Fuzz Testing with Dynamic Taint Analysis based Tools for Faster Code Coverage

Ciprian Paduraru[1,2,3], Marius-Constantin Melemciuc[2] and Bogdan Ghimis[1,2]
[1]*The Research Institute of the University of Bucharest (ICUB), Romania*
[2]*Department of Computer Science, University of Bucharest, Romania*
[3]*Electronic Arts Romania, Romania*

Keywords: Fuzzing, Tainted Analysis, Automatic, Generative Model, Test Data.

Abstract: This paper presents a novel method for creating and using generative models for testing software applications. At the core of our method, there is a tool performing binary tracing using dynamic taint analysis. Our open-source tool can learn a connection between code variables that affect the program's execution flow and their content in a set of initial training examples, producing a generative testing model which can be inferred later to produce new tests. This work attempts to maximize the code coverage metrics by focusing only on those parts of the input that affect the control flow of a program. The method can be used to automatize the test data generation on any binary x86 application. Evaluation section shows that it is producing better code coverage on applications accepting binary input formats, especially when the feedback from the test system is needed in a short time.

1 INTRODUCTION

Software security is a hot topic nowadays, especially because of the wide interconnectivity between software and hardware pieces. Before releasing software on the market, and even during development, companies are investing important resources for testing the quality of their software. We think that it is important to automatize as much as possible the generation of inputs and automatically test software pieces without human effort. One of the main purposes of an automatic test data generation system is to generate test data that covers as many lines of code of a program. A common technique is *Fuzz testing* (Godefroid, 2007), which looks for inputs causing errors such as buffer overflows, memory access violations, null pointer dereferences, etc, which in general have a high rate of being exploitable. Using this technique, testing data is generated using random inputs and the program under test is executing them for the purpose of detecting issues like the above mentioned ones. One of the main limitations of fuzz testing is that it takes a significant effort to produce inputs that covers almost all branches of a program's source code. This comes from the fact that using randomness, it results in a high chance of producing inputs that are not correct and rejected in the early outs of a program's execution.

Alternative methods that augment the classic random fuzz testing with different methods were created. Such ideas involved the use of genetic algorithms for better guiding the test data generation towards uncovered areas (Paduraru et al., 2017), or by using recurrent neural networks and predicting the probability distribution of the next character knowing a previously generated context (Godefroid et al., 2017), (Rajpal et al., 2017).

This paper discusses an open-source tool (from the authors' knowledge, the first one at the moment of writing this paper; publicly available here: https://github.com/AGAPIA/river-trace-analysis-and-fuzz) that given an x86 binary file under test, is able to learn a generative model for new tests, starting from a set of training examples (i.e., a starting set of input tests), which concentrates the fuzz process exactly on the parts used by branches inside the program. A *tracer tool* that uses tainted analysis (Stoenescu et al., 2016) is at the foundation of this work. The main idea of our method is that only specific parts of an input stream given to a program are used to evaluate the branch conditions, and our attempt is to concentrate more on learning the content of those parts in valid tests. Obtaining a model that describes where those parts are and their content

Paduraru, C., Melemciuc, M. and Ghimis, B.
Fuzz Testing with Dynamic Taint Analysis based Tools for Faster Code Coverage.
DOI: 10.5220/0007921300820093
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 82-93
ISBN: 978-989-758-379-7

format (as a general regular expression, or recursive neural networks in our current implementation), it is an important way to ensure that fuzzed inputs are not rejected from early out branches of a program and can potentially generate better code coverage.

The rest of the paper is structured as follows. Section 2 discusses related work in the field. The backend tools used for tracing and performing tainted analysis are described in Section 3. Section 4 describes our method in details, while Section 5 compares it against other new methods with the same purpose. Finally, the last section presents the conclusions and future work.

2 RELATED WORK

In the field of fuzzing techniques, there are three main categories currently: blackbox random fuzzing (Sutton et al., 2007), whitebox random fuzzing (Godefroid et al., 2012), and grammar based fuzzing (Purdom, 1972), (Sutton et al., 2007). The first two are automatic methods proving efficiency in finding vulnerabilities in binary-format file parsers. These methods are also augmented with others for better results. For example, in (Paduraru et al., 2017) authors present a distributed framework using genetic algorithms that generate new tests by looking at the probability of each branch encountered during the execution. Their fitness function scores a newly generated input test by the probability of the branches encountered in the program's execution trace. This way, the genetic algorithm tries to create input data that drives the program's execution towards rare (low probability) branches inside the program's control flow. They use Apache Spark for parallelization and dynamic tainting to know the paths taken during the execution. Their method obtains better scores than classical random fuzzers and it is one of the solutions that we compare against, using the same two examples: an HTTP parser and an XML parser.

On the other side, the grammar based fuzzing is not fully automatic: it requires a grammar specifying the input format of the application under test. Typically, this grammar is written by hand and the process becomes time consuming and error prone. It can be viewed as a model-based testing (Utting et al., 2012), and the work on it started with (Hanford, 1970), (Purdom, 1972). Having the input grammar, test generation from it can be done either (usually) random (Sirer and Bershad, 1999), (Coppit and Lian, 2005) or exhaustive (Lämmel and Schulte, 2006). Methods that combine whitebox fuzzing with grammar-based fuzzing were discussed in (Majumdar and Xu, 2007), (Godefroid et al., 2008a). Recent work concentrates also on learning grammars automatically. For instance, (Bastani et al., 2017) presents an algorithm to synthesize a context-free grammar from a given set of inputs. The method uses repetition and alternation constructs for regular expressions, then merging non-terminals for the grammar construction. This can capture hierarchical properties from the input formats but, as mentioned in (Godefroid et al., 2017) the method is not well suited for formats such as PDF objects for instance, which include a large diverse set of content types and key-value pair.

