COMPUTERS & SECURITY XXX (2015) 1–16

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Screening smartphone applications using malware family signatures

*Jehyun Lee, Suyeon Lee, Heejo Lee**

Department of Computer Science and Engineering, Korea University, Seoul 136-713, Republic of Korea

ARTICLE INFO

Article history:
Received 1 November 2014
Received in revised form
15 January 2015
Accepted 10 February 2015
Available online xxx

Keywords:
Smartphone security
Android
Malware
Variant detection
Static analysis
Family signature

ABSTRACT

The sharp increase in smartphone malware has become one of the most serious security problems. Since the Android platform has taken the dominant position in smartphone popularity, the number of Android malware has grown correspondingly and represents critical threat to the smartphone users. This rise in malware is primarily attributable to the occurrence of variants of existing malware. A set of variants stem from one malware can be considered as one malware family, and malware families cover more than half of the Android malware population. A conventional technique for defeating malware is the use of signature matching which is efficient from a time perspective but not very practical because of its lack of robustness against the malware variants. As a counter approach for handling the issue of variants behavior analysis techniques have been proposed but require extensive time and resources. In this paper, we propose an Android malware detection mechanism that uses automated family signature extraction and family signature matching. Key concept of the mechanism is to extract a set of family representative binary patterns from evaluated family members as a signature and to classify each set of variants into a malware family via an estimation of similarity to the signatures. The proposed family signature and detection mechanism offers more flexible variant detection than does the legacy signature matching, which is strictly dependent on the presence of a specific string. Furthermore, compared with the previous behavior analysis techniques considering family detection, the proposed family signature has higher detection accuracy without the need for the significant overhead of data and control flow analysis. Using the proposed signature, we can detect new variants of known malware efficiently and accurately by static matching. We evaluated our mechanism with 5846 real world Android malware samples belonging to 48 families collected in April 2014 at an anti-virus company; experimental results showed that; our mechanism achieved greater than 97% accuracy in detection of variants. We also demonstrated that the mechanism has a linear time complexity with the number of target applications.

1. Introduction

Smart devices are currently facing a serious threat posed by the surge in malware. The smartphone has become the most popular target for malware writers since it contains a great deal of user information and has mobile billing capability. The Android platform in particular, occupying the dominant position in smartphone market share (Mawston, 2014), accounted for 97% of all mobile malware in 2013, as reported by F-Secure (Aquilino et al., 2014). Recently, Android malware has increasingly

* *Corresponding author*. Tel.: +82 2 3290 3638.
E-mail addresses: arondit@korea.ac.kr (J. Lee), suyeonl@korea.ac.kr (S. Lee), heejo@korea.ac.kr (H. Lee).
http://dx.doi.org/10.1016/j.cose.2015.02.003
0167-4048/

adopted several obfuscation techniques such as metamorphism and repackaging in order to avoid detection or recognition by the user. This trend is confirmed in a 2014 report by Symantec (Wood et al., 2014) in 2014, which observes that Android malware authors focus more of their efforts on improving existing malware than on creating new malware. Indeed, the top ten Android malware families made up 76% of all Android malware reported during the first quarter of 2014 (F-Secure Labs, 2014). Compared with desktop malware, smartphone malware can cause a more direct invasion of privacy and greater potential for economic damage to users. However, the flood of Android malware variants on smartphones hampers development of efficient strategies for dealing with malware attack. Accordingly, a mechanism that prevents malware by efficiently filtering variants of known malware, is needed to retain smartphone security and user privacy.

Previous approaches for variant detection based on behavior analysis are not suitable for identifying the malware family to which a detected malware variant belongs. These approaches detect variants by assessing the similarity of behaviors such as the frequency or sequence of application programming interface (API) calls (Kwon and Lee, 2012; Aafer et al., 2013; Deshotels et al., 2014), code semantics (Crussell et al., 2013; Suarez-Tangil et al., 2014), and commonly shared byte or string patterns and strings (Faruki et al., 2013a, 2013b; Sanz et al., 2013a, 2013b, 2014), to those of known malware. Extracting and comparing behaviors from large numbers of target executables requires heavy computing overhead. Detection based on behavior similarity is a useful method for covering unknown malware, but it requires disassembling, behavior feature modeling, and complex clustering algorithms to the every inspection target applications. The behavior analysis approaches which use the dynamic analysis methods (Enck et al., 2010; Gilbert et al., 2011; Yan and Yin, 2012) cover more sophisticated variants which are hard to detect by the static analysis methods, but the number of inspection targets need to be reduced by a complement method before the heavy analysis.

The alternative approach often employed by vendors of anti-virus (AV) software using a representative signature is effective in defining and detecting malware families. In contrast to the behavior-based approaches, it is also efficient in terms of time and space complexity. However, the signatures not only have narrow detection coverage of a malware family due to strict decision conditions and naive evidence such as Android application package (APK) names and single class and method name, but also are easily defeated by malware that adopts code obfuscation such as repackaging and metamorphism. Hence, we conclude that a reinvestigation of the overall code for behavior analysis and the construction of an additional signature for a slight modulation of malware are each inefficient ways to improve malware detection when considered against the small effort that is consumed in making a variant.

In this paper, we propose an Android malware detection mechanism that screens new variants of a known malware family out. Most Android malware is a manipulated version of existing malware, and in malware belonging to the same family large portions of code and resources remain unchanged. Exploiting this feature, we detect variants efficiently and accurately by analyzing the representative parts of a family. The proposed mechanism uses a family signature that is common to the family and excludes other families via a weighting factor. The proposed signature structure consists of four parts extracted from a *Dalvik executable* (*DEX*) file, which is the executable file within an APK. The signature consists of the names classes, methods, character strings, and method bodies. A signature has multiple entries in each part, and each entity has an associated weight, according to how well the signature entity represents the identity of the malware family. In other words, the class and method names, hard-coded character strings, and reused codes that appear commonly in family members and rarely in other families have a higher weight.

In experimental evaluation, our mechanism showed high detection performance and low time consumption for variant detection. We evaluated our mechanism using 14,120 Android malware samples collected at an anti-virus company in 2014. These included 5846 family malware belonging to 48 families and 8274 samples of individual malware that were not part of any family or were part of a small family. For our evaluation, we preprocessed and refined malware families and their memberships since different AV vendors that had investigated the malware had given them different family labels. In the experiments using family signatures, our mechanism showed **97% detection accuracy**, with greater than **97% of recall performance** in the Monte Carlo validation. For individual malware detection performance, we compared the family signatures with the individual malware samples; our mechanism detected **1820 (22%)** of the malware samples out of the 8274 individuals in the malware set. This result shows that our mechanism can detect the malware manipulated in various ways such as the modifying package name, class name and part of codes, and code reordering while conventional signature-based approaches are not ordinarily able to detect such variants. Finally, in the scalability evaluation, the family signatures needed only 20 MB to cover the approximately 8000 Android malware samples. In terms of time consumption, the hashed signature matching process took only 10 s on average to screen a thousand applications against a million signature entries in a desktop PC.

Our contribution is twofold:

- We propose a type of Android malware family signature that can be used for accurately and efficiently detecting variants of known malware families. A family signature is a flexible signature for a malware family sharing the class name, method name, character strings, and code bodies of the original malware. It solves an existing malware detection issue by multiplexing decision conditions with multiple signature entries along with their representative weights. This contribution makes it possible to detect malware including the variants, even variants that have adopted an evasion technique such as metamorphism or code modification.
- We have reduced the number of signatures needed. The family signature represents a malware family by a single signature set covering a large number of family members, including newly appeared variants. The family signature consists of binary patterns and character strings shared by members of the same malware family. By estimating similarity to the various known families, our mechanism detects and classifies malware families to a practical

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

degree of accuracy. It enables an efficient response to the exponentially increasing number of malware threats.

The rest of this paper is organized as follows: We start with a definition of the problem of Android malware in Section 2, distinguishing our work from previous approaches, which are given in Section 3. In Section 4, we describe details of the mechanisms and assumptions of our proposal. Next, we present experimental results for our mechanism in Section 5. After analyzing overhead and optimization issues in Section 6, we discuss the advantages and disadvantages compared with conventional approaches in Section 7. We conclude with a summary of our work in Section 8.

2. Problem definition

As shown in a recent statistical report from Symantec (Wood et al., 2014), on the Android platform a large portion of malware consists of variants of other malware. A group of variants that share common features in their code and behavior is called a malware family. The fact that constructing variants is not as difficult as creating a totally new variety of malware has led to the rapid rise of malware entities that belong to families, but at the same time, this provides the means to screen for a large portion of new malware, by using the common features of known malware families.

