Guided Research Project

Analyzing the Variability Realization in Android

Nicolas Fußberger
Student Number: 384507

Advisor: Dr. Bo Zhang

Department of Computer Science

TU Kaiserslautern
# Table of Contents

1 Introduction .......................................................................................... 4  
1.1 Motivation ......................................................................................... 4  
1.2 Problem Statement ............................................................................ 4  
   1.2.1 Research Problems ................................................................... 4  
1.3 Research Method ............................................................................... 5  
   1.3.1 Research Questions .................................................................. 5  
   1.3.2 Research Procedure ................................................................. 6  
1.4 Research Contribution ....................................................................... 6  
1.5 Research Scope .................................................................................. 6  
1.6 Summary ............................................................................................ 6  

2 Foundations .......................................................................................... 7  
2.1 State of the Art .................................................................................. 7  
   2.1.1 Terminology ............................................................................. 7  
      2.1.1.1 Software Product Line ....................................................... 7  
      2.1.1.2 Variability ....................................................................... 7  
      2.1.1.3 Feature ........................................................................... 8  
   2.1.2 Binding Time ............................................................................ 9  
   2.1.3 Overview of Variability Realization Techniques ....................... 10  
      2.1.3.1 Conditional Compilation ................................................ 10  
      2.1.3.2 Conditional Execution .................................................... 11  
      2.1.3.3 Inheritance/Polymorphism ............................................. 12  
      2.1.3.4 Module Replacement ...................................................... 13  
      2.1.3.5 Aspect Orientation ......................................................... 13  
      2.1.3.6 Frame Technology ........................................................... 14  

3 Android .................................................................................................. 15  
3.1 Android Architecture ........................................................................ 16  
3.2 Android from a Product Line Perspective ........................................ 17  
3.3 Conceptual Architecture of Android ................................................ 18  
   3.3.1 Overview .................................................................................. 18  
      3.3.2.1 Product Configuration .................................................. 19  
      3.3.2.2 Board Configuration ....................................................... 20  
   3.3.3 Android Build System .............................................................. 20  
   3.3.4 Source Code ............................................................................ 21  

4 Analysis ................................................................................................. 22  
4.1 Configurability of Android ............................................................... 22  
   4.1.1 Qualitative Analysis ................................................................ 22  
      4.1.1.1 Configuration mechanisms ............................................ 22  
   4.1.2 Quantitative Analysis .............................................................. 23  
      4.1.2.1 Configuration Options ................................................... 23  
      4.1.2.2 Configuration Impact ..................................................... 24  
4.2 Variability Mechanisms in Configuration / Build System / Source Code .............. 26  
   4.2.1 Qualitative Analysis ................................................................ 26  
      4.2.1.1 Overview ....................................................................... 26  
      4.2.1.2 Conditional Compilation .............................................. 26  
      4.2.1.3 Conditional Execution ................................................. 29  
      4.2.1.4 Module Replacement ................................................... 30  
      4.2.1.5 Inheritance ................................................................. 32  
   4.2.2 Quantitative Analysis ................................................................ 35  
      4.2.2.1 Conditional Compilation in Source Code ....................... 35  
      4.2.2.2 Conditional Execution in Build System ......................... 42  

2
5 Conclusion & Future Work ................................................................. 43
6 References ..................................................................................... 45
1 Introduction

1.1 Motivation

With ever increasing software complexity, new engineering methods are needed to meet quality demands and a short time-to-market. Especially the customization of products for different market segments, different end users and different use-cases has been a reason for increasing complexity. To delay these design decisions as much as possible, variability is often shifted from hardware to software. Thus, a new set of engineering methods is needed to develop multiple software variants while at the same time meeting quality and time-to-market goals. Software product line engineering has emerged as a vibrant research field to address these challenges. Software Product Lines are developed by systematically managing variability while taking advantage of commonality to achieve the necessary quality and time to market by strategic reuse.

There are many successful cases in industry, where methods of software product line engineering have been successfully applied, most famously presented in the Product Line Hall of Fame\(^1\), often reporting spectacular results like a ROI of 10:1 or a productivity improvement of 360\% (Cummins inc.). However, experience reports from industry are mostly at a very abstract level since they need to protect their business from the competition. Especially the source code of these product lines is not available to researchers.

Therefore, researchers have turned to publicly available open source product lines. Most famously the Linux Kernel has been studied extensively by the product line community (e.g. \([1][2][3]\)). However, when it comes to variability realization techniques, most studies focus only on the usage of conditional compilation in source code. The analysis of Preprocessor code can be easily automated, yields quantitative results and can be scaled to a large number of product lines \([3]\). There is little empirical work (quantitatively or qualitatively) on the usage of other known variability realization techniques like polymorphism or module replacement. Additionally, most studies focus on the source code and variability in build system or configuration is often neglected. We intent to address this gap with our research by investigating the variability realization in Android, a very well-known product line that has – to the best of our knowledge – not had any attention in product line research.

1.2 Problem Statement

1.2.1 Research Problems

From these observations mentioned in the introduction we conclude the following research problems:

P1: There is little knowledge (quantitatively or qualitatively) on the usage of variability realization techniques in large product lines besides the usage of conditional compilation in source code.

P2: Existing analysis focus only on the source code and rarely investigate the usage of variability implementation mechanisms in configuration and build system.

\(^1\) http://splc.net/fame.html
1.3 Research Method

To address these research problems, we will study the variability realization of large open source product lines. We will not only focus on the usage of conditional compilation in source code but also investigate the usage of other well-known implementation techniques in source code, build system and configuration files. Since the variability realization is also highly related to the configuration mechanisms, we also study the configurability of Android. We chose the Android operating system since it is a very popular software ecosystem that has not been studied by the product line community\(^2\). Moreover, it is a very heterogeneous software system incorporating many different open source projects and is characterized by its decentralized development. Therefore, we expect to find not a consistent use of a single variability realization technique but the usage of many different techniques that are used together.

1.3.1 Research Questions

We conclude the following research questions. While RQ2 addressed both P1 and P2, RQ1 can be considered a prerequisite. Since each variation point can be ultimately traced to some configuration option, is it necessary to understand how Android is being configured in order to understand how variability is realized.

**RQ1:** How is Android being configured for a specific device?
- **RQ1.1:** What configuration options exist?
- **RQ1.2:** What configuration mechanisms are used?
- **RQ1.3:** What is the impact of the configuration?

**RQ2:** How is variability realized in Android?
- **RQ2.1:** What variability mechanisms are used on different layers?
- **RQ2.2:** Is the variability implementation localized to individual layers or distributed among multiple layers (crosscutting)?
- **RQ2.3:** What is the quantitative usage of different variability realization techniques?

These research questions are addressed in the following chapters:

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1.1</td>
<td>4.1.2.1</td>
</tr>
<tr>
<td>RQ1.2</td>
<td>4.1.1.1</td>
</tr>
<tr>
<td>RQ1.3</td>
<td>4.1.2.2</td>
</tr>
<tr>
<td>RQ2.1</td>
<td>4.2.1</td>
</tr>
<tr>
<td>RQ2.2</td>
<td>4.2.1</td>
</tr>
<tr>
<td>RQ2.3</td>
<td>4.2.2</td>
</tr>
</tbody>
</table>

\(^2\) Android has been studied from an ecosystem perspective [24], but to the best of our knowledge there has been no analysis of the variability realization in the operating system itself.
1.3.2 Research Procedure

Our research combines qualitative and quantitative data on the variability realization in Android. Qualitative data like the configuration mechanisms or examples for the usage of different variability realization techniques for different purposes are obtained by studying the official Android documentation, existing literature and manually searching through the Android OS source code. Based on these findings we developed tools to extract quantitative data on the impact of configuration mechanisms and the usage of different variability realization techniques. All tools have been developed in Python and are heavily based on Kati\(^3\), a tool developed by Google that can parse and partially evaluate Makefiles. Based on the Makefile abstract syntax tree created by Kati and specific variable names used in Android's Makefiles, we can then automatically detect different variability realization techniques. Additionally, the VITAL tool [4] is used for extracting the usage of preprocessor commands in C/C++ source code.

1.4 Research Contribution

Our main contribution in this project is the generation of empirical data (qualitative and quantitative) on the variability realization in Android, a large open source software ecosystem that is currently an open research topic. This includes qualitative data on the usage of different variability realization techniques and different configuration mechanisms in Android. We will also present quantitative data on the possible configuration options, configuration impact and the usage of variability realization techniques. To generate this data, we present a new analysis method for extracting variability data from Make-based open source systems and implemented the necessary tool support to automate this analysis.

1.5 Research Scope

There are different flavors of Android for smart watches, smartphones, televisions, cars, which can altogether be viewed as a product line. But in this report, we will look at Android OS for smartphones/tablets and treat this version of Android alone as a product line. Therefore, we will only focus on variability mechanisms bound at construction time since this is one of the main variability drivers for smartphones, although Android also includes mechanisms for runtime variability (e.g. through property files, apps, etc.). Similarly, in our quantitative analysis we will focus on variability mechanisms related to hardware variability (i.e. defined in Android board configuration).

1.6 Summary

Chapter two introduces the terminology and foundational knowledge about product lines, variability and binding times. Additionally, the variability realization techniques commonly found in literature are presented. In chapter three the target system Android, its architecture in terms of configuration, build system and source code is explained. Based upon this, chapter four contains the analysis results concerning Android's configurability and variability implementation. We present the most important configuration mechanisms and analyze their

\(^3\) https://github.com/google/kati
impact on the rest of the system. Additionally, variability realization techniques in Android are presented, both, in terms of typical examples and quantitative results on the usage of different techniques.