Autogram, mentioned in (Höschele and Zeller, 2016) learns context-free grammars given a set of inputs by using dynamic tainting, i.e. dynamically observing how inputs are processed inside a program. Syntactic entities in the generated grammar are constructed hierarchically by observing what parts of the given input is processed by the program. Each such input part becomes an entity in the grammar. The same idea of processing input formats from examples and producing grammars, but this time associating data structures with addresses in the application's address space is presented in (Cui et al., 2008). There are two main differences between Autogram and our method:

- Our method concentrates only on the part of the inputs that affect the branching conditions, while Autogram is considering all source code variables that have a connection to the input stream. For this reason, we expect that our solution is more suitable for larger input binary files than Autogram in practice.
- Autogram is suitable only for Java applications since it uses JVM to parse the callstack and correlate variables in the code to inputs used. In comparison, our method can be used to test any kind of applications which can be translated to assembly code.

Both approaches described above for learning grammars automatically require access to the program for adding instrumentation. Thus, their applicability and precision for complex formats under proprietary applications is unclear. Another disadvantage of these is that if the program's code changes, the input grammar must be learned again. The method presented in (Godefroid et al., 2017) uses neural-network models to learn statistical generative models for such formats. Starting from a base suite of input PDF files (not binaries) they concatenate all and use recurrent neural networks (RNN, and more specifically a sequence - to - sequence network) to learn a generative model for other PDF files. Their work is focused on generative models for non-binary objects.

Dynamic tainting has numerous applications such as finding and analyzing security threats (Newsome, 2005), (Arzt et al., 2014), software test generation using in combination with concolic execution (Bekrar et al., 2011), or in combination with fuzz testing and genetic algorithms (Avancini and Ceccato, 2010), (Mathis, 2017). The last two papers fall in the same target as ours, but their approach is to mark the parts of the input tainted by a program execution first, then applies genetic algorithms over those parts. Instead, our method is marking the parts of the input which are both tainted and used inside a branch evaluation in the program. The intuition is that ours should help in improving the code coverage since branch conditions can drive the number of paths explored during execution, which directly impacts the code coverage metric. As an example, think of an image processing application that iterates over some areas of an input image sent by the user, but doesn't actually modify the control flow of the program based on those iterations. In this case, **fuzzing over the entire tainted data might be useless in getting more code coverage. Instead, concentrating the fuzz process only over areas used for branching can get better result in shorter time.**

One of the most appreciated tools for practical fuzzing today, *AFL* (american fuzzy lop) (AFL, 2018), is based on genetic algorithms and various heuristics to find faster vulnerabilities and achieve good code coverage. We compared it against our solution to find out that is the difference between them. On short, AFL tends to be better on text-based inputs, while ours can get better results on binary like kind of inputs. An improved fuzzing tool, with reported results above AFL is *Angora* (Chen and Chen, 2018). It uses runtime taint analysis and keeps stack context on branch transitions to achieve improved code coverage and bugs finding. However, it is dependent on LLVM and needs access to the source code. Instead, our tool works at binary level (i.e. doesn't need access to the source code), and this is the reason we don't compare our methods against Angora.

3 BACKEND DESCRIPTION

This section is an overview of the technical tools that we use as backend for implementing the core ideas of this paper.

3.1 RIVER Tool

The tool we use to perform dynamic taint analysis and meaningful tracing from a program execution is named RIVER (Stoenescu et al., 2016), and it is available open source at https://github.com/bitdefender/river.

RIVER works at the binary level and doesn't need any other source code information from the user. At test time, different modules can be registered and produce inputs to test the user's application against them. The runtime reads instructions starting from the user's application entry point and performs dynamic instrumentations of the x86 instructions, such that they can be interpreted and executed similarly to something running inside a virtual machine (more technical details are described in (Stoenescu et al., 2016)). A *basic block* in our terminology represents a contiguous set of instructions that ends with a branch decision. Considering assembly code, basic blocks can be represented by the code inside an *if* statement, a loop one, or the entire code of a function if it doesn't have any branching decision.

3.2 Taint Analysis Description

The method of dynamic taint analysis works by marking all data originating from untrusted sources (e.g. user input, files opened, network traffic, etc). Then, at runtime, instructions having operands marked as tainted can propagate tainting to destination operands. RIVER's backend uses dynamic taint analysis at bit level (similar to the work described in (Yadegari and Debray, 2014)), tracking at runtime any source of tainted data by interpreting each instruction from the user's provided executable at the assembly level and continuously updating its internal data structures. When a branch instruction is executed, we add information about this operation such as the memory region used or the type of branch evaluation, the potential forward address, etc.

3.3 Tainted Tracer Tool

The component inside RIVER that is built over the taint analysis system to get detailed information from a program execution is called *tainted tracer tool*. The output of this tool is a combination of operations involving taint analysis operations, and addresses of basic blocks encountered in the program's execution. For each such block, we capture different information. The most important for our current paper are the module and address where the block resides, the jump instruction information (return, jump, conditional jump, call or syscall), and the option available for continuing the execution depending on the jump condition evaluation. Other details include the cost for executing a block and the number of instructions

used in the block.

Omitting the taint analysis operations, Figure 1 shows a snippet from an output log file explaining the data provided by the tracer for each basic block occurred during execution of a program.

The full output log also adds data flow analysis on top of the basic blocks tracing. The analysis is done by marking the initial input given to a program as tainted and propagating tainting information during a program execution depending on data flow and usage. Each taint propagation produces a new variable in the SSA format (Single Static Assignment). Variables are marked as $I[index]$, where index is in increasing order during program execution (i.e. inside a loop execution, even if an instruction is computed for the same variable name, in SSA format there will be many different increasing variables associated with it in the final output log).

The four operations that show taint propagation in the output log are defined below.

