved. They distinguish for instance between random fuzzers, template-based fuzzers, dynamic fuzzers, and model-based fuzzers.

The adaption controls how the input data is generated. A random fuzzer uses purely randomly generated data to test a system. This is easy to implement but promises little chance of success. A random fuzzer will usually fail already at the input check of the generated test data and has no chance to penetrate deeper into a system where the data of the message

is processed. This is contrasted by approaches like model-based fuzzing. In this, fuzzers have full knowledge of the SUT. Hence, they are also called specification-based fuzzers. Such fuzzers are not only capable of fuzzing the message structure, but also the communication flow [16].

Regardless of the type of fuzzer, classical fuzz testing is about confronting the SUT with a large amount of invalid data and evaluating its response. Originally, fuzzing is not a search or optimization problem, where knowledge from previous fuzz tests is used to improve the results.

The goal of the research project "Attack-based automation of security testing for IoT applications" is to combine the fuzz testing approach with genetic optimization. An algorithm for fuzz test case generation using a genetic algorithm from existing test cases is to be designed and implemented.

D. Related Work

Evolutionary methods have already been studied and used in the field of software testing.

For example, in "Detection of Web Vulnerabilities via Model Inference" [4], Duchene describes how EA can be used to detect cross-site scripting (XSS) vulnerabilities. XSS is an attack on a web application in which an attacker attempts to inject code into the system being attacked, which is then executed as part of the web page in the victim's browser.

A key aspect of Duchene's work in this regard is the fitness function (cf. Section 3.1). Various metrics are proposed, including the number of character classes injected successfully, number of suspicious nodes, number of transitions between the source and its reflection, to calculate a fitness value for XSS detection.

For the generation of fuzz data, Duchene employs a grammar-based fuzzer. For this, Duchene uses a grammar designed specifically for XSS attacks to generate suitable words, i.e., XSS attacks.

In "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing.", DeMott et al. [3] propose a grey-box fuzzing approach, which uses code coverage as the primary source of information for the fitness function. They show in their paper that such an evolutionary testing tool is capable of actively learning the protocol under test and yields better results than fuzzers which simply mutate captured messages or derive them from specifications.

Today's fuzzing tools like American Fuzzy Lop [6] and libFuzzer [8] also employ genetic algorithms to perform gray-box fuzzing while using code coverage as a fitness function.

III. Our Approach

For our approach, we decided to pursue a hybrid approach between using a framework for GA and developing an own implementation and to evaluate it in the later work packages. The selected approach can be summarized as such:

1) Use existing GAs and frameworks for simple user data.
2) Develop custom evolutionary algorithms for (more complex) messages and test cases

For the second approach, it is necessary to implement the genetic operators mutation and recombination and the representation of individuals according to our needs. Grammars have been chosen as the central component to generate inputs. A grammar based, evolutionary fuzzer has already been implemented in [4] for detecting XSS vulnerabilities (cf. Section II-D). An approach based on grammars is generic and widely applicable. It requires only a suitable grammar to generate test inputs. The underlying fuzzing and GA routines can stay generic and can thus be employed for many different protocols or languages. Another advantage is that grammars are already available for many communication protocols. Furthermore, grammars can be used to represent the message sequence / protocol. They are thus not only useful for describing the content structure of inputs. However, in the following we will use grammars only to represent the content structure of inputs.

An example of a simple grammar, including its production tree can be found in Figure 3. Each word of a grammar can be seen as a sequence of derivation decisions starting from the start symbol. Grammars are very well suited for fuzz testing. A suitable fuzz testing tool can randomly generate any syntactically correct word of the language. It should be noted that grammars generally only represent the syntax of a language, but disregard many semantic requirements. This directly leads to semi-invalid input data, because the inputs are syntactically correct, but likely not semantically. Such inputs are particularly valuable for fuzz testing.

To use grammar based fuzzing with GA, the operators mutation and recombination need to be adapted to grammars. In the following, we will describe our approach:

Mutation: In grammar based fuzzing, each word is the result of a sequence of derivation decisions. Therefore, to mutate a derived word, one of these derivation decisions has to be mutated. This is exemplified in Figure 7. The derivation decision for the OR operator (" | ") is changed and the word is thus mutated from the string "a" to "b". Mutating a derivation decision closer to the root of the derivation tree usually leads to a far reaching mutation. The mutated word is therefore very different from the original word. If a derivation decision is mutated closer to a leaf or terminal, the difference between the two words is typically smaller. This is relevant for genetic search procedures: If a solution is already close to a local or global maximum, rather small search steps are advisable to remain in the vicinity of the maximum. If, on the other hand, the solution is far away from a maximum, larger mutations are more likely to be successful.

