Understanding Source Code Variability in Cloned Android Families: an Empirical Study on 75 Families

Anas Shatnawi, Tewfik Ziadi, Mohamed Yassin Mohamadi

Understanding Source Code Variability in Cloned Android Families: an Empirical Study on 75 Families

Anas Shantawi, Tewfik Ziadi, Mohamed Yassin Mohamadi
Sorbonne University, CNRS, LIP6, F-75005 Paris, France
anas.shatnawi@lip6.fr, tewfik.ziadi@lip6.fr, em_mohamadi@esi.dz

***Abstract*—Software developers rely on the clone-and-own approach to rapidly develop software product variants (PVs) that meet variability in market needs. To improve the comprehension of how PVs are evolved and varied, we analyze the clone-and-own practices applied by developers of these PVs. We perform an empirical study on 75 android families to gain insights about observable phenomena related to the commonality and variability between the source code of PVs of these families. In particular, we study three research questions to identify the commonality and variability related to the organization of source code files, cloning Java methods, and configuration parameters of AndroidManifest.xml files. The results show that cloning packages, Java files and Java methods is a common practice used by developers of all android families. Maintainers should put efforts for managing the diverse implementations (bodies) of the modified cloned methods and it is essential to consider the commonality and variability of configuration parameters.**

***Index Terms*—clone-and-own reuse, source code, commonality and variability, android families, SPL**

I. Introduction

Mobile applications should be in variants to follow differences in market needs and users' demands [1]. The use of the *clone-and-own* approach becomes a common practice used by developers to rapidly customize existing applications to meet variability in market needs [2] [3] [4].

Businge et al. [1] define an android family as a group of PVs (i.e., android applications) developed and maintained in GitHub in terms of a mainline and its forks that are published in Google Play as different products. As results of their study, Businge et al. [1] identify 88 android families based on the analysis of the clone-and-own in fork-based development in GitHub. These families are implemented using 88 mainlines and 127 forks.

As a practical example, among 1200 forks of the *bitcoin-wallet/bitcoin-wallet*[1] git repository of the Bitcoin Wallet[2], Businge et al. [1] identified that 10 forks are customized to produce 10 distinguishable android applications offered in Google Play (*Globaltoken Wallet*[3], *Europecoin Wallet*[4], *FairCoin Wallet*[5], etc.) [1]. Consequently, the mainline together with the 10 forks constitute a family of 11 PVs.

[1]https://github.com/bitcoin-wallet/bitcoin-wallet
[2]play.google.com/store/apps/details?id=de.schildbach.wallet&hl
[3]play.google.com/store/apps/details?id=org.globaltoken.wallet&hl
[4]play.google.com/store/apps/details?id=madzebra.erc.wallet&hl
[5]play.google.com/store/apps/details?id=de.schildbach.wallet.faircoin&hl

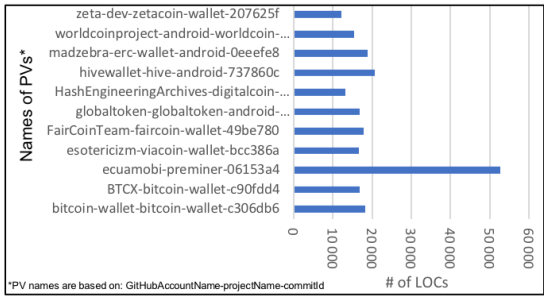

Figure 1. Number of LOCs of 11 PVs of the Bitcoin Wallet android families

Understanding the source code commonality and variability inside all these families improve the comprehension of how their PVs are evolved through the different forks. In addition, it can radically change the way how these PVs are maintained and managed by reengineered them to a systematic reuse framework such as Software Product Line Engineering (SPLE) [5]. SPLE allows us to maintain a set of PVs in parallel by managing the commonality and the variability between PVs. Indeed, many recent work have been proposed in the last years to extract SPL from cloned PVs such as [4] [6] [7] [8] [9].

For android families, understanding source code variability concerns three main dimensions: (i) the organization of the source code files through the PVs, (ii) the Java source code variability that concerns the implementation of the different PVs, (iii) the configuration parameters that define important information about android application variants to the android build tools, the android operating system, and Google Play.

As a motivation example, we identify the number of LOCs composing the source code of each PV of the Bitcoin Wallet android family in Figure 1. The diversity in the numbers of LOCs implies the existence of variability between the source code of these 11 PVs. However, this does not provide a precise knowledge about how this variability is realized compared to the three mentioned dimensions. To do so, we need to understand the commonality and variability of source code elements in terms of what packages, Java files, Java methods and configuration parameters are added, deleted, or modified.

