Data-Flow Analysis Heuristic for Vulnerability Detection on Configurable Systems

Gleyberson Andrade[1], Elder Cirilo[1], Vinicius Durelli[1], Bruno Cafeo[2], Eiji Adachi[2]

[1] Federal University of São João del-rei

[2]Federal University of Mato Grosso do Sul

[3]Federal University of Rio Grande do Norte

Abstract. *Configurable software systems offer a variety of benefits such as supporting easy configuration of custom behaviours for distinctive needs. However, it is known that the presence of configuration options in source code complicates maintenance tasks and requires additional effort from developers when adding or editing code statements. They need to consider multiple configurations when executing tests or performing static analysis to detect vulnerabilities. Therefore, vulnerabilities have been widely reported in configurable software systems. Unfortunately, the effectiveness of vulnerability detection depends on how the multiple configurations (i.e., samples sets) are selected. In this paper, we tackle the challenge of generating more adequate system configuration samples by taking into account the intrinsic characteristics of security vulnerabilities. We propose a new sampling heuristic based on data-flow analysis for recommending the subset of configurations that should be analyzed individually. Our results show that we can can achieve high vulnerability-detection effectiveness with a small sample size.*

1. Introduction

Configurable software systems offer a variety of benefits such as supporting easy configuration of custom behaviours for distinctive needs. On the other hand, it is known that the complexity induced by configuration options in the source code complicates maintenance tasks and requires additional effort from developers when adding or editing code statements [Brabrand et al. 2013]. The complexity induced by configuration options also causes developers to make mistakes that lead to more vulnerable code. In fact, security vulnerabilities have been widely reported in configurable software systems [Ferreira et al. 2016].

Studies have already proposed techniques for detecting security vulnerabilities in the source code of software systems [Sampaio and Garcia 2016], but they typically work only over a single system configuration at a time. In the context of configurable systems, the configuration space can grow exponentially with the number of configuration options, so analyzing every individual configuration becomes infeasible [Liebig et al. 2012]. To cope with this situation, developers might employ sampling heuristics to analyze only a representative subset of system configurations [Medeiros et al. 2016]. That is, instead of analysing all possible configurations, developers sample a subset of configurations to analyze each one individually. Unfortunately, the effectiveness of vulnerability detection depends on how samples are selected. And many sampling heuristics are only easy to

apply when making strong simplifying assumptions, such as that the configurable system does not contain constraints or that each file can be analyzed separately. These assumptions may not be realistic nor practical for security vulnerabilities detection and may not provide the desired code coverage. In fact, the lack of adequate sampling heuristics can lead to both undetected vulnerabilities and time-consuming code analysis.

In this paper, we tackle the challenge of generating more adequate system configuration samples by taking into account the intrinsic characteristics of security vulnerabilities. We propose a new sampling heuristic based on data-flow analysis for recommending the subset of configurations that should be analyzed individually. Our idea is to test or manually inspect only the system configuration detected as potentially vulnerable according to our data-flow analysis heuristic. The goal of our proposed sampling heuristic is to decrease the sample size while maintain good detection coverage, that is, maintain near to 100% the number of vulnerabilities that can be found only analysing the sampled configurations. Therefore, we design and execute an empirical evaluation. It aimed to improve our understanding about the use of data-flow analysis as a heuristic for creating sampling sets. Our results show that we can can achieve high vulnerability-detection effectiveness with a small sample size.

Overall we make the following contributions:

- An experience report of how to employ variability-aware data-flow analysis for vulnerability detection on configurable systems, based on an existing data-flow analysis method [Sampaio and Garcia 2016].
- An empirical evaluation that compare the performance of variability-aware data-flow analysis on detecting secure vulnerabilities with the performance of state-of-the-art sampling strategies based on web-based configurable systems.

2. Background and Related Work

A security vulnerability is a flaw in a software system that allows attackers to exploit the system in a way not foreseen by the developer [Anley 2007]. Attackers typically take advantage of security vulnerabilities for malicious purposes, such as gaining access to sensitive data or sabotaging the software system operations.