We address the problem of detection Android malware via malware family detection; to do this successfully, we must overcome their evasion techniques. Hence, we need to know the techniques used by variants to avoid detection, in order to recognize the part of variants that remain unchanged under their evasion tactics. According to one study of obfuscation techniques (Rastogi et al., 2013), several naive malware signatures have used the package name and the identifier names of a malware as a signature, leading malware authors to change this features in order to avoid detection. Repackaging and reassembling have been identified as one of the simplest and most prevalent methods for generating a variant; these methods defeat signatures that merely inspect the package name or checksum of an APK file. In code-level variants, call indirection, code reordering, and junk code insertion techniques evade simple signature approaches that use string matching with an offset or API call matching.

The previous approaches for this problem have a trade-off between efficiency and robustness. Static and dynamic analysis techniques that have been proposed to break these evasion tactics; these techniques are known to be effective, but they require heavy inspection overhead, including human resources, compared to signature-matching approaches. For cases involving several malware programs that have highly advanced evasion tactics, it may be necessary to use sophisticated techniques such as taint analysis, code semantic analysis, and dynamic analysis in a sandbox with a monkey application or a human tester. However, for malware variants that use simpler evasion tactics, screening for as many as possible with a signature-matching approach is a more efficient strategy before the depth inspection techniques. Thus, we propose a generalization of the legacy signature-matching approach that is more robust than the conventional signature approach and at the same time more lightweight than the static and dynamic behavior analysis approaches.

The proposed approach with multiple entry signatures covers the family variants. We term this composite signature a *family signature*. The key concept is based on the awareness that most malware variants use simple evasion techniques such as package renaming, class and method renaming, call indirection, code reordering, and junk code insertion because adopting multiple and more complex evasion techniques for making a variant requires greater resources investment by the malware author. In Section 4, we introduce our family signature generation and malware detection mechanism and show how it extracts the common features in a group of malware samples, how it overcomes the evasion techniques previously discussed, and how it minimizes the signature-matching overhead caused by the generalized matching approach. In Section 5, we go on to show the amount of family malware covered by the family signature, that is, what proportion of new malware can be screened by our approach before undertaking a deeper analysis.

3. Related work

Previous work for Android malware can be classified into static approaches (Kwon and Lee, 2012; Lee et al., 2010; Zhu et al., 2012; Enck et al., 2009; Crussell et al., 2013; Faruki et al., 2013a, 2013b; Suarez-Tangil et al., 2014; Zheng et al., 2013) including conventional signature matching, and dynamic approaches (Enck et al., 2010; Gilbert et al., 2011; Yan and Yin, 2012). The dynamic analysis approaches using taint analysis and API monitoring have the ability to track behavior accurately at runtime. The problem with these approaches is one of efficiency because of time and resources required for the establishment of a virtual environment, for test execution, and for test input because several malicious activities are triggered by a user input or a specific condition which are hard to be automated. The coverage and goal of our work is not exclusive to the dynamic analysis because the malware which adopts high level evasion techniques need to be investigated with the dynamic analysis techniques. Our approach is designed as a front-line filter before dynamic analysis and more sophisticated but heavy static analysis techniques.

The static approaches are mainly focused on the behavior of applications and trainable features of source code and executable files, such as permission sets, code bodies, API calls and their sequences, and code semantics. The static approaches which use permissions or code bodies without any data and control flow investigation are light-weight and scalable. However, their scalability and accuracy are highly dependent on the structure of detection features and the methods how to extract and detect the features from the target applications.

Permission-based approaches are light-weight because they do not need any code inspection and effective to detect malicious capabilities irrespective of variant generation techniques. Zhu et al.'s (2012) permission based approach extracts and uses the combination of common permissions and description texts of an Android malware to estimate the intention of the permissions. This approach overcomes the

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

false detection problem of the other permission based approaches caused by the benign applications which use similar permissions intentionally or accidently. However, because the positive words on the description texts are not mandatory and forgeable without any loss of malicious functionality, it still has a challenge on the false detection and robustness. Another sophisticate permission based approach is Enck et al.'s (2009) study. They use the permission rules for malicious activities and achieve accurate and light-weight detection. However, the permission-based detection rules need to be constructed with plenty understanding and insight on malicious activities of the malware. It means that adding a new rule for a new malicious activity needs another depth investigation. The permission-based approaches and our mechanism share a similar purpose, detecting many codes with much fewer signatures without code investigation. But the practicality can be distinct as the conditions of human resources and knowledge-bases.

Crussell et al. (2013) suggested a semantic similarity based Android malware detection mechanism using Program Dependence Graphs (PDG). Their method, *AnDarwin*, extracts a set of feature vectors from PDGs of an application, and finds the similar application clusters based on the feature vectors. This study achieves a generalized and metamorphic robust detection. However, it naively uses the feature vectors for all methods in an application, and it makes their method inefficient. When a new set of applications are occurred, it needs to perform a heavy vector comparison again with numbers of previous feature vectors. *AnDarwin* may be effective when we don't have any prior knowledge-base, but our method significantly reduces the overheads through making signatures from the family information of the known malware.

AndroidSimilar (Faruki et al., 2013a) and *DroidOlytics* (Faruki et al., 2013b) proposed by Faruki et al. utilize a sequence of bytes as a feature. Both of our proposal and *Faruki* et al.'s work consider focus on the common patterns on code obfuscated variants, but the binary patterns adopted in *AndroidSimilar* and *DroidOlytics* do not have any semantics which make a pattern hard to change for avoiding detection. Because their signature has no dependency with a part of code or instructions, the signatures can be easily avoided without any loss of malicious functionalities and heavy efforts of a malware author by making a little change on the codes and recompiling even though the signatures are the byte patterns which have not been changed on the training set. In contrast, our family signatures consist of the semantic entries which are required to construct a functionality such as the name of classes, methods, packages, libraries and constant printable strings. Changing these semantic entries from the codes requires more effort and more complex obfuscation techniques. Even more, our family signatures are organized by multiple entries per a family signature. The multi-entry signature detects the variants even though they avoid several signature entries by code obfuscation. However, the idea of robust feature selection mechanism applied in *DroidOlytics* is distinguished and can be applied to our future studies to enhance family classification accuracy.

Sanz et al. proposed a string analysis based anomaly detection methods (Sanz et al., 2013b, 2014) using machine learning techniques. In their studies, they also use the printable strings from DEX and permission file, and they give weights to the strings using Term Frequency – Inverse Document Frequency (TF–IDF) scheme for emphasizing representative strings. Their approaches share several concepts to us, but the considering the malware family as a unit of detection used in our proposal leads better detection performance than considering every malware as one class, which is applied in Sanz et al.'s. In the Sanz et al.'s another study, *MADS* (Sanz et al., 2013a), which considers multi-class classification and clustering techniques achieves a higher detection accuracy but still has room for improvement on the false positive problem.

Another code inspection approach is analyzing control flow of an application. The control flow analysis approaches have been proposed at the legacy PC environment such as code graphs proposed by Lee et al. (2010) and *BinGraph* by Kwon and Lee (2012). However, the code semantic approaches are more effective in the Android platform than the PC environment because the Dalvik bytecode used for the Android applications has much simpler diversity in their code and semantics. *Dendroid* proposed by Suarez-Tangil et al. (2014) used a basic block of DEX bytecodes as a code chunk and made a control flow graphs (CFG) to characterize Android malware families. Gascon et al. (2013) made a call graph using the function calls. *Dendroid* and Gascon's work use more specified features than our approach, but extracting and using the CFG and the call graph as a signature have to be complemented for several evasion techniques affecting the control flow like code indirection.

The API call-based approaches such as Zheng et al.'s (2013) approach, *DroidAPIMiner* (Aafer et al., 2013), and *DroidLegacy* (Deshotels et al., 2014) are widely studied and have practical detection accuracy and family classification ability. The API call sequence is one of the inherent feature which is relatively harder to obfuscate than code bodies, however, extracting and modeling API call and their sequences requires overhead for disassembling and API call generalization.

Compared with our previous approach (Lee et al., 2013) which extracts the suspicious method bodies and string features as a behavioral signature through a dynamic analysis, the major difference is that the proposed mechanism separates and enhances the static analysis part which detects the Android malware with a family signature. The proposed family signature approach fully automated family signature construction and detection mechanism. And it achieves more optimized and efficient Android malware detection. The proposed mechanism adopts a signature filtering technique and an enhanced detection mechanism overcoming the overhead caused by numbers of signature entries. This improvement increases the detection efficiency keeping the accuracy.

As a conclusion, previously proposed approaches have focused the core of the Android malware, and many of them are accurate and effective but not so efficient. The problem we attempt to solve through our study is that we do not need to inspect every new Android malware through those heavy methods if we already have enough knowledge base. Our mechanism addresses a specific position in front of code and API call level inspection which needs disassembling and heavy modeling and clustering algorithms. By generalizing the conventional signature matching to give flexibility against

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

simple evasion techniques, our mechanism detects significant portion of malware.

