View metadata, citation and similar papers at core.ac.uk
brought to you by CORE

Dynamic Tainting for Automatic Test Case Generation

Björn Mathis
Saarland University
Saarbrücken, Germany
mathis@st.cs.uni-saarland.de

ABSTRACT

Dynamic tainting is an important part of modern software engineering research. State-of-the-art tools for debugging, bug detection and program analysis make use of this technique. Nonetheless, the research area based on dynamic tainting still has open questions, among others the automatic generation of program inputs.

My proposed work concentrates on the use of dynamic tainting for test case generation. The goal is the generation of complex and valid test inputs from scratch. Therefore, I use byte level taint information enhanced with additional static and dynamic program analysis. This information is used in an evolutionary algorithm to create new offsprings and mutations. Concretely, instead of crossing and mutating the whole input randomly, taint information can be used to define which parts of the input have to be mutated. Furthermore, the taint information may also be used to define evolutionary operators.

Eventually, the evolutionary algorithm is able to generate valid inputs for a program. Such inputs can be used together with the taint information for further program analysis, e.g. the generation of input grammars.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**; • **Applied computing** → *Document analysis*;

KEYWORDS

Dynamic tainting, input generation

ACM Reference format:
Björn Mathis. 2017. Dynamic Tainting for Automatic Test Case Generation. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DOC),* 4 pages.
https://doi.org/10.1145/3092703.3098233

1 INTRODUCTION

Dynamic tainting is a dynamic program analysis method which is used to track data in an executing program. Unique identifiers, called *taints*, are attached to values based on predefined rules or manually. The propagation of those taints through the program can then be used to compute data flows.

Tainting itself has numerous applications, among others: debugging [4], finding and analyzing security threats [11], and generating grammars [8]. Especially for the dynamic analysis of programs, inputs are needed.

Böhme et al. [2] have shown that random input generation is in theory in most cases more efficient than sophisticated input generation methods. Since many programs have highly structured inputs, up to the point where they are governed by *grammars*, random test generators likely produce inputs that are rejected in the parsing phase. Therefore, random input generation is not sufficient to test the actual program functionality. Thus, programmers still need to create inputs manually or define a language model that can be used by a fuzzer [6, 7]. Both are time and cost consuming.

Together with symbolic execution, tainting has become an important part in the software engineering research [12]. In contrast to dynamic tainting, symbolic execution techniques create constraints for the input over execution paths without actually executing a program. Such constraints can then be solved to create an input that executes the path. State-of-the-art tools like KLEE [3] try to generate test inputs for arbitrary programs by using symbolic execution, which, as Godefroid et al. describe, is often not sufficient for highly structured inputs (e.g. compiler and interpreter) [6].

I propose to use tainting information to automatically generate complex and valid inputs for programs with highly structured inputs (Figure 1). To instantiate the approach, I create a tool called LTAINT, which uses LLVM bitcode to analyze an application and is therefore capable of analyzing languages (like C) that are compilable to the LLVM intermediate representation which can be compiled to machine code. LLVM is an infrastructure which enables, among

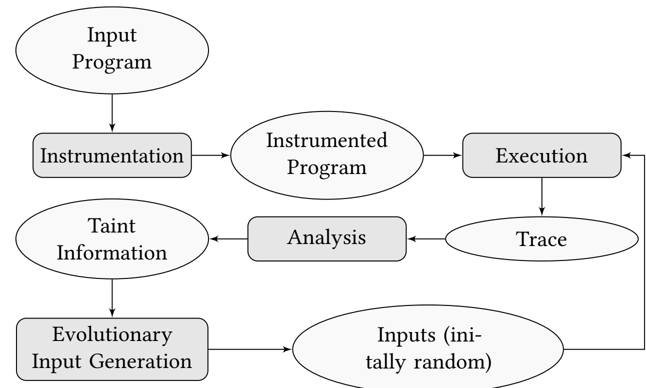

Figure 1: Dynamic Tainting and Input Generation Workflow

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA'17-DOC, July 2017, Santa Barbara, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3098233

others, code analysis, optimization and instrumentation on an intermediate representation. The taint information can be used to guide an evolutionary algorithm for automatic test input generation. For example, the taint information for a branch instruction shows which parts of the program input influence the instruction and therefore needs to be mutated to alter which branch is being taken. Furthermore, taint and execution information of my approach can be used to calculate the fitness function, e.g. by taking code and input coverage[1] into account.

