Limiting Recertification in Highly Configurable Systems

Analyzing Interactions and Isolation among Configuration Options

Christian Kästner
Institute Software Research
Carnegie Mellon University
Pittsburgh, PA 15213
kaestner@cs.cmu.edu

Jürgen Pfeffer
Institute Software Research
Carnegie Mellon University
Pittsburgh, PA 15213
jpfeffer@cs.cmu.edu

ABSTRACT

In highly configurable systems the configuration space is too big for (re-)certifying every configuration in isolation. In this project, we combine software analysis with network analysis to detect which configuration options interact and which have local effects. Instead of analyzing a system as Linux and SELinux for every combination of configuration settings one by one ($>10^{2000}$ even considering compile-time configurations only), we analyze the effect of each configuration option once for the entire configuration space. The analysis will guide us to designs separating interacting configuration options in a core system and isolating orthogonal and less trusted configuration options from this core.

Categories and Subject Descriptors

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

Keywords

composability, configuration options, certification, security metrics, network analysis

1. INTRODUCTION

Configuration options have repeatedly been a problem for security. For instance, severe security vulnerabilities are pervasive in non-browser SSL validation code mostly due to misconfigurations of SSL libraries and disabling unneeded functionality in the Linux kernel could avoide over 50 % of CVE security vulnerabilities for most users [4].

Systems with many configuration options are challenging to analyze and certify for security properties, so currently only code snapshots are certified. Changes to configuration options have crosscutting effects, requiring costly recertification. In fact, the configuration space grows exponentially with the number of configuration options. Security vulnerabilities may hide in specific combinations of configuration options only, triggered by (often unintended) interactions among configuration options. Whereas current approaches work on specific snapshots and require expensive recertifications, our approach extend underlying mathematical models (data-dependence graphs) with configuration knowledge and will thus scale analyses and reduce the need for repeating analyses. In this short paper we outline our technical approach of modeling the statics and dynamics of complex software systems.

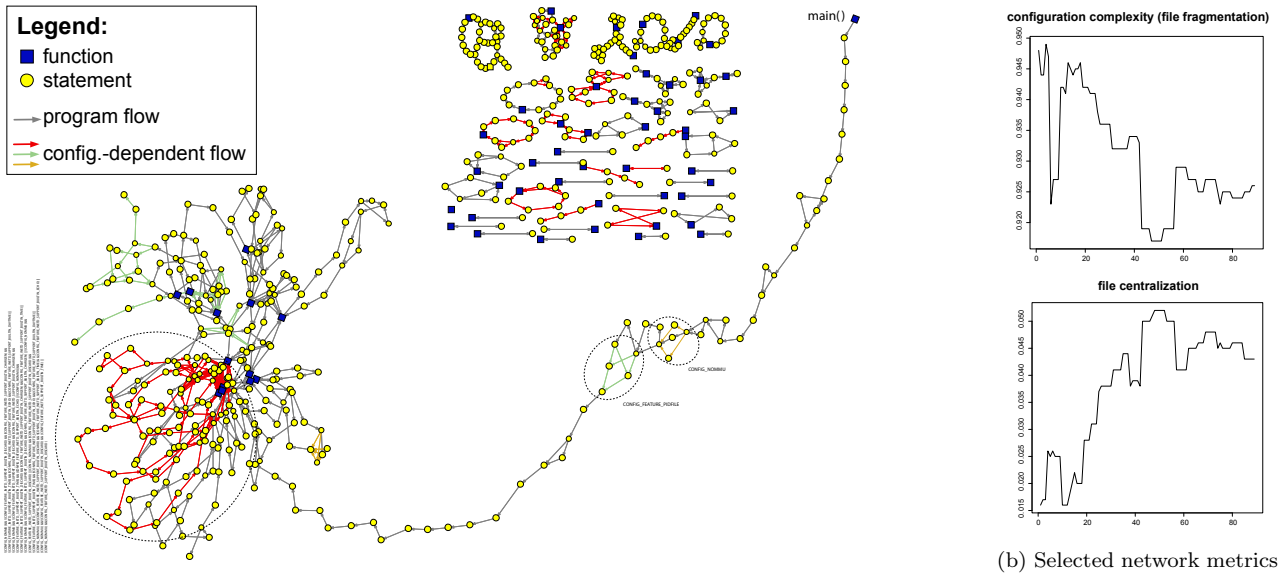
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
HotSoS '14, April 08-09 2014, Raleigh, NC, USA
Copyright 2014 ACM 978-1-4503-2907-1/14/04 ...$15.00
http://dx.doi.org/10.1145/2600176.2600199.

