ORIGINAL ARTICLE

Journal Section

A Case Study on Modularity Violations in Cyber Physical Systems

Lu Xiao[1] | Michael J. Pennock[1] | Joana L. F. P. Cardoso[1] | Xiao Wang[1]

{lxiao6, mpennock, jcardoso, xwang97}@stevens.edu

[1]School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030, USA

Correspondence
Lu Xiao
Email: lxiao6@stevens.edu

Funding information
This study was funded by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract HQ0034-13-D-0004.

In recent years, cyber-physical systems (CPS) have been widely used in different problem domains. The hardware and software components in a CPS are deeply intertwined at various levels of abstraction under changing contexts to achieve the desired goals. One way to manage this complexity is through the use of modular architecture that enables portions of a CPS to be upgraded, replaced or fixed in a plug-and-play manner. In practice, nominally modular architectures depart from this ideal. Thus, there is a need to identify, measure, and understand the causes of these departures. Techniques have been developed to accomplish this for pure software systems, but can these techniques be adapted to infer hardware-related modularity violations in a CPS? To investigate this question, we adapted methods from pure-software systems and analyzed two open source cyber-physical systems: OpenWrt and MD PnP. We found that the software architecture of these systems is well-modularized. However, we were able to detect and measure modularity violations associated with hardware and other domain related concepts. Furthermore, software components related to hardware were more likely to change frequently compared to general software components. Together, these findings suggest it is feasible to identify a subset of hardware-driven modularity violations using techniques adapted from pure software systems.

KEYWORDS
modularity, cyber-physical system, change propagation

1 | INTRODUCTION

In recent years, cyber-physical systems have achieved widespread application in diverse areas including: civil infrastructure, energy, health-care, transportation, automotive, smart appliances, and others [1]. With the advance of technology, the recognition of new consumer needs, and the detection of deficiencies in current systems, components need to be upgraded, replaced, and fixed—frequently, in many domains. Operators and users of these systems need to be

able to accomplish these actions quickly and without disrupting the rest of the system. In principle, a modular system architecture should facilitate such actions, but nominally modular system designs are not always modular in practice. An understanding of when and how these deviations occur could enable the development of improved design techniques, more effective project management approaches, and more successful upgrade efforts for cyber-physical systems.

Departures from nominally modular architectures have already been studied in pure software systems. In software systems, a module encapsulates code and data to implement certain functions and provides uniform interfaces for its clients (i.e. other modules) to use those functions [2]. In a large-scale software system, what constitutes a module depends on the perspective of the stakeholder. Various stakeholders may view components, packages, and source files as modules for addressing separate concerns [3]. From the perspective of developers, source files are fine-grained modules since each source file provides a group of cohesive functions that interact with functions provided by other source files. Each source file is a separate unit in the system that is designed, implemented, and maintained by different developers [4, 5, 6, 7]. Ideally, sources files without any explicit structural dependencies should be able to evolve independently without impacting each other. However, in practice, it has been observed that source files without any explicit structural dependencies were frequently changing together when new features or bug fixes were implemented. Wong et. al. [6] call this phenomenon a modularity violation.

The analysis of modularity violations in software systems relies on identifying discrepancies between the structural dependencies among source files and how the source files change together in evolution history. That is, if two source files are structurally independent, but they frequently change together, they are considered to be involved in a modularity violation. Modularity violations in software systems are usually the result of shared “assumptions" among source files. For example, in a case study of a commercial project, a modularity violation was caused by the assumption of a certain time unit being used in a group of mutually independent source files. Whenever the time unit changed from milliseconds to seconds in one source file, the other source files had to change accordingly to avoid errors [8]. Studies of various software systems revealed that modularity violations could cause significant maintenance consequences to software systems [9, 10].

Despite the prevalence of cyber-physical systems in recent decades, there are a lack of techniques and empirical experience that would allow stakeholders to detect, measure, and understand modularity violations in developed and acquired cyber-physical systems. The challenge is that a cyber-physical system is composed of both software sub-systems and hardware sub-systems. Existing engineering techniques mostly either focus on the cyber perspective or the physical perspective. There are only a few approaches, such as [11], that integrate both perspectives. A natural question is whether the techniques developed to analyze modularity violations in pure software systems can be adapted to understand the latent connections between the cyber and physical aspects of a system.

To that end, the goal of this study is to evaluate the feasibility of adapting techniques from software engineering to the detection, measurement, and analysis of modularity violations in cyber-physical systems. We propose that modularity violations in cyber-physical systems should be organized into three different categories: 1) software-software level violations, which happen between two software modules and are the same as modularity violations in traditional software systems; 2) software-hardware level modularity violations, which happen between software modules and hardware modules; and 3) hardware-hardware level modularity violations, which happen between two hardware modules. We hypothesize that when hardware-software or hardware-hardware violations are encountered during operations, it is much more likely that these are addressed via software changes than hardware changes due to the differences in both cost and time required. If that hypothesis were correct, there should be evidence of hardware driven changes in the maintenance logs of the software components. An example of this evidence would be logs indicating that software source files that have been changed due to updates in hardware-related concepts, such as “zigbee" and “bluetooth". This propagation of change indicates implicit connections between the physical and cyber sides

in a cyber-physical system.

This paper contributes a first-of-its-kind case study of real-life cyber-physical systems to analyze modularity violations that are potentially triggered by hardware-related concepts. It demonstrates the feasibility of identifying potential hardware-driven modularity violations in cyber-physical systems by analyzing a software version control system. More importantly, the approach enables the identification of the shared concepts that potentially propagate maintenance actions between the cyber and the physical sides. Understanding the role of these shared concepts is central to improving approaches to developing modular cyber-physical systems.

The remainder of this paper is organized as follows: Section 2 describes the nature of the modularity problem in cyber-physical systems as well as prior work in detecting modularity violations. Section 3 describes our approach to adapting the prior work from pure software systems to analyzing OpenWRT and MD PnP, and Section 4 discusses the results. Section 5 discusses both the implications and limitations of this study as well as future work we plan to address those limitations. Finally, Section 6 concludes the paper.

