A Negative Input Space Complexity Metric as Selection Criterion for Fuzz Testing

Martin A. Schneider(✉), Marc-Florian Wendland, and Andreas Hoffmann

Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
martin.schneider@fokus.fraunhofer.de

Abstract. Fuzz testing is an established technique in order to find zero-day-vulnerabilities by stimulating a system under test with invalid or unexpected input data. However, fuzzing techniques still generate far more test cases than can be executed. Therefore, different kinds of risk-based testing approaches are used for test case identification, selection and prioritization. In contrast to many approaches that require manual risk analysis, such as fault tree analysis, failure mode and effect analysis, and the CORAS method, we propose an automated approach that takes advantage of an already shown correlation between interface complexity and error proneness. Since fuzzing is a negative testing approach, we propose a complexity metric for the negative input space that measures the boundaries of the negative input space of primitive types and complex data types. Based on this metric, the assumed most error prone interfaces are selected and used as a starting point for fuzz test case generation. This paper presents work in progress.

Keywords: Security testing · Risk-based testing · Fuzz testing · Security metrics

1 Introduction

Today's system are getting more and more complex, becoming systems of systems, such as Cyber-Physical Systems and Internet of Things. This has several implications, in particular with respect to the security point of view. Security relevant vulnerabilities are found and exploited nearly everywhere and have an impact of 3 trillion dollars on the economy [2]. As systems grow and getting more complex, the risk for security-relevant faults is also increasing. This is true for several reasons: complex systems such as Cyber-Physical Systems are a heterogeneous network of sensors, actuators and components that process sensor data and control the actuators. Different transport mechanisms and processing algorithms, e.g. HTTP and SQL, may also lead to vulnerabilities. Complex interfaces, data types and dependencies between different fields of complex data types as well as between different parts of an interface are exacerbating such problems.

Fuzz testing is a technique that tests for faulty input validation mechanism of a system under test (SUT) and their effects if invalid input data is not recognized

K. El-Fakih et al. (Eds.): ICTSS 2015, LNCS 9447, pp. 257–262, 2015.
DOI: 10.1007/978-3-319-25945-1_17

and rejected but processed by the SUT. Input validation mechanisms determine whether a given value is valid or not in terms of a specification. They specify the boundaries of the input space for valid data. The complement of the input space of valid data is the negative input space comprising all invalid values. We suppose that the more boundaries have to be checked for given data, the higher is the risk of a faulty implementation of the corresponding validation mechanisms.

The rest of this paper is organized as follows: Sect. 2 presents related work with respect to metrics for vulnerabilities that supports our hypothesis. Section 3 describes the different elements our metric is composed of. Since we propose to use this metric as a selection criterion for fuzz testing, we describe in Sect. 4 how this could be achieved. Section 5 presents first results from the MIDAS project and Sect. 6 concludes with an outlook.

2 Related Work

There are several investigations on complexity of code and its error proneness. This includes vulnerabilities as well. Corresponding metrics can be distinguished whether they are based on source code, code changes or interfaces.

Shin et al. [7] employed a combination of different code complexity, code churn, and developer metrics in order to predict vulnerabilities and vulnerable source code files, built prediction models using logistic regression and showed their prediction capabilities on the Mozilla Firefox web browser and the Linux kernel. Their goal was to reduce the effort for code inspection and testing.

Chowdhury and Zulkernine [6] presented a framework for vulnerability prediction based on complexity, coupling, and cohesion metrics applied to source code. They built vulnerability predictors employing C4.5, random forest, logistic regression, and naive-Bayes classifier. A vulnerability prediction accuracy of 74 % has been achieved by this approach.

Cataldo et al. [5] showed that there is not only a correlation between source code based metrics and vulnerabilities but also between interface complexity and error proneness. Since he considered only errors that occurred for systems in the field, this correlation is not only of statistical but also of practical significance. Cataldo employed the metrics interface size and operation argument complexity that were used by Bandi et al. [3]. Operation argument complexity was dependent of the type of the operation's arguments. A constant value is assigned to each type, e.g. 0 is assigned to Boolean, 2 to Real, and 6 to Record, Struct, and Objects. The operation argument complexity is determined by the sum of the complexity of each argument's type [3]. The interface size is defined the product of the number of parameters and the sum of their sizes (operation argument complexity).

3 Negative Input Space Complexity

The works of Cataldo [5] and Bandi [3] form the basis for a negative input space complexity suitable for security testing. In contrast to Shin [7] and Chowdhury [6],

the metrics used by Cataldo do not require access to source code but only to the interfaces while preserving a correlation to the error proneness. Thus, this correlation is appropriate for black box approaches. Therefore, we call such metrics black box metrics.