In this paper, we perform an empirical study to gain insights about observable phenomena of the commonality and variability across the source code of PVs of 75 android families from

the results of Businge et al. [1]. The goal of such phenomena is to support maintainers to understand the clone practices in the android families which helps them to take maintenance decisions related to these PVs. We select the 75 families due to the availability of their source code on GitHub repositories and their PVs are published on Google Play. Our empirical study is guided based on three Research Questions (RQs) that are related to three dimensions discussed above:

- **RQ1: what is the commonality and variability in the organization of source code files of PVs?** We identify the impact of clone practices on the source code organization including adding, removing and modifying Java files and packages. To study this, we rely on similarities and differences in packages and Java files composing PVs of the 75 android families.
- **RQ2: what is the commonality and variability related to the cloned Java methods across PVs?** We study the commonality and variability corresponding to the cloned Java methods across source code files of PVs in terms of adding, removing and modifying methods. We identity exact cloned methods and modified cloned methods based on analyzing the similarities and differences between headers and bodies of these methods in different PVs of the 75 android families.
- **RQ3: what is the commonality and variability in the configuration parameters of android applications of PVs?** We investigate the influence of clone practices in the variability of configuration parameters (e.g., access permissions, activities...) that characterize the android applications corresponding to the PVs. We identify the configuration variability by analyzing the *AndroidManifest.xml* files of PVs to identify common and variable configuration parameters.

The results show that developers rename some packages of cloned Java files in different PVs of 90.66% android families (the Oracele's name convention pattern). Also, we identify that cloning exact method is a common practice used by developers of all families and developers modify the body of cloned methods but with low percentage in many families. Configuration parameters of PVs of families are varied and it is essential to consider their commonality and variability while maintaining these families.

The contributions of this paper are:

- To the best of our knowledge, our study is the first empirical study that performs a deeply analysis of the source code variability of PVs of 75 android families.
- We define seven software metrics to measure the commonality and variability in PVs.
- We propose a methodology for understanding code variability at different variability dimensions (organization, implementation details and configuration parameters).
- We develop a parameterized tool that realizes the proposed methodology for understand code variability and it works for any java-based family.

The rest of this paper is organized as follows. We describe the dataset of 75 android families and their PVs in Section II. Then, we discuss our methodology to answer research questions in Section III. Next, we discuss the results of answering the three research questions in Section IV. We discuss the usefulness of the results of our empirical study and the threats to validly in Section V. Related works are discussed in Section VI. We conclude our empirical study and draw future directions in Section VII.

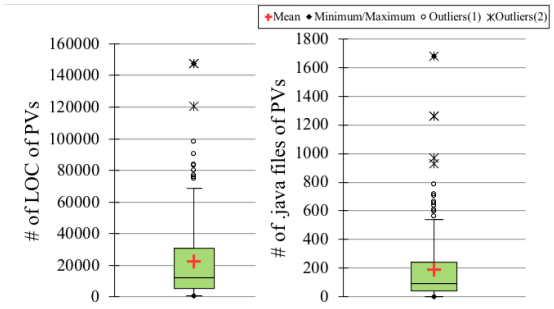

Figure 2. The size of PVs of android families

II. Dataset of Android Families

Businge et al. [1] identified 88 android families in terms of PVs developed and maintained together based on the clone-and-own approach in fork-based development in GitHub repositories. Each family is a composition of one mainline PV and its forks represent other PVs. To identify the 88 families, they developed a search methodology to guarantee that: (i) each family consists of at least two PVs (the mainline and at least one active fork), and (ii) each PV has a unique corresponding application offered in the Google Play. For details of their identification methodology please refer to [1].

In our study, we include 75 real android families in which all of their PVs are implemented using Java and their corresponding GitHub repositories are still accessible as public. We exclude 13 families due to that: (i) the PVs are implemented using non-Java languages (e.g., C++ and Kotlin), (ii) the GitHub repositories of PVs are not found, (iii) the PVs do not have any commonality in their source code files at all levels of abstraction and (iv) PVs of one family are exactly identical products. We consider as input for our study the source code of the last commit[6] of each PVs of the 75 families.

Figure 2 presents the numbers of Lines Of Code (LOCs) and Java files of PVs[7] to better understand the nature of the families used in our study. We have a diversity in terms of the number of PVs composing each family (2, 3, 4, 5, 6, 11 PVs) and the number of LOCs and Java files of these PVs. We have families composed of PVs including very similar numbers of LOCs and Java files and families composed of PVs having varied numbers of LOCs and Java files. Moreover, Businge et al. [1] reported the diversity of developers of PVs of the same

[6]We downloaded last commits of product variants in March 2019.

[7]Software metrics are calculated using the Understand tool, and the statistics are identified using the XLSTAT tool.

families in which 74% of forks (94 out of the 127 forks) do not have any mutual developers with their mainlines.

III. Methodology for Answering Research Questions

A. General Methodology to Identify Commonality and Variability

To understand variability and to identify the statistical data describing the commonality and variability across PVs of each android family, we develop a process presented in Figure 3. This process extends the BUT4Reuse (Bottom-Up-Technologies for Reuse) framework [9] to analyze commonality and variability corresponding to each RQ. We select to extend BUT4Reuse because: (i) it is generic that allows to use it to analyze different software artifact types (e.g., source code and XML configuration files), and (ii) it is an open-source extensible framework that allows us to (re)implement new algorithms, visualization capabilities and metric extractors.

To compare PVs, BUT4reuse is based on three main steps: identifying elements, identifying blocks and identifying variability metrics.