4. Detection of malicious applications using family signatures

In this section, we introduce an Android malware detection mechanism that uses a malware family representative signature. A family signature which is a group of binary patterns covering a group of Android malware entities clustered as a family.

4.1. *Mechanism overview*

Our detection mechanism is a generalization of the signature-based approach. On a smartphone, the number of inspection targets (i.e., applications) is steeply increasing, and the majority of new malware entities are malware variants. The legacy signature-based approach scans many applications in a timely manner. However, since the success of the signature-based approach is highly dependent on the specific signature, it is not robust against malware variants. Therefore, we add flexibility to the signature by composing multiple signature entries, thereby overcoming the weakness of the signature-based approach.

Fig. 1 shows the overall architecture of our proposed mechanism. The mechanism consists of a signature construction phase and a detection phase. Once a family signature is constructed, the detection mechanism uses the signature to investigate the Android applications, without any additional signature training.

The key feature of our signature structure compared to the other Android malware detection approaches is that it uses signature entries to estimate similarity to a family rather than simply scanning a file for the existence of an entry. In terms of the signature entry, exactly matched binary patterns, especially bytecode instructions, support high detection precision, and the combination of multiple signature entries complement the variant detection ability. In addition, we estimate the significance and representativeness of each entry and assign it a weight according to its relative importance in the determination of a detection.

To detect malware variants belonging to a family, we use signature entries consisting of four types of binary patterns; class name, method name, character string, and method body. Except for the method body, each entry consists of printable character strings. The method body consists of *Dalvik* bytecode instructions which are compiled instructions. In the Android environment, an application's executable code is a *Dalvik executable* (*DEX*) file, and we extract all four types of signature entries from the DEX file.

Malware detection is performed by a simple and lightweight process using the static family signatures. The malware family classifier decides whether a target DEX file belongs to a known family, using a detection metric that estimates the similarity between a target DEX file and each family signature. We devote the rest of this section to explaining the process in detail and describing the signature construction, similarity calculation, and family classification mechanism.

4.2. *Android malware family signature*

A family signature represents a malware family; in other words, it is capable of determining whether the target application is a new member of a known malware family.

4.2.1. *Family signature structure*

A family signature consists of the four categories of main entries; class name, method name, method body, and character strings. As discussed in Section 2, the purpose of the family signature is to detect malware variants that are detectable with a static signature while overcoming simple evasion techniques such as repackaging, reassembling, string renaming, code reordering, and junk code insertion. To address repackaging and reassembling, we check the inside of the DEX code and use the four categories of features as a signature. The features that we use for a family signature and the reasoning for each are detailed as follows.

- **Class names** and **method names**. Our signature first checks the class names and method names irrespective of their order and position. One class name and method name may not indicate malware, but a set of identically matched names is more indicative of a positive detection.
- **Character strings**. Character strings are a form of signature typically used in past approaches. A legacy signature containing only one character string and its offset in a DEX file is easily evadable by the techniques of code reordering and

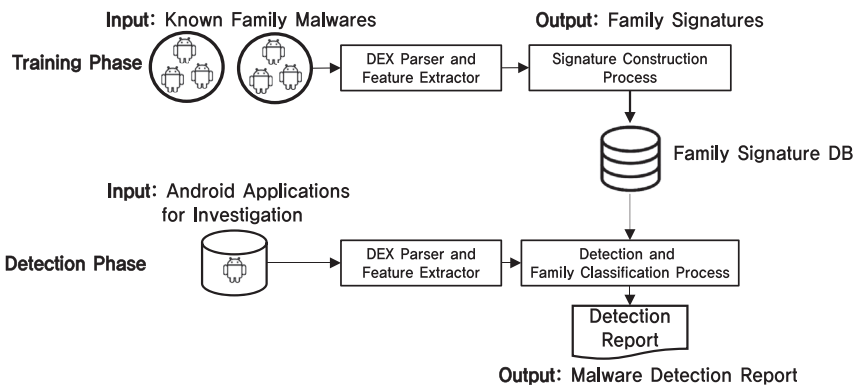

Fig. 1 – Overall architecture of proposed Android malware detection mechanism.

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

string encoding. However, our signature extraction mechanism, which finds statistically common and multiple character strings is robust to many kinds of reordering and encoding if the variants share identical strings.

- **Bytecode bodies.** By including bytecode bodies of each method, a signature is robust against the code reordering, string renaming, and call indirection as well as to repackaging. The reassembling tactic causes the bytecode bodies to change, but several simple methods keep their bytecode body because of the low diversity of Dalvik Bytecode. To address junk code insertion, we divide a bytecode body by the continuous nop operation. It is still evadable via a garage code insertion but can cover those variants that were generated without going to great effort.

A family signature consists of multiple entries, and we termed each entry as a *signature entry*. Each signature entry is stored in hashed format and has an associated weight ranging from 0 to 100 which means the percentage. Of the four categories, the character strings and bytecode bodies may have codes too lengthy to store and match. However, the hashed signature entries have a static size, and the total size of a family signature is only dependent on the number of signature entries. Fig. 2 illustrates the structure of a family signature, but the implementation of signature management and storage in a database is not limited to a particular example. Table 1 gives a brief description of each structure field, and Table 2 is an example of a family signature, which is a part family signature of the *Plankton* family. An entry "Ljava/util/regex/Matcher" in the class name category with the weight 80 means that the class was used at the 80% of the training set which are classified into the *Plankton* family. An entry with a higher weight is the more effective and representative for detection, but it does not means that the entries with a lower weight are unnecessary. For example, detecting two entries with weight 50 lead more reliable detection than detecting only one entry with weight 80.

The *Plankton* family is known that it performs collecting the browser history and device status, modifying the browser's bookmarks, and connecting a remote server to download and install a file. As shown in Table 2, our family signature includes the classes, permissions, and key terms to perform the

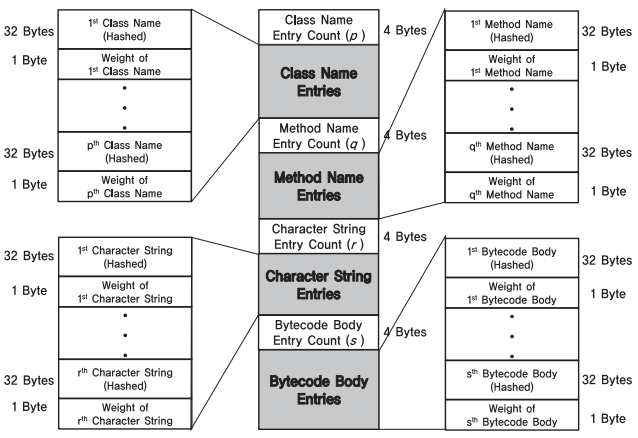

Fig. 2 – Structure of a family signature with hashed entries.

Table 1 – Description of signature fields.

Field name	Description
# of entries	The number of entries in a catergory
Signature entry	A pattern body in a 32 bytes hashed form for pattern matching
Entry weight W	The weight of above signature entry between 0 and 100 for estimating a cumulated credit

set of known malicious activities of *Plankton* even though the entries are automatically selected.

4.3. *Family signature construction*

Signature construction begins by classifying the malware families that are used as the signature training set. The signature construction process extracts the binary patterns and character strings from known malware and calculates the weight of each pattern and string according to its contribution in representing the family to which it belongs. Lastly, the signature refinement process removes those redundant signature entries that would adversely affect the accuracy or efficiency.

4.3.1. *Malware family definition*

The accuracy of our detection mechanism is significantly affected by the definition of a given malware family. Because a family signature is a set of common patterns within a malware family, it is hard to determine a family signature and identify a member malware if the family members rarely share patterns. In practice, a misclassified family member in the training set causes corruption in the family signature. Thus, we utilize the family classification information labeling from multiple AV vendors as the ground truth for our signature construction and evaluation. From the malware family names reported by 52 distinct AV vendors, we assessed the family membership of each Android malware sample through a family definition process, as illustrated in Fig. 3.

We classify malware samples into their respective families based on their malware labels. Because a malware label has various types of information such as platform, infection method, major functionality, variant label, and hash value like *Android.Trojan.YZHCSMS* and *AndroidOS/GenBl.BA6A6AF0*, we extract key terms, which are used for recognizing a malware family, from the full label. Because each AV vendor uses different key terms even though they are semantically equivalent from a human perspective, we adopt heuristic rules in order to make them consistent. For example, the family name *basebridge* is given to various forms such as *bbridge*, *basebridg*, and *basbridge*. To avoid misclassification, we apply the heuristic rules only in obvious cases.