Dynamic tainting can be used together with the generated inputs to perform further program analysis, e.g. the generation of context free grammars. Höschele et al. are currently building a grammar generation tool, AutoGram [8], which automatically infers a context free grammar based on the program structure with the help of dynamic tainting. Their approach uses a program and a set of valid sample inputs. The executing program gets analyzed by a dynamic tainter for Java, called JFlow, which creates a list of taint information events. This list contains *a)* Method entries and exits, *b)* Array accesses and *c)* Field accesses with the taints of their operands. AutoGram analyzes those events to infer a context free grammar. Such a grammar can then be used by a grammar-based fuzzer to fast generate valid and complex inputs.

A large amount of software is written in LLVM bitcode compilable code. Nonetheless, compared to Java, the bitcode delivers less information, important details, like the size of an array may not always be present. This information is needed to report taints of all elements in an array if it is referenced by a pointer. Additional computations need to be done to cope with the drawbacks induced by the lower level of LLVM. I propose to use dynamic tainting together with static and dynamic program analysis to create *high level* taint information, e.g. the taints of all elements of an array. Furthermore, implicit data flows are a problem while analyzing programs with highly structured inputs. For example, a string to token conversion often happens implicitly by comparing a string and assigning a variable with a token based on the outcome of the comparison. This leads to *under-tainting* (the token would not get the taints of the string) and needs to be detected. I plan to automatically detect such parts of the program, e.g. with pattern matching. This additionally increases the precision of the tainting approach.

Eventually, the *high level* taints and implicit data flows can be used together with automatically generated inputs to enhance or even enable further program analysis, like grammar learning or to enforce confidentiality and integrity. In the work of Tsankov et al. [13], high level taints would be needed to report the taints of a char array at a sink to extract if information from a non-confidential source is leaked or to enforce integrity by reporting and anonymizing data that has to be sanitized.

My hypothesis is that dynamic tainting can be used to answer the following research questions positively:

- RQ1 Is it possible to extract enough information from a program in LLVM bitcode format to generate *high level* taint information events?
- RQ2 Is it possible to detect relevant implicit data flows that would result in *under-tainting*?
- RQ3 Can dynamic tainting be used together with an evolutionary algorithm to automatically generate valid and complex inputs for programs with highly structured inputs?

2 BACKGROUND

Many tainting approaches exist that focus on specific tasks [12]. Nonetheless, to the best of my knowledge there exists no tainting approach on the abstraction level of LLVM , which is able to report *high level* taints. I propose to take program analysis into account in order to report, among others, taints of full arrays, structure names associated with taints or implicit data flows which would cause *under-tainting*. Problems like implicit data flows are already investigated by some approaches [9], but not in the context of automatic input generation and not together with *high level* taints.

To the best of my knowledge, dynamic tainting is not used for automatic input generation. Nonetheless, other approaches try to generate inputs from scratch, like symbolic execution approaches. They collect constraints over program paths with respect to the input variables of the program. Those constraints are solved in order to get input values that trigger those paths. One state-of-the-art tool which uses symbolic execution is Klee [3]. Klee shows, that symbolic execution can perform well on programs with small input structure. Nonetheless, complex programs, which use highly structured input, usually have complex path constraints. Such constraints are often hard to solve, in general constraint solving is NP-complete. Therefore, input generation with symbolic execution does not perform well on complex programs with many paths and a highly structured input [6].

Evolutionary algorithms are also used for input generation. They iteratively refine the set of inputs by using metrics obtained from program analysis (like branch distance). For example, American Fuzzy Lop [1], an "instrumentation guided genetic fuzzer" [1] analyzes the program under test while it is executing and adapts the already used inputs by mutating them. If such a mutant discovers a new program path, it is added to the set of inputs from which new mutations can be generated.

First tests with afl on cJson [5] have shown, that afl needs more information than just one json object as test input to generate meaningful and more sophisticated inputs. This problem is already known[2]. Concretely, the generated inputs from afl still only contained one json object with only small alternations, even after days of running. I believe that dynamic tainting can be used to enhance the selection of new inputs and the mutation of existing ones such that an evolutionary approach is able to successfully generate inputs for programs with highly structured input.

3 PROPOSED APPROACH

Figure 1 shows an overview of my proposed framework. It consists of four main parts that are discussed in more detail in this section: *a)* Instrumentation, *b)* Execution, *c)* Analysis and *d)* Input generation. After instrumenting the program, it can be executed with inputs from the evolutionary input generation (which are initially random). The execution is analyzed with dynamic tainting and

[1]The amount and usage of input that was actually read by the program in comparison to the input that was given.

[2]http://lcamtuf.coredump.cx/afl/README.txt,
http://lcamtuf.blogspot.de/2015/04/finding-bugs-in-sqlite-easy-way.html

the taint information is used in the evolutionary input generation phase to generate new inputs that might trigger new paths.