2.1. Security Weakness with Input Dependency

Most security vulnerabilities in web-based software systems stem from input dependency weaknesses. According to OWASP Top 10 Security Risks [1], the most critical weaknesses in web-based software systems are the ones associated to improper input validation. They are characterized by missing validation or incorrect verification of input properties that are required to process data safely. We can cite as examples of which are considered as security weakness with input dependency: (i) **Injection**: Injection weakness (e.g SQL injection) occurs when untrusted input is provided to an interpreter as part of a command or query; and (ii) **Cross site scripting**: XSS weakness occurs whenever a software system receive untrusted input and sends it to a web browser without proper validation or escaping.

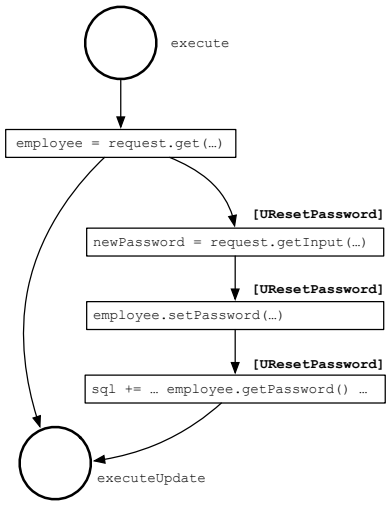
Figure 1 illustrates an example of a potential security weakness with input dependency. The statement in Line 10, in the `UpdateEmployeeDataService` class,

[1] `owasp.org/www-project-top-ten/`

Figure 1. Relevant points

```
01 public class UpdateEmployeeDataService extends Command {
02     ...
03     public void execute() throws Exception {
04         Employee employee = request.get(Login.EMPLOYEE);
05         updateEmployeePassword(employee);
06         employeeRepository.save(employee);
07     }
08
09     public void updateEmployeePassword(Employee employee) {
10         String newPassword = request.getInput("newPassword");
11         employee.setPassword(newPassword);
12     }
13 }
```

```
01 public class EmployeeRepository extends Repository {
02     ...
03     public void save(Employee employee) throws Exception {
04         ...
05         sql += "'" + employee.getPassword() + "')";
06         ...
07         stmt.executeUpdate(sql);
08         stmt.close();
09     }
10 }
```



retrieves from the `request` object the parameter ``newPassword'' and assigns it to the `newPassword` string variable. The input retrieved from the `request` object flows to the Line 05, in the `EmployeeRepository` class, without any input validation. Thus, an untrusted input from an attacker can trick the interpreter into executing unintended commands or accessing critical data.

2.2. Techniques for detecting security vulnerabilities

Proper input validation is the most common technique applied in practice to avoid untrusted input and can be applied to raw data (e.g., strings, numbers) or even to headers in order to ensure that inputs can be safely processed by the source code. There exist other techniques which attempt to transform dangerous input into ones safer as filtering or encoding/escaping. However, whether to introduce or not input validation can be by nature a unwieldy decision, specially when input dependency weakness can be included by unanticipated variability as illustrated in Figure 1. We can classify vulnerability detection techniques as:

- **manual inspection**: code review conducted by developers in order to uncover potential weakness. Code review has been massively applied in practice however it still be prone to errors.
- **static analysis**: a method which attempt to uncover potential weakness by employing static analysis techniques as *Taint Analysis* and *Data Flow Analysis*. Theses techniques aim to collect dynamic information about software systems while they are in a static state.
- **dynamic analysis**: a method which dynamically exercises the software systems behaviour using the same techniques that hackers and malicious individuals might adopt when trying to breach the source code security.

Regardless studies in the literature suggest that static analysis tools are efficient even when they report false positives [Sampaio and Garcia 2016], input dependency vulnerabilities are usually more effectively detected by dynamic analysis techniques [Aggarwal and Jalote 2006]. However, when we consider large numbers of variants in configurable system, exercising every possible input is not possible or impractical.

Also, creating and maintaining large test suites demands many human resources and might take too long. Therefore, as pointed out by [Medeiros et al. 2016], large samplings pose complex challenges to developers as testing several products variations is costly.