(1) Identifying elements. It aims to identify the set of *elements* from each PV. BUT4Reuse allows us to define the granularity of the elements to be identified corresponding to each RQ. For the same PV, we can select from coarse to fine granularity (e.g., package level versus statement level for source code). To identify the elements corresponding to each RQ, we extend BUT4Reuse by integrating two parsers: Java and XML parsers. The Java parser aims to break the source code of PVs into packages and Java files for RQ1, and methods for RQ2. It relies on the Abstract Syntax Tree (AST) of JDT.The XML parser aims to identify a set of XML tags corresponding to configuration parameters based on the analysis of *AndroidManifest.xml* files of PVs.

(2) Identifying blocks. It aims to identify the commonality in terms of *interdependent similar* elements across PVs and the variability in terms of *dissimilar* elements across PVs. Based on a user-defined similarity metric between any pair of elements, a set of elements is considered as interdependent if and only if they belong to exactly the same PVs. We perform this identification based on reusing algorithms implemented in BUT4Reuse which automatically identify sets of elements that correspond to the distinguishable features from PVs. These sets of elements are named *Blocks*. Each block contains a set of interdependent elements belonging to exactly the same PVs.

We categorize the identified blocks into *complete common*, *partial common* and *product exclusive* blocks based on the number of PVs sharing the interdependent elements of blocks, as presented in Figure 4. The *complete common block* includes elements that are interdependent in all PVs of the android family. The *partial common blocks* contains elements that are interdependent in subsets of PVs but not all of them. The *product exclusive blocks* are composed of elements that are specific to individual PVs and does not have any interdependent relation with other elements in other PVs.

(3) Identifying variability metrics. We measure the commonality and variability by performing statistics that describe behaviour of PVs based on the complete common, partial common and product exclusive blocks. We define a set of commonality and variability metrics to measure four characteristics: the complete commonality in all PVs, the partial commonality in subsets of PVs, the product exclusive variability and the general variability degree.

1) Complete commonality across all PVs: this aims to measure the interdependent code elements in *all* PVs. We define two metrics: (i) percent of complete common block (%CCB) and (ii) percent of complete common elements (%CCE). We measure %CCB based the ratio between the complete common block and the total number of identified blocks ($\frac{\#ofCompleteCommonBlocks}{\#ofIdentifiedBlocks}$) and %CCE based on the ratio between number of elements of common block and number of elements of all identified blocks ($\frac{\#ElementsInCompleteCommonBlock}{\#DistinctElementsInAllPVs}$). %CCB and %CCE values are always situated in [0–1]. The higher values mean that all PVs share more identical code elements and vice versa.

2) Partial commonality in subsets of PVs: this aims to measure the interdependent code elements across a subset of PVs. We define two metrics (i) percent of partial commonality block (%PCB) and (ii) percent of partial commonality elements (%PCE). %PCB is the ratio between the number of partial common blocks and the total number of identified blocks ($\frac{\#ofPartialCommonBlocks}{\#ofIdentifiedBlocks}$), and %PCE is the ratio between the number of elements of partial common blocks and the total number of elements of all identified blocks ($\frac{\#ElementsInPartialCommonBlocks}{\#DistinctElementsInAllPVs}$), Values of %PCB and %PCE are situated in [0–1]. The higher values mean that PVs share more interdependent code elements and vice versa.

3) Product exclusive variability: it measures the code elements that are exclusive only for individual PVs. We define two metrics (i) percent of product exclusive blocks (%PEB) and (ii) percent of product exclusive elements (%PEE). %PEB refers to the ratio between the number of product exclusive blocks and the total number of identified blocks ($\frac{\#ofProductExclusiveBlocks}{\#ofIdentifiedBlocks}$). %PEE is calculated based on the ratio between the number of elements of product exclusive blocks to the total number of elements of all identified blocks ($\frac{\#ElementsInProductExclusiveBlocks}{\#DistinctElementsInAllPVs}$). Values of %PEB and %PEE are situated in [0–1]. The higher values mean that PVs are different from each others.

4) General variability degree (GVD): it aims to measure the diversity between PVs of an android family based on the ratio between the number of identified blocks and the maximum number of possible blocks (

GVD = $\frac{\#ofIdentifiedBlocks}{(2^{\#ofPVs}-1)}$

). The values of GVD are situated in [0–1]. The height value refers to more diversity between PVs. If GVD = 1, then PVs reach the maximum possible diversity level. If GVD = 0, then PVs are identical in their source code.

We identify the seven commonality and variability metrics for each android family for the three RQs. Then, we use the XLSTAT tool to analyze these metrics.