2 Foundations

2.1 State of the Art

2.1.1 Terminology

2.1.1.1 Software Product Line

“A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [5]. Instead of developing each product from scratch, product line engineering is about taking advantage of commonality and carefully managing variability. New products can be assembled from already existing parts (core assets) tailored to individual costumers. Despite tailor-made products, the expected benefits from adopting a product line over single system development include reduced costs, improved quality and fast time to market [6]. It is clear that these advantages do not come for free. A product line requires a serious upfront investment for domain engineering processes like scoping or development of core assets. Figure 1 shows the expected costs for developing several systems from scratch compared to a product line approach. A small number of systems can still be developed with less effort using a single-system development approach since there is no investment into strategic reuse required. Only if many different (but similar) systems are developed such an upfront investment pays off in the long term.

![Figure 1: Comparison of costs for single system development and product-line development](image)

2.1.1.2 Variability

Variability is a very general concept within product line engineering and can refer to the variability within a software development process as well as its resulting artefacts like requirements specifications, architecture documents, source code or test cases [8]. For our
purpose, we want to use a definition of variability that is more targeted towards variability implementation. Variability can be defined as “the ability to derive different products from a common set of artefacts” [6]. Anastasopoulos and Gacek [9] differentiate between multiple types variability which are shown in Table 2. In general, one can differentiate between positive and negative variability, where positive variability adds functionality and negative variability removes functionality. Other categories denote the optional inclusion of code/requirements/components, their replacement (alternative) or change in their functionality. Another variability type is concerned with a change in the platform or environment of a system. An example for such variability is the migration of a system from a Unix-based environment to a Windows-based environment.

<table>
<thead>
<tr>
<th>Variability Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>Functionality is added</td>
</tr>
<tr>
<td>Negative</td>
<td>Functionality is removed</td>
</tr>
<tr>
<td>Optional</td>
<td>Code is included</td>
</tr>
<tr>
<td>Alternative</td>
<td>Code is replaced</td>
</tr>
<tr>
<td>Function</td>
<td>Functionality changes</td>
</tr>
<tr>
<td>Platform / Environment</td>
<td>Platform or environment changes</td>
</tr>
</tbody>
</table>

To describe variability even further Pohl introduced the notion of variability subject and object [7]. A variability subject is “a variable item of the real world or a variable property of such an item” and describes what can vary. A variability object is “a particular instance of a variability subject” and describes how an item varies. An example for a variability subject is the cooking time of a steak where possible instances (variability objects) are rare, medium rare or well done. The manifestation of variability subjects in artefacts such as requirements, architecture or code is referred to as a variation point. Jacobson et al. first introduced this term as “one or more locations at which the variation will occur” [10]. Analogously, the “representation of a variability object within domain artefacts” [7] is called a variant. Consider the color of Apple’s iPhone. Possible variability objects include green, blue, black, yellow, etc. However, Apple only offers silver, gold, rose gold and black as possible variants for this variability object. Therefore, variants might only be a subset of possible variability objects.

2.1.1.3 Feature

The term feature describes “a characteristics or end-user-visible behavior of a software system” [6] and is typically used in the context of a product line to distinguish multiple products. A product can be thought of as a set of features, a so-called feature selection. This concept allows to easily communicate product characteristics to all stakeholders while also enabling to manage variability and commonality in all software lifecycles. Anastasopoulos and Gacek [9] define multiple feature types (cf. Table 3) that constrain feature selection. A mandatory feature must be present in all products, while an optional feature may only exist in a subset of products. Constraints between features include alternative features (i.e. only one of several features can be selected) and mutually inclusive and exclusive features.
Features and their relations are typically specified in a feature model whose graphical representation is called a feature diagram. An example for a feature diagram of a graph library is depicted in Figure 2. The diagram is a tree whose nodes denote features and different notations for parent-child connections encode different feature types. Additionally, constraints for feature selection may be added as propositional logic.

In Figure 2, an empty circle on top of a feature denotes an optional feature, while a filled circle denotes a mandatory feature. An empty circle at the bottom of a node defines a one-out-of-many relationship, so that only one of the child nodes may be selected for any product. A filled circle defines a some-out-of-many relationship and any number of children may be selected for a product. Mutually inclusive and exclusive feature types are defined via constraints. For example, the MSG algorithm can only be selected if the graph is undirected and weighted (MSG \( \Rightarrow \) Undirected \( \land \) Weighted).

![Figure 2: A feature diagram for a graph library [6]](image)

### 2.1.2 Binding Time

When deriving a product from a common set of assets, at one point the decision needs to be made how to resolve variability added through variation points (i.e. which features to include or exclude in a product). The time at which this decision is made is called the binding time. Depending on the domain and on the business objective for a product line, it can make sense to make this decision sooner or later.

For example, in embedded systems where resources are often scarce variability is bound rather sooner than later so that the code for a single product can be optimized and needs as little memory as possible. In other domains like the smartphone industry where additional functionality can be added by users by installing additional apps, such variability needs to be present at runtime.
Svahnberg et al. [8] define the following phases of a system’s lifecycle where variability may be bound: *Product architecture derivation, compilation, linking and runtime*. During product architecture derivation, variation points in the product line architecture need to be set to a particular variant to derive the architecture for a particular product. Examples for variation points may be optional components that correspond to an optional feature or selecting a particular specialization of a general component (e.g. depending on the target hardware platform). The most common technique for bounding variability at compile time is the C/C++ preprocessor which scans the source code and adds or removes certain parts. During linking, the object files created by the compiler are built into an executable. This step depends on the chosen programming language and technologies that are used. During runtime, it may be possible to extend the system by adding new variants (e.g. downloading apps on the smartphone) or to choose among a set of predefined variants (e.g. by changing the settings in an application).

### 2.1.3 Overview of Variability Realization Techniques

In this section, we will give a brief overview of variability realization techniques commonly found in literature. We will only consider mechanisms that bind variability as early as compile-time. In particular we will not cover product architecture derivation as an earlier binding time [8]. We will look specifically at variability mechanisms in source code, although some mechanisms can also be used on other artefacts like requirements specification or architecture documentation.

<table>
<thead>
<tr>
<th>Variability Mechanism</th>
<th>Binding Time</th>
<th>Implementation</th>
<th>Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional Compilation</td>
<td>Compilation</td>
<td>Preprocessor</td>
<td>Any</td>
</tr>
<tr>
<td>Conditional Execution</td>
<td>Runtime</td>
<td>Conditional</td>
<td>Limited, mostly</td>
</tr>
<tr>
<td>Inheritance/Polymorphism</td>
<td>Compilation/</td>
<td>Language</td>
<td>Statements</td>
</tr>
<tr>
<td>Module Replacement</td>
<td>Linking</td>
<td>Constructs</td>
<td>Any</td>
</tr>
<tr>
<td>Aspect Orientation</td>
<td>Compilation</td>
<td>Specific tools</td>
<td>Limited, mostly</td>
</tr>
<tr>
<td>Frame Technology</td>
<td>Compilation</td>
<td>Code weaving</td>
<td>statements</td>
</tr>
<tr>
<td>Cloning</td>
<td>Compilation</td>
<td>Copy&amp;Paste,</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Branches</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3: Overview of variability mechanisms commonly found in literature. Data adapted from [11].*

#### 2.1.3.1 Conditional Compilation

**Description**

Conditional compilation is a widely used variability mechanism in open source software as well as in industrial software [12]. It allows programmers to conditionally add or remove code before compilation based on a configuration. This variability is bound at construction time which allows for highly optimized and memory efficient code. This mechanism can be used to conditionally compile single statements, classes, modules – there is no limit for application in terms of granularity.
Implementation

Conditional compilation is typically implemented using a preprocessor, most famously the C/C++ preprocessor. Using `#ifdef` directives, any part of the code can be conditionally compiled based on some `#define` definitions.

```
1. #ifdef __WINDOWS__
2.   #include <winsock2.h>
3. #else
4.   #include <sys/socket.h>
5. #endif

1. #ifdef HAS_HEAT_SENSOR
2.   Sensor sensor = new HeatSensor();
3.   Observer.register(sensor);
4. #endif
```

Figure 4 shows two examples for the usage of the C/C++ preprocessor. The left example demonstrates the conditional inclusion of files based on the target operating system (OS is Windows if `__WINDOWS__` is defined through `#define __WINDOWS__`) and the right example shows the addition of a heat sensor based on if the target product supports such a sensor (if `HAS_HEAT_SENSOR` is defined through `#define HAS_HEAT_SENSOR`). Although preprocessors are mostly used for code, they work on any text-based artefact and can also be used for requirements documents, architecture documentation, etc.

Advantages & Disadvantages

Conditional compilation is very easy to apply using available preprocessors. These are not limited to code but can be applied to any text-based document since they are oblivious to the syntax. However, for a long term benefit their usage requires some discipline. Since the preprocessor is also used for other things beside variability implementation (e.g. constant definitions), its usage as a variability mechanism should be clearly separable by employing a prefix like `HAS_PREFIX`. This also allows further automated analysis regarding code erosion and dead code detection [13]. The developer should also care for simple logical expressions inside `#ifdef` statements and avoid highly nested `#ifdef` code to keep the code maintainable [14].