2 | BACKGROUND

To provide some background for this work, in this section, we first review the literature associated with architecting cyber-physical systems. This literature reveals the importance of modularity when architecting cyber-physical systems but also indicates a lack of techniques to assess modularity in these systems. However, these techniques do exist for pure software systems, and so we next introduce an existing methodology used detect and analyze modularity violations in traditional software systems. Finally, we discuss the challenges to adapt this technique to detect modularity violations in cyber-physical systems. These challenges ultimately motivated this work

2.1 | Architecting Cyber-physical Systems

The term "cyber-physical systems" emerged around 2006, when it was coined by Helen Gill at the National Science Foundation [12, 13]. Gill defined a cyber-physical system as an integration of computation with physical processes. Usually cyber-physical systems are composed of diverse subsystems consisting of both physical and software components developed by different vendors. These components are deeply intertwined at various levels of abstraction (e.g. data flow, physical connections, and logical connections etc.) under changing contexts to achieve the desired functionalities and the respective quality attributes, such as performance, security, scalability, maintainability, etc. It has been recognized that the design, modeling, and maintenance of cyber-physical systems is more challenging than for traditional engineering systems due to the intrinsic heterogeneity and nondeterministic nature of the interactions among cyber and physical components [13, 14, 15]. Architecting and evolving a system under these circumstances is a formidable problem.

To address this issue, stakeholders have increasingly emphasized modular and open approaches to system development. According to Baldwin and Clark [16], modularity allows for both independence of structure and integration of functions in large and complex systems. The goals of modularity are to improve interoperability, facilitate system evolution and technology insertion, and foster competition. However, it can be difficult for the stakeholders to assess whether the resulting architectures and systems are truly modular. In other words, can modules in a cyber-physical system actually be upgraded, replaced, and fixed in a plug-and-play manner without affecting one another to maximally leverage the benefits of modularization?

One of the challenges to architecting and managing cyber-physical systems is to capture the cyber processes and

the physical processes at the same time. Existing techniques mostly either represent the cyber side or the physical side [17, 18]. From the "cyber" perspective, there are different techniques to represent the architecture of software systems, such as architecture description languages, Unified Modeling Language (UML) models, and component models. From the physical perspective, there are different traditional engineering techniques to model the development of physical systems.

There is only limited research that takes a more holistic approach to architecting cyber-physical systems. For example, Kernschmidit and Vogel-Heuser developed a modeling approach, based on the Systems Modeling Language (SysML), to analyze interdisciplinary change influences in production plants [11]. Rajhans, et al. [19] contributed a new cyber-physical architectural style to present the interconnections and interactions between physical and cyber components. They defined three related families of general components and connectors pertaining to the cyber domain, the physical domain, and their interconnections. From the architectural assessment perspective, Sinha [20] developed a theoretical framework for structural complexity quantification and its implications for the design of cyber-physical systems. Derler et al. [12] focused on the challenges of modeling cyber-physical systems that arise from the intrinsic heterogeneity, concurrency, and sensitivity to timing of such systems, while describing some promising approaches that use domain-specific ontologies to enhance modularity and jointly model the functional and implementation architectures. Finally, Cristalli et. al. [21] provides a representative example of a modular approach to developing a cyber-physical system through their modular design for a Smart Robotic Cell, composed by several interconnected sub-systems.

What all of the above have in common is a focus on architecting modular cyber-physical systems. However, there is no guarantee that the realized system will exhibit the intended modularity. The need to assess the actual modularity of a cyber-physical system is particularly acute. As Lee [22] points out, cyber-physical systems have always been held to a higher standard of reliability and predictability than general-purpose computing. Low reliability and predictability constitute a significant barrier to the application of cyber-physical systems to safety-critical domains such as traffic control, automotive safety, and healthcare. While a modular design is certainly an option to improve reliability, predictability and maintainability, when employed, it is desirable to understand how well that goal is accomplished. Approaches to assess modularity in pure software systems have been developed, but to the best of the authors' knowledge no approaches have been developed that specifically address the unique challenges of assessing the modularity of cyber-physical systems. Thus, the question is whether existing approaches from the software domain can be adapted to the cyber-physical domain.

2.2 | Detecting Modularity Violations in Software Systems

The term modularity violation was first proposed by Wong et al. [6] to describe the phenomenon where independent software modules frequently change together during the evolution of a system in the process of fixing bugs or adding features. Methodologies and tools have been built for analyzing modularity violations in software systems [23, 6, 7]. The identification of a modularity violation follows three steps:

1. Reverse-engineering the modular structure of a software system from the code base: The modular structure of a software system can be calculated based on the structural dependencies among software entities. Xiao et al. developed a new architectural representation, called Design Rule Space (DRSpace) modeling to capture the modular structure of a software system as multiple overlapping design spaces [7], applying Baldwin and Clark's design rule theory [16]. The modular structure can be visualized in the form of a Design Structure Matrix (DSM). The items in the DSM represent software entities, the cells represent the structural dependencies among entities.

The entities can be clustered into modules based on selected dependency types for supporting analysis of different focuses.

2. Extracting the evolutionary coupling (i.e. the number of co-changes) between software modules: Evolutionary coupling can be calculated based on the data recorded in version control systems, which automatically keep track of changes made to software entities. Consequently, we are able to determine who changed what and when and for what reason. Analyzing such data allows us to determine the frequency of concurrent modifications of any two software entities in the course of maintenance activities. In other words, it enables us to extract the evolutionary coupling between software entities. The more frequently two entities are modified together, the stronger is their evolutionary coupling—implying latent connections.
3. Calculating the discrepancy between the above two data sets to point to modularity violations among software modules: If two software entities/modules are structurally independent according to step 1, but they have high evolutionary coupling according to step 2, there is a potential modularity violation. According to prior empirical studies of industry software systems, modularity violations usually indicate implicit assumptions shared among software modules. For example, it could be an implicit assumption regarding the usage of time units in two modules [8].

In software systems, the consequences of modularity violations could be higher bug-rates (when the assumptions are not consistent among different parties) and increased maintenance costs (in order to keep the assumptions consistent) [23, 7, 9]. Studies of real-life open source projects indicate that up to 85% of bug-fixing efforts in a software system involve modularity violations [9].

2.3 | Detecting Modularity Violations in Cyber-Physical Systems

The described approach for detecting modularity violations is limited in that it only addresses software systems implemented in a single programming language. Complex cyber-physical systems, however, comprise both heterogeneous hardware and software sub-systems, which makes them particularly difficult to analyze. Indeed, incurring modularity violations that result from latent relationships between software and hardware modules endanger the system's long term evolution, maintenance, and success. In particular, due to the inflexibility and high cost of evolving hardware modules, modularity violations in cyber-physical systems could more easily lead to vendor lock-in compared to a pure software environment.