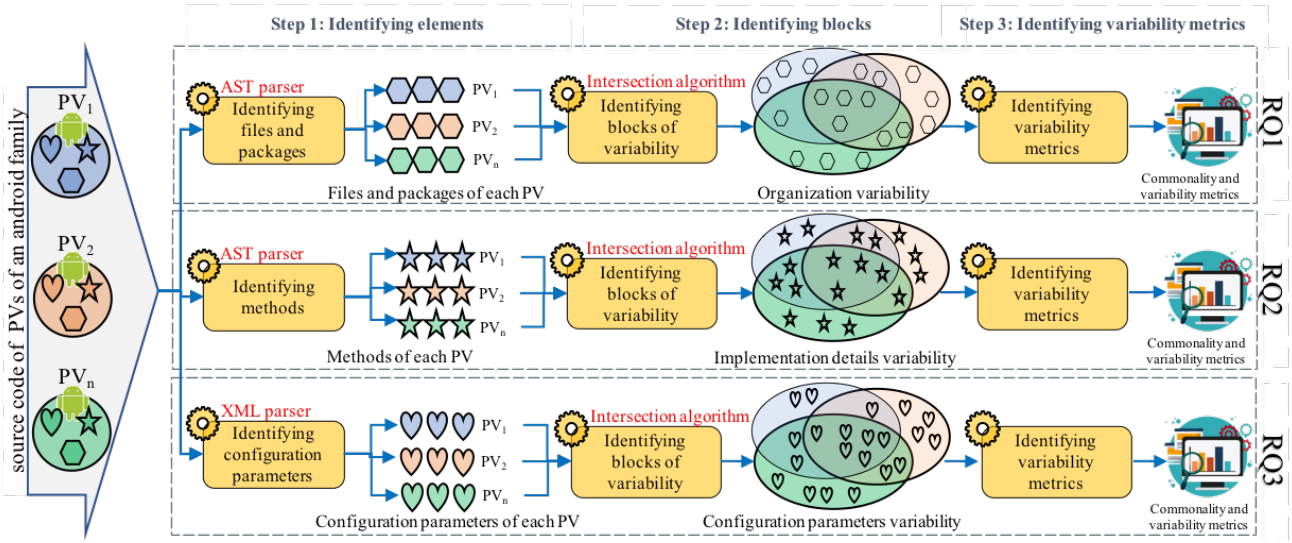

Figure 3. Process to identify variability for answering the research questions

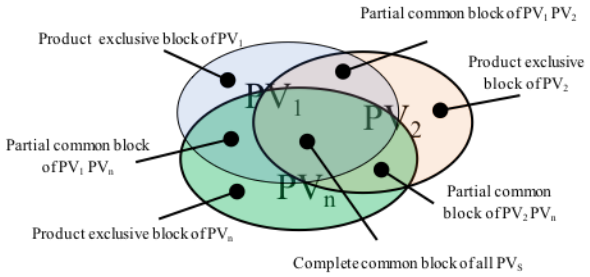

Figure 4. Commonality and variability using blocks of interdependent elements

B. Specific Methodology for Research Questions

In this section, we present the specific methodology used to answer each research question.

1) RQ1: What is the Commonality and Variability in the Organization of Source Code Files of PVs: We compare PVs based on *packages and Java files* constituting these PVs. We consider *packages and Java files* as representative of the source code organization (i.e., an android application is structured in terms of *.java* files organized in packages based on their categories or functionalities). To understand the general behaviour of cloning practices related to adding and deleting packages and Java files across PVs, we study the amount of the commonality and variability of packages and Java files across PVs. We identify if the cloning of PVs causes the resettlement of the same packages and Java files from one package to another across different PVs of the same family based on the analysis of the commonality and variability based on two scenarios. In the first scenario, we consider as interdependent packages and Java files those sharing exactly their full qualified names across PVs (e.g., *com.preminer...BitcoinIntegration.java* should be identical across PVs). In the other scenario, we consider those sharing only the same base-names across PVs (*com.preminer...BitcoinIntegration.java* and *de.schildbach.wallet...BitcoinIntegration.java*[8] are equal Java files organized in different packages). We test the amount of commonality and variability corresponding to packages and Java files of the two scenarios by comparing the values of the seven metrics. If the values of these metrics are different in the two scenarios, then the resettlement of packages and Java files across PVs exists. Otherwise, there are no resettlement of packages and Java files across PVs.

2) RQ2: What is the Commonality and Variability Related to the Cloned Java Methods across PVs: We compare PVs based on *methods* implemented in the cloned Java files across different PVs (i.e., methods of classes having the same base-name across the PVs). We focus on Java methods because they implement the behaviour of functionalities of PVs.

We identify Java method variability based on similarities and differences in their headers and bodies to allow maintainers to understand the types of variability existed in cloned methods. We focus on two types of cloned methods: exact cloned methods and modified cloned methods.

Exact cloned methods are methods having identical headers and bodies across PVs of the same family. This represents type-1 method cloning [10]. We identify these methods by defining the similarity metrics of But4Reuse to detect as interdependent methods only those having *identical* headers and bodies except for white spaces.

Modified cloned methods are methods having the exact identical headers but different bodies across cloned Java files of PVs of the same family. This type of cloned methods includes type-2 and type-3 method cloning [10] because we consider bodies as different if they have at least one variation in their identifiers, literals, types, layout, comments or statements. We give attention to these types of cloned methods because they need a special processing corresponding to merge