2.1.3.2 Conditional Execution

Description

Conditional execution uses standard conditional statements to implement variability. In contrast to conditional compilation these statements are executed at runtime and can thus be used to implement runtime variability. Since these statements are part of the programming language, they need to obey its syntax rules and can only be used to implement variable blocks of statements.

Implementation

The build-in if-else or switch constructs of programming languages are used to implement conditional execution. In order to differentiate variability implementation from regular usage
of conditional statements, it is recommended that a specific variable naming pattern (e.g. a common prefix) is chosen.

Advantages & Disadvantages
Conditional execution is very easy to use and required no further tools since conditional statements are part of every higher programming language. Since conditional statements are also used for other things, there is risk of mixing the implementation of variability with the rest of the code. A naming pattern of variables can solve this issue. A disadvantage of this approach is the mixing of commonality and variability inside a single file. For example, a new variant cannot be added without first understanding and then altering the existing code. Like with conditional compilation there is also the risk of creating highly nested or complex conditional expressions that increase maintainability over time.

2.1.3.3 Inheritance/Polymorphism

Description
There are multiple different variability mechanisms related to inheritance or polymorphism ranging from binding at compile time to binding at runtime. Mechanisms commonly found in literature include subtype polymorphism, parametric polymorphism (often called static polymorphism) and overloading [11][15] as well as multiple inheritance, mixin-based inheritance, object-based inheritance [9] and ad-hoc polymorphism and casting [15].

Implementation
The implementation of inheritance-based variability mechanisms typically depends on the specific type of mechanisms and the programming language. Not all mechanisms can be implemented in all object-oriented languages. For example, Java does not support multiple inheritance (in contrast to C++).

More information on how to implement subtype polymorphism and parametric polymorphism in C++ can be found in [16] and [17].

Advantages & Disadvantages
A common advantage of inheritance-based mechanisms is the separation of commonality and variability. For example, an abstract class called Sensor implements the common behavior across all sensors, while each specific sensor is implemented in a subclass. This technique is often used in frameworks to allow developers to define extensions to the existing functionality [11].

On the other hand, this approach of implementing variability and commonality in different files does not scale well. With increasing variability, the amount of subclasses increases as well, creating a complex inheritance tree. Runtime binding always adds a performance penalty, and may also result in runtime errors (null pointers) [11].
2.1.3.4 Module Replacement

Description
Module replacement is used for selecting between files or subsystem often stored in sibling directories. Depending on the implementation technique this selection can take place during compilation or linking.

Implementation
Module replacement is commonly implemented in C++, where an (possibly conditionally) included header file selects the right file to include or the header files have identical names among sibling directories, in which case the selection can be done using the –I flag to select an include directory (compile time) or –L flag to select a path to a linked library.

Advantages & Disadvantages
Like inheritance-based mechanisms, module replacement allows the separation of variants but has no runtime overhead since variability is bound during compilation or linking. However, variation points are not easily visible in code (in case the headers have identical names) and may also involve the build system (e.g. to specify –I or –L flags). This problem can be addressed by tool support. Another problem is the handling of defaults (e.g. in case of optional variability) since a module needs to be selected at all time. A dummy module may solve this problem [11].

2.1.3.5 Aspect Orientation

Description
Aspect orientation is a technique to address the problem of crosscutting concerns that result in code duplication, scattering or tangling. It uses so-called aspects to localize the code related to these concerns into one code unit. “An aspect is a programming construct that encapsulates the implementation of a crosscutting concern” [6]. An aspect is then weaved into the rest of the program at specific locations (called joint points) using a process called aspect weaving.

Implementation
The implementation typically requires special tool support (e.g. AspectJ) since code weaving is not supported in most programming languages.

Advantages & Disadvantages
Similar to module replacement, this technique allows the separation of commonality and variability. Additionally, it provides the means to localize implementation of features which would otherwise be scattered across the code and result in duplicated code. However, this technique can only be used with additional tool support since code weaving is not supported in most programming languages. In contrast to techniques like inheritance or conditional execution which use constructs that every programmer is familiar with, this technique will most likely require a learning phase before it can be applied.
2.1.3.6  Frame Technology

Description
Frame Technology is a variability mechanism developed by Paul G. Basset [18] that allows to separate those modules that change frequently from those that change less frequently. For example, the implementation of common functionality across all products may not change very often, whereas new features or changes to existing features may be more frequent. If commonality and variability reside in the same file, each change has the potential to break the common code although only the variable code needs to be changed. Using Frame Technology “arbitrary text parts can be managed as variabilities, even syntactically incomplete code such as partial loops or isolated return statements” [19].

Implementation
Similar to aspect orientation, the implementation of Frame Technology requires special tool support. A detailed example can be found in [19].

Advantages & Disadvantages
Like module replacement or inheritance, this technique allows the separation of common and variable code but unlike other mechanisms, it does not need to respect the programming language syntax when doing so. This technique is not well known and not widely used [11]. It requires special tool support and additional training.
### 3 Android

Android is the most successful mobile operating system of all time, currently owning a market share of 84.1% (according to Gartner, May 2016). It is not only the major operating system for smartphones, but also runs on tablets, wearables like smartwatches, televisions and automotive multimedia systems. Clearly, such a platform needs to carefully manage variability. After all, these different kinds of devices greatly differ in terms of their provided functionality, user interface, hardware, etc.

In this report, we will focus solely on Android (version 6) for smartphones and tablets, since this is the primary usage of Android.

---

**Figure 5: Android 6 devices and their HW/SW characteristics**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Display</strong></td>
<td>LED-backlit 7.0 inches 1200 x 1920 pixels</td>
<td>LED-backlit 7.0 inches 1200 x 1920 pixels</td>
<td>IPS LCD 8.9 inches 1536 x 2048 pixels</td>
<td>AMOLED 5.96 inches 1440 x 2560 pixels</td>
<td>True HD IPS+ 4.95 inches 1080 x 1920 pixels</td>
</tr>
<tr>
<td><strong>Chipset</strong></td>
<td>Qualcomm Snapdragon S4Pro</td>
<td>Qualcomm Snapdragon S4Pro</td>
<td>Nvidia Tegra K1</td>
<td>Qualcomm Snapdragon 805</td>
<td>Qualcomm MSM8974 Snapdragon 805</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Quad-core 1.5 Ghz Krait</td>
<td>Quad-core 1.5 Ghz Krait</td>
<td>Dual-core 2.3 Ghz Denver</td>
<td>Quad-core 2.7 Ghz Krail 450</td>
<td>Quad-core 2.3 Ghz Krail 400</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>Adreno 320</td>
<td>Adreno 320</td>
<td>Kepler DX1</td>
<td>Adreno 420</td>
<td>Adreno 330</td>
</tr>
<tr>
<td><strong>Primary camera</strong></td>
<td>5 MP Geo-tagging, touch focus, face detection</td>
<td>5 MP Geo-tagging, touch focus, face detection</td>
<td>8 MP Geo-tagging, touch focus, face detection</td>
<td>13 MP Geo-tagging, touch focus, face detection, panorama, HDR</td>
<td>8 MP Geo-tagging, touch focus, face/smile detection, panorama, HDR</td>
</tr>
<tr>
<td><strong>Sensors</strong></td>
<td>Accelerometer, gyro, proximity, compass</td>
<td>Accelerometer, gyro, proximity, compass</td>
<td>Accelerometer, gyro, proximity, compass</td>
<td>Accelerometer, gyro, proximity, compass, barometer</td>
<td>Accelerometer, gyro, proximity, compass, barometer</td>
</tr>
</tbody>
</table>

Figure 5 depicts the specification for different smartphones and tablets, all running the current Android operating system version 6. The obvious variability among these devices is their underlying hardware. They use different CPUs, GPUs, different display technologies with different dimensions, different sensors and cameras. This includes optional variability (like different sensors) and alternative variability (like CPU or GPU). In fact, the underlying hardware is the only way, that a regular smartphone can stand out from the competition. Since the Android app store is open to all Android smartphones, they can all download the same applications. Therefore, any pure software variability can be easily replicated by third party apps. For example, the LG Nexus 5 is the only device whose camera has built-in smile detection. However, a quick search in the Google play store instantly reveals third party apps that enable all other devices to achieve the same functionality.
3.1 Android Architecture

A look at the Android architecture gives a first clue on how hardware variability is being supported within Android.

The topmost layer consists of the application framework containing the standard Android APIs used by app developers to access basic functionality. The Binder IPC proxies enable the inter-process communication between the application framework and the system services which are used by the application framework to access the underlying hardware.

The hardware abstraction layer exposes a consistent interface to higher layers while allowing the underlying hardware implementation to change. This is a common variability mechanism at the architectural level that enables the support for cameras, sensors, etc. without having to change any of the upper layers.

Android is built on top of the Linux kernel whose device drivers control the hardware.

The Android architecture already gives some insight into how this operating system supports hardware variability:

1. Android makes use of HALs that hide all hardware related variability from upper layers by providing a consistent interface. This is the location were hardware vendors can add their custom implementations.