- Extract: Cutting a part from the tainted data to a new variable, by specifying the offset where to cut from and size in bits. A concrete example for this operation in a programming language would be getting a slice of an array or the last two bytes of an integer represented on four bytes.

  $$I[k] <= I[index][offset : size]$$

- Concatenation: Concatenate two tainted variables to another. Consider operations where the code puts together a new variable from two existing ones.

  $$I[k] <= I[p] \quad | \quad I[q]$$

- Generic execution: Represents a set of instructions on one or two variables that propagates tainting to the resultant. Consider math operations in programming languages as an example for this, both unary and binary transformations.

  $$I[k] <= I[p] \quad ++ \quad I[q]$$

- Constants: Every constant used in the code is considered tainted since it modifies the data flow execution.

  $$I[427] <= const0x0000000A(32)$$

A snippet of a full output log can be observed in Figure 2. Other than the explanation given below the figure, **a key observation here is that when in assembly code there is a jump instruction evaluated by a condition related to parts of the input stream, that condition evaluation depends in the end on a single tainted variable if SSA format is used.**. The variable in the cause is the last one appearing before the block description entry in the log file. In Figure 2, variable $I[431]$ is evaluated for checking the branch condition. This is a key aspect because it shows how we connect the branch decisions in a program by the input stream, i.e. through SSA variables outputted by the tainted tracer tool.

4 CREATING AND USING THE MODEL TO GENERATE NEW TEST DATA

This section presents the solution we use for obtaining a generative model by learning the patterns from the tainted tracer output logs using an existing set of training inputs. The purpose is to learn the general format of the inputs given to a program, and the input data patterns especially around the parts that are used in the source code for branching decisions. By concentrating fuzzing only on those parts we expect to hit two targets:

- Use more efficiently the computational resources for fuzz process and thus obtain faster a good coverage than other fuzzing methods.
- Obtain a good pass rate (i.e., number of inputs not rejected in the early branches of a program's execution) for inputs generated using our generative model since it learns the valid format of the parts used for branching.

With enough input training examples, we also expect that our model to generalize well and create enough diversity.

4.1 Overview

Considering that we have a software application and a database of recorded user inputs to it, our target is to find a generative model that can explain the existing sets of inputs and produce similar new inputs.
As a concrete example, consider a program doing image processing and where its control flow depends mostly on a small part of the input, the image's header. The format of most of the inputs will be $[header \mid image\ content\ in\ bytes]$, where *header* can be binary data such as 4x2 bytes describing the resolution of the image, 1 byte specifying the number of channels, and a string of

Module name	Offset	Cost	Jmp type	Jmp instr	Esp	Nr instr	Taken	Offset	Not taken	Offset
libsimple-address.so	0x000005DF	5	0	3	0xF5BBF25C	5	libsimple-address.so	0x000005DB	???	0x00000000
libsimple-address.so	0x000005DB	2	0	0	0xF5BBF260	2	???	0x00000000	???	0x00000000
libsimple-address.so	0x000005EB	6	0	3	0xF5BBF24C	6	libsimple-address.so	0x00000450	???	0x00000000
libsimple-address.so	0x00000450	1	1	1	0xF5BBF24C	1	???	0x00000000	???	0x00000000
libsimple-address.so	0x00000456	2	0	1	0xF5BBF248	2	libsimple-address.so	0x00000430	???	0x00000000
libsimple-address.so	0x00000430	2	1	1	0xF5BBF244	2	???	0x00000000	???	0x00000000
ld-2.26.so	0x000154C0	6	0	3	0xF5BBF234	6	ld-2.26.so	0x0000F130	???	0x00000000
ld-2.26.so	0x0000F130	6	0	3	0xF5BBF220	6	ld-2.26.so	0x0001B57D	???	0x00000000
ld-2.26.so	0x0001B57D	2	0	0	0xF5BBF224	2	???	0x00000000	???	0x00000000
ld-2.26.so	0x0000F13B	22	0	2	0xF5BBF1F8	22	ld-2.26.so	0x0000F2BF	ld-2.26.so	0x0000F181
ld-2.26.so	0x0000F181	3	0	2	0xF5BBF1F8	3	ld-2.26.so	0x0000F260	ld-2.26.so	0x0000F18D

Figure 1: The figure shows a part of the output returned by the tracer using raw tracing option (omitting the taint analysis output for easier presentation).The first two columns tell the module name and offset where the block resides. Jump type shows where the condition for branch evaluation resides and can be a value in the set:0 - immediate jump, 1 - memory, 2 - register. Jump instruction column shows the kind of jump instruction that ends the block, and can be a value in the set: 0 -call, 1 -return, 2 - syscall, 3 - jmp, 4 - jcc. The last four columns specify the pairs of modules and offsets of the following basic block if the branch is taken or not.

```
I[0] <= p[0]
I[1] <= I[0][0:24]
I[2] <= I[0][0:32] ++ I[1][0:24]
I[3] <= p[1]
I[4] <= I[1][0:8] | I[3][16:16]

.............................

I[430] <= I[17][24:8]
I[431] <= I[430][0:8]
libsimple.so   +00000540 (10) (0) (2) (F60AE228) (10) libsimple.so   +00000540 libsimple.so   +0000055F
```

Figure 2: Figure showing a snippet of output log using tainted tracer tool. The first part of the image shows some taint propagation instructions. Note that the initial input stream is marked with *p* in the log, and the default memory addressing is on 4 bytes, i.e. p[0] extracts the first four bytes from the initial input stream sent by the user. The other extraction and concatenation operators used in the picture are explained in Section 3.3. In the second part of the picture, there is a related taint propagation and block execution. The tainted instructions used inside a block appear before the entry of that block in the output log. For example, variables I[430] and I[431] are generated by tainting inside the block. In this case, if the branch is taken, the next block will be inside the same module at address 0x00000540, while if not taken, the new offset is 0x0000055F.