[8]Real examples from the Bitcoin Wallet android families of 11 PVs

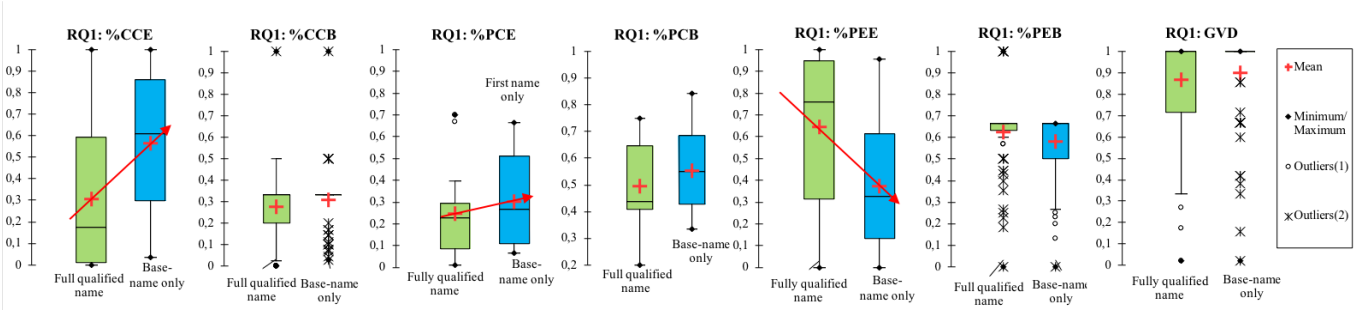

Figure 5. RQ1: the commonality and variability of packages and Java files across PVs

these method bodies while reengineering PVs to SPLs. To identify these methods, we define the similarity metrics of But4Reuse between any pair of methods as follows. Methods are consider as interdependent ones if and only if they have the exact identical headers and different bodies across PVs.

3) RQ3: What is the Variability in the Configuration Parameters of Android Applications of PVs: We compare PVs based on the configuration parameters identified from XML tags in the *AndroidManifest.xml*[9] of each PV. Developers use these XML tags to configure parameters for characterizing important information concerning each PV to the Android operating system, the build tools and the Google Play store. We consider in our study configuration parameters that describe:

- The android application components of PVs including *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. This enables to identify the commonality and the variability in application components of PVs.
- The *permissions* granted to PVs to access private user data or to perform sensitive operations (e.g., *send SMS*, *access camera*, etc.). This allows us to test how developers change permissions of PVs.
- The package names of PVs that are used by the Android build tools to identify the directory of source code files. This enables to check whether PVs source code files are replaced or not while cloning them.
- The hardware and software features defining the *device compatibility* of PVs. This allows us to identify if PVs are customized to be compatible with different devices.

IV. Results for Answering RQs

A. RQ1: What is the Commonality and Variability in the Organization of Source Code Files of PVs?

In Figure 5, we present the box plots of the seven metrics corresponding to the two scenarios of the full qualified name vs the base-name of packages and Java files of PVs in all families. As all of the box plots are tall, families hold quite different patterns compared to their commonality and variability elements in both scenarios of base-name and full qualified name. The results show that we have more commonality (%CCE and %PCE are increased) and less variability (%PPE is decreased) when the base-name is considered compared to the full name for 90.66% (68/75) families. We observe that the reason behind these different values is that developers *rename* some root packages in different PVs while keeping the same sub-packages of cloned Java files. These cloned Java files are identified as interdependent ones across PVs only when we consider their base-names. We interpret this observation by the Oracle's naming conventions[10] as a clone practice where companies should start their package names by reversing their Internet domain name. For example from 11 PVs of the Bitcoin Wallet family, we find that 8 PVs have the same root packages as *de.schildbach.wallet* while they have been changed in 3 PVs to *com.hivewallet*, *com.viacoin.wallet* and *com.preminer*.

For 9.33% (7/75) families, we identify that their PVs have exactly the same commonality and variability level based on the identical values of our commonality and variability metrics in the two scenarios. This means that clone practices in these families do not include any resettlement in the organization of PVs and they do not adhere to the Oracle's naming conventions (they are anti-patterns).

We note that the complete commonality (%CCB and %CCE) is decreased in favour to the partial commonality (%PCB and %PCE) when the number of PVs is increased. Also, the exclusive variability is decreased for families of large number of PVs compared to those of small number of PVs based on %PEB and %PEE. For GVD, it is decreased as well as the number of PVs is increased. The GVD reaches the peak (GVD = 1) for most of families of 2 PVs because the maximum number of possible blocks is 3: common block and 2 exclusive ones for each PV. We find that 2 families have identical packages and Java files in their PVs (have one interdependent common block) and 5 families that one PV is subset of the other PV in terms of packages and Java files (have 2 blocks: 1 common: 1 exclusive).

B. RQ2: What is the Commonality and Variability Related to the Cloned Java Methods across PVs?

Based on the results of RQ1, we consider that Java files having the identical base-names across PVs as the same cloned Java files.