2. Android is built on top of the Linux kernel, which provides its own sophisticated mechanisms for handling variability that have been studied in detail by the product line community. For example, the Linux kernel can be compiled for many different hardware architectures (like x86 or ARM) which allows it to run on tablets, smartphones, TVs, desktop pcs, etc. As we will later see, the Linux kernel is actually not part of the Android source code, but is included as a precompiled binary. Therefore, a lot of the variability is already bound and does not need to be considered by Android.
### 3.2 Android from a Product Line Perspective

*Table 4: Android source code structure [20]*

<table>
<thead>
<tr>
<th>Directory</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abi</td>
<td>Minimal C++ Run-Time Type information support</td>
</tr>
<tr>
<td>Bionic</td>
<td>Android’s custom C library</td>
</tr>
<tr>
<td>Bootable</td>
<td>OTA, recovery mechanism and reference bootloader</td>
</tr>
<tr>
<td>Build</td>
<td>Build System</td>
</tr>
<tr>
<td>Cts</td>
<td>Compatibility Test Suits</td>
</tr>
<tr>
<td>Dalvik</td>
<td>Dalvik VM</td>
</tr>
<tr>
<td>Development</td>
<td>Development tools</td>
</tr>
<tr>
<td>Device</td>
<td>Device-specific files and components</td>
</tr>
<tr>
<td>Docs</td>
<td>Content of <a href="http://source.android.com">http://source.android.com</a></td>
</tr>
<tr>
<td>External</td>
<td>External projects imported into the AOSP</td>
</tr>
<tr>
<td>Frameworks</td>
<td>Core components such as system services</td>
</tr>
<tr>
<td>Hardware</td>
<td>HAL and hardware support libraries</td>
</tr>
<tr>
<td>Libcore</td>
<td>Apache harmony</td>
</tr>
<tr>
<td>Libnativehelper</td>
<td>Helper functions for use with JNI</td>
</tr>
<tr>
<td>Ndk</td>
<td>Native development Kit</td>
</tr>
<tr>
<td>Out</td>
<td>Build output will be placed here</td>
</tr>
<tr>
<td>Packages</td>
<td>Stock Android apps, providers and IMEs</td>
</tr>
<tr>
<td>Pdk</td>
<td>Platform Development Kit</td>
</tr>
<tr>
<td>Prebuilt</td>
<td>Prebuilt binaries, including toolchains</td>
</tr>
<tr>
<td>Sdk</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>System</td>
<td>Embedded Linux platform that houses Android</td>
</tr>
<tr>
<td>Tools</td>
<td>Various IDE tools</td>
</tr>
</tbody>
</table>

From an external perspective Android can clearly be seen as a Software Product Line. Figure 5 showed some sample products containing commonality and variability that are all running the same Android version.

As it turns out, Android also conforms to the model of a Product Line from an internal perspective. Table 4 shows the folder structure of the Android source code, where core assets and application specific assets are clearly separated. The “Device” folder contains product specific assets like custom apps, layout files and other configuration files. The folders highlighted in blue can be considered as core assets. These are Android specific modules, but also other open-source projects that have been integrated into the Android platform. They may contain variability that is resolved during the build process based on the selected configuration. A specific product is derived by building Android with a selected product configuration. The resulting files are placed in the “Out” folder which does not contain any build-time variability any more.
3.3 Conceptual Architecture of Android

3.3.1 Overview

Conceptually, we decompose Android into three layers (cf. Figure 7). The configuration layer comprises Android’s configuration files that define its configuration space and act as input for product derivation. Android defines different kinds of configurations at different layers of abstraction, namely the product and board configuration. The product derivation process is implemented by Android’s build system that resolves variability based on the selected configuration. This can have an effect on the overall build process (global Makefiles) or on individual modules (module Makefiles). The source code is organized in modules that can be hierarchically grouped. Each module contains an Android.mk file that defines its compilation process.

3.3.2 Android Configuration

Table 5: Android Configuration Layers [21]

<table>
<thead>
<tr>
<th>Layer</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>myProduct, myProduct_eu</td>
<td>The product layer defines the feature specification of a shipping product such as the modules to build, locales supported, and the configuration for various locales. In other words, this is the name of the overall product. [...]</td>
</tr>
<tr>
<td>Board/Device</td>
<td>Trout, goldfish</td>
<td>The device/board layer represents the physical layer of plastic on the device (i.e. the industrial design of the device). For example, North American devices probably include QWERTY keyboards whereas devices sold in France probably include AZERTY keyboards. [...]</td>
</tr>
<tr>
<td>Arch</td>
<td>Arm, x86, mips, arm64</td>
<td>The architecture layer describes the processor configuration and ABI (Application Binary Interface) running on the board.</td>
</tr>
</tbody>
</table>

Android specifies three different layers of configuration with decreasing abstraction level. We will focus on the first two, the product and board configuration since the architecture configuration is part of the Linux kernel.
Configuration files in Android are regular Makefiles, although they only make use of a subset of the Make language. They assign values to a set of predefined constants that will be used during the build process to decide which modules to build, how to resolve their variability, etc. Note that there is no tool support for creating a configuration. The recommended approach is to copy an existing configuration and adjust it as necessary.

3.3.2.1 Product Configuration

Number inheritance from the common Open Source product configuration
$\texttt{(call inherit-product, }$\texttt{\$(SRC\_TARGET\_DIR)/product/aosp_base_telephony.mk)}$

\begin{verbatim}
PRODUCT\_NAME := aosp\_shamu
PRODUCT\_DEVICE := shamu
PRODUCT\_MODEL := AOSP on Shamu
PRODUCT\_MANUFACTURER := motorola

PRODUCT\_PACKAGES += \
    Launcher3
PRODUCT\_COPY\_FILES += \
    device/moto/shamu/init.shamu.rc:root/init.shamu.rc
PRODUCT\_PROPERTY\_OVERRIDES += persist.ims.disableDebugLogs=1
PRODUCT\_LOCALES += en\_GB de\_DE es\_ES fr\_CA
\end{verbatim}

Figure 8: Product configuration excerpt from device/moto/shamu (device.mk)

The product configuration is the configuration on the highest level of abstraction. Although it is documented as a “feature specification”, we will see that the abstraction to the level of features is inherently missing. Other product lines like the Linux kernel allow the user to select which features should be included in the current build and then maps this information to a lower level configuration containing which modules to compile, which dependencies to resolve and which parameters to choose. Android on the other hand does not provide any such mapping and puts the burden on the configurator to select modules and to set parameters in such a way that the desired features are included in the current build. Figure 9 shows an example product configuration from a Nexus phone. The configuration mechanisms in the product configuration will be explained in detail in section Product Configuration on page 22.
3.3.2.2 Board Configuration

According to the Android specification, the board configuration represents the “physical layer of plastic on the device”. This configuration certainly defines more technical characteristics of the device on a lower level of abstraction than the product configuration. Examples are the processor architecture, cpu or wlan device. However, in contrast to the product configuration, the variable names of the board configuration and also their possible values are not defined by the Android specification which makes the configuration a tedious and error prone task. The constants defined in this configuration are referenced by global Makefiles as well as local Makefiles of individual modules and often appear in conditional statements. Therefore, there exists an implicit set of possible values for each constant. Since there is also no tool support, the only option is to manually trace a constant through the build files in order to know their possible values and impact.

3.3.3 Android Build System

Androids build system is based on GNU Make. But unlike most Make-based build systems, it does not use it recursively (i.e. build each subsystem independently by a recursive Make call). Instead, it relies on file naming conventions and includes all files with a certain name throughout the source code to build a gigantic Makefile during built time. A schematic overview of the build system is depicted in Figure 10. The build system is decomposed in the build configuration (using the “lunch” tool), globally defined Makefiles can control the overall build process (residing in build/core/) and module Makefiles (named “Android.mk”) that define how an individual module is being compiled.
The Android build process starts by selecting a product configuration to be built using the “lunch” tool. The build process is then initiated by running Make, which executes the globally defined Makefiles in the build/core directory. These Makefiles include (the Make “include” operation is similar to a preprocessor #include) the configuration files of the previously selected configuration and make use of their definitions to resolve variability and adjust the build process accordingly.

The Android source code is decomposed into modules. Each module has a file called “Android.mk” that defines how each module should be compiled. These files might also include conditional statements referencing the variables from the product and/or board configuration. The global Makefiles will include all files named “Android.mk” and resolve their variability by evaluating their conditional statements and execute their compilation commands. In total Android 6 contains 2834 modules, each containing a file called Android.mk. Figure 11 shows a sample Android.mk file. These files make use of a small subset of the Make language and assign values to a set of predefined constants similar to the product configuration files. After setting the path to the current directory (line 1) and clearing all previously set variables (line 2, this is necessary since these Makefiles are not executed in isolation but included by a global Makefile), the list of source files of the current module is defined (line 3) together with the name of the module (line 4) and any compiler flags that should be added when building the compiler command (line 5). The compiler command is then constructed according to the predefined rules for compiling a static library (line 6).

3.3.4 Source Code

![Android source code](image)

The source code of Android is organized into 2834 modules which can be hierarchically nested. According to the analysis conducted by Zhang [22] the majority of Android is implemented in C/C++ code (~120,000 C/C++ files) followed by Java (~50,000 files). With 4475 Makefiles, the build system in Android is highly distributed with each module having their own Makefiles. Overall, it is clear that the Android source code is highly heterogeneous by allowing the use of a variety of different programming languages like Java, C, C++, Python, Go, Javascript, etc.
4  Analysis
4.1  Configurability of Android
4.1.1  Qualitative Analysis
4.1.1.1  Configuration mechanisms

Product Configuration

Figure 13 depicts an excerpt from the product configuration of a Nexus phone (codename shamu). Notice that the configuration is not created from scratch, but inherited from another configuration (line 1-2). It starts by initializing basic information like the product name, device name, model and manufacturer (line 3-6).