After key term extraction and refinement, we classify malware families according to the most frequently assigned key term of each malware. For example, if a malware entity is labeled *Plankton* by 15 AV vendors and as *Airpush* by 5 AV vendors, we classify it as the *Plankton* family. All the malware having the same best-key terms are considered to belong to the same family. The empirical result of family classification is listed in Table 3.

Table 2 – An example of *Plankton* family signature.

Category	Signature entry	W
Class and method name	Ljava/util/regex/Matcher;	80
	Ljava/util/regex/Pattern;	83
	Lcom/apperhand/common/dto/ApplicationDetails;	99
	Lcom/apperhand/common/dto/BaseDTO;	99
	Lcom/apperhand/common/dto/Bookmark;	99
	Lcom/apperhand/common/dto/Build;	99
	Lcom/apperhand/common/dto/Homepage;	98
Plain string	9774d56d682e549c	82
	android.permission.ACCESS_COARSE_LOCATION	81
	android.permission.ACCESS_FINE_LOCATION	81
	android.permission.ACCESS_NETWORK_STATE	88
	android.permission.ACCESS_WIFI_STATE	89
	android.permission.INTERNET	94
	android.permission.READ_PHONE_STATE	91
	com.android.browser.permission.WRITE_HISTORY_BOOKMARKS	96
	com.startapp.android.APP_ID	90
	com.startapp.android.DEV_ID	90
	CRoQAlVGS1keGVoEHgRLEBoOGRdLEUE+agQtJzsiJj8tABJOHhYdGw	96
	Didnotconsumetheentiredocument.	93
	Didn'tattemptochangethehomepage	98
	LazilyParsedNumber.java	91
	TgsRHg4aEE8dGwc=	93
	TgwLHQQEBVwCHQ==	97
	Th0MHR0dB1sdHQ==	97
	ThsKFxcZAU0dCxAAFhdLEgYGGB0	97
	http://www.searchmobileonline.com/$CATEGORY$?sourceid=7&q=$QUERY$	94
	sd789rdme4984mx34590345345834cm353890573m45897feryitoet7r89e74545	98
Method body	95000405DF0104FFB561B610B070B090B03098010008D9020820BA20B610	96
	95000406DF0106FFB551B610B070B090B03098010008D9020820BA20B610	96
	97000405B760B070B090B03098010008D9020820BA20B610B0400F00	96
	DF0006FFB640B750B070B090B03098010008D9020820BA20B610B0400F00	96

4.3.2. *Signature construction process*

A family signature is constructed by means of the four steps shown in Fig. 4. Construction starts with the labeled Android malware samples as a training set and follows these four steps.

- DEX extraction: For APKs, we use only a Dalvik executable (DEX) file to construct a signature. Likewise, the detection process also needs only the DEX file. In this first step, the signature construction process extracts the DEX file from the Android malware samples. DEX extraction is a simple task because an Android package is compressed into a simple file compression format.
- Candidate entry extraction: The second step is to extract patterns from the DEX files. From each DEX file, we extract the four kinds of patterns mentioned above. We refer the positions of the patterns which are indexed in the DEX header.
- Entry weight calculation: In order to select useful patterns from the many candidate entries extracted, we give an entry weight for each pattern. The weight of an entry is calculated by the pattern sharing ratio PSR.
- Signature refinement: Lastly, we select the effective signature entries by removing entries according to their weights, leaving a minimized but accurate set of family signatures. After entries are filtered by their weights, we remove redundant entries that are commonly included from different families including the benign applications.

According to the DEX file structure, the class name, method name and character strings are extracted from the resource section; these are pointed to offsets in the DEX header. A method body is a binary string of Dalvik virtual machine (DVM) bytecode, and the bytecode body of each method is also pointed to the offsets.

A weight of a given signature entry represents how commonly this pattern is shared within the family. The weight W is calculated by the PSR which is the percentage of malware samples sharing the same pattern within the training set, as given in Equation (1).

When $M(p,a)$ indicates the number of malware entities having pattern p in their family a,

$$W(p,a) = \frac{M(p,a)}{\sum^{i \in f} M(i,a)} \quad (1)$$

A signature entry can be occurred in multiple families, and it has different weight for each family.

4.3.3. *Signature refinement and redundant signature*

Refinement of the number of signature entries directly affects the space and time consumption for signature storage and matching. Even though a family signature set may be efficient because it covers many family members (more than 50 members according to a statistics Wood et al., 2014), it still has room for improvement. Because some family members that share large portions of code may make too many signature entries in our signature construction mechanism, we need to

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

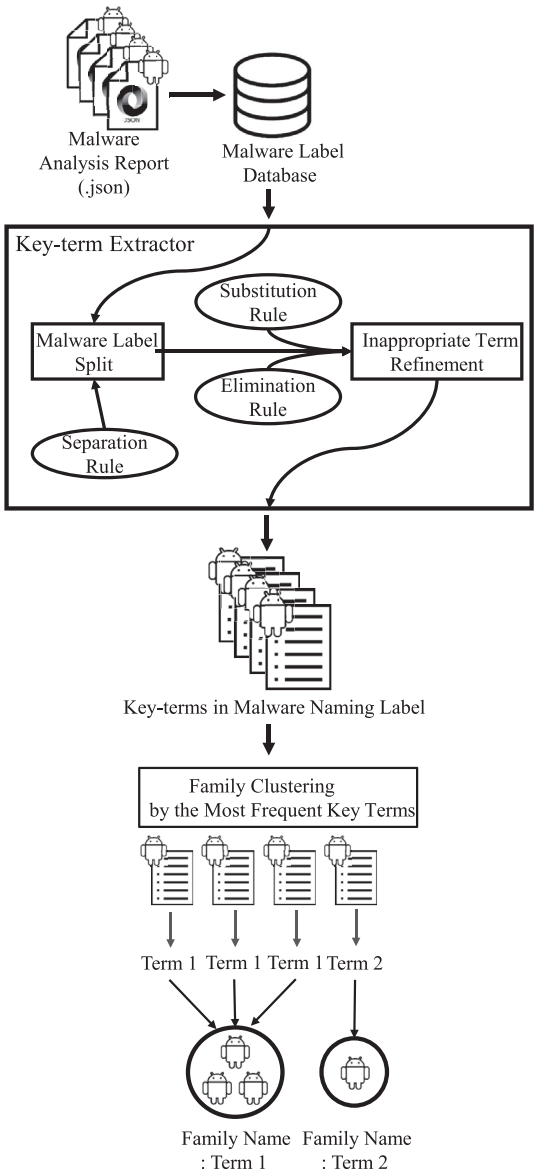

Fig. 3 – Android malware family classification from pre-investigated malware labels by multiple anti-virus (AV) vendors.

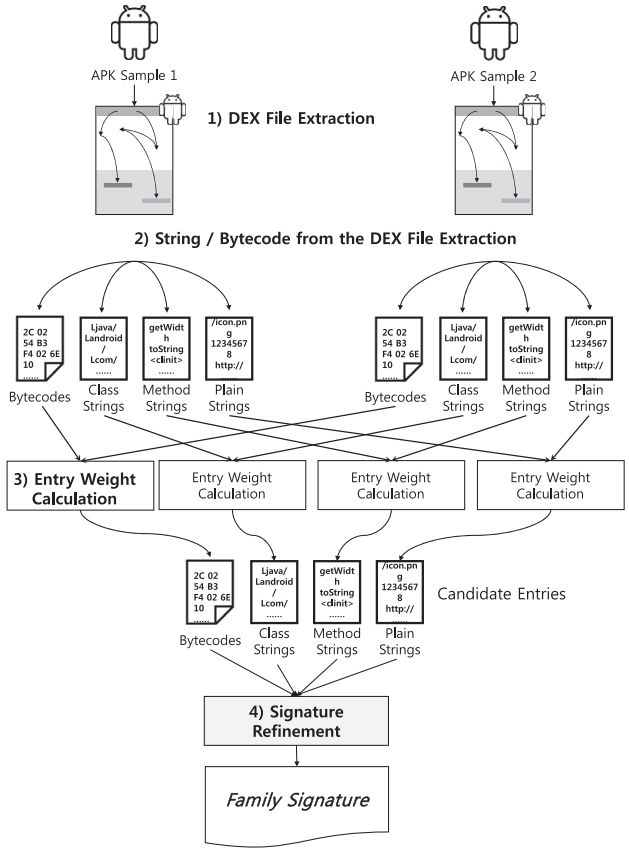

Fig. 4 – Family signature construction process from APK samples.

Table 3 – Top ten malware families classified, by name.

Rank	Family name	# Members
1	plankton	1412
2	kuguo	591
3	stealer	436
3	waps	374
5	mseg	355
6	airpush	369
7	droidkungfu	298
8	gingermaster	292
8	utchi	279
10	dowgin	270
	Total	4676

choose the more effective and efficient signature entries, using selection criteria.