In this paper, we organize modularity violations in cyber-physical systems into three different types: 1) software vs. software, 2) software vs. hardware, and 3) hardware vs. hardware. The first type of modularity violation has been extensively investigated in prior work [6, 8, 7, 23, 9]. There, the approach to identify and measure modularity violations relies on the analysis of source code and operational data such as maintenance activity records. As previously mentioned, such information can be automatically and comprehensively tracked in version control systems. The immediate challenge that arises when trying to extend the same approach to identify hardware-specific modularity violations is that, oftentimes, it can be difficult to find records that extensively describe changes made to the different hardware components. Given these limitations, the question naturally follows, could a subset of hardware modularity violations be inferred from an analysis of records of software changes? More specifically, some software changes may be a consequence of hardware related modularity violations. If that were the case, analysis of version control systems could be used to detect some hardware related modularity violations outright and flag other potential violations for further investigation. The issue is whether or not there is actually enough information in a version control system to infer a hardware modularity violation. Consequently, the goal of this study was to test the feasibility of leveraging software

maintenance data to infer hardware related issues that are indicative of software-hardware modularity violations.

3 | APPROACH

In this section, we first introduce the two cyber-physical systems that served as the subjects of this study. Next, we describe our approach to analyze and understand potential modularity violations in these two projects using a three step process.

We chose to analyze the open source software infrastructures for two real-world cyber-physical systems: OpenWrt and MD PnP. OpenWrt is a Linux based core that is commonly used to support the Internet of Things (IoT). MD PnP stands for the Medical Device "Plug-and-Play" interoperability program which aims to improve patient care by integrating information from medical devices. Table 1 summarizes basic facts of these projects. It shows the scale (measured by number of files, methods, and lines of code), the development team size, and the age (measured by the number of revisions and starting time). These projects were chosen because, in addition to providing sufficient research data, they also involved two very different application domains.

We analyzed each project using the following process:

Step 1: Reverse Engineering

We recovered the modular structure of the two projects using reverse engineering techniques. Specifically, we leveraged the tool set previously built by Xiao et. al. [7]. The input of this step included the code repository and code revision history. This step generated two source file level Design Structure Matrices for each project: one for the static structural dependencies and the other for the dynamic evolutionary coupling as shown in Figure 1 and Figure 2 for OpenWrt and MD PnP, respectively. As discussed earlier, we chose source files as the fine-grained modules in these two systems. The first DSM describes the design structure formed by source files, while the second DSM describes how the source files actually change together in system evolution. The detailed interpretation of these matrices will be elaborated later.

Step 2: Modular Structure Measurement

We measured the modular structure of the software portion of each system using two complementary metrics called Decoupling Level developed by Mo et. al. [23] and Propagation Cost developed by MacCormack et. al. [24].

The Decoupling Level describes how well a system is decomposed into small, manageable modules [23]. In order to compute the Decoupling Level of a system, the structural DSM of a system is clustered into hierarchical layers using the Architectural Design Rule Hierarchy clustering algorithm [25]. After the clustering, the source files in the DSM are grouped into hierarchical layers based on their structural dependencies: the lower layers depend on the upper layers, but upper layers do not depend on the lower layers. For example, in Figure 1a, the layers are highlighted in rectangles and the dependencies are all aggregated in the lower triangle of the DSM as a result of the clustering algorithm. Decoupling Level measures how well source files in each layer are decoupled from the entire system. We measure two aspects to calculate a separate Decoupling Level for each layer: 1) the number of dependencies from the lower layer source files; and 2) the number of source files and their dependencies within a layer. Adding the Decoupling Level of each layer yields the overall *Decoupling Level* of the entire system. Due to its construction, the highest possible value of the overall Decoupling Level of a system is 1. This means that the system is perfectly modularized, which is not likely in practice. A lower value for Decoupling Level indicates a less modularized system. For example, if there are 100 source files in a system and these 100 files are completely independent from each other, the Decoupling Level will be 1. In comparison, if the 100 source files are completely interconnected, the Decoupling Level will be 0. The detailed

calculation process can be found in Mo. et. al.'s work [23]. In this study, we compared the *Decoupling Level* of the two case study projects with the Decoupling Level of 129 traditional software systems previously evaluated by Mo et. al. [23]. This facilitated the understanding of the overall modular structure of the two case study projects, as well as how modularized they are relative to other systems.

The Propagation Cost assesses the modularity of a software system by measuring the degree to which a change to any single source file causes a potential change to other source files in the system, either directly or indirectly (i.e., through a chain of dependencies that exist across source files) [24]. The Propagation Cost of a system is calculated as the density of the n-transitive closure of its structure DSM. The structure DSM of a system describes how source files are directly connected to each other. The density of the DSM is the number of dependencies among source files divided by the square of the total number of source files in a system. Thus, a value 100% means that every source file is directly connected to every other source file in the system. The n-transitive closure of a DSM is found by multiplying the DSM by itself n times. The result is still a DSM, but it now describes how source files are indirectly connected to each other through maximally $n - 1$ hops [26]. For example, in a 2-transitive closure of a DSM, a dependency between two files means that these two files could be indirectly connected to each other through at most one other source file. More details of the calculation of n-transitive closure and Propagation Cost can be found in [24, 26]. The maximum value is 100%, meaning that every element is connected (directly or transitively) to another element in the system. Therefore, a lower value for Propagation Cost indicates greater independence among the modules.

Note that Decoupling Level and Propagation Cost measure the modular structure of a system from opposite perspective and thus, complement each other. Due to the nature of these measurements, they tend to be negatively correlated. However, according to previous research [23], Decoupling Level is more sensitive than Propagation Cost in terms of consistently capturing maintenance difficulties encountered in a system. In order to be comprehensive, we used both metrics to the measure the modular structure of the two case study projects.

Step 3: Modularity Violation Analysis

As discussed earlier, there are three levels of modularity violations in cyber-physical systems: 1) software-software level; 2) software-hardware level; and 3) hardware-hardware level. The identification of modularity violations in a cyber-physical system is challenging because maintenance operations on hardware components are not as comprehensively recorded as in traditional software systems. We hypothesized that software components that are changed frequently due to hardware related concepts are indicative of potential modularity violations. As such, the third level of modularity violation is not the focus of this study. This study focuses on the first two levels of modularity violations by analyzing the software side changes that are potentially triggered by hardware-related concepts.

While the previously described techniques for identifying potential modularity violations that use frequent co-changes can be applied to the case study projects, they do not tell us the cause the modularity violation. In pure software systems, we can assume that any modularity violations are software-software level violations. That is not the case for cyber-physical systems. In this study, we hypothesized that we could determine which modularity violations are hardware related by analyzing source files and change messages. For example, if two supposedly independent software modules frequently change together while referencing the same hardware device, it suggests modularity violation induced by that hardware device.

To evaluate the feasibility of this approach, we extracted project specific hardware-related concepts by manually reviewing the project documentation and wiki pages. Next, we performed a co-change analysis on the project source files and screened for sets of files with high co-change rates as likely sources of modularity violations. Of those with high co-change rates, we identified files that were related to project-specific hardware concepts. We constructed representative sets of hardware-related keywords for OpenWrt and MD PnP. We searched change messages and

source file content and identified those that contained the hardware-related keywords. This enabled us to identify software side changes that were triggered by a hardware-related concepts. We then analyzed the change frequency of files impacted by hardware related concepts relative to the project as a whole.