4 bytes for specifying the order of the channels. (e.g. of headers ”1024—768—4—ARGB”, ”1920—1080—3—BGR”). Giving such a set of images to our model training methods, it will learn a model where the input that does most of the control flow of the program resides in the first part of the input, and where the first two numbers have a memory footprint of 4 bytes, with values within some range, the third is a single byte number which is either 1,3 or 4, then a string of 4 bytes representing a string that satisfies a regular grammar. Then, this model can be used to produce new inputs by fuzzing only on those areas. It is up to the user to decide how to fuzz those areas of interest. Our current out-of-the-box implemented methods offer customized regular grammars and recursive neural networks to understand certain categories of inputs such as numbers / strings / date, but the user is free to inject other methods such as genetic algorithms to learn and fuzz over the targeted content (Godefroid et al., 2017), (Paduraru et al., 2017).

It is important to note that the rest of the content which is not used for any branching evaluation is not of interest for our method since it doesn't affect the control flow of the program, thus, neither affects the code coverage if we modify it. Going back to the example above, think that pixels in the image are not used for any kind of branch evaluation. Randomizing that image's pixels doesn't modify the program's control flow at all. This is the reason why in our method, we concentrate the fuzzing process only over areas detected as being used in branch evaluation, while the rest is randomized just to keep a similar structure to the original input.

The foundation of our system is based on the tainted tracer tool described in Section 3. The generative model is obtained by executing the program against each input in the training set and understanding the patterns in the resulted logs for various parts of the input stream used in branch decisions. The only type of applications that are not currently supported in our method are the ones that can imply asynchronous / non-deterministic read of the input streams. These streams cannot be currently learned as a generative model since pattern matching poses some serious challenges, but this is added in our fu-

ture work investigations.

Figure 3 shows in more details the flow of obtaining a generative model by tracing and learning the patterns inside tainted logs, from various input streams recorded. The input streams can have different formats and lengths, there is no restriction on their content.

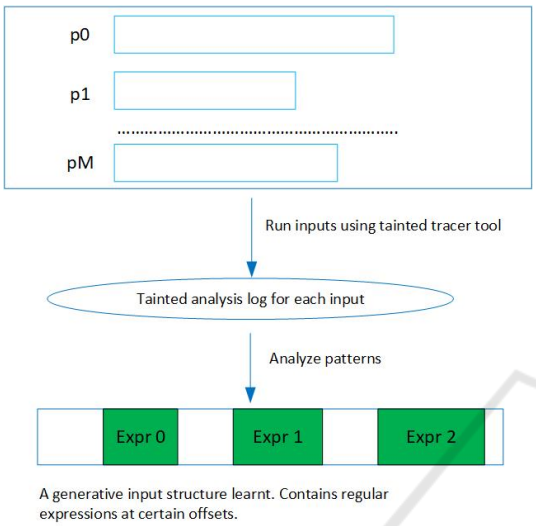

Figure 3: Overall flow for obtaining a generative model, from a set of inputs. It starts with a recorded set of inputs for the evaluated program. The tainted tracer tool is executing the program against each input in the set, obtaining an output log. Then, this log is parsed and patterns over different input sequences are analyzed to create the final model. The green areas are corresponding to parts of an input structure which are used for branch decisions. Thus, by concentrating the fuzz methods over those areas, it is the same result as doing fuzzing everywhere in the input, but with less computational resources used.

4.2 Steps to Create the Generative Model

There are four steps involved in creating the model, defined below.

Step 1 - Create a DAG Associated with the Output Logs

The first step when analyzing a tainted output log is to create the DAG (Directed Acyclic Graph) of the tainted variables. For each variable, we store the interval of bits used from the initial input stream (more specifically, a union of intervals). For example, Figure 4 sketches a set of tainted instructions along with the DAG obtained from it. The initial input stream given to the evaluated program is marked with the variable named p. The graph contains as nodes

Listing 1: Simple loop using parts of input stream: array a.

```
for (int i = 0; i < N ; i++)
{
    x = a[i];
    y = a[i + 1];
}
```

all the tainted variables, while the edges mark the data dependencies between them.

Step 2 - Cluster the Nodes Corresponding to the Same Variable in the Program's Source Code

A disadvantage when using the SSA format for instructions is that when looping over variables consuming directly or indirectly input data in the user's program assembly, a new SSA variable is created for each original variable at each loop iteration. As an example, for the source code associated with the one in Listing 1, there will be N SSA variables created for both x and y in the output log ($I[index]$, where index varies between 0 and $2*N-1$, variable x being represented by even indices, while variable y by odd indices. Each of these tainted variables consume a single entry of the input stream.). This is a disadvantage because our model has to understand that all the odd indices for example actually belong to the same cluster, i.e. pointing to the same purpose variable in the source code. Same is valid for the even indices. In the end, the purpose of the second step is to replace all the intermediary tainted variables nodes in the example, with just two tainted variable nodes which store the union of all input stream intervals used in the cluster they represent (x uses even entries, while y uses odd entries of the input stream a).

Fortunately, this can be easily obtained by parsing the output log and simulating a stack using the modules and offsets information for the lines describing a block entry. Following them, we can understand if a jump was made and where. When a group of tainted variables $I[index]$ appears at the same stack context (module, offset), it means that they are in the same cluster. At parsing time, we hold a hashing scheme (indexed by the pair module and offset) for finding if a variable is in the same cluster with another, and if it is, then its interval of used input stream is appended to the existing variable, otherwise the new tainted index variable is added to the DAG. A concrete example of this behavior based on a sample output log can be observed in Figure 5.