The constant PRODUCT_PACKAGES lets you define a list of modules to be build. Using this mechanism, you can also add your own modules (defined in the product-specific device folder) to extend Android. Each module throughout the Android source code contains a module specification in the form of a special Makefile that defines among other things the name of the module. Notice that there is neither a global definition of all modules nor is there any definition of dependencies between modules.

PRODUCT_COPY_FILES defines a list of files that are copied during the build process to another location (syntax source:target, line 8). This is a customization mechanism commonly used to define your own layout, look & feel, runtime configurations or other resource based information. Extracting variability from code and putting it in a resource file like an xml or configuration file is good practice since it lets you alter your product without recompilation or touching the code. Using this mechanism, you can for example define your own layout files and copy them during the build process to override the default layout.

Not all variability is resolved during build-time. Android makes extensive use of property files which are read at runtime. Values for properties can be set using the PRODUCT_PROPERTY_OVERRIDES constant (line 9).

PRODUCT_LOCALES allows to define which languages should be preinstalled (line 10).
Board Configuration

As explained in section Board Configuration on page 20, the Board Configuration has no defined configuration mechanisms. Neither the possible configuration constants, nor their possible values or their semantics are defined in any way.

4.1.2 Quantitative Analysis
4.1.2.1 Configuration Options

To get an impression on what is actually configurable within Android, we analyzed how many configuration options exist. Specifically, we studied the board configurations available in Android 6 since most of the variability implementation is hardware-related. The data extraction was performed by building a simple Makefile parser that is able to parse Androids board configurations and extract the (constant, value) pairs. In this step we ignored any conditional statements (which rarely occur), but did include the constants in their body in our result. By parsing all available board configurations and merging the results we know how many different values where used for each constant. Although we cannot claim to get a complete result with this approach, it is a first approximation and the best we can do with the data that is publicly available.

The resulting data is visualized in Figure 14 in form of a histogram. We analyzed all available 26 board configuration files of Android 6 and obtained 158 unique constants. The y-axis shows the amount of configuration constants and the x-axis depicts the frequency of the amount of different values used.

This result clearly shows that only a small fraction of all configuration constants actually has more than one possible value among all available configuration files. 113 constants only have a single value (71%) while only 20 configuration constants (13%) have 5 or more values and can be regarded as real variability.

The natural questions that follows is: What do the configuration constants define that are highly variable? To party answer this question we also included Figure 15 displaying the name of each configuration constant. Obviously, extensive domain knowledge is required to fully answer this questions but we can make some observations from the constant names.

![Figure 14: Possible configuration options for board configuration constants. The results were obtained by analyzing all available 26 board configuration for Android 6 release 1.](image-url)
From the names of configuration constants in Figure 15 we can draw the following conclusions:

First, several of those constants that are highly variable depict partition sizes or block sizes with numbers (*_PARTITION_SIZE, *_BLOCK_SIZE). Second, some of these configuration constants are used for implementing module replacement and therefore specify include directories (e.g. BOARD_SEPOLICY_DIRS or *_BDROID_BUILDCFG_INCLUDE_DIR). Third, many constants refer to the CPU or architecture of the target device (TARGET_ARCH_VARIANT, TARGET_CPU_ABI, TARGET_ARCH, TARGET_2ND_ARCH_VARIANT). Thus, this seems to be one of the major source of variability among the different board configurations.

Looking at the naming pattern of constants, it seems like at one point a naming pattern was chosen (e.g. prefix TARGET_* for specifying properties of the target device or BOARD_* for encoding that this constant is defined in the board configuration). But through the evolution of the system the configuration files got cluttered with new constants that did not follow this rule (e.g. OVERRIDE_RS_DRIVER or MAX_EGL_CACHE_SIZE) which makes any automated analysis were difficult.

### 4.1.2.2 Configuration Impact

The logical next question to ask is:

1. Which of these configuration constants has the highest impact?
2. Where do these configuration constants have an impact? I.e. is variability resolved in global Makefiles or locally in individual modules?

To answer these questions, we searched all of Android’s Makefiles for the names of the board configuration constants. Obviously, a true analysis of Makefiles (including scanning and parsing) is not feasible within this project. Therefore, we rely on a purely textual search here making use of regular expressions based search programs like grep.

In this analysis, we differentiate between three locations within Android. First, the device folder which contains (besides the configuration) vendor-specific modules and apps. Second, the build folder which contains global build files that direct the overall build process and define reusable constants and functions. Any other Makefile must belong to ordinary Android modules which form our third category. By making this distinction, we can infer whether constants have a global impact (if they are referenced mostly in global build files) or whether
the impact is distributed among multiple modules (if they are mostly referenced in local module-specific build files).

The result of our analysis is depicted in Figure 16 in a stacked area plot. The total amount of area colored for a specific constant is the total amount of references of this constant. This number is then broken down into the individual categories DEVICE, BUILD and MODULE by using different colors.

Figure 16: Textual References of board configuration constants in Androids build files. The figure distinguishes between build files in the device-folder (DEVICE) which belong to vendor-specific modules, build files in the build-folder (BUILD) and files in individual Android modules (MODULE). Only the 30 most references constants are shown.

The target architecture (TARGET_ARCH) is the most referenced board configuration constant with almost 400 references. It is being referenced in device-specific Makefiles, global Makefiles but mostly in the Makefiles of individual Modules. Followed by the target board platform (TARGET_BOARD_PLATFORM) which is referenced about 200 times. Interestingly, this constant is only referenced in device-specific modules (DEVICE) and local Android modules (MODULE) and not in global Makefiles. With about 100 references, the third most referenced constant is the second target architecture (TARGET_2ND_ARCH) which has been introduced for 64-bit Android builds and is referenced mostly in global Makefiles.

From this analysis, we can draw the following conclusions:

1. Only a small set of constants have a large amount of references throughout the Android build system while the great majority only have very few references. The amount of references follows a power law distribution, a phenomenon that also describes the number of links on the internet, file sizes, etc.
2. The target architecture and board platform are by far the constants with the highest impact, mostly on individual modules.
3. The majority of references are in device-specific or regular Android modules (~70% of all references). Therefore, the architecture and board platform variability in Android has to be resolved in individual modules.
4.2 Variability Mechanisms in Configuration / Build System / Source Code

4.2.1 Qualitative Analysis

4.2.1.1 Overview

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Conditional Compilation</th>
<th>Conditional Execution</th>
<th>Module Replacement</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Build System</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Source Code</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

4.2.1.2 Conditional Compilation

Build System

```
ifeq ($(TARGET_ARCH),arm)
LOCAL_SRC_FILES += \
src/asm/pvmp3_polyphase_filter_window_gcc.s \ 
src/asm/pvmp3_mdct_18_gcc.s \ 
src/asm/pvmp3_dct_9_gcc.s \ 
src/asm/pvmp3_dct_16_gcc.s
else
LOCAL_SRC_FILES += \
src/pvmp3_polyphase_filter_window.cpp \ 
src/pvmp3_mdct_18.cpp \ 
src/pvmp3_dct_9.cpp \ 
src/pvmp3_dct_16.cpp
endif
```

Figure 17: Example for the use of conditional compilation in Makefiles on a file level. Example taken from frameworks/av/media/libstagefright/codecs/mp3dec/Android.mk

Typically, conditional compilation uses a preprocessor to insert or remove code parts before the compilation process. Android does not use a preprocessor for Makefiles. However, they employ conditional execution to control which source files will be compiled.

Figure 17 depicts an example for the use of conditional compilation in Makefiles. The variable TARGET_ARCH (line 1) refers to the target architecture which is defined in Androids board configuration. Based on the target architecture, this Makefile assigns which source files gets compiled (by assignment to the LOCAL_SRC_FILES variable).

Therefore, Android employs conditional compilation to implement file-level variability within its modules.
Conditional compilation is probably the most used technique for implementing small grained variability in C and C++ code. It uses the preprocessor to insert or remove parts of the source code before compilation.

What is remarkable about Android is that the usage of conditional compilation in the source code is not localized to the source code level but crosscutting through build system and configuration. You would expect to find a header file with the definition of preprocessor flags, but Android uses a different technique. Since the configuration layer is implemented in Makefiles, the source code cannot directly access it to derive values for preprocessor constants. Therefore, the build system acts as a translation layer to translate between the configuration and the implementation technique in the source code.

Figure 18 depicts an example for such a case. In the board configuration, a constant called `BOARD_CHARGER_ENABLE_SUSPEND` is defined and set to `true` which controls whether the charger is allowed to go into a suspend state. In order to implement this variability using conditional compilation in the source code, the preprocessor flag

```
ifeq($(strip $(BOARD_CHARGER_DISABLE_INIT_BLANK)),true)
LOCAL_CFLAGS += -DCHARGER_DISABLE_INIT_BLANK
endif
ifeq($(strip $(BOARD_CHARGER_ENABLE_SUSPEND)),true)
LOCAL_CFLAGS += -DCHARGER_ENABLE_SUSPEND
endif
```

```
#ifdef CHARGER_ENABLE_SUSPEND
static int request_suspend(bool enable)
{
    if (enable)
        return autosuspend_enable();
    else
        return autosuspend_disable();
}
#else
static int request_suspend(bool /*enable*/)
{
    return 0;
}
#endif
```
CHARGER_ENABLE_SUSPEND is conditionally added through the build system by defining a compiler flag (the flag “-D” is equivalent to #define in a source file). The preprocessor then chooses the right implementation for the method request_suspend based on if the preprocessor constant CHARGER_ENABLE_SUSPEND is defined. Therefore, Android employs conditional compilation to implement small grained variability based on the configuration files by conditionally introducing preprocessor flags via the module Makefiles.