4 | RESULTS

In this section, we discuss the case study results for OpenWrt and MD PnP in three parts. First, we describe the modular structure of these two projects as measured using Decoupling Level and Propagation Cost. Second, we describe the results of the co-change analysis among project source files and identify potential hardware linkages. Third, we describe the relative change proneness of those source files associated with hardware-related keywords relative to the change proneness of all files in each project.

4.1 | Measuring the Modular Structure

First, we analyze how well the two case study projects are modularized as compared to the 129 pure software systems studied in the past [23].

Figure 1a shows an overview of the modular structure identified in OpenWrt, represented in the form of a DSM [16]. This DSM is a square matrix showing the structural dependencies among source files in OpenWrt. The rows and columns represent source files, and a black dot represents a structural dependency from the row to the column. Figure 1a is just a qualitative overview of the interdependencies among the source files in OpenWrt, without capturing the details, such as file names and dependency types. Similarly, Figure 2a shows the overview of the modular structure identified in MD PnP. Based on the size of the shaded areas, OpenWrt shows less coupling than MD PnP, and thus has better modular structure.

To get a quantitative understanding of how modularized these two projects are, we calculated two metrics: 1) the decoupling level (DL) and 2) the propagation cost (PC), based on the DSM.

The Decoupling Level values for OpenWrt and MD PnP are 0.78 and 0.68 respectively. Figure 3 shows the cumulative distribution of the Decoupling Level value for 129 projects (108 open source projects and 21 industrial projects) from our prior study [23]. The red stars in Figure 3 mark the decoupling levels of OpenWrt and MD PnP compared to these 129 projects. The data show that the Decoupling Level values of OpenWrt and MD PnP are higher than about 85% and 70% respectively of the 129 software projects. The implication is that both OpenWrt and MD PnP are better decoupled into mutually independent modules when compared to the majority of the previously studied pure software projects.

Propagation Cost yields complementary results as the analysis shows that the Propagation Cost values for OpenWrt and MD PnP are only 1.4% and 6.5% respectively. The Propagation Cost, again, suggests that both projects are well modularized and decoupled.

These results indicate that the software components in both OpenWrt and MD PnP are more modularized compared to traditional software systems. We conjecture that this is because the hardware components in a cyber-physical system increase the overall complexity of the system relative to pure software systems. Increasing the modularity of the software may be a way to cope with that complexity. Furthermore, by talking to IoT experts, we realized that the first priority of the software components is to stay concise to reach the quality goals of energy and cost efficiency in IoT systems.

4.2 | Identifying Software-Software Level Modularity Violations

We applied the approach described in Section 3 to identify software-software level modularity violations [23, 6, 7]. First, we calculated the evolutionary coupling among source files in OpenWrt and MD PnP in order to identify the modularity violations. The evolutionary coupling between two source files is the number of times the two files change together in the revision history. The overview of the evolutionary coupling among files in OpenWrt and MD PnP is illustrated in Figure 1b and Figure 2b respectively. Similar to the structural dependencies shown on the left side of each figure, the rows and columns represent the source files in a project. Each red dot indicates the evolutionary coupling between the file on the row and the file on the column. In OpenWrt, we observe that the weight of most of the evolutionary coupling is below 4, indicating software source files do not change together frequently. In particular, this is tracked from more than a decade of revision history (between 2004 and 2017) of OpenWrt. The evolutionary coupling is much lower compared to other traditional software systems we studied in the past [23]. In comparison, the co-changes among source files in MD PnP are more frequent [27, 28].

Second, we computed the discrepancy between the structural dependencies (left side) and the evolutionary coupling (right side) in Figures 1 and 2. Figure 4 shows the Design Structure Matrix (DSM) formed by files involved in the software modularity violations in OpenWrt. The rows and columns represent the 23 source files that are involved in software level modularity violations. The 23 source files are arranged in the same order on both the columns and the rows. Each cell describes two possible types of relationship between the file on the row and the file on the column: 1) the structural dependencies embedded in the source code; and 2) the evolutionary coupling reflected in the system's evolution history. The structural dependencies are described in text, such as "Call" in cell[11,12], which indicates that the source file on row 11, *yesno.c*, calls some function(s) defined in the source file on column 12, *util.c*. The evolutionary couplings are captured by the numbers in each cell. They indicate the frequency of co-change between the file on the row and the file on the column. For example, *cell[10, 6]* says "5", indicating file 10, *mach-nbg6716.c*, and file 6, *mach-wlr8100.c* changed together 5 times in the revision history. Cells that indicate evolutionary coupling but have no corresponding structural dependencies are highlighted in red. As shown in Figure 4, the 23 source files exhibit frequent evolutionary coupling but are largely structurally independent from each other. Thus, these 23 source files are potential modularity violations, which suggests shared but latent assumptions among them.

Further manual inspection of the source files involved in modularity violations indicated that hardware-related concepts contributed to these software level modularity violations. For example, the naming conventions of the source files imply that these files share hardware-related concepts. As highlighted in Figure 4, from row 1 to row 10, the naming of the files all contain "mips", which is a typical microprocessor architecture for cyber-physical systems. In addition, row 20 to 23 shows that the file names all contain "firmware", which is held in non-volatile memory devices, such as ROM or flash memory. Based on these observations, we conjecture that the hardware concepts are actually the causal factors of these software-software level modularity violations in OpenWRT.

In MD PnP, we identified 11 instances of modularity violations, involving a total of 120 files. Since the co-change frequency in MDPnP is higher, we filtered out co-changes that happened less than 6 times to focus on the most intense co-changes in the system. Figure 5 shows the DSM of one of the 11 instances of modularity violations in MD PnP. The rows and columns represent the 15 involved source files, arranged in the same order. Similar to Figure 4, we highlighted the background color of cells in red when there is evolutionary coupling but no structural dependencies between source files. We can observe that these 15 files are largely independent from each other, but frequently change together in the evolutionary history. For example, cell[15,4] says 13, indicating that the file on row 15, *DemoCapnostream20.java*, and the file on column 4, *DemoN595.java*, changed together 13 times! Similar to the modularity violations in OpenWrt, these software source files are associated with hardware concepts. It should be noted that in the name space each file

path contains the keyword “device", as highlighted in the brown rectangle on the left side. Furthermore, there are terms related to medical devices in the specific file names. For example, the name of the file on row 5 contains “Oximeter," which is a specific medical device used to measure oxygen level.