The weight attached to a signature entry is a straightforward criterion to estimate its relative effectiveness; a signature entry having a higher weight normally has a greater effect in the process of identifying a malware family member. However, it is not always true that entries with higher weights lead better detection, because one of the key contributions of a family signature is the complementary efficacy by the other less important signature entries against the evasion techniques that hide critical features. Thus, we empirically evaluate the effect of removing low-weight signature entries on detection accuracy. The goal of the evaluation is to find the most efficient removal point, that which achieves a significant reduction in the number of signature entries while allowing the minimum damage to detection accuracy. For this evaluation, we check the detection accuracy and signature size reduction after stepwise removing those signature entries that have a weight less than a removal threshold T_{psr}.

Another refinement method is removing the common entries appeared in multiple family signatures as well as by benign applications. To address common patterns occurring among distinct families, we adopt a *redundant signature* instead of updating every signature entry weight whenever a new malware family is reported. The redundant signature is a set of commonly detected binary patterns in any categories of Android applications irrespective of the maliciousness. We

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

construct the redundant signature by finding the patterns appearing from the multiple families exceeding a certain threshold. In our experiments in Section 5, the threshold is 2 families out of 4 training families. The redundant signature also includes the common patterns of the benign applications by making a benign family signature considering all of the benign applications as a family. The redundant signature is effective at removing patterns that adversely affect detection accuracy and have little information in family classification. Patterns that are usually selected as the redundant signature are the class, method, and package names of common Android or Java libraries and automatically generated methods that are widely used in Android applications. In our detection and classification mechanism, signature entries in the redundant signature are ignored.

4.4. *Malicious application detection process*

4.4.1. *Family signature matching and classification process*

Our malware detection mechanism is a simple and light-weight process that uses family signatures to find the most similar malware family. The actual comparison process uses hashed values, not the longer binary patterns. Because our mechanism detects only exact pattern match, the signature entries and the patterns extracted from an investigation target application can be quickly compared through their hash values. In addition, the detection process determines whether a pattern in a signature exists in the pattern hashes of a target file in a constant time using a hash map that has a constant search time.

Signatures are loaded into memory in a dictionary data structure, and the detection process uses a dictionary search method to find matching signature entries. The dictionary is a hash table that holds the value for each key. Each signature entry which is a key in the dictionary contains the family name and its weight as a value. A dictionary is a time-efficient data structure for a static set of data consisting of the same data type because it has a search time of $O(1)$. For a more time efficient investigation, the signature entries are separated by their categories for each different types of entry. A pattern type is searched within a dictionary of signature entries having the same type as the target pattern. We analyze the time complexity of the hashed signature matching process in Section 6.1.

Given a target file, the detection process first scan all binary patterns belonging to the four signature types that are placed in a DEX file as same as the signature construction phase. All of its patterns are hashed with the same hash function to compare to the signatures. Next, the detection process obtains a detection metric, the similarity $S(P, A)$ between a pattern set P of a given DEX file and family signature A, by summarizing and normalizing the weights of the signature entries that are hit by the hashed patterns. $S(P, A)$ presents the similarity between a target file and a signature as Equation (2).

$$S(P, f_a) = \frac{\sum_{j=1}^{n} W\left(p_j \middle| p_j \in (P \cap f_a), a\right)}{\sum_{i=1}^{m} W(p_i, a)} \tag{2}$$

$S(P,A)$ is calculated for every family. For malware detection, the detection process checks whether any S exceeds a detection threshold T_s, which is a numeric value between 0 and 100. Lastly, the detection result of a target application is determined by the classification result, that is, the malware family most similar to the target application. If there is no family having a similarity higher than detection threshold, T_s, the application is determined to be benign. The concrete algorithm of the detection process in a pseudo code is as Algorithm 1.

Algorithm 1 Malware detection and family classification

Input: Android application set $D = \{d\}$
Input: Family signature dictionary $F = \{a, e, w\}$ // {**f**amily name, **e**ntry, **w**eight}
Output: Detection result $R = \{d, s, a\}$

```
T_s = DetectionThreshold
W = {w_p, a} // {weight for pattern p in family a, family}
while d is not the end of D do
    // Family similarities for each application
    P ← GetPatternsFrom(d);
    s_a ← 0;
    while p is not the end of P do
        W ← TryGetValue(p, F);
        // Set of matched families and weights for pattern p
        while {w_p, a} is not the end of W do
            s_a ← s_a + w_p/GetTotalWeightof(a)
            // Similarity of family a and pattern set P
        end while
        p ← GetNextPattern(P);
    end while
    r_s ← GetLargestValue(s_a);
    r_a ← GetLargestValueFamily(s_a);
    if r_s > T_s then
        R ← R ∪ {d, r_s, r_a};
        // Detection result for each application
    end if
    d ← GetNextFile(D);
end while
```

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

5. Performance evaluation

To evaluate the practicality of our malware detection mechanism, we performed experiments to measure detection performance and time efficiency. The experiments were performed on a desktop PC with a 3.3 GHz Intel dual-core CPU, 32 GB of RAM, and Microsoft Windows 7 (64 bit) as the operating system. Our self-developed experimentation program written in C# was used to measure time consumption and detection accuracy in the detection of malware variants.

5.1. Data sets

For the performance evaluation, we gathered 15,165 samples reported and classified as Android malware at an AV company, from which we were able to extract 14,120 DEX samples.

Specifically, the data set contained the malware class information for each malware sample; these included detection results and malware naming labels as assigned by 52 AV vendors utilized by the AV company. The classification of family members differs by AV vendor, but when we assigned family membership according to the most frequently used name, the resulting data set contained 5846 malware entities in families having at least ten family members each. The average family size was 112.7 members. The remaining 8274 malware samples (all in families having sizes less than ten) were classified as individual malware. Lastly, for the false detection testing, we collected 3648 benign Android applications at random from the Internet and checked them using two commercial AV tools. The DEX files of the malware entities averaged 340 KB in size, and that of the benign applications, 260 KB.

5.2. Malware family definition

Because the malware naming labels, including the family names, are different at each AV vendor, family classification is not consistent. In our experiments, we tested our mechanism using the family name as classified by the most AV vendors for a given Android malware sample. Table 3 shows the top 10 largest malware families generated according to this labeling method.

5.3. Signature training

Family signatures of known malware for the detection test were constructed from pre-analyzed and published malware. We extracted class and method names, method instruction bodies and character strings from the DEX files. For detection performance evaluation, we randomly chose a training set from a malware family and tested the trained signature on the entire set. Because a family signature contains just a part of its patterns from the training set samples, we needed to evaluate whether a trained signature would accurately detect the trained samples accurately as well as samples in the untrained test set. Thus, the training set was included in the test set.

The training set is used for constructing the family signature, and the test set is used for checking the detection result. After evaluating the performance using training set configuration ranging from 70% to 90%, we used 70% for the remainder of the performance evaluation because the configurations showed slight differences as shown in Fig. 5. The larger the training set showed the higher precision but the lower recall.

In our experiments, the signatures that were extracted from the randomly chosen training set had sizes that varied according to signature refinement and family characteristics, but generally a family signature took several tens to a few hundred KB for a given malware family.

5.4. Experimental result

5.4.1. Malware detection performance

The proposed system detects and identifies new malware as a variant of a known malware family. At the center of our malware detection mechanism is the similarity calculation. In contrast to the legacy signature-matching method, our detection method investigates how similar a new application is to the known malicious ones. The degree of similarity between a malware family and an application is determined by the ratio of shared signatures, as explained in detail in Section 4.4. If it is determined in this step to be a new unknown, it means either that the target application is the benign or that a new malware family has been found, one that does not correspond to any known malware families.

For out evaluation of malware detection and identification performance, we performed a *Monte Carlo* variation analysis with the real-world malware samples. We took a random 70% of samples from a family for signature construction and used the entire sample sets as detection targets. We performed this training and testing ten times and took the average performance.

The variant detection system demonstrated reliable detection performance. Fig. 6 shows precision (upper) and recall (lower) of the validation results. Where the detection threshold $T_s = 15$ and the PSR threshold for signature entry removal $T_{psr} = 60\%$ which was the case having the best F-measure metric, 96%, it achieves 96% precision and 97% recall with a 0.74% FPR. On the other hand, at $T_s = 85$, which is the minimum T_s achieving 0% FPR, 100% precision and 75% recall were attained. The overall performance metrics shown on

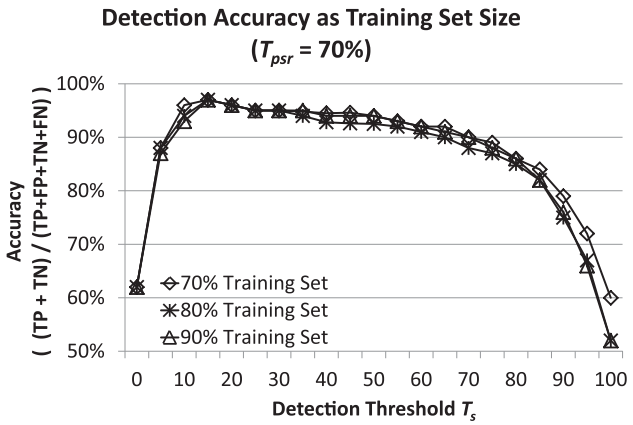

Fig. 5 – Effect of training set size on detection accuracy.