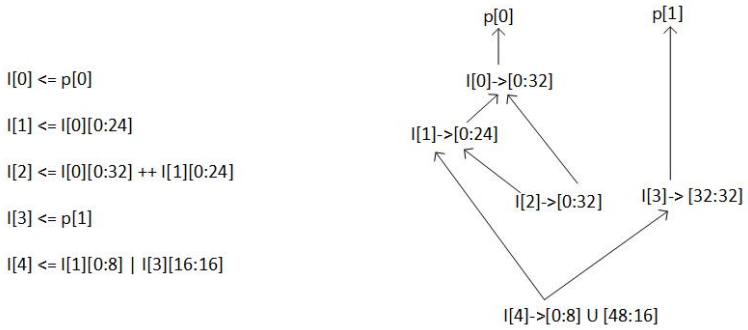

Figure 4: A set of tainted instructions and their corresponding DAG. On the right side of each node, it is shown the interval in bits from the input stream used by that node, in the same format as the extraction operation: $[startbit : size in bits]$. Tainting starts from the original stream p, and by default, if not specified, the tainting is done on 4 bytes. In this case, I[0] uses the first 4 bytes from the initial input stream. I[1] does an extraction operation and uses the first 3 bytes from the initial input. Since I[3] uses p[1], its real used interval in bits is starting at bit 32 and has length 32 bits, thus bytes 4 to 7 inclusive are used. I[4] is part of an operation involving extracted parts of I[1] (the first byte from it) and I[3] (its last two bytes). On complex applications, these intervals can grow significantly and overlap.

```
I[430] <= I[17][24:8]
I[431] <= I[430][0:8]
libsimple.so   +00000540 (10) (0) (2) (F60AE228) (10) libsimple.so   +00000540 libsimple.so   +0000055F
I[432] <= I[17][16:8]
I[433] <= I[432][0:8]
libsimple.so   +00000540 (10) (0) (2) (F60AE228) (10) libsimple.so   +00000540 libsimple.so   +0000055F
I[434] <= I[17][8:8]
I[435] <= I[434][0:8]
libsimple.so   +00000540 (10) (0) (2) (F60AE228) (10) libsimple.so   +00000540 libsimple.so   +0000055F
```

Figure 5: Figure showing a snippet of output log for a program containing a loop statement. The loop can be recognized by observing multiple basic block entries at the same address (module *libsimple.so* and offset $0x00000540$), each of these with a jump to a different block address if the branch condition is no longer evaluated as true (jumping to the same module, but offset $0x0000055F$. In the complete output log, after all iterations are finished, the next basic block entry will show the same module and address $0x0000055F$ if the loop is a simple iteration over iterative code which doesn't crash or performs any exception). The only variables retained by our DAG after step 2 are $I[430]$ and $I[431]$, since all others are in the same cluster with them. Also, notice that variable $I[430]$ holds an interval of 8 bits ($I[17][24:8]$ means that starting from bit 24 of $I[17]$ it uses the next 8 bits), and in the next instruction the log says that $I[431]$ depends on the entire interval used for $I[430]$. Therefore, after step 3, the only leaf node (and the remaining node of interest) is variable $I[431]$. To identify and connect this variable between the execution of the same program with different inputs, the DAG will contain instead of $I[431]$, a label containing the module and offset where it belongs to: ["*libsimple.so*", 00000540]. If in a different execution the same corresponding source code variable would have index $I[498]$ because the program took a different flow by using a different input, the variable/cluster is still recognizable by its module and offset origin, i.e. it will have the same label.

Step 3 - Finding the Leaf Nodes in Each Input Test Example; Connect and Cluster the Leaf Nodes between Different Examples' Execution Logs

The concept of *leaf node* in our context is not the usual one in the graph theory. In our context, a leaf node represents a node that doesn't have its entire stored interval being reused by the set of nodes depending on it. For instance, in Figure 4, $I[0]$ is not a leaf node because its input interval is reused by nodes $I[1]$ and $I[2]$, while node $I[3]$ is still a leaf node since its first two bytes are not reused by $I[4]$. The leaf nodes in our context are therefore $I[4]$, $I[3]$ and $I[2]$.

Internally, for each of the leaf nodes, we also store the basic block (the pair [module, block offset]) associated with that variable. An important observation at this point is that **for each basic block, there will be at most one leaf node**. The explanation is based on remembering the key observation discussed in Section 3, that each basic block will propagate tainting information to a single variable used for evaluating the jump condition, and considering our contextual definition for leaf nodes given above. The group of internal variables which use parts of the input stream and in the end contributes to the branching condition of a basic block (i.e. affecting the execution flow of the program), are therefore grouped in a single node variable.

An example of these two last steps can be visualized in Figure 5. There is only one leaf node in the end: $I[431]$. All other variables are either be-

ing reused by dependants or in the same cluster with $I[431]$.

Executing the same program with different inputs can cause different SSA variable indices for the same corresponding source code variables. More specifically, if before the loop in Listing 1 there would be a simple decision statement (*if* statement) based on the input, then if we compare the tainted variables in the SSA format, the variable indices created for x and y could be different between two executions of a program if one would take the branch before loop, while the other doesn't (e.g. inside the simple decision statement there could be other variables consuming tainted input data, and a new SSA variable index is therefore created for each one). We can still connect the variables between different execution paths in a cluster variable by observing that after each variable usage, the output log contains the basic blocks information where that tainting was propagated. If indices in two (or more) execution logs were created in the same basic block entry, then they actually represent the same variable. Thus, in the final DAG (and model output) we store the pair $[module, offset]$ instead of the variable index. We still denote this pair as *tainted variable* in the continuation of the paper.