Furthermore, it should be noted that with ambiguous languages, the same word can be generated from different derivation paths. Thus, in the case of a mutation of word#1, it can happen that the resulting word#2 is the same as word#1. Such occurrences need not be a problem for the genetic algorithm if they are rare, but should be avoided for efficiency.

Recombination: Analogous to mutation, recombination can also be realized with the help of derivation decisions (or rather their subtrees). To realize recombination, one can exchange subtrees of decisions which start at matching grammar sym-

bols. This is illustrated in Figure 8. Two derivation trees are formed from the given grammar S ->1*3 ("a" | "b"). The recombination or crossover then takes place at the derivation decision from the OR operator (" | "). The entire subtree starting from the OR operator is copied from word#1 to the corresponding position in word#2 and vice versa. The result is two new words that are a mix of the two original words. Essential for this type of crossover is that both original words have the selected crossover root symbol in their derivation trees. Otherwise, no crossover would be possible. This is illustrated in Figure 5. You can see that a crossover on the symbol "A" would be not possible as it has not been derived in the other word (due to a different derivation decision above). This problem is not trivial, because with more complex grammars the effort increases greatly for the identification of the derivation points at which a crossover is possible.

Similar to mutation, it can be assumed that also in recombination, a crossover near the leaf of a derivation tree leads to new words with only minor changes compared to the initial values, while a crossover on higher tree layers leads to strong changes. Knowing that can again be useful for approaching local optima in the search space. Similarly, there is a chance that the subtrees selected for crossover will be identical. This should be avoided.

IV. Implementation

This section gives a deeper insight into how genetic operators for grammars have been implemented and illustrates the theory described before. Practical implementation problems are also discussed and solutions proposed.

A. The Implementation Approach

As discussed in Section III, we are using a hybrid approach to employ genetic algorithms. This approach considers the use of GA frameworks as well as own implementations. The framework we chose is ECJ. Our own implementations have been implemented by extending Fuzzino. In the following, we will discuss both tools and our results with them.

B. ECJ

"ECJ is a mature and widely used evolutionary computation library with particular strengths in genetic programming, massive distributed computation, and coevolution" [10]. ECJ stands for Evolutionary Computation Toolkit in Java and is a framework for GA. It was developed by Sean Luke, a computer science professor at George Mason University, and offers extensive features for solving optimization problems with evolutionary algorithms. Information about it can be found in [10], a paper about the history and planned extensions, the ECJ Handbook [2] and of course on the project's website [9].

ECJ is particularly suitable for optimization problems which can be easily encoded with its standard representations (especially vectors). However, it tends to be more intricate for problems which do not naturally lend themselves to such representations (e.g., test cases consisting of an arbitrary number of test steps). In this case you will likely have to implement your own representation of individuals.

This is one of the reasons which informed our decision to use GA frameworks mainly for simpler data types. By doing this, we can especially benefit from their strengths. ECJ has performed well in experiments with standard optimization problems and also custom ones (i.e., SQL injection input generation). Nonetheless, our focus mainly shifted towards the development of our own tools, as it offers more flexibility. So far, we can report that ECJ has shown promising capabilities for the usage in GA-based fuzzing. However, deeper investigations are still to come.

C. Fuzzino

Fuzzino is a fuzzing library which has been developed by Fraunhofer FOKUS. It constitutes a *smart fuzzer* as it employs knowledge about the input structure to generate test data. The library has been implemented in Java and is available as open source [5]. It provides several generators as well as mutation routines for standard datatypes (e.g., dates, hostnames, SQL injections). Furthermore, it is possible to extend Fuzzino with custom heuristics[2] for other datatypes, as demonstrated in [15].

In the following, we first discuss the grammar fuzzing features of Fuzzino[3]. Next, we will describe what is necessary to extend Fuzzino with GA capabilities. Finally, we present our concrete implementations.

