Characterizing Complexity of Highly-Configurable Systems with Variational Call Graphs

Analyzing Configuration Options Interactions Complexity in Function Calls

Gabriel Ferreira[1], Christian Kästner[1], Jürgen Pfeffer[1], Sven Apel[2]
[1]Carnegie Mellon University, United States
[2]University of Passau, Germany

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*

Keywords

configuration options, complexity metrics, variational call graphs, pointer analysis

1. INTRODUCTION

Security has consistently been the focus of attention in many highly-configurable software systems. Several vulnerabilities on widely-used systems, such as the Linux kernel and OpenSSL, are reported every day in the National Vulnerability Database (NVD). The configurability of these systems enables the rapid generation of customized products, but also creates security challenges in the development and maintenance processes. For instance, interactions caused by configurations may create serious security threats and make generated products more susceptible to attacks [6], but the causes of these problems may be harder to detect because they occur only in specific configurations.

In this poster presentation, we outline our technical approach of characterizing configuration complexity of highly-configurable systems using call graphs. Our main technical contribution is the implementation of a variational call graph, including a pointer analysis that improves its accuracy by also including indirect function calls made by pointers. The resulting variational call graph will provide a foundation for subsequent analysis based on network analysis graph-based metrics. Our long-term goal is to explore these metrics results to provide means to developers to reason about the complexity of configuration interactions and to help them to identify critical systems' parts, increasing their awareness to potential threats when performing development and maintenance tasks.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the Owner/Author.
HotSoS '15 Apr 21-22, 2015, Urbana, IL, USA
Copyright is held by the owner/author(s).
ACM 978-1-4503-3376-4/15/04.
http://dx.doi.org/10.1145/2746194.2746211.

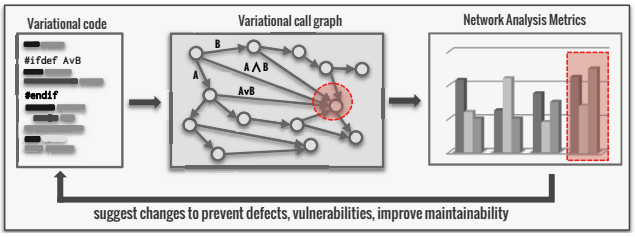


Figure 1: Simplified view of our intended approach

2. TECHNICAL APPROACH

Several works have focused on alternative methods to characterize complexity, predict defects, and identify vulnerabilities in software systems [8, 5], because classical methods rely on imprecise size-based measures [2]. For example, McCabe's cyclomatic complexity [4] was a commonly used, though heavily criticized, metric to measure the complexity of source code. More sophisticated metrics, based in statistics, data mining, and networks analysis, have been studied to capture complexity caused by various structural abstractions of software systems [7, 8]. According to Zimmerman [8], network-based metrics on dependency graphs can be used to identify critical parts and predict the number of occurring defects in software systems.

Our initial goal is to assess variability-related complexity of systems using call graphs as abstraction. The main technical challenge is to gather accurate variational call graphs that include configuration information. A variational call graph includes all possible function definitions and function calls of any configuration and labels each one with a presence condition (a propositional formula over configuration options) characterizing exactly in which configurations a function definition or function call would be included. The result is a labeled graph that can be used for subsequent analysis; when analyzing configuration complexity, we are especially interested in those labels. In Figure 3 we show an example of a variational call graph for the code in Figure 2. Most existing infrastructures for analyzing source code are designed for individual configurations (i.e., analyzing code after preprocessing). Instead, our goal is to create a unique variational call graph that represents all possible products but also describes in which configuration individual function calls are possible, but in a scalable fashion without actually analyzing each configuration individually in a brute-force fashion.

```
1     void foo() { ... }
2     void bar() { ... }
3     void baz() { ... }
4
5     int main() {
6     #ifdef B
7        void (*fp)();
8     #ifdef A
9        foo();
10       fp = &bar;
11    #else
12       fp = &baz;
13    #endif
14       (*fp)();
15    #endif
16    }
```

Figure 2: Example of variational code in C with use of pointers to increase flexibility of function calls on runtime.

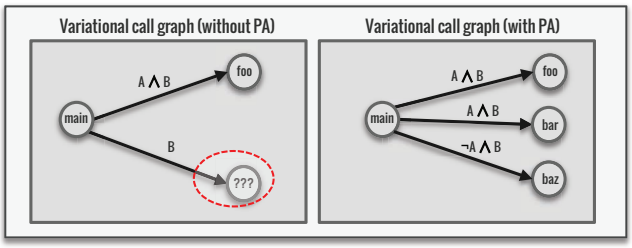


Figure 3: Precision comparison of the resulting call graphs generated from Figure 2; (a) call graph without pointer analysis, (b) call graph with pointer analysis.

Indirect calls through pointers or other forms of dynamic dispatch add further complexity when creating call graphs. Many programs rely on these calls to create flexibility in their run-time behavior. Therefore, ignoring indirect function calls can drastically decrease the accuracy and usefulness of call graphs of these programs. Figure 3 illustrates the difference of accuracy between approaches that differ in using a pointer analysis to compute a call graph. To compute a variational call graph from our example, we would create nodes for functions `main`, `foo`, `bar`, and `baz` and an edge from `main` to `foo` under the condition $A \wedge B$ for the call in Line 9 as shown in Figure 3a; with our pointer analysis, we re able to identify to where the pointer function call in Line 14 is directed.

Our main contribution is the implementation of a variational call graph including a pointer analysis that enriches the otherwise imprecise extraction of call graphs as part of the TypeChef infrastructure [1, 3]. The variational pointer analysis reasons about possible targets for function pointers *in all possible configurations*. The improved variational call graph will provide a foundation for subsequent analysis based on network analysis graph-based metrics. In the long run, we expect these metrics to provide feedback to developers about critical parts of these systems, which could be used as indicatives of defects or potential vulnerable portions of the program (Figure 1).

3. CONCLUDING REMARKS

We are currently computing and analyzing the variational call graph of Busybox[1]. The resulting variational call graph has 3179 distinct function definitions and 14397 distinct function calls. From all function calls, 10797 are included in all configurations and 3600 (25%) have presence conditions. Next, we need to extend classical network metrics (e.g., centrality, community detection, and concentration metrics) to enable them to be applied to variational graphs, since the classical metrics do not support graphs with propositional information. After this initial analysis, we intend to compute the variational call graphs of other large open-source software systems, including the Linux kernel and OpenSSL. Once we gather variational call graphs representations from each of these configurable systems, we intend to evaluate the usefulness and accuracy of our approach by comparing our graph-based metrics results with reported vulnerabilities of highly-configurable systems from public available databases, such as the NIST National Vulnerability Database (NVD).

[1] http://www.busybox.net/

Acknowledgments

G. Ferreira is supported by a doctoral research grant from CAPES, Brazil. Grant n. 13713/13-2.

4. REFERENCES

[1] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM Press, 2011.

[2] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2013.

[3] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, 2013.

[4] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng. (TSE)*, 2(4):308–320, 1976.

[5] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 452–461, 2006.

[6] A. Nhlabatsi, R. Laney, and B. Nuseibeh. Feature interaction: The security threat from within software systems. *Progress in Informatics*, (5):75–89, 2008.

[7] S. Sobernig, S. Apel, S. S. Kolesnikov, and N. Siegmund. Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines: An exploratory study. *Empirical Software Engineering*, ((forthcoming)), October 2014.

[8] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 531–540, 2008.