4.3 | Identifying Hardware-Software Level Modularity Violations

Reasoning based on the above observations, we infer that hardware related concepts could also be the contributing factors for software-hardware modularity violations. That is, software components that change frequently due to hardware related concepts. As an example, we found this comment when developers changed a software entity in OpenWrt: *“set chip type directly in ar8216_id_chip"*, where chip is obviously a hardware term. We assume it is less likely the other way around: software concepts contribute to hardware changes, because it is, in most cases, more affordable to change software components.

We manually extracted 16 keywords from project documents from OpenWrt. The details are shown below in Table 2 containing keywords such as “radio”, “WiFi”, “zigbee”, etc. By matching these manually extracted keywords in developers’ change comments, which are usually used to explain why they made the change, we identified 70 source files in OpenWrt (of the 1052 total files) that are potentially involved in software-hardware modularity violations.

Similarly, we manually extracted 12 keywords from project documents in MDPnP. The details are shown below in Table 3 containing keywords such as “oximeter”, “xray”, “ultrasound”, etc. By matching these manually extracted keywords in developers’ change comments, we identified 156 source files in MD PnP (of the 831 total changed files) that are potentially involved in software-hardware modularity violations.

To further understand the significance of hardware-related concept as change contributors, we calculated the change-proneness ranks of the 70 and 156 files involved in hardware-software level modularity violations in OpenWrt and MD PnP and compared these with the change-proneness ranks of all source files in these two projects.

The comparison results for OpenWrt are shown in Figure 6. The x-axis represents the change frequency threshold of source files. For example, a threshold of 20 means source files changed at least 20 times in the project history. The y-axis represents the cumulative probability of a file residing above the respective change-prone threshold on the x-axis. In Figure 6, the upper line shows the cumulative probability of change frequency for those files identified as related to hardware-software modularity violations. The lower line shows the cumulative probability for all files in the project. For instance, the data shows that 19% of files involved in hardware-software modularity violations change 10 or more times; while only 5% of the total set of source files (among the total 657 files changed between 2004 and 2017) change 10 or more times. The data show that these 70 files are at least twice as likely to change compared to average files. This suggests that hardware related concepts could be the main contributing factor to changes made to software modules in OpenWRT.

We can make similar observations for MDPnP as shown in Figure 7, especially for change frequencies above 5. In particular, 20% of the 156 hardware-related modularity violation files changed 20 or more times compared to only 6% of average files. This indicates that files involved in hardware-related modularity violations are more likely to change frequently relative to average files. Therefore, we also infer from MD PnP that hardware-related concepts could be the main trigger for changes to software components, and thus imply latent software-hardware modularity violations that merit attention from stakeholders.

5 | DISCUSSION

In this section, we will first summarize the key observations achieved in this study. Next, we will discuss the limitations of this paper and the potential future research directions.

1. Both projects studied exhibited high levels of modularity in their software architecture (Figure 3). Despite this modularity, there were significant numbers of potential modularity violations (Section 4.2 and 4.3).
2. Most of the modularity violations seemed to involve to hardware or other non-software domain concepts (Figure 4, Figure 5)
3. Files associated with hardware-software violations had a higher probability to change frequently (Figure 6 and Figure 7)
4. Hardware and other non-software domain concepts appear to constitute shared assumptions among the cyber and physical layers that drive these hardware-software modularity violations (Section 4.2 and 4.3)

While the analysis of two projects is not sufficient to draw general conclusions, it is sufficient to establish the feasibility of the approach. Both projects studied seemed to be intentionally highly modular to allow the relatively easy swap out and integration of hardware devices. Even so, there seemed to be latent assumptions within the architecture about how these devices would behave. Furthermore, MD PnP has the additional complication of considering medical practices and human biology. It appeared that many of the apparent violations of the modular architecture were driven by these latent assumptions about the physical layer. Because of this asymmetry, there seemed to be sufficient information in the software version control systems to detect and measure at least a subset of hardware-software modularity violations using techniques derived from software engineering. That opens the door to a more comprehensive and longer-term analysis of the drivers of modularity violations in cyber-physical systems.

With that in mind it is important to acknowledge the limitations of this study. First, we treated source files as fine-grained modules. This is appropriate from the perspective of a low-level developer. However, we acknowledge that other stakeholders may view the system from different granularity levels. For example, product managers usually view modules as cohesive functional components to deliver the product value and competitiveness. This results in a very different definition of a module. From that perspective, a module may contain many source files, and the project owner may not be concerned with modularity violations among source files within a given module. In our future work, we plan to explore and investigate different criteria to decompose a system into modules and analyze the modularity violations across these alternative decompositions .

Second, we were limited by the need to manually extract and search for domain concepts and keywords. Not only is this approach too time consuming to scale to a larger study, but also it is almost certain that we missed many subtleties. In short, we were able to detect the most obvious cases, but each cyber-physical system is going to have an application specific network of concepts and assumptions. If our hypothesis is correct, it is this network of concepts that is going to drive many of the modularity violations in a cyber-physical system. Consequently, it would be necessary to learn a new network for each system one wishes to analyze. To make this problem tractable, in future work, we plan to apply text analytics and other machine learning techniques to extract this network of domain concepts from each project's documentation and source files.

Finally, a level of modularity violation that was not addressed at all in this study is the hardware-to-hardware modularity violation. This is largely due to the lack of systematic, centralized maintenance records on the hardware side. Unlike the software components of a cyber-physical system, the revision history of the hardware parts are largely unavailable. This paper leverages software data to infer hardware related issues, but this approach cannot detect all of

the pure hardware level modularity violations. We will leave this challenge to future work.

6 | CONCLUSIONS AND FUTURE WORK

In this paper, we conducted a case study on two real-world cyber-physical systems, OpenWrt and MD PnP, to investigate the feasibility of identifying and measuring modularity violations involving hardware components. To the best of our knowledge, this is the first work to address this problem in the area of cyber-physical systems. What we found was that while the software architectures of OpenWrt and MD PnP are well-modularized, there were a number of software modularity violations that appear to involve hardware-related concepts. Furthermore, we found that software components related to hardware were much more likely to change frequently. Combined, these findings suggest hardware components are a source of latent modularity violations in OpenWrt and MD PnP. Our study demonstrated the feasibility of adapting methods to identify and measure modularity violations from pure software systems to cyber-physical systems.