1) Grammer-based Fuzzino: Fuzzino does not only provide generators and mutation routines for standard datatypes but is also able to handle grammars in the Augmented Backus-Naur Form (ANBF). It can thus derive syntactically correct words for a given grammar. Moreover, it also able to generate words which violate its structure. This is realized by fuzzing the grammar itself and then deriving words from that modified grammar.

Generating syntactically invalid words can be very important. Consider for example buffer overflows. These can lead to malicious code execution by going beyond the syntactic max length of an input. Nevertheless, violating the grammar can also drastically reduce the chance of passing initial checks of the SUT. For instance, because the input requirements are very restrictive. Therefore using syntactically invalid inputs can decrease the probability of uncovering issues in deeper code layers.

To deterministically generate each word of a grammar, Fuzzino uses derivation decisions (in the following often simply referred to as decisions). These are predefined selections for grammatical operators with degrees of freedom, e.g., an alternation. For instance, it could specify which selection to take in an alternation. Decisions can therefore be compared to derivation steps, since derivation steps also resolve the non-determinism of operators by making derivation choices. Figure 3 depicts an example of a word, its grammar, and the necessary decisions to derive that word from the grammar. Nodes which

[2]We use the term heuristic to refer to generation and mutation routines.

[3]Please note, these capabilities are not available in the open source version.

have been used in the derivation are highlighted green. We use blue boxes to indicate taken decisions. Our grammar notation is based on the ABNF.

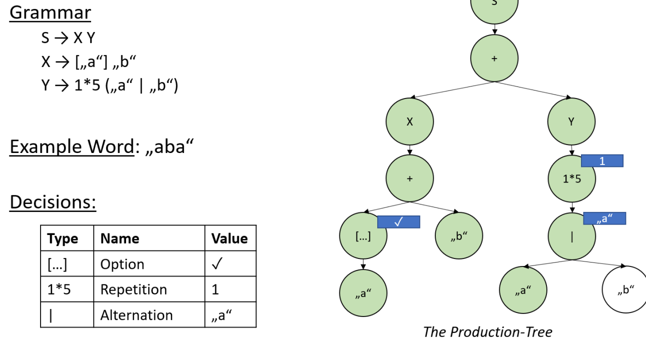

Fig. 3. Decisions in grammars

Each word can be defined by its unique combination of derivation decisions. Fuzzino systematically generates different permutations of decisions and uses them to derive the respective words. In Fuzzino, each decision is associated with its node[4] in the production tree. Thus, each node can check whether there is a predefined decision for it. The derivation process is realized with a recursive top-down traversal of the production-tree. Terminal strings are generated at the leaves and passed back up. These strings are processed by the upper nodes (mostly concatenated) which produces the final output once traversal finishes.

To control the derivation procedure, a premade set of decisions is passed initially and handed down in each descend. Operator nodes can thus check whether there is a premade decision for them or not. If so, they use that decision. Otherwise, they make a random choice.

2) Tasks and Procedures for Extending Fuzzino with GA: To use Fuzzino for evolutionary fuzzing we made some extensions to the original source. A first issue was that we could only set one decision per operator node. This is not enough, as an operator node can be visited multiple times in complex derivations. It is therefore necessary, to be able to define multiple decisions per operator node. We illustrate this with an example in Figure 4. The alternation here is visited several times due to the repetition. On each visit a new selection is made between the terminals *a* and *b*, overall, three times. If we could only store one of those decisions for the alternation, then we could not fully describe the word.

Another challenge was that derivations did not consider their derivation path. This means that decisions had no information about to the full derivation path at which they shall apply. They so far only knew its last element, namely their operator node. This can be problematic if we have different decisions for the same operator node. In this case, the operator could not decide which decision to take. This can also be illustrated with

[4]We use the term node to refer to one instance of an operator or non-terminal in the production tree.