2.3. Sampling and Variability-aware Analysis

Dynamically exercising software systems is a costly task, but this may become impractical when it comes to configurable systems as variants grow exponentially as the number of variability increases. Therefore, in general, classic analysis methods do not scale in practice and researchers have begun to develop a new category of variability-aware analyses. Their aim at reduce analysis effort by exploit the similarities among variants. In contrast, sampling strategies handle the exponential grow of variants suggesting that, in the place of analyzing all variants, one selects a representative subset to be analyzed individually, one by one.

There are several algorithms that aim at to generate the best subset: the one which maximize our chances of finding bugs and has the fewest number of variants. Two classical sampling algorithms are one-enabled and one-disabled. The one-enabled produces the subset by selecting all variants that has only one variability enabled. In the other way around, the one-disabled proceeds by selecting all variants which has only one variability disabled. The resulting samplings are of the size N in terms of the number of variability. LSA [Medeiros et al. 2016] is another sampling algorithm which generates system configuration samples which size grows linearly in terms of the number of variability, in contrast to t-wise algorithms [Medeiros et al. 2016]. The LSA algorithm have been presenting goods result in terms of bug-detection capabilities and sample set sizes [Medeiros et al. 2016].

3. Data-Flow Analysis Heuristic in the Presence of Variability

Our heuristic uses a variability-aware parser to create an abstract syntax tree (AST) enhanced with variability information. Next, it applies a traditional data-flow analysis to suggest a subset of variants to be tested or manually inspected. In this section, we present our sampling heuristic detailing the generation of the variable AST (Section 3.1) and the variable data-flow analysis heuristic (Section 3.2).

3.1. Variable Abstract Syntax Tree

In contrast to other sampling strategies, our strategy aims at employing a vulnerability detection method that analyzes the complete set of variants and selects a subset to be tested or manually inspected. Therefore, we adopted a compact representation of abstract syntax tree (AST) as commonly used in other variability-aware analyses [Liebig et al. 2012]. Such a representation is called variable AST [Ferreira et al. 2015] and contains variability information about all variants. The variable AST as proposed by [Liebig et al. 2012] reassemble a non-variability aware AST, however it additionally includes information about whether a path should be considered as included in one variation or not. This representation is compact because it shares almost all paths that are common across variants. As stated by Liebig et al. [Liebig et al. 2012] it makes variability-aware analysis faster than any brute-force solution.

To build the variable AST we proceed by walking in the tree and, based on the configuration knowledge, stamping nodes with the variability active at that point

in the source code. We illustrate variable source code in Figure 1. In principle, we can use simple nodes in AST to represent mandatory statements, as the one in Line 04 (`UpdateEmployeeDataService`). Mandatory nodes do not carry out any information about variability. To reason about variability, we additionally stamp in the AST which sub trees have to be included in which variants. As an example, in Figure 1, the statement in Line 10 – `UpdateEmployeeDataService`, is only included whether Feature *User Reset Password* (`UResetPassword`) is selected. In our example, presence conditions refer only to a single Feature (`UResetPassword`), however more complex presence conditions are possible and common in practice.

3.2. Variable Data-Flow Analysis Heuristic

To perform data-flow analysis for vulnerability detection in the presence of variability, we construct a variable data-flow graph (DFG) which incorporates all possible paths in the configurable system. A data-flow graph is a model where nodes represent operations and predicates applied to data, and edges represent communication channels which moves data from a producing node to a consuming node. In DFGs, control- and data-flow are represented in one integrated model. Therefore, as pointed out by Liebig et al. [Liebig et al. 2012], variable data-flow graphs are conservative static approximations of the real behavior of the configurable software system.

In a variable DFG, nodes corresponds to statements in the variable AST and edges represents how data flows among statements. As in configurable systems the successors of a statement may differ across variants, we resort to include variability knowledge about successors nodes sets in the variable DFG. For example, in Figure 1 we illustrate an excerpt of the corresponding variability-aware DFG. The successor of the assignment statement `employee = request.get()` in Line 04 (`UpdateEmployeeDataService`) vary across variants: whether Feature `UResertPassword` is selected, assignment statement `newPassword = request.getInput()` in Line 10 (`UpdateEmployeeDataService`) is the direct successor; whether Feature `UResertPassword` is not selected, `stmt.executeUpdate()` in Line 07 `EmployeeRespositoy` is the only successor. By evaluating the variability on edges we can analyze individually the DFG of each variant in a compact and general manner as proposed by [Liebig et al. 2012].