We aim at using black box metrics in order to assess the risk for a vulnerability within an implementation of an interface. Since there is a correlation between interface complexity, operation argument complexity and error proneness, we suppose this correlation holds true for security-relevant errors as well.

We would like to exploit the supposed correlation for prioritization of security test cases generated by using data fuzzing techniques and to select types as a starting point for fuzzing. Data fuzzing techniques inject invalid or unexpected input data to a system under test in order to reveal security-relevant implementation flaws based on missing or faulty input validation mechanisms [4]. Semi-valid input data is generated to test each input validation mechanism separately in order to ensure that each constraint that must be hold by valid input data is correctly implemented. Our presumption is: The more constraints apply for an input date of a certain type, the higher is the chance that one of these validation mechanism is faulty.

The negative input space of a certain type is determined by the boundaries of the positive input space comprising all valid values. The boundaries between the positive and negative input space is specified by the constraint a valid input data has to respect. Therefore, the negative input space metric is expressed with respect to these constraints.

Hypothesis: A high negative input space complexity is an indicator for a higher risk of a faulty implementation of an input validation mechanism.

However, given the fact that there is a faulty implementation of an input validation mechanism, there may be two cases. On one hand, the validation mechanism is too loose, i.e. an actually invalid input date is considered to be valid. Such a situation may pose a security-relevant fault, i.e. a vulnerability. On the other hand, the validation mechanism may be too strict and reject actual valid values assuming that they are invalid. This may constitute a functional error. Whilst the focus of this work is on the first case, it may also have an impact on assessing the functionality of a system considering the second case.

Since our metric is based on the constraints for valid input data, we have to carefully investigate the different kinds and the structure of them and how they may be assessed by the metrics.

Basically, we can distinguish two kinds of separations between valid and invalid input data: static and dynamic boundaries. A static boundary is defined by an expression that does not contain any variable despite that one whose value shall be decided whether it is valid or not. Considering $x > 5$ as a constraint for valid values. x is the variable whose valid range is defined to be all values greater than 5. Obviously, such a constraint is quite easily implemented and there is only very little chance for a faulty implementation of a corresponding input validation mechanism. However, considering as input type a string, there

may be a lot of such constraints that has to be tested in order to decide whether a given value is valid leading to a higher risk of a faulty validation mechanism.

In contrast, a dynamic boundary depends on other variables, e.g. parts of a given input data, in order to determine if a provided data is valid. Considering a calendar date, the lower boundary of the day value is static, i.e. day > 0, while its upper boundary is dynamic because it depends on the month:

$(day < 29 \wedge month = 2) \vee$
$(day < 31 \wedge (month = 4 \vee month = 6 \vee month = 9 \vee month = 11)) \vee$
$(day < 32 \wedge (month = 1 \vee month = 3 \vee month = 5 \vee month = 7 \vee$
$month = 8 \vee month = 10 \vee month = 12))$

This boundary is dynamic, i.e. it is a different value depending on the value of another variable, the month. The implementation of an appropriate input validation mechanism is more error prone than one of a static boundary because another variable has to be evaluated. Obviously, the above-noted expression is not correct because leap years are not yet considered. Considering leap years increases the complexity of the expression because additional logical clauses have to be added that take the year variable into account. On the whole, the validity of the day number depends on two other variables (month and year) and on a complex expression that specifies four boundaries (28, 29, 30, and 31 may be the upper boundary of a valid day number).

Until now, the metric shall take into account the following aspects: whether a boundary is static or dynamic, the number of different boundaries, and on how many variables a dynamic boundary depends on.

The complexity of the expression for a boundary does also have an impact on the complexity and may increase the error proneness of the implementation. The complexity of the boundary expression can be measured using the corresponding abstract syntax tree that depends on its height. This is the fourth aspect that shall influence the metrics. A first approximation of the metric is defined as follows:

$$|b_{stat}| + \sum_{i=1}^{|b_{dyn}|} |vars_i| \cdot height_{AST} \tag{1}$$

where $|b_{stat}|$ denotes the number of static boundaries, i.e. boundaries that do not depend on any other variable as for instance day > 0,

$|b_{dyn}|$ denotes the number of dynamic boundaries, i.e. boundaries that do depend on other variables, e.g. 28, 29, 30, and 31 for a day,

$|vars_i|$ denotes the number of variables on which a dynamic boundary depends, e.g. 2 for the day its dynamic boundaries that depends on month and year,

$height_{AST}$ denotes the height of the abstract syntax tree of the expression of a dynamic boundary, e.g. 1 for the expression $(day < 29 \wedge month = 2)$, and 2 for the expression $(day < 31 \wedge (month = 4 \vee month = 6 \vee month = 9 \vee month = 11))$.