1) Exact Cloned Methods: Figure 6 shows the results of the commonality and variability metrics corresponding to the exact cloned methods across PVs. The results of these metrics

[9]https://developer.android.com/guide/topics/manifest/manifest-intro

[10]https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

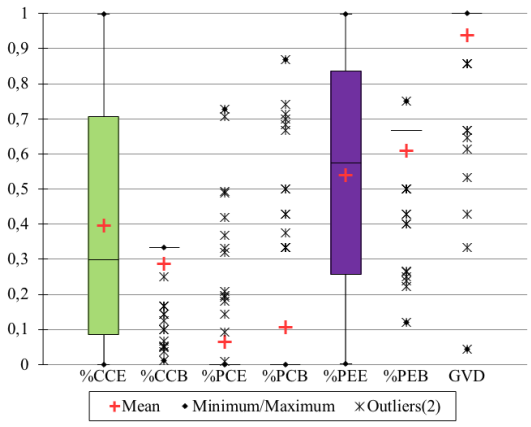

Figure 6. RQ2: the commonality and variability of exact cloned methods

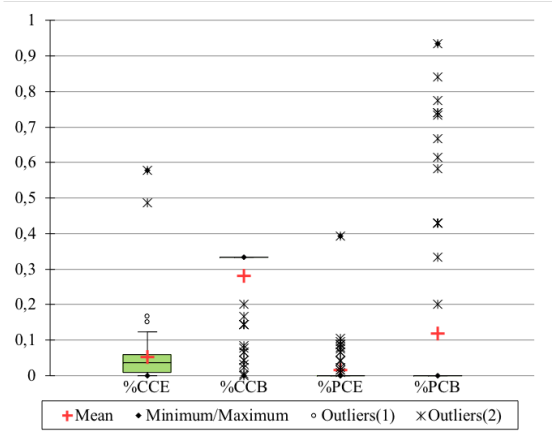

Figure 7. RQ2: the commonality of modified cloned methods

show that cloning exact method (type-1) is a common practice used by developers of the 75 families.

The very short box plot of %CCB shows a sort of agreement between families in terms of the existence of a common block of exact cloned methods and its percentage compared to shared and exclusive blocks identified in these families. This percentage of the common block is relatively decreased compared to the number of PVs composing the families. We find that the centrality of this percentage is between 29% and 33% where the mean and the median are placed respectively.

The relatively tall box plot of %CCE shows a disagreement in terms of the number of exact cloned methods in all PVs of different families. The number of exact cloned methods compared to the complete commonality is less than 8% for 24% (18/75) of the families, more than 72% for 20% (15/75) of the families, and between 8% anf 72% for 56% (42/75) of the families. We find that the number of the exact cloned methods in all PVs of a given family is decreased relatively when this family has more PVs.

Concerning the partial commonality, the box plots of %PCB and %PCE show the percentages of the shared blocks compared to common and exclusive blocks and the number of exact cloned methods compared to the partial commonality. As all families of 2 PVs do not have any partial commonality naturally, the box plots of %PCB and and %PCE are very short and their centrality is at 0. The values of %PCB and %PCE are increased compared to the number of PVs composing families (more PVs means more partial commonality). By comparing %PCB with %CCB and %PCE with %CCE, we find that the complete commonality of exact cloned methods is transformed to the partial commonality as well as more PVs are cloned in a given family. The reason is that new cloned PVs may not consider a subset of exact cloned methods corresponding to the complete commonality which transforms these methods to be a part of the partial commonality. For instance, in the family of 11 PVs, we identify that 1.8% of the methods are exact cloned methods in the 11 PVs (complete commonality) compared to 33.1% of the methods are extract cloned ones across subsets of these 11 PVs (partial commonality).

The results of %PEB and %PEE indicate to product exclusive methods that are not cloned across PVs. The short box plot of %PEB means families have an agreement in terms of the percentage of exclusive blocks compared to common and shared blocks. The tall box plot of %PEE denots to that families have different amounts of product exclusive methods. This means that PVs have different amounts of product-exclusive feature extensions. The box plot of GVD shows that most of families reach a high variability degree compared to the exact cloned methods across the PVs. Large families have less general diversity compared to small ones, e.g., all families of 2 PVs reach the maximum possible diversity level (GVL = 1), while the family of 11 PVs is considered relatively similar (GVD is 4%).

2) Modified Cloned Methods: We provide the results of the complete and partial commonality metrics corresponding to modified cloned methods in Figure 7. As all box plots are comparatively short ones, we have a sort of agreement between different android families in terms of the number of modified cloned methods compared to all methods of the corresponding family. %CCB and %CCE show that 98.6% (74/75) of android families have applied modified cloned methods across all PVs in which each PV has a different implementation of the same method. The centrality of the box plot is between 1% and 6% of the methods are modified cloned methods. The outliers of this centrality is realized in terms of 4 families that have 15%, 16.5%, 48.6%, and 57.8% of their methods as complete modified cloned methods.

Based on %PCE, all families of 3, 4, 5, 6 and 11 PVs contain modified cloned methods in subsets of their PVs. The %PCE of families of 2 PVs is zero because families of 2 PVs cannot have a partial commonality. In general, the centrality pattern of %PCE for families of 4, 5, 6 and 11 PVs is between 6% and 17% of their methods as modified cloned methods in subsets of the PVs. We have one outlier family that have 39.3% of its methods. For families of 3 PVs, the centrality pattern of %PCE is between 1.8% and 3.9%.