In future work, we plan to address some previously described limitations of this study. First, we intend to explore different criteria for decomposing a system into modules. In this paper, we defined a module as a source file. However, as noted previously, the definition of a module is a matter of perspective. We plan to investigate modularity violations involving modules at different levels of resolution such as packages. Second, we manually extracted keywords by reviewing project revision logs to identify hardware-related concepts. This approach does not scale as a new set of keywords would need to be manually extracted for each cyber-physical system one wanted to analyze using the approach described in this paper. In the future, we plan to leverage natural language processing techniques to automatically extract hardware-related concepts from the repositories of cyber-physical systems. Finally, we plan to evaluate our approach to detect hardware-related modularity violations on a broader set of cyber-physical systems. Moreover, we believe that similar approach can be easily extended to identify modularity violations in any critical embedded systems. We plan to widen the scope of this study beyond cyber-physical systems in future work.

ACKNOWLEDGMENTS

This material is based upon work supported, in whole or in part, by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract HQ0034-13-D-0004. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Department of Defense. This work was also supported in part by the National Science Foundation of the US under grants CCF-1823074.

REFERENCES

[1] Rajkumar R, Lee I, Sha L, Stankovic J. Cyber-physical systems: the next computing revolution. In: Design Automation Conference IEEE; 2010. p. 731–736.

[2] Schach SR. Classical and Object-Oriented Software Engineering W/Uml and C++. McGraw-Hill, Inc.; 1998.

[3] Kruchten PB. The 4+ 1 view model of architecture. IEEE software 1995;12(6):42–50.

[4] Allen EB, Khoshgoftaar TM, Chen Y. Measuring coupling and cohesion of software modules: an information-theory approach. In: Proceedings Seventh International Software Metrics Symposium IEEE; 2001. p. 124–134.

[5] De Souza C, Froehlich J, Dourish P. Seeking the source: software source code as a social and technical artifact. In: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work ACM; 2005. p. 197–206.

[6] Wong S, Cai Y, Kim M, Dalton M. Detecting software modularity violations. In: 2011 33rd International Conference on Software Engineering (ICSE) IEEE; 2011. p. 411–420.

[7] Xiao L, Cai Y, Kazman R. Titan: A toolset that connects software architecture with quality analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering ACM; 2014. p. 763–766.

[8] Schwanke R, Xiao L, Cai Y. Measuring architecture quality by structure plus history analysis. In: 2013 35th International Conference on Software Engineering (ICSE) IEEE; 2013. p. 891–900.

[9] Xiao L, Cai Y, Kazman R, Mo R, Feng Q. Identifying and quantifying architectural debt. In: Proceedings of the 38th International Conference on Software Engineering ACM; 2016. p. 488–498.

[10] Mo R, Cai Y, Kazman R, Xiao L. Hotspot patterns: The formal definition and automatic detection of architecture smells. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture IEEE; 2015. p. 51–60.

[11] Kernschmidt K, Vogel-Heuser B. An interdisciplinary SysML based modeling approach for analyzing change influences in production plants to support the engineering. In: 2013 IEEE International Conference on Automation Science and Engineering (CASE) IEEE; 2013. p. 1113–1118.

[12] Derler P, Lee EA, Vincentelli AS. Modeling cyber–physical systems. Proceedings of the IEEE 2011;100(1):13–28.

[13] Lee EA. The past, present and future of cyber-physical systems: A focus on models. Sensors 2015;15(3):4837–4869.

[14] González-Nalda P, Etxeberria-Agiriano I, Calvo I. Flexible, modular, standard, free and affordable model for CPS control applied to mobile robotics. In: 2015 10th Iberian Conference on Information Systems and Technologies (CISTI) IEEE; 2015. p. 1–6.

[15] Jantunen E, Zurutuza U, Ferreira LL, Varga P. Optimising maintenance: What are the expectations for cyber physical systems. In: 2016 3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC) IEEE; 2016. p. 53–58.

[16] Baldwin C, Clark K, Design rules: The power of modularity (Vol. 1) MIT press; 2000.

[17] Baheti R, Gill H. Cyber-physical systems. The Impact of Control Technology, T. Samad and AM Annaswamy. IEEE Control Systems Society 2011;1.

[18] Malavolta I, Muccini H, Sharaf M. A Preliminary Study on Architecting Cyber-Physical Systems. In: ECSA Workshops; 2015. p. 20–1.

[19] Rajhans A, Cheng SW, Schmerl B, Garlan D, Krogh BH, Agbi C, et al. An architectural approach to the design and analysis of cyber-physical systems. Electronic Communications of the EASST 2009;21.

[20] Sinha K, et al. Structural complexity and its implications for design of cyber-physical systems. PhD thesis, Massachusetts Institute of Technology; 2014.

[21] Cristalli C, Boria S, Massa D, Lattanzi L, Concettoni E. A Cyber-Physical System approach for the design of a modular Smart Robotic Cell. In: IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society IEEE; 2016. p. 4845–4850.

[22] Lee EA. Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC) IEEE; 2008. p. 363–369.

[23] Mo R, Cai Y, Kazman R, Xiao L, Feng Q. Decoupling level: a new metric for architectural maintenance complexity. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) IEEE; 2016. p. 499–510.

[24] MacCormack A, Rusnak J, Baldwin CY. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science 2006;52(7):1015–1030.

[25] Wong S, Cai Y, Valetto G, Simeonov G, Sethi K. Design rule hierarchies and parallelism in software development tasks. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering IEEE Computer Society; 2009. p. 197–208.

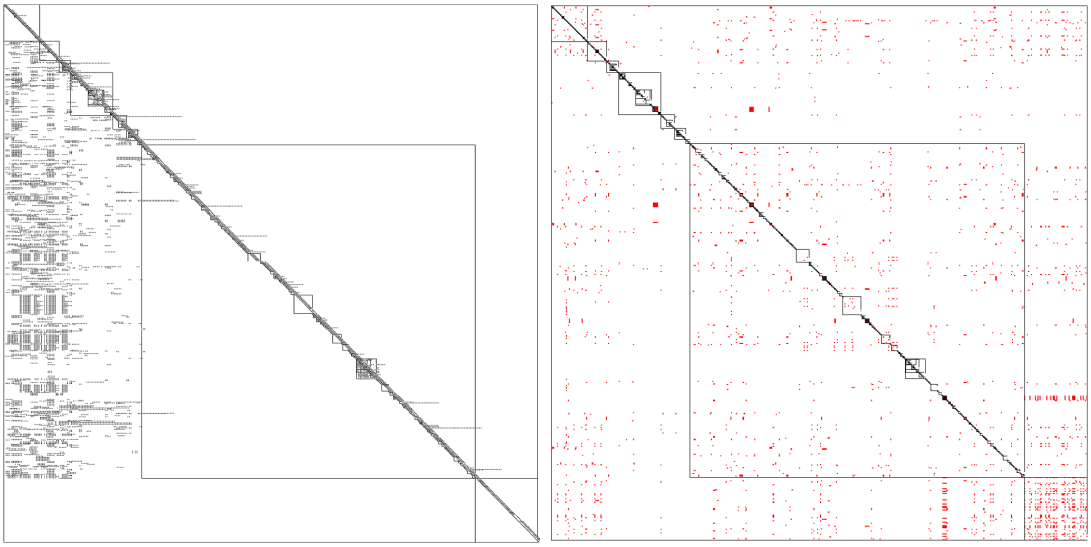
[26] Fischer MJ, Meyer AR. Boolean matrix multiplication and transitive closure. In: 12th Annual Symposium on Switching and Automata Theory (swat 1971) IEEE; 1971. p. 129–131.