2. TECHNICAL APPROACH

Various forms of abstract interpretations have been explored to statically analyze a program's source for various security properties. Recently, we and others have have extended such analyses to precisely cover large configuration spaces [1, 5]. Network analysis has been applied to research problems in many different fields [2]. In the context of software research source code [8] and developer networks [6] have been analyzed with network models to quantify various quality attributes, predict defects, and identify critical interactions. Here, we combine static analysis and network analysis with a focus on configuration options.

We extract network information from the source code, precisely tracking information about configuration options. Initially, we focus on C code with #ifdef directives, because, in such implementations, configuration options are explicitly tagged. Subsequently, we will analyze also load-time configurations, such as command-line parameters and configuration files, with additional analysis to first establish the mapping between configuration options and corresponding activated code fragments.

We track configuration options with our TypeChef infrastructure, that can parse and process unpreprocessed C code and build an accurate control-flow graph for the entire system, covering *all* configurations [3, 5]. Technically, we determine for every code structure a (propositional) *presence condition* describing in which configurations the structure is included; for example, we derive a presence condition $X \wedge \neg Y$ if the structure is only included if configuration option X is enabled and Y is disabled. That is, instead of representing the structure of each product separately, we represent the structure for all configurations at once, where presence conditions describe under which configurations certain structures are included. On top of existing control-flow analysis, we track data dependencies throughout the system across the configuration options to build a program dependence graphs [7]. The graph covers all configurations without exploring every configuration separately and with-



(a) Control flow of *networking/inetd.c*.

(b) Selected network metrics over time for 89 releases.

Figure 1: Graph representation of program code and configuration options; system: busybox.

out loosing precisions through approximation or heuristics. In Fig. 1a, we show an example of such graph (here based on control-flow information), where highlighted edges represent configuration-dependent control flow with specific presence conditions. This modeling approach creates significant challenges for network analysis. State of the art network analysis can handle *multiplex graphs*, but just on the level of describing isolated relations—friendship, email communication, face-to-face communication, etc. We are not aware of any network analytical research that has been done to study graphs with complex (propositional) presence conditions.

3. VULNERABILITY ANALYSIS

We use the resulting graphs representing the system's dynamic and static structure including all configuration options for further analysis. We also have hierarchical information along several dimensions: structural elements are typically nested in other structural elements (e.g., statements in functions in files in directories) and configuration options often are structured in a hierarchical variability model (options and sub-options, often grouped by topic) with known constraints between configuration options (e.g., X may only be selected if Y is enabled). We use network measures to identify isolated nodes and critical nodes in the graph, similar to identifying critical persons in a social network. For example, we identify combinations of configuration options that are essential for exchanging critical information between otherwise separate subsystems and functions in which the control flow is complicated by many configuration options. We expect that well-selected network metrics can predict code that is more likely to be vulnerable or in which vulnerabilities have a more severe impact. In addition, observing network metrics over time (Fig. 1b) allows us to detect subtle creeping increases of configuration complexity over time and allowing developers to intervene.

For validation, we correlate known vulnerabilities and their severities from the NIST National Vulnerability Database with our findings to determine to what degree configuration-related complexity can indicate implementations that are more prone to vulnerabilities or in which vulnerabilities have more severe consequences

4. REFERENCES

[1] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL^{LIFT}: Statically analyzing software product lines in minutes instead of years. In *PLDI*, pages 355–364, New York, 2013. ACM.

[2] M. Henning, U. Brandes, and J. Pfeffer. *Studying Social Networks*. Compas Verlag/University of Chicago Press, Frankfurt/Chicago.

[3] 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 *OOPSLA*, pages 805–824, New York, 2011. ACM.

[4] 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 *NDSS*, pages 1–18. The Internet Society, 2013.

[5] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/FSE*, pages 81–91, New York, 2013. ACM.

[6] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *FSE*, pages 13–23, New York, 2008. ACM.

[7] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE*, pages 177–184, New York, 1984. ACM.

[8] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08*, pages 531–540, New York, 2008. ACM.