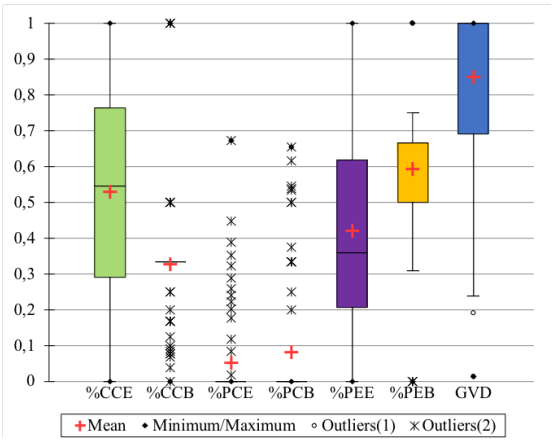

Figure 8. RQ3: The commonality and variability of configuration parameters

C. RQ3: What is the Variability in the Configuration Parameters of Android Applications of PVs?

The results of the commonality and variability metrics are presented in Figure 8. %CCB is reduced in favor of %PCB as well as families have more PVs. We find that the main reason of this reduction in %CCB is mainly due to the differences in the application components (*Activities*, *Services*, *Broadcast Receivers* and *Content Providers*) compared to slice differences in the other parameters in PVs of the same families.

The box plot of %CCB is comparatively short that means all families have the same pattern in terms of the ratio between the common block and the total number of identified blocks. On the other hand, the box plot of %CCE is very tall. Therefore, families hold quite different patterns in terms of the volume of complete common configuration parameters. The reason behind the tall box plot is the outliers of some families. For examples, we identify one family that has no common parameters in its PVs and 4 families that their PVs have exactly the same configuration parameters.

Unlike %CCB and %PCB, %CCE is not reduced in favor of %PCE by comparing %CCE and %PCE of the same families. For example in the family of 11 PVs, the number of configuration parameters in the common block are equal to those in the shared blocks (%CCB = %PCB = 32). Therefore, it is not important to have less complete commonality if we have extra partial commonality. Based on the comparatively tall box plots of % PEE, families have disagreement in terms of variability degree in their configuration parameters.

V. Discussions

A. How the Results Help Maintainers?

The results obtained in our empirical study provide useful information for maintainers of cloned android families. As we find that most of android families follow the Oracele's name convention that makes packages and Java files that have the exact identical base-names and different full qualified names across PVs are the same cloned Java files, we recommend to ignore the differences in full qualified names related to the variability of root packages of PVs of these families to be able to identify cloned method variability.

The cloning exact method is a common practice used by developers of all families. This produces a large number of methods that have the exact headers and bodies across all or a subset of PVs. To better maintain (e.g., fixing bugs) these cloned methods, we recommend to reengineer these cloned methods to construct reusable software libraries (i.e., APIs).

The presence of modified cloned methods in PVs highlights challenges to be encountered during reengineering these PVs to SPLs. Such challenges are related to merge the different implementations (bodies) of the same cloned methods by creating reusable and representative abstractions for the SPL core assets. We recommend to reuse existing approaches that contribute to cope with method merging challenges [11] [12]. Also, configuration parameters of PVs of android families are varied and it is essential to consider their commonality and variability while migrating these families to SPLs.

B. Threats to Validity

Internal validity. While measuring the partial commonality using %PCB and %PCE metrics, we treat all shared blocks equally without considering the number of PVs represented by each shared block. This means that our results do not represent this dimension in a deep way.

To obtain the commonality and variability metrics, we extend the But4Reuse tool by adding two components for parsing Java source code and XML files of PVs. We rely on the Eclipse Java development tools and the DOM parsers that are well-tested, evaluated and used in several tools and research papers. DOM is officially recommended by the World Wide Web Consortium (W3C). For identifying blocks, we use But4Reuse intersection algorithms that are also tested in several research papers [**?**] [9]. To mitigate related threats, we validate the correctness of our extension of But4Reuse based on the Java unit test and the manual validation of toy examples.

External validity. The main concern against the generalizability of our results is that we rely on a dataset of open-source android families developed in GitHub repositories based of fork-based development. Therefore, our results cannot be generalized for android families developed using other clone practices in the industrial companies.

VI. Related Work

To the best of our knowledge, our study is the first empirical study that performs a deeply analysis of the source code variability of PVs of 75 android families. We identify two crosscutting domains: analyzing reuse in android ecosystem and reengineering PVs to SPLs.

Analyzing reuse in android ecosystem. Li et al. [13] analyzed 1.5 million android applications and classified them to 75,963 different families. They identified applications developed by the same company based on package name and the certificate signed by developers. They only performed a high abstract code comparison to remove potential outliers. The resulting families are not open-source. Thus, we cannot include

them in our study. Businge et al. [1] identified 88 android families by analyzing the fork-based clone-and-own in GitHub. Businge et al. [1] studied four research questions to analyze: the characteristics of android families, the behaviour of pull requests, the diversity of developers of PVs, and the type of product customization. Shatnawi et al. [14] [15] invested the reuse practices of android APIs to identify reusable components corresponding to functionalities of android APIs. They extracted reuse practices in terms of frequent usage patterns of API classes and methods by analyzing how developers reused classes and methods of android APIs in their applications. Mojica et al. [16] studied 200,000 applications (not families) to analyze the inheritance, library and framework reuse practices in these applications based on class signatures. However, none of these existing studies performed a deep analysis of the source code commonality and variability in android families related to our RQs.