[27] Xiao L, Cai Y, Kazman R. Design rule spaces: A new form of architecture insight. In: Proceedings of the 36th International Conference on Software Engineering ACM; 2014. p. 967–977.

[28] Cai Y, Xiao L, Kazman R, Mo R, Feng Q. Design rule spaces: A new model for representing and analyzing software architecture. IEEE Transactions on Software Engineering 2018;.

TABLE 1 Case Study Subject

Project Name:	OpenWrt	MD PnP
# of Files:	1052	866
# of Methods:	6061	7872
# of Lines of Code:	163,114	73,616
# of Developers:	137	12
# of Revision Records:	38099	1605
Range of History:	2004 to 2017	2013 to 2017

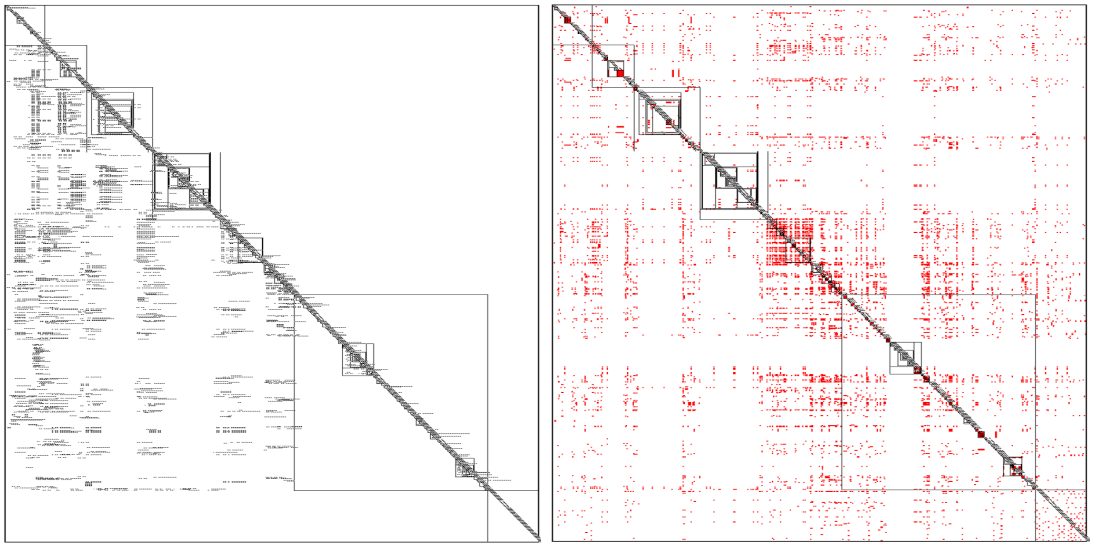


(a) Modular structure (b) Evolutionary coupling

FIGURE 1 OpenWrt modular structure and evolutionary coupling

TABLE 2 Domain/Hardware keywords in OpenWrt

"radio", "WiFi", "zigbee", "btle", "mips", "ramips", "mtd", "broadcom",
"routerboot","router", "firmware", "bluetooth", "energy", "power", "soc",