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

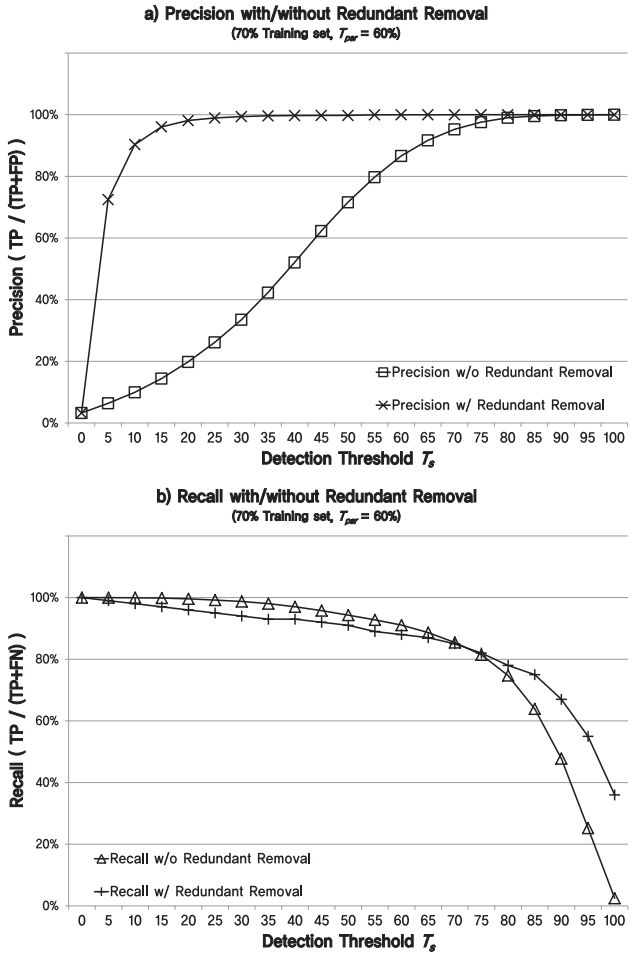

Fig. 6 – Precision (a) and recall (b) obtained in the configuration using 70% as the training set and having a pattern sharing ratio threshold $T_{psr} = 60\%$.

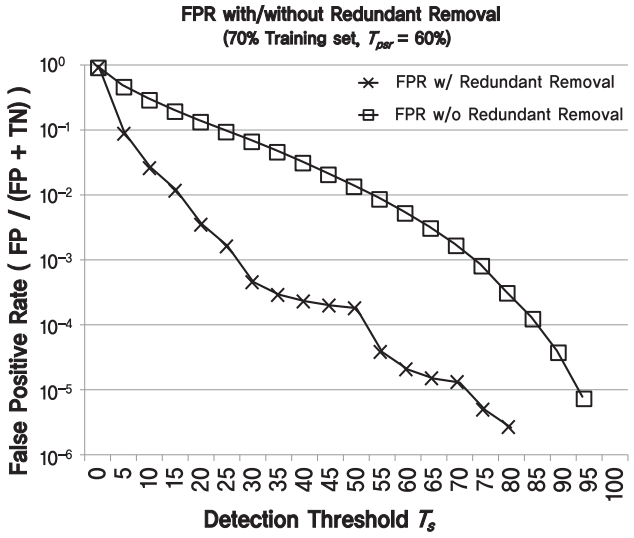

Fig. 7 – False positive rate obtained in the configuration using 70% as the training set and having a pattern sharing ratio threshold $T_{psr} = 60\%$.

Table 4 have equivalent or better degrees compared to other previous approaches even though our proposal considers the variants which employ a few simple evasion tactics. This result supports our assumption that a large portion of variants are generated with a simple evasion tactics and can be screened out with a generalized signature.

In terms of family classification, the family signatures classify the sample Android malware into the corresponding family out of 48 malware families near 70% of accuracy. We estimate the family classification accuracy with the rate of correctly classified samples over the entire samples, on average of the ten times of Monte Carlo validation. As Fig. 8, the classification accuracy fluctuates as the increase of T_{psr}. The reason why the accuracy increases again is that the set of correctly classified family is different for each T_{psr}. For example, where $T_{psr} = 50$, the accuracies of the family *Adrads* and *Zsone* are 98% and 8% each. However, they are 41% and 79% where $T_{psr} = 100$. From this result, we can guess an optimized T_{psr} for a family may be different to the other families. One of the considerable reasons that makes the optimized T_{psr} different is the distinct number of training samples.

Table 4 – Detection performance comparison.

Method	Accuracy	Recall	Precision	F-measure
Androguard	93.04%	49.58%	99.16%	66.11%
DroidMat	97.87%	87.39%	96.74%	91.83%
DroidLegacy	94.03%	92.73%	97.32%	94.97%
Proposed	97.86%	97.00%	96.05%	96.75%

5.4.2. *Effect of redundant signature entry removal*

For evaluating the effectiveness of redundant removal, we used a redundant signature set that had been constructed in our previous study (Lee et al., 2013). This redundant signature includes the binary patterns shared by 1680 benign applications and 79 malware samples belonging to the *DroidDream*, *Geimini*, *KMIN*, *and PjApps* families. The set of malware samples and benign applications used for extracting the redundant signature is distinct to the evaluation data set in Section 5.1.

We checked 3648 benign Android applications using the same family signature sets used in the malware detection experiments. Fig. 7 shows the effect of adopting the redundant

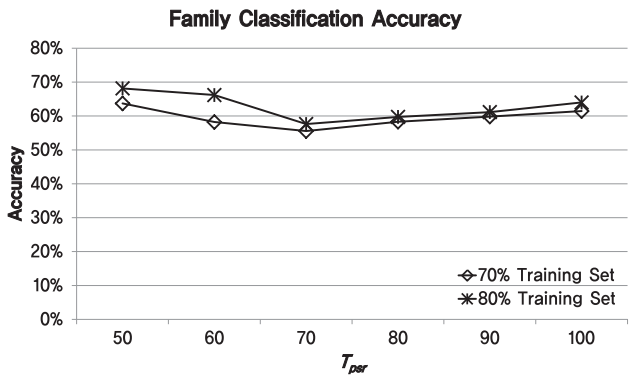

Fig. 8 – Family classification accuracy as T_{psr}.

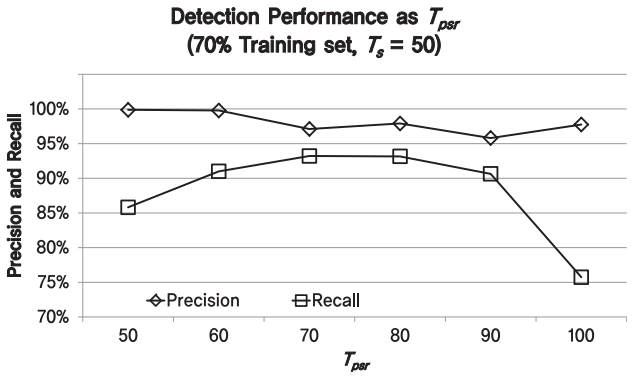

Fig. 9 – Detection precision and recall as a function of pattern sharing ratio threshold T_{psr}.

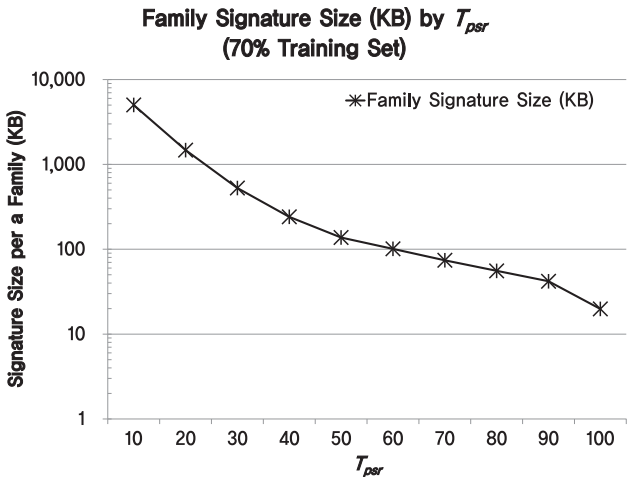

Fig. 11 – Average family signature size as T_{psr}.

signature in reducing false positive. In all ranges of false positive occurrence, the rates were significantly decreased. However, the drawback of redundant removal is loss on recall illustrated on the lower side of Fig. 6.