Step 4 - Format and Store the Model for Inference

Depending on how an input stream is used by a program, each leaf node variable can use different areas from it, corresponding to the union of intervals stored inside the node (Figure 6). Some variables might not appear at all in a tainted log output, for example, if the program execution with an input stream doesn't use at all those specific variables. The content detected as used by each individual variable in each of the input test examples is concatenated into a contiguous array of bytes (a string), then (by default) a regular expression generator (DevonGovet,) is used to create a regular expression that is able to match all the strings from each individual input stream. The user has the option to hook a function and decide what to do with the detected pattern and try other methods too. Another method used by us, as stated in the evaluation section, is by using recursive neural networks for learning the format and the possible content used for a variable.

The output of this step is a set containing meta-data for all variables. The meta-data stored for each variable is in the form $\{[module, offset], DataPattern\}$, where $DataPattern$ is the data pattern learned for the respective variable (i.e. by default a regular expression matching all data in the input streams; user has another out of the box option to chose a RNN model instead of regular expression to learn and produce content for that specific variable; another generic option is to attach own hooks and create customized models based on the existing framework), and the pair $[module, offset]$ is the address where this variable was used as a branch condition evaluation. A mapping from all variables to their corresponding meta-data is created and denoted further by $VarToDataPattern$.

To support different inference methods, the implementation also saves a snapshot of different input streams examples resulted from the outputs logs (variable $InputsDatabase$). Concretely, for a subset of the inputs from the training examples set, we store a data structure with the following format: $\{InputIndex; InputLength; VariablesUsed; Offsets; Intervals\}$. The first two parameters represent the index and the length of the input stream in the original set. The third parameter is a set of tainted variables used, sorted by the order of their appearance in the respective stream of input, while the fourth parameter is the offset in the input stream where each variable is used (i.e. the first byte used by the variable). The last parameter represents the union of intervals used by each variable on the initial input stream.

4.3 Producing New Inputs using Inference

To generate a new input, one of the saved snapshots is sampled and the method tries to produce an input that follows the same structure, but for each variable used inside the snapshot we use the global data pattern learned from all training examples. Basically, we follow the next steps:

(1) Select one random test ($index$) from $InputsDatabase$ (we need to follow its structure closely to obtain a valid input), and create a random array (NEW_INPUT) of bytes with its original length (using $InputsDatabase[index].InputLength$).

(2) For each variable V in $InputsDatabase[index].VariablesUsed$

   (a) Generate a new string (S) using the registed model (i.e. by default regular expression): $VarToDataPattern[V]$.

   (b) Apply the generated string over NEW_INPUTS array, starting at the offset specified by the attribute

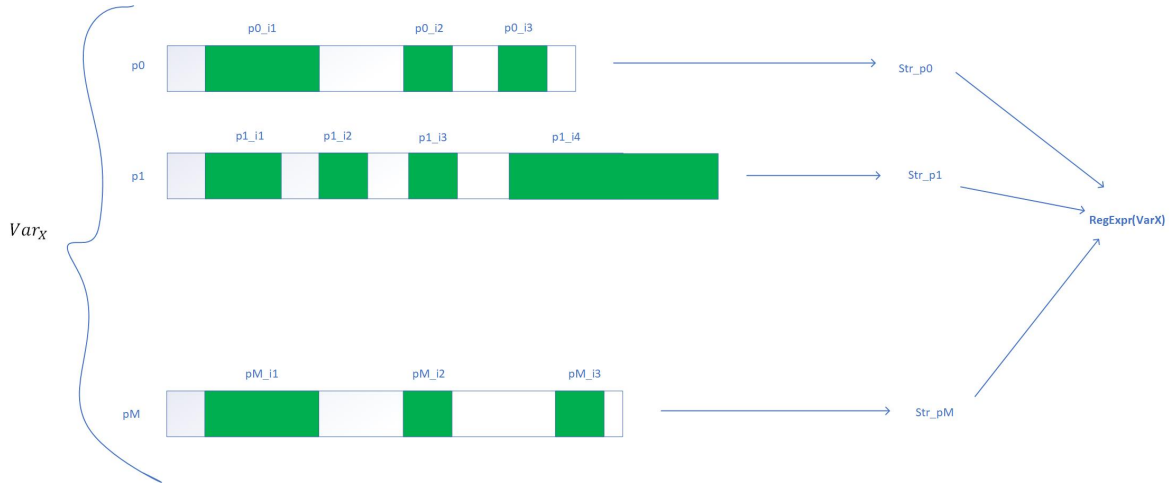

Figure 6: Considering a set of M input streams in the training examples, the green areas show the parts used by a tainted variable in each of them. The content in those areas are concatenated in a string, per each input stream, and by default are given to the regular expression matching, which in the end builds a regular expression matching all usages. If RNN method is used to learn the data pattern instead of regular expressions, the model learns for that specific variable the probability of the next character, similar to *seq2seq* models in (Godefroid et al., 2017). Note that our implementation allows the user to hook a function for learning on its own the data pattern model, and another one for performing inference to generate new data based on the learned pattern.

$InputsDatabase[index].Offsets[V]$ for the corresponding variable and try to match as much as possible the set of intervals specified in $InputsDatabase[index].Intervals[V]$.

4.4 Practical Usage of Models and Inference

In practice, inputs sets for a program can be updated online, i.e. new sets of inputs can be added for improving the existing model. When a new input is added, it must be executed through the tracer and a new tainted log is obtained. The existing model is updated by:

- Adding the transformed log data to $InputsDatabase$.
- Updating the generative model for the variables used in the new test, i.e. data structure $VarToDataPattern$, because the new test could contain new patterns for those variables.

The next time inference is used, it will take into account the new input test too. The intervals used by the variables can still overlap sometimes when applied NEW_INPUT array. Also, the offsets where generated strings are applied might be dependent on the content of the generated strings (e.g. think about branching variables representing the size of an image). Still, by using the method described above we can learn models and generate inputs which intuitively should provide better code coverage than many fuzzing techniques.