Example 2

Figure 19: Example for the usage of conditional compilation in the source code to implement (module-) local variability. Example taken from external/wpa_supplicant_8/wpa_supplicant/

Figure 19 depicts an example for the usage of conditional compilation to implement (module-) local variability. While in the previous example variability has been imposed by Android’s configuration, in this example the module is being configured locally independent of Android’s configuration. The file Android.config defines a high level configuration where each feature can be turned on or off by assigning “y” or “n” to its variable. In contrast to Android’s variability management where the abstraction to the level of features is inherently missing and any semantics of configuration constants needs to be reverse-engineered, this module allows a high level configuration where each feature is explicitly documented. The dependencies between features and the mapping to the feature implementation is implemented in the Android.mk file.

For example, the feature CONFIG_P2P is selected in the Android.config file for the current configuration. The Makefile Android.mk conditionally adds source files for compilation based on the feature selection.

This example shows that individual modules like wpa_supplicant_8 (Figure 19) can have a more systematic way of handling variability than the overall Android codebase. This is due to the fact that Android uses a lot of external open source software in its codebase. The example shown is clearly itself a product line that has been integrated into the Android code base as a core asset.
### 4.2.1.3 Conditional Execution

#### Configuration

**Example 1**

Figure 20 depicts an example for the usage of conditional execution in Android’s configuration files (specifically the board configuration). This example depicts the usage of dex-preoptimization, which is an optimization that enables a fast boot time for Android OS and Android apps at the cost of additional memory usage. This feature is only available if Android is built on a Linux machine and the target build variant is set to “user” (which is a release build; an alternative build variant would be “debug”). This configuration constraint is implemented in the board configuration using the conditional execution mechanism of Makefiles. Therefore, Android uses conditional execution at the configuration level to implement dependencies between configuration items (TARGET_BUILD_VARIANT and WITH_DEXPREOPT) and between the build environment (HOST_OS).

**Example 2**

The second example depicts a different usage of conditional execution in Android’s configuration file (specifically the board configuration). Android’s configuration is not implemented in a single configuration file, but Android divides its configuration into two layers, namely the product configuration and board configuration. The example in Figure 21 shows how Android uses conditional execution to implement dependencies among these layers. There exists a variant of the LG Nexus 5 device (codename “Hammerhead”) that has a fingerprint sensor. Android handles this variability by defining a specialized configuration for this device which inherits from the configuration of the basic device and adds the fingerprint functionality. However, both devices share the same board configuration. Therefore, any fingerprint related configuration constants in the board configuration need to be tied to the fingerprint product configuration. Android implements this constraint using conditional execution by comparing the name of the chosen target product (Figure 21, line 1).

Therefore, Android uses conditional execution in its configuration files to implement dependencies between its different layers of configuration (i.e. board and product configuration).
Build System

We have already seen multiple examples of conditional execution in Android’s build system in previous sections. For example, Figure 18 shows how conditional execution is used to connect the configuration with the variability implementation technique (in this example conditional compilation) by introducing preprocessor constants based on the value of the configuration constant BOARD_CHARGER_DISABLE_INIT_BLANK.

Source Code

```c
1. // This function checks for the parameters which can be offloaded.
2. // This can be enhanced depending on the capability of the DSP and policy
3. // of the system.
4. bool AudioPolicyManagerBase::isOffloadSupported(const audio_offload_info_t& offloadInfo)
5. {
6.  […]
7.  // Check if offload has been disabled
8.  char propValue[PROPERTY_VALUE_MAX];
9.  if (property_get("audio.offload.disable", propValue, "0") == 0) {
10.     if (atoi(propValue) != 0) {
11.        ALOGV("offload disabled by audio.offload.disable=%s", propValue);
12.        return false;
13.     }[…]
14.  }
```

Figure 22: Example for the usage of conditional execution in source code to implement variability with respect to a configuration/policy file

Figure 22 shows an example for the usage of conditional execution in source code for implementing variability. The function `isOffloadSupported` (line 4) refers to a feature called hardware-offloaded audio processing. This term describes the possibility of performing the main audio processing tasks outside the computer’s main CPU. Whether this feature is turned on or off is defined in a property file which is loaded at runtime and accessed using the key “audio.offload.disable” (line 9). The value of this entry is used in conditional statements (line 10) and affects the functions behavior.

While we have seen the implementation of build-time variability (i.e. variability which gets bound during the build process) using conditional compilation a lot, Android makes use of conditional execution for implementing runtime variability via property/policy files in source code.

4.2.1.4 Module Replacement

Build System

While module replacement is mostly associated with include directories in C/C++ files it can also be used in the build system for conditionally including Makefiles. Figure 23 shows an example for such a case in Android 6. The content of the top-level Android.mk Makefile for the module GPS is shown. GPS has several submodules called msm8960, msm8974, msm8084 that implement different GPS sensors for different board platforms. The compilation process
is being directed to the correct GPS submodule based on the chosen target board platform (conditional statements in line 3,6,9) by including all Makefiles of the submodule (include statement in line 5,8,11). The variable TARGET_BOARD_PLATFORM has been defined in the board configuration and describes the chosen board platform.

Interestingly, there are further constraints that might prevent the GPS module from building any of its submodules. Namely, the variable BOARD_VENDOR_QCOM_GPS_LOC_API_HARDWARE must not be empty and the variable BOARD_VENDOR_QCOM_LOC_PDF_FEATURE_SET must be set to true. These constraints are not visible at the configuration level where they are defined and make the configuration process for a new device an error prone and tedious task.

### Source Code

#### Configuration

```
ifeq ($(TARGET_PRODUCT), bt_shamu)
  BOARD_BLUETOOTH_BDROID_BUILDCFG_INCLUDE_DIR := device/moto/shamu/bluetooth_extra
else
  BOARD_BLUETOOTH_BDROID_BUILDCFG_INCLUDE_DIR := device/moto/shamu/bluetooth
endif
```

```
# Setup bdroid local make variables for handling configuration
ifeq ($(BOARD_BLUETOOTH_BDROID_BUILDCFG_INCLUDE_DIR),)
  bdroid_C_INCLUDES := $(BOARD_BLUETOOTH_BDROID_BUILDCFG_INCLUDE_DIR)
  bdroid_CFLAGS += -DHAS_BDROID_BUILDCFG
endif
```

#### Build System

```
ifdef HAS_BDROID_BUILDCFG
  #include "bdroid_buildcfg.h"
endif
```

---

The first thing to notice about the module replacement example in Figure 24 is that this variability implementation mechanisms cuts across all layers similar to the example for conditional compilation in Figure 18. The example is concerned with Androids Bluetooth module (system/bt) which uses a configuration file in form of a c-code header file that differs among different products. This variability is resolved by means of module replacement.
The include-directory is conditionally defined at the configuration layer (variable `BOARD_BLUETHOOOTH_BDROID_BUILDCFG_INCLUDE_DIR`) based on the chosen target product. The content of this variable is then specified as an include directory in the build system by assigning it to `bdroid_C_INCLUDES`. This will later generate a compiler command using the –I flag for defining an include directory. Additionally, a preprocessor constant `HAS_BDROID_BUILDCFG` is introduced through the build system that indicated whether an include-directory has been defined at all. This way, module replacement can also be used to implement optional variability opposed to just alternative variability. Based on this variable, the source code includes the header file “`bdroid_builcfg.h`” which resides in the include-directory introduced via the build system.

Interestingly, while in the previous example in Figure 23 we highlighted the inappropriate implementation of constraints on the module level which make configuration a difficult task, in this example the issue is resolved by shifting the variability to the configuration level where it belongs.

4.2.1.5 Inheritance

Configuration

![Inheritance tree of product configurations of Android 6. The leaf nodes represent the final configurations for a device. The configurations inherit from their parent nodes (incoming edges). The red circle highlights the configuration for HTC Nexus 9 32- and 64-bit versions.](image)

Androids configuration layer makes great use of inheritance to reduce the complexity of configuration files. Figure 25 depicts the inheritance tree of Androids product configuration that are part of the Android 6 source code. The leaf nodes at the bottom represent final configuration files whereas all other nodes represent intermediate configuration files. Most of these intermediate files are provided by Android to support vendors in configuring Android. However, vendors also make use of this inheritance mechanism for their own purpose and use it to encode commonality and separate variability. For example, the two nodes highlighted in red represent the product configuration for HTC Nexus 9 as a 32-bit...
version and a 64-bit version, both inheriting from their parent node that encapsulates their commonality.

In the following we will present some concrete examples from different product configuration files.

**Example 1**

```
1. PRODUCT_PACKAGES := fingerprint.hammerhead
2. PRODUCT_LOCALES := en_US
3. $(call inherit-product, device/lge/hammerhead/aosp_hammerhead.mk)
4. PRODUCT_NAME := aosp_hammerhead_fp
5. PRODUCT_DEVICE := hammerhead
```

This example refers to the variant of the Nexus 5 device (codename “Hammerhead”) that contains a fingerprint sensor. The implementation of this variability in Android’s board configuration has already been presented as an example for conditional execution in Figure 21. In this example, we see how this variability is resolved in the product configuration using inheritance.