(a) Modular structure (b) Evolutionary coupling

FIGURE 2 MD PnP modular structure and evolutionary coupling

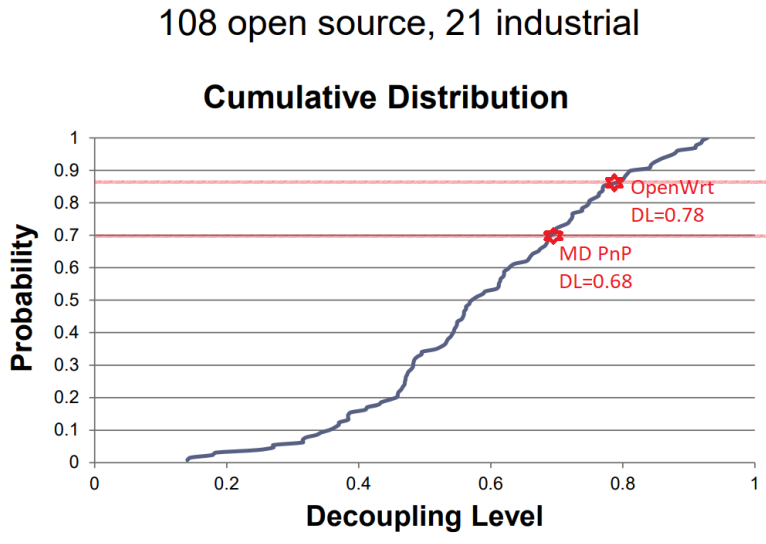


FIGURE 3 OpenWrt and MD PnP Decoupling Level Percentile among 129 Systems

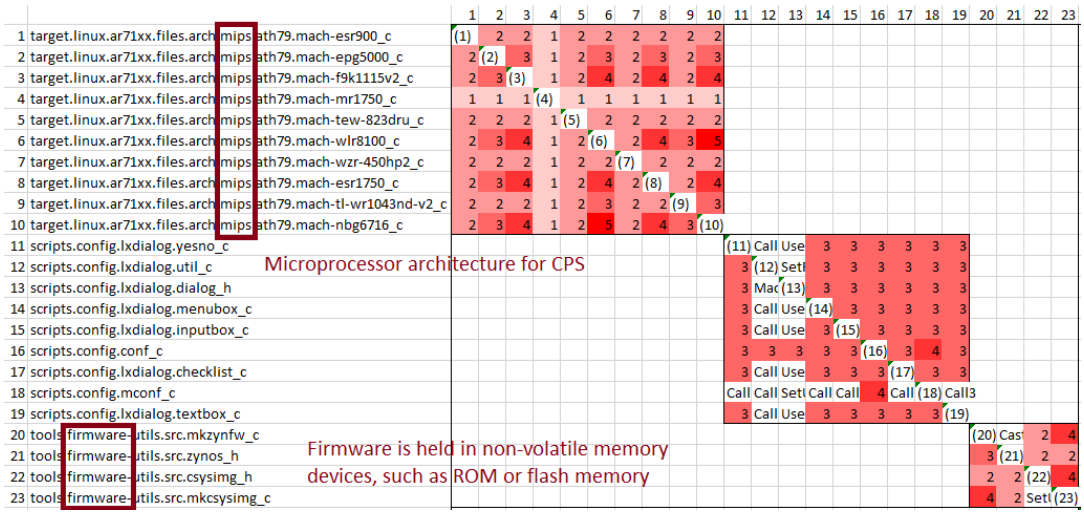


FIGURE 4 Software-software modularity violations in OpenWrt

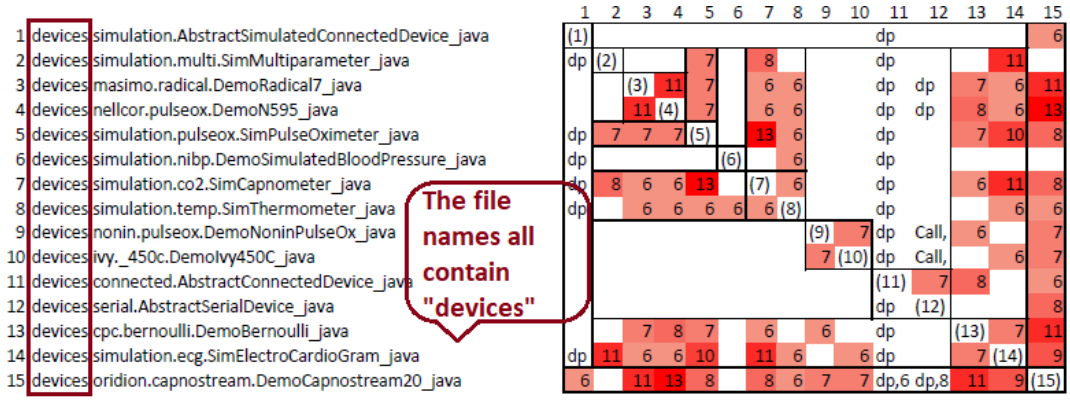


FIGURE 5 Software-software modularity violations in MD PnP

TABLE 3 Domain/Hardware keywords in MDPnP

"ventilator", "beaglebone", "pump", "electrocardiogram", "xray", "anesthesia",
"analgesia", "oximeter", "oxygen", "respiration", "telemetry", "ultrasound", "capnograph",

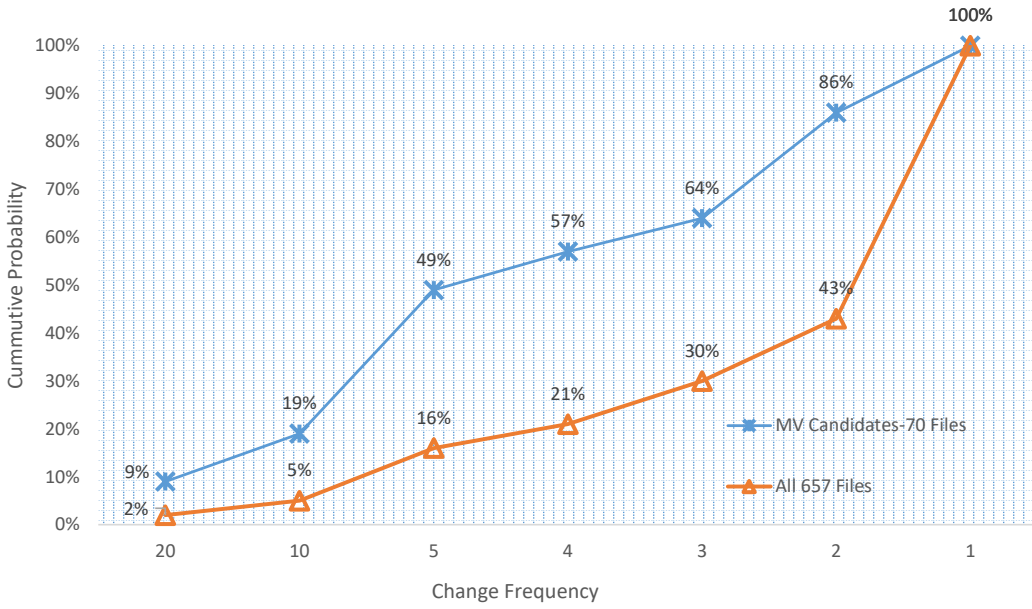


FIGURE 6 Hardware-related and general change-proneness for OpenWrt

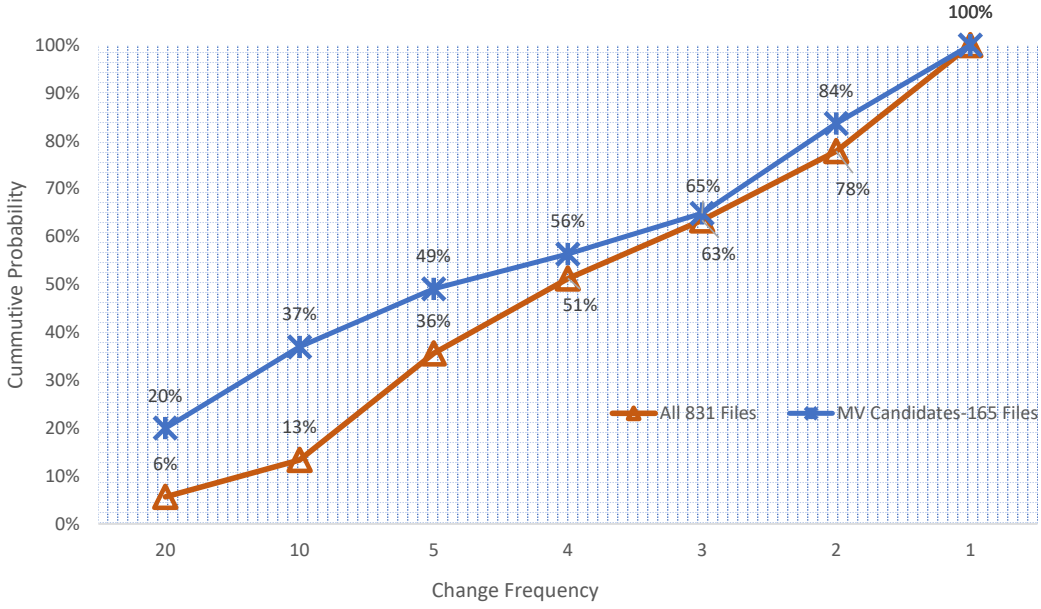


FIGURE 7 Hardware-related and general change-proneness for MD PnP