5 EVALUATION

The evaluation between different methods was done using two open source applications that accept binary files as input - *zlib* (http://www.zlib.net/) and *libjpeg* (http://www.libpng.org/pub/png/pngcode.html), and one text based input file - *libxml* (xmlsoft.org). The purpose for *zlib* evaluation is to create a model that learns to create new compressed files. Same purposes are targeted for *libjpg* - create a generator for images that will be later given to the library to read data from, and *libxml* - a generator for XML files.

The three applications will be compared against the five methods described below. The method presented in (Höschele and Zeller, 2016) couldn't be evaluated since there is no source code / executable publicly available at the moment.

(1) Method 1: Presented in (Godefroid et al., 2008b), where Recurrent Neural Networks are used to create a model able to create content similar to the input tests. Our evaluation used the same inference method which authors suggested as performing the best in their tests, *SampleFuzz*, i.e. sampling a new random number after each space character.

(2) Method 2: Presented in (Paduraru et al., 2017), where genetic algorithms are used to improve fuzzing techniques and get potential more code coverage.

(3) Method 3: Described in this paper, applying directly the algorithm in Section 4.

(4) Method 4: On top of Method 3, but using an additional fuzzing with probability 5%. This means that after generating the string with our method, 1 out of 20 bytes will be replaced by a random byte.

(5) Method 5: Using american fuzzy lop (AFL) (AFL, 2018).

The models trained by Methods 1,3,4 used an internal collection of input files consisting of 10000 XML files, 5000 jpg pictures, and 10000 small archived content files, all randomly crawled from the internet. The Method 1 model was trained for about 15h (until the model converged even on the jpg input files). At the same time, Methods 3 and 4 took around 13h to parse tainted logs and process data, on the same system (a cluster of 8 PCs, each one with 12 physical CPU cores, totaling 96 physical cores of approximately the same performance - Intel Core i7-5930K 3.50 GHz; Each of the PC had one GPU device, an Nvidia GTX 1070). One of the advantages of Methods 2 and 5 related to this aspect, is that they don't need any training time.

To evaluate the five methods we use a metric very similar to code coverage. But instead of showing the number of lines of code, we show how many branches (different basic blocks of code) of a program are evaluated using all the available tests generated. There are two main reason for this: counting only the basic blocks is less intrusive (i.e. the software performing analysis needs to count only the jumps taken in the source code), and we are more interested in seeing programs taking different flows in the execution path. This last reason is based on the observations that sometimes applications have a huge percent of source code lines on the main execution flow, while some other interesting blocks of code have just a few lines. The number of different basic blocks touched during execution was evaluated using our tracer tool, but this time with taint analysis disabled. The models for Method 3 and 4 have no difference in inference time.

It is important to note that nowadays testing methodologies are applied to check binaries after each or a bunch of consecutive source code changes submitted to a repository to verify against vulnerabilities or bugs. In this context, using fuzz testing can be a good approach in general since it can get good code coverage in a short time. Left infinitely to run, all methods are expected to converge to the same results. Considering this, we think it is important to show how our method compare to others in short intervals of time too, since users might want to get quick feedback from a fuzz test system, testing their new changes in the source code of a program. To evaluate fairly between the time needed to produce new inputs and the quality of the results, first, we let all five methods produce new tests (inferring their model) for fixed time moments, choose between 10 minutes and 24 hours. For each time range, method and test application, we let each method produce a different folder of new tests. Then, for each folder with tests, we used our tracer tool to execute the evaluated libraries and get the number of different basic blocks executed.

First, Table 1 shows the code coverage for each method and application with the setup defined above. The results confirm the initial intuition: our method is producing generative models which are better in terms of code coverage metric, for applications which receive binary files as inputs and whose branch decisions are made only through a small part out of that file. In comparison with Methods 1 and 5 our method randomizes the rest of the input content and concentrates more on modifying the parts of an input that seems to affect the branching conditions and program's execution flow, instead of learning a model for the entire set of training input files. Combined with a small fuzzing percent at inference, our method obtains a greater variety which seems to improve the results. Compared to Methods 1 and 5, the average improvements were between $13-20\%$.

However, on applications with inputs based more on text files (*libxml*), where the branching conditions can be affected by the text context, our method is still better than genetic fuzz testing, but worse than Methods 1 and 5. This was again expected since there are too many points of branch conditions in ratio with the input's length, thus using a single RNN model for the entire set of data is faster for inference. Note that for evaluating the *libxml* library we also used a RNN to learn the variables' data patterns and generate similar content. In this case, our model stores variables with sub-models learned from almost the entire input streams in the set of training examples.

Figures 7 and 8 show that our method is even more useful when users want to get quick feedback for evaluating a bunch of new source code changes. Note that after 10 minutes of testing, our methods tend to be 50% better than other methods fuzz methods. This can be a valuable optimization of resources used for testing software in industry.

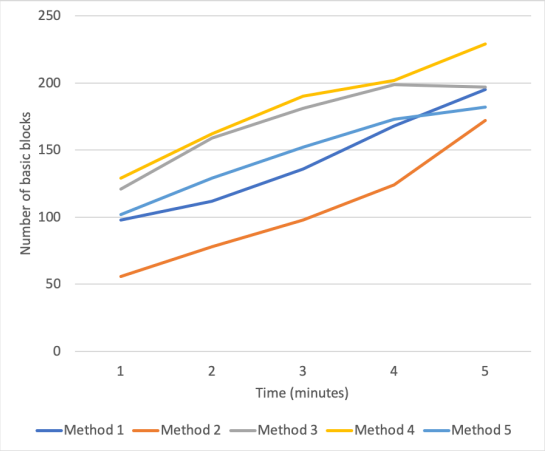

Figure 7: Shows the number of different basic blocks executed by each fuzzing method for *zlib* library. The horizontal axis shows the results for different moments of time (minutes).