We conduct our variability-aware data-flow analysis by selecting potentially insecure paths in the variable DFG. We consider a path as potentially insecure when it starts in a statement which receives external input data (source-point) and stops in a method call whose output goes beyond the boundaries of the configurable software system (sink-point), without passing through any input validation (sanitization-point). In Figure 1, we can observe a potentially insecure path which starts in the assignment statement in Line 10 (`UpdateEmployeeDataService`) and reach reach Line 07 (`EmployeeRepository`) without any input validation. That is, there is no sanitization-point between the source-point `employee = request.get()` and the sink-point `stmt.executeUpdate()`. As discussed by Sampaio and Garcia [Sampaio and Garcia 2016], such a heuristic drastically decrease the rate of false positives and present satisfactory results on detecting vulnerability in web-based software systems.

4. Experiment Setup

In this section we present the experiment we carried out to improve our understanding of how data-flow analysis can be used as a heuristic for creating sampling sets. To perform our study, we instantiated our strategy as a plug-in for a variability-aware tool. We chose two medium-sized, Java web applications: Health Watcher (HW) [2] and NCO [3]. These two web-based applications are configurable systems similar to the ones used in previous work. Additionally, we selected three common sampling strategies: one-enable, one-disabled and LSA. An in-depth discussion of the chosen sampling strategies is provided by [Medeiros et al. 2016].

4.1. Research Questions

As mentioned, our goal is to shed some light on the use of data-flow analysis as a heuristic for creating sampling sets in the context of configurable software systems. To this end, we devised the following research questions (RQs):

- **RQ_1:** How effective are the chosen sampling algorithms at detecting vulnerabilities?
- **RQ_2:** Which is the most cost effective sampling algorithm in terms of the size of the resulting sample?
- **RQ_3:** How many test paths have to be taken into account when employing each of the sampling algorithms?

4.2. Corpus of Vulnerabilities

We used the fault injection method to estimate the effectiveness of our sampling strategy. To answer our RQs, we used fault-seeding: that is, we injected faults (i.e., vulnerabilities) into the two web applications in our sample. Specifically, we inserted three types of data flow related vulnerabilities: (i) data used in object instantiation – an untrusted data is used as parameter in object instantiation; (ii) data used as method parameter – an untrusted data is used as parameter in a method call; and (iii) data as method return – an untrusted data returned by a method is used in assignment statements or method call. Overall, the corpus of vulnerability comprises 25 injected in the HW and 23 injected in the NCO.

5. Results and Discussion

In this section, we present and discuss our results in terms of the RQs (Section 4.1).

5.1. RQ_1: How effective are the chosen sampling algorithms at detecting vulnerabilities?

We found that all sampling strategies detected more than 90% of the vulnerabilities in the configurable software systems in our sample. One-enabled detected the lowest number of vulnerabilities, while all vulnerabilities in our corpus can be detected by exercising only 3 products in HW and NCO as selected by GVD. In the worst scenario, one-enabled missed 13 vulnerabilities in HW because it requires developers to select some variability and disable all the others. We observed that to detected some vulnerabilities it is necessary to

[2] ptolemy.cs.iastate.edu/design-study/hw/HealthWatcherOO$_{a}ll.tar.gz$

[3] www.inf.puc-rio.br/ lsampaio/nco/nco.zip

HW				
Strategy	**Num. Variants**	**Vulnerabilities**	**Vulnerable Paths**	**Paths Total**
GVD	3	28/28	46/46	46/1632
One-enabled	61	15/28	25/46	651/1632
One-disabled	61	25/28	43/46	1403/1632
LSA	124	25/28	43/46	1403/1632
NCO				
Strategy	**Num. Variants**	**Vulnerabilities**	**Vulnerable Paths**	**Paths Total**
GVD	3	24/24	78/78	78/2672
One-enabled	37	22/24	54/78	2286/2672
One-disabled	37	24/24	78/78	2672/2672
LSA	76	24/24	78/78	2672/2672

Table 1. Health Watcher and NCO results

explore specific combinations of code blocks, which is not always achieved by existing traditional algorithms. Considering NCO, most of the sampling algorithms performed well, detecting all the vulnerabilities. In contrast, HW has configuration constraints that render some combinations impossible to be generated by traditional sampling algorithms to perform poorly.