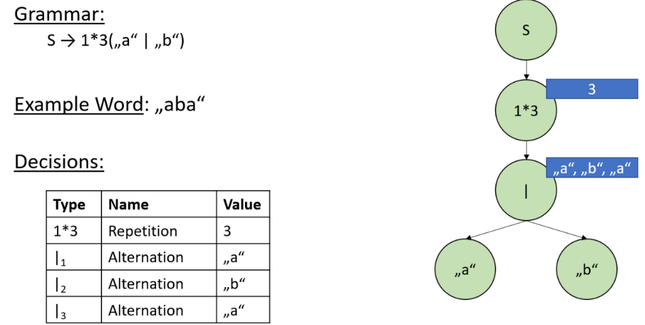

Fig. 4. Multiple decisions per non-terminal

Figure 4. As we have multiple decisions for the alternation, we need to know when to take which. This can be realized by associating each decision with its full derivation path. For instance, a mapping of the path ”S $\rightarrow$1*3$\rightarrow$ $|_1$” to the terminal ”a”. Moreover, we also needed to introduce completely new routines to randomize the values of decisions for mutation or to exchange groups of decisions for crossover.

These operations also raise some additional concerns, e.g., how to prevent mutation and crossover from taking place disproportionately often close to the derivation tree leaves. Intuitively, one could select the decisions for mutation and crossover randomly over all available ones. However, many decisions occur only towards the bottom of the tree. Consider for instance repetitions of terminal selections, e.g., a repeated selection of alphanumeric characters. This might for example occur if you randomly generate the content of an html-tag. Such repetitions occur immediately above the tree leaves, as they deal with terminals. If we have many such selection repetitions (which are furthermore quite long), they can easily contribute the majority of decisions. Especially if the tree is rather shallow and we thus only have few decisions in the regions above. If we would thus randomly select decisions uniformly over all available ones, the chance would be higher to select ones near the leaves. At the same time, decisions closer to the leaves are usually not as important for the overall structure of the derivation tree. This is because they typically only affect small subtrees. Consequently, they would be also less interesting for mutation and crossover.

Another concern is to ensure that the operator or non-terminal we crossover on is contained in both words. Otherwise, we could not perform the crossover as we could not exchange the subtrees (or rather the decisions belonging to the subtrees)[5]. This situation is depicted in Figure 5. Here, a crossover on the non-terminal *A* is impossible, as it has not been derived in the second word.

[5]Please mind: We often speak about exchanging subtrees when performing crossover. However, our implementation works like this that we only exchange decisions. Namely, all decisions which belong to the corresponding subtrees that we conceptually exchange. This has an equivalent effect to the exchange of derivation trees.

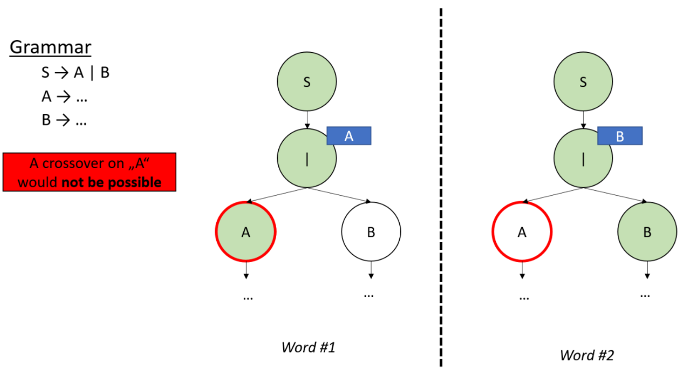

Fig. 5. Example of a failed crossover

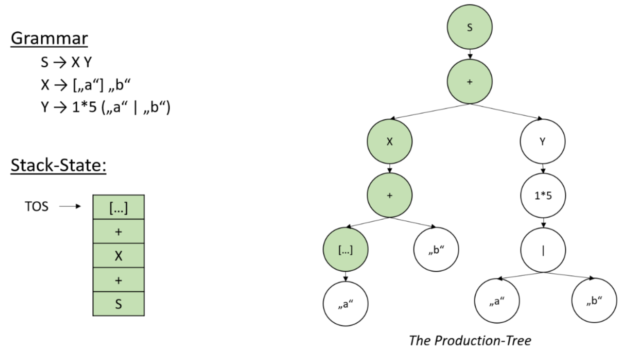

Fig. 6. The stack state during a derivation

Finally, it is also important to prefer nodes for crossover which would result in a change of the original words. Because it could easily be the case that we have subtrees in two words which are identical. If we would exchange those (or rather their decisions), then we would have practically no effect from the crossover. Thus, it is important to focus on decisions which differ in their subtrees.

3) The implemented approach: The first issue we resolved was that we could only store one decision per operator node. We fixed it by storing a list of decisions for each operator node (instead of one at max).