For managing this variability, Android uses a common product configuration for Nexus 5 devices that contains their commonality. For implementing variability, a new product configuration is created that inherits from this configuration and adds any additional information. The product configuration of the Nexus 5 variant with fingerprint sensor is depicted in Figure 26. In this variant, the module implementing the fingerprint sensor capability is added for compilation (line 1). The common settings are inherited from the parent product configuration (line 3) and the product name is set accordingly (“_fp” refers to “fingerprint”). Therefore, Android make use of inheritance at the configuration level to separate variability and commonality.

**Example 2**

```
1. # Inherit from the common Open Source product configuration
2. $(call inherit-product, $(SRC_TARGET_DIR)/product/aosp_base_telephony.mk)
3. PRODUCT_NAME := full_hammerhead
4. PRODUCT_DEVICE := hammerhead
5. PRODUCT_BRAND := Android
6. PRODUCT_MODEL := AOSP on HammerHead
7. PRODUCT_MANUFACTURER := LGE
8. PRODUCT_RESTRICT_VENDOR_FILES := true
9. $(call inherit-product, device/lge/hammerhead/device.mk)
10. $(call inherit-product-if-exists, vendor/lge/hammerhead/device.mk)
```

Figure 27 depicts the product configuration of a basic Nexus 5 device and demonstrates how inheritance at the configuration level can also be used for simplifying the configuration files. First, the configuration inherits from a base configuration provided by Android (*aosp_base_telephony*, line 1) that adds basic functionality common to all devices and has its own large inheritance tree. Next, some basic properties of the device are defined (line 3-8)
and the detailed configuration is inherited from a separate file (device.mk, line 9). Finally, some vendor specific configuration settings might be added if such file exists (line 10). This example showed how inheritance among configuration files can be used to simplify individual configuration files by implementing the principle of separation of concerns.

Source Code

Figure 28: Example for the usage of inheritance in source code to separate commonality and variability

Figure 28 depicts an example for the usage of inheritance in source code. Similar to the usage of inheritance in configuration files (cf. Figure 26) the purpose is to encapsulate commonality in a common base class and separate variability in individual subclasses. The example shows the implementation of different sensors of different manufacturers in Android. The commonality among these sensors is captured in a common base class SensorBase. Each sensor inherits from this class to implement its specific features. Hence, variability and commonality are separated using inheritance.
Summary

Table 6: Summary of the results obtained by qualitative investigation of variability realization techniques in Android on different layers. * These mechanisms are not localized to a single layer but are crosscutting through all layers

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Layer</th>
<th>Conditional Compilation *</th>
<th>Conditional Execution</th>
<th>Module Replacement *</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td></td>
<td>Implement dependencies among configuration constants and build environment and between different layers of configuration</td>
<td></td>
<td></td>
<td>Simplification of configuration files &amp; separation of commonality and variability</td>
</tr>
<tr>
<td>Build System</td>
<td></td>
<td>Implement file-level variability</td>
<td>Connect configuration to variability implementation technique</td>
<td>Implement variability at the granularity of submodules</td>
<td></td>
</tr>
<tr>
<td>Source Code</td>
<td></td>
<td>Implementing small-grained statement/function level variability</td>
<td>Implementation of runtime variability</td>
<td>Implementation of variability at the granularity of files</td>
<td>Separation of commonality and variability</td>
</tr>
</tbody>
</table>

4.2.2 Quantitative Analysis

4.2.2.1 Conditional Compilation in Source Code

In the chapter on qualitative analysis of variability realization techniques we have already presented examples on how variability is realized using conditional compilation. In this chapter, we will gather quantitative data on the usage of conditional compilation in source code to answer the following questions:

- **Q1:** To what extend is conditional compilation used in Android’s source code?
- **Q2:** To what extend is it used to implement variability?
- **Q3:** What is the distribution of variation points among the Android code?
- **Q4:** Which constants have the highest impact?

The process for analyzing the usage of conditional compilation is depicted in Figure 29. In this analysis, we focus on conditional compilation in source code (C/C++ files) which is typically implemented using the C/C++ Preprocessor. The analysis is split into two parts: (1) analyzing Preprocessor usage and (2) tracing Preprocessor constants to their definition. For (1), we extract the variation points introduced through conditional compilation using the VITAL tool [4] and then calculate the file scattering per constant to reason about the extent to which conditional compilation is used (Q1). But since the preprocessor is also used for other purposes like constant or macro definition, not all of these variation points correspond to the implementation of real variability. Since Android’s configuration files are actually Makefiles,
Figure 29: Process for analyzing the usage of conditional compilation in Android’s source code. We use the VITAL tool to extract variation points in source code introduced by the preprocessor and trace these constants to their definition in code or Makefiles.

their configuration constants can only be referenced in other Makefiles. This means, that any Preprocessor constant that is defined based on Android’s configuration, must only be introduced via Makefiles. As a result, we consider only those variation points as real variability, whose preprocessor constants have been introduced via Makefiles. This motivates part (2) of our analysis where we trace each Preprocessor constant found in source code, to its original definition (i.e. in source code or Makefiles). For identifying definitions in source code, we use a simple string-based search through all source code files and for Makefiles, we use the Kati tool to look for Preprocessor definitions introduced via the –D compiler flag. When filtering out those constants which are defined in source code, we can reason again about the distribution of variation points and the impact of individual constants by calculating the scattering degree (Q3) (Q4). In the following, each analysis step is explained in more detail.

Extracting Preprocessor Variation Points

As a first step, we need to extract the usage of conditional preprocessor directives like #ifdef, #ifndef. We use the VITAL tool [4] to automatically extract all variation points in Android 6 introduced through the preprocessor. As a result, we found 334744 variation points (excluding header include guards ending with _H or _H_) containing 42531 unique preprocessor constants introduced through conditional compilation. On average, every preprocessor
constant results in 7,8 variation points. Assuming the names of all preprocessor constants are unique throughout the Android source code, we can calculate the file scattering as simply the number of files that have at least one variation point with this constant. The results are visualized in Figure 30. The distribution of the scattering degree shows the same phenomenon we identified in section 4.1.2.2 when we studied the configuration impact of individual configuration constants, namely the heavy-tailed distribution. There are only four preprocessor constants (0,0094%) scattered across more than 1000 files, 24 scattered across 1000-500 files (0,056%) and 42293 constants scattered across less than 100 files (99,44%). For comparison, there are in total 151296 C/C++ files (including header files) in Android 6.

Figure 30: The file scattering of preprocessor constants that were extracted using the VITAL tool. Only the 100 most scattered constants are shown.

Figure 31 offers a different visualization of the same data using a so-called Treemap that shows the scattering degree of each constant encoded as the size of rectangles. This kind of visualization allows for a better comparison of the scattering degree of Preprocessor constants.

Figure 31: The file scattering of preprocessor constants that were extracted using the VITAL tool. Each box corresponds to a preprocessor constant and the size of each box visualized its scattering degree. Only the 100 most scattered constants are shown.
Tracing Preprocessor Constants to their Definition

Figure 32: Summary of part (2) of our analysis. We trace variation points in source code to the definition of their Preprocessor constants which can be either defined in source code or Makefiles.

Figure 32 summarizes part (2) of our analysis. Now that we have extracted all variation points in C/C++ code introduced through the Preprocessor our goal is to identify to what extent these variation points actually implement variability. For this purpose, we trace each preprocessor constant to its definition, which can be either in C/C++ code or Makefiles. In C/C++ code, Preprocessor constants are defined via `#define`. In Makefiles, Android introduces special variable names that contain the compiler flags used for building a compiler command. One of these flags (`-D`) can be used to introduce Preprocessor constants. Since we know that the Android configuration files are really Makefiles and their parameters can only be accessed via Makefiles, we assume that any Preprocessor constant that is used for implementing variability must have a connection to Androids configuration and can thus only be introduced via Makefiles.

For finding the definition of Preprocessor constants in code, we make use of the UNIX programs `grep`\(^4\) and `awk`\(^5\) embedded in a Python script that allow us to search all C/C++ files highly parallelized.

For finding the definition of Preprocessor constants in Makefiles we use a slightly modified version of Kati\(^6\) which is a tool developed by Google that parses and evaluates Makefiles to convert them to the Ninja build system currently used by Android (master-branch). We modified Kati to produce a log entry of each Makefile statement and its evaluation. Since many conditional statements depend on configuration parameters, operating system, build system environment, etc. not all paths are evaluated by Kati. For example, if the expression of a conditional statement `if(expression)` is evaluated to `true`, the `else` part will not be evaluated. We modified Kati to also include those statements into its log that would not be considered by default and evaluate them when possible. This enables us to consider all the Makefiles statements, mostly independent of the chosen configuration. The limitations of this approach are discussed at the end of this section.