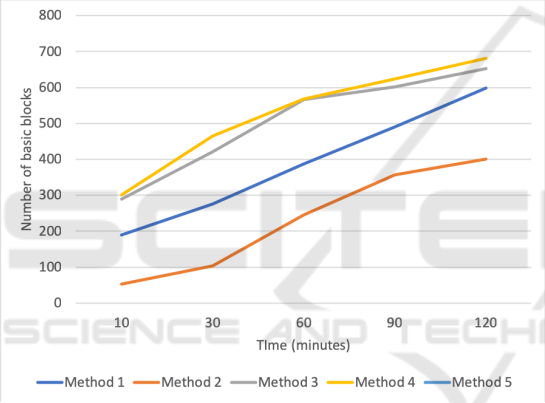

Figure 8: Shows the number of different basic blocks executed by each fuzzing method for *libpng*. The horizontal axis shows the results for different moments of time (minutes).

6 CONCLUSION AND FUTURE WORK

This paper presented a novel method based on dynamic taint analysis and a tracer tool to obtain a generative model starting from a set of training examples, which can be used to generate a new set of input tests for different evaluated programs. As the results show, the method is more efficient than existing ones for generating inputs for applications where the input stream is binary and the control flow of the program is affected only by a small part of it. However, our future plan include improvements for evaluation methods such as: getting more libraries for evaluation (similar to the corpus in AFL (AFL, 2018)), include other important metrics (detection of crashes and their type, overflows, hangs), use LAVA benchmark for automatic evaluation (Dolan-Gavitt et al., 2016). In terms of methods used, our future plans are to invest more in using machine learning to learn the relationship between the binary content of a file and the branch decisions inside a program, and, as previously mentioned in the paper, to adapt our method to asynchronous programs that use the input stream in a non-deterministic way.

Table 1: The number of branch instructions touched in comparison between the five methods and the three applications under test. Each method was let to produce inputs for 24h, then tracer was executed on each application and generated test folder to obtain the results below.

Method used	zlib	libjpg	libxml
Method 1	218	783	1286
Method 2	199	698	1129
Method 3	215	789	1150
Method 4	238	875	1227
Method 5	235	781	1240

ACKNOWLEDGMENTS

This work was supported by a grant of Romanian Ministry of Research and Innovation CCCDI-UEFISCDI. project no. 17PCCDI/2018 We would like to thank our colleagues Teodor Stoenescu and Alexandra Sandulescu from Bitdefender, and to Alin Stefanescu from University of Bucharest for fruitful discussions and collaboration.

REFERENCES

AFL (2018). American fuzzing lop (afl). In *http://lcamtuf.coredump.cx/afl/*.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. L., Octeau, D., and McDaniel, P. D. (2014). Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*.

Avancini, A. and Ceccato, M. (2010). Towards security testing with taint analysis and genetic algorithms. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 65–71, New York, NY, USA. ACM.

Bastani, O., Sharma, R., Aiken, A., and Liang, P. (2017). Synthesizing program input grammars. *SIGPLAN Not.*, 52(6):95–110.

Bekrar, S., Groz, R., Mounier, L., and Bekrar, C. (2011). Finding software vulnerabilities by smart fuzzing. In

2011 Fourth IEEE International Conference on Software Testing, Verification and Validation(ICST), volume 00, pages 427–430.

Chen, P. and Chen, H. (2018). Angora: Efficient fuzzing by principled search. *CoRR*, abs/1803.01307.

Coppit, D. and Lian, J. (2005). Yagg: An easy-to-use generator for structured test inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 356–359, New York, NY, USA. ACM.

Cui, W., Peinado, M., Chen, K., Wang, H. J., and Irun-Briz, L. (2008). Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 391–402, New York, NY, USA. ACM.

DevonGovet, h. Regular expression generator.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W. K., Ulrich, F., and Whelan, R. (2016). LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, pages 110–121. IEEE Computer Society.

Godefroid, P. (2007). Random testing for security: blackbox vs. whitebox fuzzing. In *RT '07*.

Godefroid, P., Kiezun, A., and Levin, M. Y. (2008a). Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA. ACM.

Godefroid, P., Kiezun, A., and Levin, M. Y. (2008b). Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215.

Godefroid, P., Levin, M. Y., and Molnar, D. (2012). Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: machine learning for input fuzzing. In Rosu, G., Penta, M. D., and Nguyen, T. N., editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 50–59. IEEE Computer Society.

Hanford, K. V. (1970). Automatic generation of test cases. *IBM Syst. J.*, 9(4):242–257.

Höschele, M. and Zeller, A. (2016). Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA. ACM.

Lämmel, R. and Schulte, W. (2006). Controllable combinatorial coverage in grammar-based testing. In Uyar, M. Ü., Duale, A. Y., and Fecko, M. A., editors, *Testing of Communicating Systems*, pages 19–38, Berlin, Heidelberg. Springer Berlin Heidelberg.

Majumdar, R. and Xu, R.-G. (2007). Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, New York, NY, USA. ACM.

Mathis, B. (2017). Dynamic tainting for automatic test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 436–439, New York, NY, USA. ACM.

Newsome, J. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

Paduraru, C., Melemciuc, M., and Stefanescu, A. (2017). A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In Bosman, P. A. N., editor, *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, pages 1857–1863. ACM.

Purdom, P. (1972). A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375.

Rajpal, M., Blum, W., and Singh, R. (2017). Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR*, abs/1711.04596.

Sirer, E. G. and Bershad, B. N. (1999). Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13.

Stoenescu, T., Stefanescu, A., Predut, S.-N., and Ipate, F. (2016). River: A binary analysis framework using symbolic execution and reversible x86 instructions.

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312.

Yadegari, B. and Debray, S. (2014). Bit-level taint analysis. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 255–264, Washington, DC, USA. IEEE Computer Society.