The next thing was that we needed to resolve the selection ambiguity for decisions which appear at the same operator node. To solve this, we now associate each decision with its full derivation path and not only its operator node. For this we introduced a tracking of the derivation path. This is realized with a stack which pushes nodes on itself when visiting them. Pushed nodes are removed once we recursively ascend out of them. The current state of the stack thus represents the current derivation path. We illustrate this in Figure 6. It shows the relationship between the stack content and the derivation path. Due to this relationship, we can use the stack to associate decisions with the path at which they have been made (e.g., to store this information for later usage). Conversely, the stack can also be used by operator nodes to check whether a premade decision fits to the current path. It is important to mention that we also use indices for each node on the derivation path. This is to distinguish different visits of the same operator node on one subtree layer. An example of this is presented in Figure 4. You can see that we use indices to distinguish different alternation-decisions occurring at one layer.

To implement the evolutionary operations mutation and crossover, we have added methods which allow to adapt and exchange derivation decisions. We have implemented mutation such that we first select a random decision. The value of that decision is then non-deterministically changed. Finally, a new word is derived based on the now adapted set of decisions. This adapted set is a copy of the original one so that we can keep the original one intact for later operations. We have provided an example of a simple mutation in Figure 7. In this example, the alternation-decision is set from “a” to “b” after the mutation.

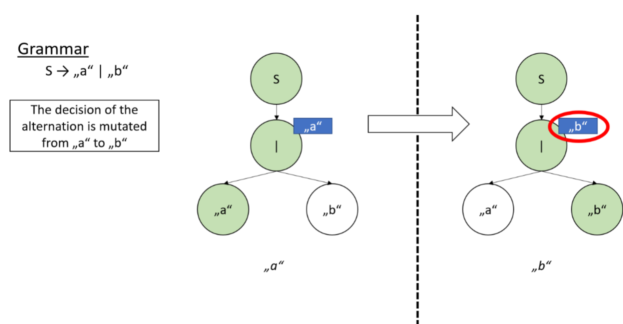

Fig. 7. Example of a mutation

The crossover implementation is more complex. For one, it takes two words as input (since we are exchanging subtrees between two words). Moreover, it also requires more processing steps. First, we select two random but matching decisions from these words. By matching we mean that they belong to the same operator type or non-terminal (e.g., they both belong to an alternation). The selected decisions serve as the roots of the subtrees to be exchanged. Second, we identify all decisions which depend on these roots. This refers to the decisions which are made as a consequence of these root decisions. Finally, these decisions are relocated to their new position in the other word. We ensure to also update the derivation paths of the decisions accordingly, as they are moved to a new tree structure. Finally, we use these updated sets of decisions to generate two new words. These words constitute the result of the crossover. During the entire procedure we again only work on copies to keep the original words intact and separate. We have provided a simple example of a crossover in Figure 8. You can see that we exchanged the second alternation decision of the first word (b) with the second one of the second word (a). This results in two new words where the corresponding letters are exchanged at the respective places.

Root decisions for crossover are randomly and uniformly selected over the tree levels. This means that we first randomly select a tree level. We then use this level to select a random

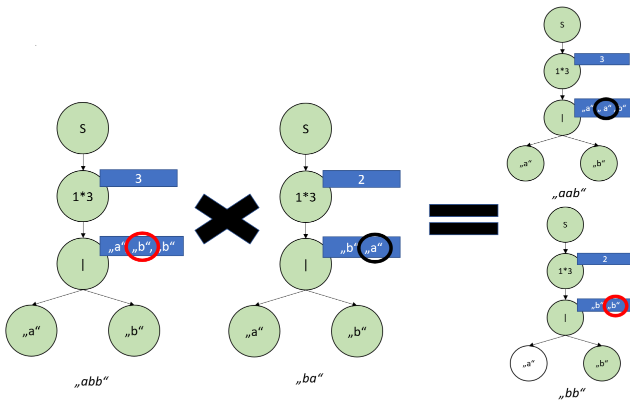

Fig. 8. Example of a crossover

decision at that level. This decision serves as the first root of the crossover. Both selection processes follow a uniform distribution. By doing this, we can prevent selecting disproportionately often crossover-roots near the tree-leaves. We use an analogous procedure for the selection of mutation decisions. As mentioned before, it is important for crossover to select root-decisions whose operators are present in both words. To achieve this, we limit our candidates to those decisions which fulfill that criterion during the selection.