\(^6\) [https://github.com/google/kati](https://github.com/google/kati)
The results for tracing Preprocessor constants to their definition is depicted in Figure 33. We differentiate between four different categories: the definition was found in code, the definition was found in a Makefile, the definition was found in code and Makefiles or the definition was found neither in code nor Makefiles. Unsurprisingly, the majority of definitions (64.6%) of preprocessor constants were found in code which confirms our assumption that most of the variation points using preprocessors are not related to variability. A lot more surprising is the fact that for 33.9% of the constants, there is no definition in code or Makefiles. For the remaining 1.5% (i.e. 655 constants) we found that 1.1% are only defined in Makefiles and 0.5% are defined in Makefile and code. The latter may be the result of our general assumption that the name of Preprocessor constants is always unique. The fact that many Preprocessor constants are not defined at all, may be explained by the fact that Android includes many open source projects that are configured for their use within the Android source code. As a result, only those Preprocessor constants are defined that enable the functionality required by Android.

The purpose of the previous step was to identify those constants that are used for implementing variability. As stated before, we assume that only those constants introduced through Makefiles (i.e. the categories Makefile and Code+Makefile from above) satisfy this property. Now that we have identified those constants we can revisit the scattering degree. Figure 34 shows the file scattering only for those Preprocessor constants that are defined via Makefiles.
When comparing Figure 34 showing the file scattering of Preprocessor constants introduced via Makefiles and Figure 30 showing the file scattering of all Preprocessors constants, we notice that those constants introduced via Makefiles are a lot less scattered. First of all, the constant with the maximum scattering degree (introduced via Makefiles) is scattered across 1608 files whereas the most scattered constant in our previous analysis was scattered across almost 10000 files. Secondly, the average scattering degree is also slightly smaller with 4.8 for constants introduced via Makefiles and 5.1 for all Preprocessor constants.

When we have extracted the Preprocessor constants introduced via Makefiles, it is time to revisit our assumption that only those constants are used for variability implementation. When studying the names of constants (cf. Figure 35) we see that at least the most scattered constants do not seem to be related to features. Rather, they are used for encoding technical variability like whether to compile with a debug option (e.g. DEBUG, NDEBUG), how memory is organized (e.g. _LITTLE_ENDIAN, BITS_PER_LONG) or whether to run the static code analysis tool lint. However, we can identify some naming patterns for the less scattered constants. In a systematic usage of conditional compilation for implementing variability one would expect a consistent prefix for Preprocessor constants so they can be distinguished from other constants. For example, an industrial product line from Danfass uses the prefix HAS_FEATURE [23]. And indeed, also for Android we see 160 constants beginning with HAVE_ (e.g. HAVE_MMAP, HAVE_JPEG, HAVE_IPV6) and 28 with the prefix USE_ (e.g. USE_OPENSSL, USE_SIMULATOR, USE_CPUSETS). Overall 28.7% of the constants correspond to this naming pattern. But as we have seen in the examples for the usage of conditional compilation in Android (cf. Figure 18) this must not be the case and cannot be assumed for our analysis.
Limitations of the Analysis Approach

As Make is a Turing complete language, its analysis is never without shortcomings. Existing techniques based on static analysis tend to be tailored to a specific system and cannot be applied for other systems [3]. Our Kati-based approach for extracting data from Makefiles solves this issue since it is independent of a particular system and can therefore be applied to any Make-based system.

On the other hand, Kati is not a static analysis tool. Its purpose is to translate build rules from Make into the Ninja build system. For this reason, it only considers a certain path through Makefiles and ignores those execution paths were conditional statements evaluate to false. In particular, it makes use of a specific configuration and does not explore the complete configuration space. We addressed this issue by forcing Kati to look at as many execution paths as possible. This increases the amount of data we can extract from Makefiles dramatically, but also brings some side-effects.

![Figure 36: Disadvantages of Kati-based analysis of Makefiles. Illustrating the problem of the variable LOCAL_CFLAGS being influenced by multiple execution paths (left) and its possible dependence on configuration constants](image)

Figure 36 illustrates the major shortcomings of our Kati-based solution. LOCAL_CFLAGS is the variable name in Android that is used to store compilation flags (such as \(-DXY\), to add preprocessor constants). On the left, we illustrate the problem of having a different value for LOCAL_CFLAGS due to a variable reference that is conditionally set. Although we extract the knowledge that FLAG_NAME can have two different values based on the value of the constants \(_UNIX\), this knowledge is not incorporated into our analysis yet. Therefore, any variable reference of FLAG_NAME will only yield the value of the last executed assignment, in this case the Else-case. The right side of Figure 36 illustrates the dependence of configuration constants (let us assume BOARD_CONFIG_PATH is defined in the board configuration of an Android device). Since a configuration needs to be chosen before running Kati, all configuration constants are set to a specific value. The variable reference $(BOARD_CONFIG_PATH)$ will yield the value of BOARD_CONFIG_PATH that was set in the chosen configuration. In the case that a configuration constant is referenced in a LOCAL_CFLAGS assignment (i.e. it is introduced as a preprocessor constant), Kati will only use the value of the configuration constant that is defined in the chosen Android configuration and not explore the complete possible configuration space.
Conclusion

We presented an analysis method to extract data on the usage of conditional compilation in make-based systems. We first analyzed the usage of the C/C++ preprocessor to implement variation points using the VITAL tool and found that the distribution of Preprocessor constants among source code files follows a heavy-tailed distribution with only a small fraction being highly scattered. In part two of our analysis we traced the Preprocessor constants used in variation points to their original definition to identify which of these constants are introduced via Makefiles. Since the configuration in Android is written in the form of Makefiles, only those constants might have a connection to Androids configuration and therefore may implement real variability. As a result, we found that the same heavy-tailed distribution for the file scattering persists for those constants introduced via Makefiles, although the maximum scattering degree is much lower as well as the average scattering degree. Finally, even for those constants introduced via Makefiles, there was no heavily used naming pattern and it is therefore impossible to further constrain the amount of constants that relate to feature variability, i.e. whose variation points are actually used to implement variability. As future work, we intend to look at the variation points in Makefiles and focus on those Preprocessor constants that are introduced conditionally, based on some configuration constant. This way we can establish links between Androids configuration and the implementation mechanism conditional compilation in source code and therefore distinguish feature variability from merely technical variability.

4.2.2.2 Conditional Execution in Build System

![Figure 37: The usage of conditional execution in the Android build system. We differentiate between local Makefiles (blue) which belong to individual modules and global Makefiles (orange) which are executed globally independent of individual modules. Different bars denote the usage of different kinds of variables (i.e. board configuration constants, product configuration constants, etc.) and their height indicate the number of variation points. The bar label denotes the number of variables that belong to each variable class in brackets.](image-url)
Androids build system is defined in Makefiles which allow the usage of conditional statements much like a programming language. It supports regular if-statements like `ifeq`, `ifneq` but also statements like `ifdef`, `ifndef` in the style of the C/C++ preprocessor. We will consider all of these statements as the implementation of conditional execution since they change the control flow. Note that even though conditional execution has been presented as a technique to implement runtime variability (cf. section 2.1.3) its usage in Makefiles clearly refers to construction time of the overall system since Makefiles are only executed at construction time.

We analyzed the usage of conditional execution in Android’s build system by parsing a log of Make statements created by our modified Kati tool, like we did in the previous analysis. The results are shown in a bar chart in Figure 37. We distinguish between local Makefiles (blue bar), which belong to individual modules and global Makefiles (orange bar) which reside in Androids `build` folder and have an impact on the overall build process. Looking at the variables used in these conditional expressions, we can identify several different classes of variables. We differentiate between variation points containing board configuration constants, product configuration constants, constants defining the build environment (e.g. operating system) and everything else. We can clearly identify board configuration constants based on the results of our previous analysis in section 4.1.2.1. Additionally, we also considered constants with the prefix “BOARD_” as board configuration constants. Product configuration constants are identified with the prefix “PRODUCT_”. The usage of the constants `HOST_OS` and `TARGET_OS` define the build environment class whose variation points depend on the operating system. Every other variation point falls in the Others class.

From the results in Figure 37 we can make the following observations: First, the majority of variation points reside in local Makefiles which means that binding to a particular variant is done mostly locally for each module and not globally. Second, conditional execution is only used for resolving variability introduced through the board configuration and not variability introduced through the product configuration. The board configuration constants are used in almost 500 variation points, whereas the product configuration constants are used in close to zero variation points. Therefore, variability introduced through the product configuration must be resolved using other mechanisms and possibly at other binding times. Third, the build environment, i.e. the operating system is another major source of variability implemented using conditional execution in 372 variation points. Thus, in a system that is being developed in distributed way by people all over the world with different hardware and software, build environment variability seems to be one major kind of variability.

5 Conclusion & Future Work

In this research project, we have explored the realization of variability in Android – a famous open source software product line. We started in chapter 2 by introducing the necessary terminology and covered foundational knowledge about product lines, variability and binding times. We presented variability mechanisms commonly found in literature and weighted their advantages and disadvantages. In the following chapter, we started to introduce Android, its architecture, configuration and build process. Before going into the analysis on the usage of variability mechanisms we covered the configurability of Android and highlighted its shortcomings. We found a lack of systematic variability management in the sense that there is no definition of possible configuration constants, possible values, dependencies or trace to
any higher level feature model. Nevertheless, we could reverse engineer some of the properties of Android’s configuration. Next, we presented the results of our qualitative research on the variability mechanisms used in Android. We found that many of the mechanisms found in literature are also used in Android, however sometimes for a different purpose. The last part of our project was the quantitative research on the usage of variability realization techniques. We presented an analysis process and its results for Android for the usage of conditional compilation in source code and conditional execution in Android’s build system. In future work, we intend to apply this method to other mechanisms as well such as module replacement.
6 References