3.1 Instrumentation

Each LLVM instruction gets instrumented such that dynamic information for the analysis can be written out in the execution phase as a trace. The information for each instruction consists of: *a)* Name of the surrounding method *b)* Opcode *c)* Names of all operands *d)* Types of all operands *e)* Values of all operands (excluding the assigned variable) and is needed to perform the taint propagation.

Furthermore, information about global variables, functions and structures are collected statically in the instrumentation phase. The information contains names, types and for structures also the size in bytes. Additionally, information for the detection of implicit data flows can be collected. This information is later used in the analysis phase to achieve a more precise taint propagation.

3.2 Execution

In the execution phase sample inputs are used to run the program under test and to generate the trace. Those inputs can be provided by the evolutionary input generation. In this case, the initial inputs are random and are iteratively refined by the evolutionary input generation with the help of the computed taint information (Figure 1). The generated trace gets consumed by the analysis to perform the taint propagation.

3.3 Analysis

The analysis takes as input the generated execution trace and performs the actual taint propagation based on the propagation semantics I define for LLVM bitcode, which follow the actual LLVM instruction semantics. For example, for an addition, the sum of both operands is assigned to a variable, thus, the taints of both operands are assigned to the variable. Furthermore, my approach performs additional computations to generate *high level* taint information, e.g.: *a)* full bounds check for arrays to report taints for pointer and *b)* mapping of addresses to structure elements. The output of the analysis are the taints of the operands for each executed instruction.

To achieve *a)*, the proposed approach has to collect information about memory allocations. For each pointer the size of the allocated memory has to be stored as well as the address it points to. If the memory is later accessed, the size of the allocated memory is known and the taints of the elements contained in the array are reported.

For *b)* a similar approach is used. If a structure is created, the information where the structure is stored to gets connected with the static information about the structure. At a structure access this information can be used, together with the dynamic information regarding the index of the element that is accessed, to report the name of the used structure element for instance.

Additionally, I propose to implement a control flow analysis to detect implicit data flows which would lead to *under-tainting*. Figure 2 gives an example. The first program argument is compared to a static string and a token is assigned to a variable if the strings are equal. Therefore, the assigned variable containing the token should have the taints of the input string. Otherwise, the information that the token assignment is based on the value of the first argument is lost.

```
//input: token1
enum tokens {TOKEN1, TOKEN2, TOKEN3}
int main(int argc, char *argv[]) {
  char* a = argv[1];
  enum tokens t;
  if (strcmp(a,"token1")) {
    t = TOKEN1;
    //t should have taints of the input now
  }
  return 0;
}
```

Figure 2: The variable containing the token should still have the taints of the originating string after conversion.

The code in Figure 2 could be part of a lexer, where strings are often converted to tokens that are then used in the parsing part of the application. To analyze the parsing precisely, the taints must "survive" the implicit data flow. In my thesis I investigate how such flows can be detected with focus on programs with highly structured inputs. A good starting point is to apply pattern matching on such parts of the code. In this case the pattern is simple: the name of the token fits with the name of the string it is derived from.

3.4 Automatic Input Generation

A central contribution of my research is the automatic creation of inputs. I propose to use the taint information of LTaint to guide the input generator. As a starting point I plan to use an evolutionary algorithm, which in general consists of two components: *a)* a generator for new inputs and *b)* a fitness function.

It is possible to use dynamic tainting for creating evolutionary operators on the fly as opposed to static operators usually applied by such algorithms. At the beginning, default mutation operators are used, such as an adaption of *bit-flipping* [10] on strings (i.e. character vectors). This approach randomly picks parts of an individual input and changes the values at the respective positions arbitrarily.

With dynamic tainting this process can be more precisely controlled. While executing the program, additional mutation operators are generated and added to the list of available mutations. For example, a JSON parser may require a file to start with a '{'. LTaint would report, that a branch instruction checks whether the first character is a '{', so one possible mutation is setting the first character of an input to '{'. Even for less specific cases, LTaint is able to report which parts of the input influence a specific branch instruction by reporting the taints of the operands of the instruction. To alter which branch is taken the mutation algorithm can use the mutation operators only on this part of the input instead of mutating randomly. To the best of my knowledge, the on-the-fly generation of mutation operators based on program executions is not done by any existing approach.

The fitness function is able to use tainting and execution information produced by dynamic tainting to evaluate the quality of any given input. Different heuristics can be used to achieve this.

Possible heuristics are the number of executed (branch) instructions or the coverage on the input, i.e. the portion of input that was actually read by the program. I expect that a lexer or parser stops shortly after reading an unexpected character or token. Therefore, the executed (branch) instructions and the coverage on the input may correlate with the correctness of the input.

4 PLANNED EVALUATION

To evaluate the feasibility, effectiveness and efficiency of the approach I plan to perform experiments to answer my research questions.