Finally, we also optimize our chances of creating changed words when performing a crossover. This is done by tweaking the selection of the second crossover-root. If we have several candidates at our disposal, we always favorite those which have a different decision subtree compared to the first root. This maximizes our chances of generating changed words after performing the crossover.

V. Preliminary Evaluation

To evaluate the Fuzzino extensions towards a genetic fuzzing tool, we used the Damn Vulnerable Web Application (DVWA) [1] to search for XSS reflection vulnerabilities. We use the attack grammar from Duchene [4] in a slightly adapted form. To evaluate the fitness of the individual test data similar to the approach in [4] the reflection is examined for the following properties:

- The corresponding reflection was checked for certain characters. The characters of interest were divided into different classes C. If a reflection contains a character of a class c_j (e.g. c_1={<, >}, c_2={", '}, ...), the fitness is increased by the value w_j.
- In addition to the characters that appear, their arrangement in the reflection is also crucial for the fitness value. Thus, reflections containing an opening angle bracket followed by a closing angle bracket have a higher fitness value than reflections containing an opening but no closing bracket or a closing bracket before the opening bracket.
- The reflection is compared with the test data regarding similarity. The more similar the test data and the reflection are, the fewer characters have been modified by the system and the more reliable it is to find a potential XSS vulnerability.

For the test execution, we randomly generate 1000 words. These were then submitted to the DVWA and their fitness values calculated. For the development of the next generation, the 250 fittest individuals were selected. Using the mutation and recombination operators described above, the next generation was created from this set, which was in turn submitted to the DVWA to calculate their fitness values. These new individuals (children) are then added to the previous set of individuals, which was then reduced again to the 250 fittest. By using this very simple algorithm, it was possible within 23 iterations to produce a test data which points to a XSS-reflection vulnerability, cf. Table I, no. 1. The three words of the last generation with the highest fitness can be found in Table I.

No	Test data	Fitness
1	$< div\ onload = "alert(123)" > \ /div >$	81.83
2	$< svg\ onbeforeunload = alert(123) > /svg >$	81.75
3	$< svg\ onerror = alert(123)' > \ /svg$	78.88

TABLE I
Three fittest test data after 23 iterations

Figure 9 shows how the fitness values of the individuals evolved from one generation to the next. Each generation consists of 250 individuals. In the first five iterations, it is noticeable that the individuals have a fitness below 65 and that the individuals move closer and closer to this value. In the generations eight and nine, we observe jumps in the fitness values resulting from significantly fitter individuals generated by the genetic algorithm. In the following generations, it can be seen that the number of individuals with a fitness value close to 82 is continuously increasing.

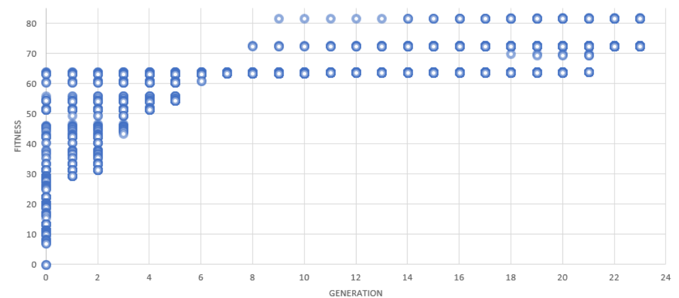

Fig. 9. Evaluation of Individuals

VI. Summary and Outlook

This paper summarizes first results on genetic fuzzing on grammars to identify vulnerabilities in IoT applications. In the first part, we presented an introduction to genetic optimization and search algorithms and adapted them to the field of software testing. In the second part, we presented two tools and their extension for genetic fuzzing using grammars. The main result

is the idea to follow a hybrid approach for the adaptation in the area of security testing for IoT that uses a traditional GA framework to evolve test data for simple data types using a specific encoding and genetic operators specifically implemented for grammars to evolve test data for complex data types.