The correct implementation of an input validation mechanism for a static boundary is rather easy to implement, we set it to one, resulting in a complexity that depends on the number of static boundaries $|b_{stat}|$. For each dynamic

boundary, we determine its complexity by the number of variables it depends on ($|vars_i|$) and the complexity of its expression in terms of the height of the expression's abstract syntax tree ($height_{AST}$). We use the sum of the complexity of all boundaries as metric of the negative input space complexity.

4 Using the Metric as Selection Criterion for Fuzz Testing

The metric described above presents an approach to assess the proneness of the implementation of an interface for being faulty in terms of input validation and thus, for potential vulnerabilities. The easiest way to use this metric is to perform a prioritization of testing efforts based on decreasing metric score. Of much more interest would be to find a threshold of the complexity metric beyond the most implementations, e.g. 80 percent, are faulty. Of course, this threshold may depend on other factors such as the programming language. Another aspect would be whether the interface size by Bandi et al. [3] is still of statistical significance when being based on the presented metric.

5 Examples from the MIDAS Project

Within the MIDAS European research project [1], we are currently building a test platform on the cloud for testing of service-oriented architectures. As part of this project, we are implementing different fuzzing techniques in addition to the metric presented above. Our joint input model is a Domain Specific Language (DSL) based on Unified Modeling Language (UML) and UML Testing Profile (UTP) which already provides mechanics for defining fuzzing operators for test cases.

Within the project, we are working together with industrial partners. One is from the logistics domains. Its web services are implementing the Global Standards One (GS1) Logistics Interoperability Model (LIM)[1]. It provides a large number of different types specified using XML schema.

We parsed the type specifications and their constraints and calculated a few complexity scores depicted in Table 1. The first type String80Type is a simple string with a length restriction of at least one character and at most 80. The calculated score is determined by the number of boundaries, in this example 2, one for the lower bound and one for the upper bound. The remaining two types are of interest because the type validity is specified by a regular expression. In order to determine their complexity score, we resolve predefined character classes and evaluate the different number of character ranges as well as quantifiers. The difference between the two types GIAIType and GRAIType results from the predefined character range \d that comprises many more than the Arabic digits and thus, are constituting an interesting starting point for fuzz testing.

[1] http://www.gs1.org/lim.

Table 1. Complexity Score Examples from GS1 LIM

Type name	Expression	Complexity score
String80Type	$length \geq 1 \wedge length \leq 80$	2
GIAIType	[-!"&'()*+,./0–9:;<=>?A–Z_a–z]4,30	27
GRAIType	\d{14}[-!"&'()*+,./0-9:;<=>?A–Z_a–z]4,30	102

6 Outlook

Within the MIDAS project, we will validate the metrics using our pilots from the Logistics domain and the Healthcare domain, using HL7-based web services and thus, a complex type system as well. We will adjust the metric based on our experiences and are considering also different aspects, such as the different number of constraint types, e.g. regular expressions and simple constraints, e.g. the length constraints. Investigating a threshold beyond that components, interfaces, and types shall be selected for fuzzing testing is a second task of importance. A comparison with other metric-based vulnerability prediction frameworks described in Sect. 2 can be achieved by applying the metric to the Mozilla Firefox web browser.

Acknowledgments. This work was partially funded by the EU FP7 projects MIDAS (no. 318786) and RASEN (no. 316853).

References

1. EC FP7 MIDAS Project, FP7-316853 (2012–2015). www.midas-project.eu
2. Risk and responsibility in a hyperconnected world. Technical report, World Economic Forum/McKinsey (2014)
3. Bandi, R., Vaishnavi, V., Turk, D.: Predicting maintenance performance using object-oriented design complexity metrics. IEEE Trans. Softw. Eng. **29**(1), 77–87 (2003)
4. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: Finding software vulnerabilities by smart fuzzing. In: IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST) 2011, pp. 427–430 (March 2011)
5. Cataldo, M., Souza, C.R.B.D., Bentolila, D.L., Mir, T.C., Nambiar, S.: The impact of interface complexity on failures: an empirical analysis and implications for tool design (2010)
6. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit.- Embed. Syst. Des. **57**(3), 294–313 (2011). http://dx.doi.org/10.1016/j.sysarc.2010.06.003
7. Shin, Y., Meneely, A., Williams, L., Osborne, J.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. **37**(6), 772–787 (2011)