RQs 1 and 2 concentrate on the effectiveness of the tainting. Thus, the evaluation is done on handcrafted and real world subjects. For both RQs I need to evaluate the number of true positives/negatives and false positives/negatives as well as metrics like precision and recall based on the ground truth of a micro-benchmark suite.

To the best of my knowledge no micro-benchmark suite[3] exists to test the effectiveness of dynamic tainting tools. Therefore I create a handcrafted benchmark suite which shows that LTaint is able to handle specific explicit and implicit tainting challenges. Nonetheless, this benchmark is artificial and can therefore only partially show the effectiveness of my approach.

For RQ 1 I need test subjects that reveal the missing *high level* information of LLVM. For example, my approach is planned to report the taints of all elements of a char array (i.e. string) for an operand which is a pointer to a char array. Since array bounds are not known in LLVM, the sizes of arrays have to be tracked.

For RQ 2 I need to create test subjects like the one in Figure 2. They are planned to contain typical implicit data flow patterns that are used in real world lexers and parsers. Also, I plan to implement test subjects which contain *"noise"* in the implementation to show that the implicit flow detection is sufficiently conservative. Such *noise* are implicit data flows which lead to *over-tainting*, e.g. a loop which reads a file line by line. A naive approach, which taints all control flows, would taint all values in the loop with the taints of the read line, which may lead to full tainting of all data and a decrease in the tainting precision.

Additionally, I evaluate my approach on real world applications with highly structured inputs to show that it is able to analyze large applications. RQs 1 and 2 are also evaluated on those applications. Since no ground truth exists for real world applications, I have to compare the tainting with and without *high level* taints and implicit data flows to confirm that the additional information is used and reported by the tainting framework. Furthermore, I might be able to manually verify reported taints to show the effectiveness on real world applications.

RQ 3 concentrates on the creation of sample inputs. I plan to use the same real world applications as for RQs 1 and 2 and evaluate the efficiency and effectiveness of the input generation. The reference tools are Klee and afl, the state-of-the-art tools for automatic input generation. Also, Klee uses symbolic execution and afl genetic algorithms, the main approaches for input generation from scratch.

All approaches are evaluated in terms of code coverage (e.g. line and branch coverage), which indicates the efficiency of the methods.

[3] A benchmark suite with small applications. Each application concentrates on a specific tainting challenge.

5 EXPECTED CONTRIBUTION

The central contribution of my proposed work is the automatic generation of test inputs for complex software with highly structured input by using dynamic taint information in an evolutionary algorithm. The generated inputs can be used in software testing and software analysis. Also, the proposed tainting approach is planned to deliver more information than the state-of-the-art for software which is solely available on the abstraction level of LLVM bitcode to enable advanced code analysis, e.g. the generation of grammars.

The automatic creation of test cases which deeply test programs is still an open question. For highly structured input and complex programs the state-of-the-art is not able to provide sufficient test cases. Thus, I plan to use taint information in an evolutionary algorithm to direct the creation of new offsprings and mutations.

Furthermore, the additional *high level* taint information of my approach can be used by tools which rely on dynamic tainting to enhance their analysis or even enable it on the level of LLVM bitcode. One use case is the generation of grammars, e.g. with AutoGram. Among others, AutoGram needs *high level* taint information, like the taints of all array elements. In contrast to others, my approach is able to report such taint information, e.g. by tracking array bounds to report full array taints based on a single pointer if needed.

REFERENCES

[1] afl. 2017. american fuzzy lop. (2017). http://lcamtuf.coredump.cx/afl/

[2] M. Böhme and S. Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering* 42, 4 (April 2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[4] James Clause and Alessandro Orso. 2009. Penumbra: Automatically Identifying Failure-relevant Inputs Using Dynamic Tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1572272.1572301

[5] Dave Gamble. 2017. cJson - Ultralightweight JSON parser in ANSI C. (2017). https://github.com/DaveGamble/cJSON

[6] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, Vol. 43. ACM, 206–215.

[7] Nikolas Havrikov, Matthias Höschele, Juan Pablo Galeotti, and Andreas Zeller. 2014. XMLMate: Evolutionary XML Test Generation. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 719–722. https://doi.org/10.1145/2635868.2661666

[8] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, 720–725. https://doi.org/10.1145/2970276.2970321

[9] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation.. In *NDSS*.

[10] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[11] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*.

[12] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 317–331. https://doi.org/10.1109/SP.2010.26

[13] Petar Tsankov, Marco Pistoia, Omer Tripp, Martin Vechev, and Pietro Ferrara. 2016. FASE: Functionality-aware Security Enforcement. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, New York, NY, USA, 471–483. https://doi.org/10.1145/2991079.2991116