For the next steps, it is planned to refine parameters of the genetic algorithm, e.g., the mutation rate, and to apply our approach to a case study from the research project for a realistic evaluation, targeting additional vulnerabilities than XSS. We will combine the implementation in Fuzzino presented in this paper with the capabilities of ECJ according to the hybrid approach. We will evaluate whether our approach is able to find vulnerabilities more efficiently than traditional fuzzing approaches and investigate how to adapt the fitness function to cover more than XSS vulnerabilities.

REFERENCES

[1] *Damn Vulnerable Web Application (DVWA)*. https://dvwa.co.uk/

[2] *The ECJ owner's manual* https://cs.gmu.edu/~eclab/projects/ecj/manual.pdf

[3] DEMOTT, J. ; ENBODY, R. ; PUNCH, W. F.: Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In: *BlackHat and Defcon* (2007). https://www.blackhat.com/presentations/bh-usa-07/DeMott_Enbody_and_Punch/Whitepaper/bh-usa-07-demott_enbody_and_punch-WP.pdf

[4] DUCHENE, F. : *Detection of web vulnerabilities via model inference assisted evolutionary fuzzing*, Grenoble University, Diss., 2014

[5] FRAUNHOFER FOKUS: *Fuzzino*. https://github.com/fraunhoferfokus/Fuzzino

[6] GOOGLE: *American Fuzzy Lop*. https://github.com/google/AFL/releases

[7] JONATHAN KNUDSEN: *What is Fuzz Testing: The Poet, the Courier, and the Oracle* https://www.synopsys.com/software-integrity/resources/white-papers/what-is-fuzzing.html

[8] LLVM: *LibFuzzer*. https://llvm.org/docs/LibFuzzer.html

[9] LUKE, S. : *ECJ 27, A Java-based Evolutionary Computation Research System*. https://cs.gmu.edu/~eclab/projects/ecj/

[10] LUKE, S. : ECJ then and now. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. New York, NY : ACM, 2017. – ISBN 9781450349390

[11] LUKE, S. : *Essentials of metaheuristics: A set of undergraduate lecture notes*. Second edition, online version 2.0. Morrisville, N.C. : lulu.com, June 2013. – ISBN 978–1–300–54962–8

[12] OEHLERT, P. : Violating Assumptions with Fuzzing. In: *IEEE Security and Privacy Magazine* 3 (2005), Nr. 2, S. 58–62. http://dx.doi.org/10.1109/MSP.2005.55. – DOI 10.1109/MSP.2005.55. – ISSN 1540–7993

[13] SCHNEIDER, M. A. ; GROSSMANN, J. ; TCHOLTCHEV, N. ; SCHIEFERDECKER, I. ; PIETSCHKER, A. : Behavioral Fuzzing Operators for UML Sequence Diagrams. In: HAUGEN, Ø. (Hrsg.) ; REED, R. (Hrsg.) ; GOTZHEIN, R. (Hrsg.): *System Analysis and Modeling: Theory and Practice - 7th International Workshop, SAM 2012, Innsbruck, Austria, October 1-2, 2012. Revised Selected Papers* Bd. 7744, Springer (Lecture Notes in Computer Science), 88–104

[14] SIVANANDAM, S. N. ; DEEPA, S. N.: *Introduction to genetic algorithms*. Berlin: Springer, 2008. – xix + 442 S. http://dx.doi.org/10.1007/978-3-540-73190-0. http://dx.doi.org/10.1007/978-3-540-73190-0. – ISBN 978–3–540–73189–4

[15] STEFFEN LÜDTKE: *Adaption des Fuzztest-Ansatzes für HL7: Erstellung von Fuzzing-Heuristiken, Implementierung einer TTCN-3 basierten Testumgebung sowie Evaluation in einer Fallstudie: Diplomarbeit*. 2016

[16] TAKANEN, A. ; DEMOTT, J. ; MILLER, C. : *Fuzzing for software security testing and quality assurance*. Boston, Mass. : Artech House, 2008 (Information security and privacy series). – ISBN 9781596932142

[17] THIERENS, D. : Adaptive mutation rate control schemes in genetic algorithms. In: *Proceedings of the 2002 Congress on Evolutionary Computation, CEC'02*. Piscataway, NJ : IEEE Service Center, 2002. – ISBN 0780372824

[18] THOMAS BÄCK ; MARTIN SCHÜTZ: Intelligent mutation rate control in canonical genetic algorithms, Springer, Berlin, Heidelberg, 158–167