5.4.3. *Effect of signature entry reduction via pattern sharing ratio threshold* T_{psr}

In our signature definitions, a pattern shared by more family members has more influence in the determination of family classification as well as in the determination of maliciousness. Based on this fact, we can use the entry weights to select the more influential signature entries from the candidate entries. We denote the minimum threshold PSR value of a signature entry to be included in a family signature as T_{psr}. Because reducing the number of signature entries directly decreases the size of signature as shown in Fig. 11, and signature-matching overhead, finding the optimal value of T_{psr} in the trade-off between detection performance and signature entry reduction is an important issue. From Fig. 9, we can see that at the detection threshold $T_s = 50$, as optimized for recall ranged between 70 and 80. However, when considering overall accuracy, T_{psr} was optimized at a value of 60. The removal of signature entries affected the recall at every value of T_s, as shown in Fig. 10. The increase in the number of refined signatures filtered out by increasingly higher value of T_{psr} resulted in higher and slowly decreasing recall as T_s increased.

A higher T_{psr} value increases recall but also decreases precision and family classification accuracy. A signature entry with a higher PSR has a greater possibility of being shared with other families, including benign applications. When signature entries are removed, the influence of PSR is increased because the detection process decides the maliciousness of a target application with a normalized summary of the PSR. Following this effect, the higher T_{psr} values also give the more influence to the signature entries which incur false positive.

Family classification accuracy continuously decrease as T_{psr} is increased. As fewer signature entries remain, the chance that a signature entry belonging to one family is shared with other families will increase. If a pattern shared among multiple families that is not in the redundant signature has more influence than do the entries for each corresponding family, the target malware will be misclassified into the wrong family.

Lastly, as shown in Fig. 9, the detection result for T_{psr} above 80 can be interpreted as being due to over-filtering. In these conditions, loss of diversity by the reduction in entries damages the recall rate. The lower recall rate means that the family signature has less generality and robustness against the variants, and that it is approaching the legacy single-entry signature, which has no diversity. Thus, the optimal point appropriate to the goal for the number of family signatures is the point resulting in the maximum reduction in signatures without damaging recall.

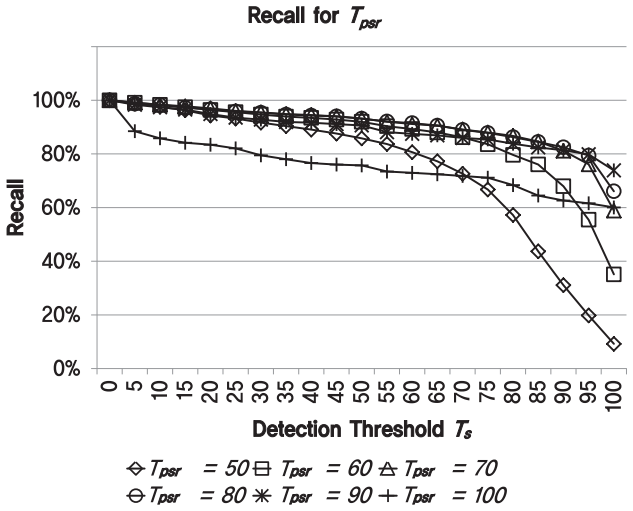

Fig. 10 – Detection recall as a function of pattern sharing ratio threshold T_{psr}s.

6. Overhead analysis

6.1. *Temporal overhead for family signature matching*

The temporal overhead required for signature matching is dependent on the number of signatures and the number of

patterns in each target file. It is normal to assume that the number of signature entries will steadily rise as the amount of malware increases, and this is the case for conventional signature approaches which do not need to concern the number of patterns in a target file. The conventional signature approaches use a pattern offset for each signature, and a signature requires only one comparison with a pattern at a specific offset. However, because in our mechanism the family signature is checked against every possible pattern in a target file and is able to accommodate code reordering evasion tactics, the number of patterns, while plural, has a constant expected value. Therefore, the key to minimizing temporal overhead is to decrease the computation time that is dependent on the number of signature entries. As we described in Section 4.4, we reduce the temporal overhead caused by the relatively larger number of signature entries by using a dictionary.

During the detection process, signature entries loaded into memory are stored in a dictionary structure. A search of the dictionary for a pattern consists of hashing and traverse. In a single search round, the time complexity of $h(k)$, where h is a hash function of a key value k, is $O(1)$, and a traverse takes $\Theta(1+a)$ of time complexity. The a in the traverse overhead is a load factor for the hash table, and is given by (n/m) where n is the number of elements in the hash table, and m is the size of the hash table. Thus, a is equal to the expected length of a linked list of a hash bucket. If a hash function $h(k)$ is capable of simple uniform hashing that distributes its keys in a uniform way, the expected search time is $O(2+a/2-a/2n)=O(1+a)$ when a key is in the hash table. On the other hand, when a key is not in the hash table, it needs to search to the end of the list whose expected length is a, and the search time for an unsuccessful search is $\Theta(1+a)$. Therefore, the expected search time considering both successful and unsuccessful searches is estimated as $O(1)+\Theta(1+a)=O(1)$.

Typically, the number of all signature entries is greater than the number of patterns in a given file being investigated. Thus, finding hashed patterns of a target file from the hashed signature table is faster than the opposite case. When the expected number of hashed patterns is β in α target files, the investigation requires $\alpha\cdot\beta$ time for hashing and the same time for searching. Because the value β is a constant value, independent of α, as derived in Equation (3), our detection mechanism using a hashed signature search method has linear time complexity that is dependent only on the number of target files α.

When the number of signature entries is γ, a search takes a constant time λ. Then, for α target files having β expected patterns per file, the time complexity T is as

$$T(\alpha,\beta)=\alpha\cdot\beta\cdot h(k)+\alpha\cdot\beta\cdot\lambda=\alpha\beta\cdot(h(k)+\lambda)=O(\alpha) \quad (3)$$

We also empirically evaluated the time consumption for family signature matching through pattern hashing and dictionary search. We implemented our Android malware detection and family classification mechanism in the C# language and performed time consumption analysis on a desktop PC using our real-world malware samples. The detailed environment configuration is given in Section 5. For this time consumption analysis, we performed the detection ten times, increasing the number of target applications and the number of signature entries, and calculated the average time consumption. Fig. 12 shows the linear increase in computation time with the increase in the number of target applications.

Because the total number of family signature entries that remained in our best configuration was less than 10,000, we artificially inserted randomly generated signature entries into each family signature uniformly for the larger scale experiments. The randomly generated entries had no chance of being hit by the patterns in the target files, and their presence only served to increase the worst-case search time.

An increase in the number of signature entries increases the size of the hash table and the length of the chains in each hash bucket. The detailed internal algorithms and implementation in the *.Net Framework*, that are used to implement the mechanism, are not fully public, but the consequential computation time, which included some additional tasks such as summarization of weights and similarity sorting, was found to increase in a logarithmic manner with the number of entries.

Another major external contributor to time consumption is context switching or paging overhead due to a large hash table, but this effect is difficult to assess and is outside the scope of our study. Loading only a portion of the signatures at once to reduce the size of the hash table in memory may

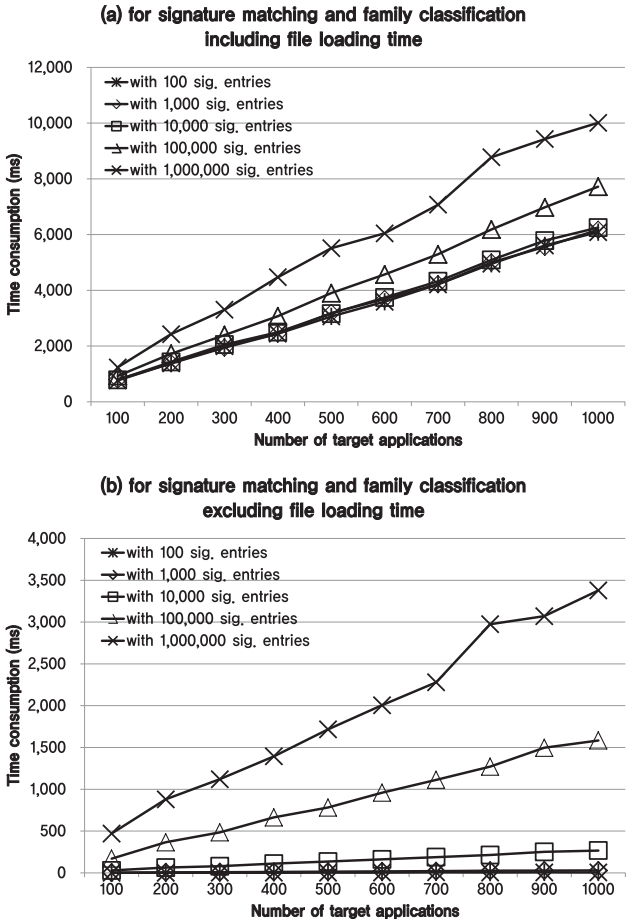

Fig. 12 – Time consumption for detection as a function of the number of target applications and the number of signatures, with and without file loading time.

reduce the memory management overhead, but it would also cause multiple hash table searches and a linear increase in search time for the same number of signature entries.