Reengineering PVs to SPLs. Different approaches were proposed to support reengineering PVs to SPLs. These approaches aim to identify features [6] [9], software product line architectures [4] [8], reusable assets [7] [17], and feature model synthesis [18] [19] . They analyze variability between PVs based on several algorithms including FCA [4] [19], interdependent element identification [9], latent semantic indexing [6], clustering algorithm [17] and clone detection [20]. Although these approaches contribute to manage the commonality and variability of PVs by reengineering to SPLs, their aim is not to understand the commonality and variability patterns based on empirical studies on cloned android families. We plan to adapt a number of these approaches to migrate android families into SPLs in our future works.

VII. Conclusion and Future Works

We performed an empirical study on 75 android families developed using clone-and-own approach in fork-based development. We investigated three main RQs for analyzing the commonality and variability of: (i) the organization of source code files, (ii) the cloning of Java methods and (iii) the configuration parameters in PVs of the 75 families.

We defined seven software metrics to measure the commonality and variability and proposed a methodology to extract values of these metrics for corresponding to the three RQs. We automated this methodology as an extension of But4Reuse.

The results of our empirical study show that developers of 90.66% of families rename root packages in different PVs while keeping the same sub-packages of cloned Java files following the Oracele's name convention pattern. Regarding the cloning of Java methods, we identify that cloning exact method is a common practice used by developers of all families and developers modify the body of cloned methods but with low percentage in most of families. The presence of modified cloned methods in PVs indicates that maintainers should put efforts for merging the diverse implementations (bodies) of the same cloned methods for creating reusable abstracted methods for the SPL core assets. The variability in configuration parameters are due to the differences in the application components (*Activities*, *Services*, *Broadcast Receivers* and *Content Providers*) compared to slice differences in the other parameters in PVs of the same families.

As future works, we plan to identify more cloned families from open-source repositories, to analyze the variability over time based on the version history and to reengineer the cloned android families into SPLs.

Acknowledgment

This work was supported by the ITEA 3 REVaMP^2 project.

References

[1] J. Businge, M. Openja, S. Nadi, E. Bainomugisha ,and T. Berger, "Clone-based variability management in the android ecosystem," in ICSME. IEEE, 2018, pp. 625–634.

[2] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in ICSME. IEEE, 2014, pp. 391–400.

[3] V. Saini, H. Sajnani, and C. Lopes, "Cloned and non-cloned java methods: a comparative study," Empirical Software Engineering, vol. 23, no. 4, pp. 2232–2278, Aug 2018.

[4] A. Shatnawi, A.-D. Seriai, and H. Sahraoui, "Recovering software product line architecture of a family of object-oriented product variants," Journal of Systems and Software, vol. 131, no. C, pp. 325–346, 2017.

[5] P. Clements and L. Northrop, "Software product lines: practices and patterns," 2002.

[6] A.-D. Seriai, M. Huchard, C. Urtado, S. Vauttier, H. Eyal-Salman et al., "Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing," in SEKE, 2013.

[7] C. Parra and D. Joya, "Split: An automated approach for enterprise product line adoption through soa." J. Internet Serv. Inf. Secur., vol. 5, no. 1, pp. 29–52, 2015.

[8] A. Shatnawi, A. Seriai, and H. Sahraoui, "Recovering architectural variability of a family of product variants," in ICSR. Springer, 2015, pp. 17–33.

[9] J. Martinez, T. Ziadi, T. F. Bissyandé, J .Klein, and Y. L. Traon, "Bottom-up technologies for reuse: automated extractive adoption of software product lines," in ICSE Companion. IEEE Press, 2017, pp. 67–70.

[10] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 577–591, 2007.

[11] K. Narasimhan, C. Reichenbach, and J. Lawall, "Cleaning up copy-paste clones with interactive merging," Automated Software Engineering, vol. 25, no. 3, pp. 627–673, 2018.

[12] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in CSMR-WCRE. IEEE, 2014, pp. 104–113.

[13] L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Mining families of android applications for extractive spl adoption," in SPLC. ACM, 2016, pp. 271–275.

[14] A. Shatnawi, A.-D. Seriai, H. Sahraoui, and Z. Alshara, "Reverse engineering reusable software components from object-oriented APIs," Journal of Systems and Software, vol. 131, pp. 442–460, 2017.

[15] A. Shatnawi, A. Seriai, H. Sahraoui, and Z. Al-Shara, "Mining software components from object-oriented APIs," in ICSR. Springer, 2015, pp. 330–347.

[16] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," IEEE software, vol. 31, no. 2, pp. 78–86, 2014.

[17] A. Shatnawi and A.-D. Seriai, "Mining reusable software components from object-oriented source code of a set of similar software," in IRI. IEEE, 2013, pp. 193–200.

[18] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans, "Feature model extraction from large collections of informal product descriptions," in FSE. ACM, 2013, pp. 290–300.

[19] J. Carbonnel, M. Huchard, A. Miralles, and C. Nebut, "Feature model composition assisted by formal concept analysis," in ENASE, 2017, pp. 27–37.

[20] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in CSMR. IEEE, 2008, pp. 163–172.