5.2. RQ_2: Which is the most cost effective sampling algorithm in terms of the size of the resulting sample?

The sizes of the sample sets range from 3 to 124 in HW and from 3 to 76 in NCO. According to our results, GVD was the sampling strategy that yielded the smallest sample sets. LSA selected the largest sample sets considering both configurable software systems in our sample. These results raise one major issue: these sampling strategies are not enough to cope with large configuration spaces because most of them yield a large number of configurations that need to be tested properly. However, the number of configurations is often too large for exhaustive testing and generating test inputs for testing all combinations is in general unwieldy. This supports our assumptions that sampling strategies should mainly take into account the intrinsic characteristic of vulnerable code, instead of taking into account only the information in configurable files or feature models.

5.3. RQ_3: How many test paths have to be taken into account when employing each of the sampling algorithms?

The total number of paths corresponds to the the number of paths that have to be exercised when we have to execute the N products recommended by the sampling strategy. For example, for HW, 651 paths out of 1632 are covered by exercising the 61 products recommended by one-enable. It is worth noting that these paths are the ones that start in a source-point and reach a sink-point. We can observe that there is a redundancy issue associated with the variants selected by traditional sampling algorithms. For example, when traversing only 46 paths, as recommended by GVD, we were able to detected all 28 vulnerabilities in HW. In contrast, when exercising the 1403 paths yielded by LSA, we we able to detect only 25 out of 28 vulnerabilities in HW. The same behavior can be observed in NCO – exercising 2286 out of 2672 paths turned out not to be enough for detecting all vulnerabilities.

6. Concluding Remarks

In this paper, we tackle the challenge of cost-effective vulnerability detection by selecting more adequate system configuration samples by taking into account the intrinsic characteristics of security vulnerabilities. We propose and evaluated a new sampling heuristic based on data-flow analysis for recommending the subset of configurations that should be analyzed individually. To do so, we analyzed the behaviour of our proposed sampling heuristic in comparison with three common sampling strategies (one-enable, one-disabled and LSA) considering 48 vulnerabilities injected in two configurable systems. Our results show that we can can achieve high vulnerability-detection effectiveness with a small sample size.

Several avenues for future exploration are possible. As mentioned, our results suggests that enhancing weakness detection heuristics with variability information could be a better sampling strategy than generating sample sets by solely considering variability information. Therefore, as a future work, it would be interesting to investigate whether other bug detection heuristics can be also enhanced with variability information. Additionally, since our results show that a reduced number of paths might have to be taken into account when testing a configurable software, we believe that existing test case generation strategies can take advantage of such observation.

References

Aggarwal, A. and Jalote, P. (2006). Integrating static and dynamic analysis for detecting vulnerabilities. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 1, pages 343–350. IEEE.

Anley, C. (2007). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2nd edition.

Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., and Borba, P. (2013). Intraprocedural dataflow analysis for software product lines. In *Transactions on Aspect-Oriented Software Development X*, pages 73–108. Springer.

Ferreira, G., Kästner, C., Pfeffer, J., and Apel, S. (2015). Characterizing complexity of highly-configurable systems with variational call graphs: Analyzing configuration options interactions complexity in function calls. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*.

Ferreira, G., Malik, M., Kastner, C., Pfeffer, J., and Apel, S. (2016). Do ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *International Systems and Software Product Line Conference (SPLCâ16)*.

Liebig, J., Von Rhein, A., Kästner, C., Apel, S., Dorre, J., and Lengauer, C. (2012). Large-scale variability-aware type checking and dataflow analysis.

Medeiros, F., KÃstner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 643–654.

Sampaio, L. and Garcia, A. (2016). Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software*, 113:337–361.