According to the results of our empirical analysis, time consumption for family signature matching increases linearly as the number of target applications increases, and an increase in the number of signature entries increases temporal overhead in a logarithmic manner when overhead for the decision tasks is included. As a practical example, when we increase the number of entries by a factor of ten, from 100,000 to 1,000,000, the computation time for detection only doubles, from 1584 ms to 3278 ms. Taking into account that the family signature covers a number of new malware entities with a single signature, the experimental results demonstrate the efficiency and practicality of our mechanism.

6.2. Spatial overhead for runtime and storing family signatures

In terms of spatial overhead, the management and use of the family signatures require a file system space for the signatures and a memory space for loading the signatures in a dictionary structure, and these are straightforward. The spatial overhead for storing signatures is fully dependent on the implementation of the file system or a signature database. Because all of the patterns in a family signature are stored in a hashed form having a static size of 32 bytes, the spatial complexity for storing the signatures increases linearly as the number of signature entries γ increases. However, the hashed form of signature entries is a considerable issue. Because all of the patterns in a family signature are stored in 32-byte hashed form, this arrangement offers a spatial advantage when the average size of the signature entries is greater than 32 bytes. However, the hashed form is not mandatory and has no effect on computation time or detection accuracy because a dictionary implementation already has the hash function included in the construction time. Therefore, for a case where the average length of the entries is less than 32 bytes, we can reduce the space needed for the signatures by storing the signature entries in their original binary forms, without hashing. It should be noted that at runtime, the dictionary size will also increase linearly as γ increases because of the increase in the size of the internal hash table.

7. Discussion

7.1. Comparison with signature-based detection methods using string matching

We analyze the ways in which our detection mechanism differs from the legacy signature-matching approach that uses pattern offset and single signature entry per a malware entity. Compared with the legacy signature-matching approach based on string matching, matching a target DEX file with a family signature has greater temporal overhead due to the higher number and unspecified position of signature entries. However, we minimize the temporal overhead to a constant time by storing and searching the hashed signature entries on a dictionary data structure. The objective is to keep the time consumption due to the number of signature entries to a constant. Consequently, temporal overhead is only linearly dependent on the number of target files.

The legacy string-matching-based signature consists of the offset of a specific binary pattern and a pattern body, and a signature needs to be compared just once per target file with the binary pattern at the offset. In our mechanism, however, because we consider the signature strings and bytecode bodies at an unspecified position on each malware variant, the offset technique is hard to be adopted. The information that we can get from a DEX file is only the area of each signature category. Therefore, a signature-matching method without the offset, as our mechanism, should consider the overhead for matching all of the signature entries with every possible candidate binary pattern in a target file. We reduce this matching overhead by using a hash table search instead of string matching.

The main overhead in the use of multiple signature entries is the computing time needed to check whether a pattern is matched with λ signature entries or not. When the λ signature entries are compared to the β binary patterns per file, the entire matching process needs $(\lambda \cdot \beta \cdot \delta)$ time for string comparison, where $(\beta \cdot \delta)$ is the expected number of rounds of matching to find a matched pattern. In the legacy string-matching approach using a signature offset, $(\beta \cdot \delta)$ is 1. Thus, the time complexity of the legacy signature matching is dependent only on λ because it only needs to compare a binary pattern at the specific position per file. On the other hand, our matching mechanism requires $(\lambda \cdot \beta \cdot \delta)$ time for string comparison. In a realistic investigation situation, δ is close to 1.0 because the target files have many more unmatched patterns than matched patterns. Therefore, the key to reducing the comparison overhead lies in making the effect of λ and β as small as possible.

We use a dictionary to reduce the temporal overhead due to λ and β. A dictionary is one of the data structures that lead to constant search overhead with little influence from the number of data it has. A dictionary uses a hash table, and the search is performed by a hash table lookup. The family signature matching method using a dictionary has negligibly low overhead from an increase in λ because it uses additional memory space for a hash table. Given that the number of new malware entities and their signatures is continuously increasing, the constant time consumption of the hash table search approach offers a great advantage in terms of computing time. See Section 6.1 for a detailed analysis of the overhead of our detection method.

7.2. Comparison with API call analysis and behavior-based detection methods

Analyzing API call sequences is a well-known malware detection approach, but it requires excessive overhead to analyze every possible execution path in a static way. On the other hand, dynamic analysis at runtime requires a sandbox environment and instrumentation overhead. To overcome these problems, modeling and finding commonly appearing patterns of API calls from malware can be an efficient solution in the detection of suspicious applications that have critical capabilities. However, this method suffers from a false

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003

detection problem, especially in the smartphone environment because legitimate smartphone applications have many more capabilities and behaviors that are similar to malicious applications than is the case in legacy PC environments. This feature affects the accuracy of API-based approaches in both static and dynamic analysis methods. To conclude, API calls are a representative feature that can detect malicious capability in applications, but they need to be complemented by additional features and metrics. In our mechanism, we significantly mitigate the false detection problem by multiplexing decision conditions with multiple types of signature entries. This approach makes the detection mechanism more robust to false detection and evasion attempts by decreasing dependency on a single type of feature.

7.3. *Differences from variant detection methods in PC environments*

The first key feature of Android malware compared to malware in the legacy PC environment is the simpler and more semantic code set of an executable file. Most Android applications consist of Dalvik bytecode, which has a much smaller instruction set and richer semantics than Assembly code or even Java bytecode. This highly encapsulated and abstracted code directly contributes to the success of our signature structure. Code patterns in an instruction have not been an effective signature in the PC environment, even though they provide strong evidence in identifying reused code. Finding exact matches of long code strings that are easy to change supports precise detection but has low robustness to variants. However, Dalvik bytecode has a much simpler and smaller instruction set than do x86- or JVM (Java virtual machine)-coded executables. This means that a shorter string of Dalvik code can express the same semantics as its counterpart code in the PC environment even though it has less variety. For example, a method for obtaining the sum of integer values in an array that can be coded with just five lines of Java code are encoded to 14 instructions in 25 bytes of JVM bytecode but require only 6 instructions in 18 bytes of DVM (Dalvik virtual machine) bytecode. This feature enhances the suitability of using bytecode patterns as a detection signature.

Another significant feature of the Android malware is flood of family malware. As supported by the analysis reports from AV-vendors, the dominant portion of Android malware is constructed by repackaging technique, and most of the new malware belongs to a malware family. Even in 2014, the average family size is increased while the number of families is decreased. It means that we have higher possibility to face a variant of a known malware than unknown one. This fashion is supported by the repackaging and code reuse techniques. Because of this similarity feature, a family signature approach is much more effective than the similar approach in the PC environment such as a generic signature.

8. Conclusion

In this paper, we have proposed a scalable and accurate approach for Android malware detection. The proposed system overcomes the lack of robustness against Android malware variants that afflicts the conventional signature-based approach. The proposed generalized signature for Android malware families that are groups of variants sharing code and resources is responsive to their evasion techniques. The malware detection and family classification mechanism detects family malware through the family signature constructed from sets of known malware family samples. The diversity of the family signature powered by multiple signature entries and the similarity estimation approach increases coverage over that of typical single-entry signatures, to include greater numbers of family members.

We have shown experimentally that the family signature approach improves detection accuracy compared with the previous static approaches. In performance testing with thousands of real-world Android malware entities and benign applications, it showed an accuracy and recall rate of more than 97%, and the samples were detected with a few megabytes of signatures. In addition, the proposed detection mechanism consumes reasonably small time, requiring only tens of seconds to investigate a million targets. Moreover, time consumption increases only linearly with increasing numbers of targets. In conclusion, the proposed system has enough investigation performance for responding the rapidly growing numbers of Android applications, both of the malicious and the benign. Considering the efficiency and detection accuracy of our mechanism, it is well suited for screening Android applications before they are uploaded on public app markets as well as for the sever-side investigation of samples by network-based AV systems, or further depth investigation methods.

Acknowledgment

The preliminary version of this paper was presented in the IFIP Int'l Information Security and Privacy Conference (IFIP SEC 2013) (Lee et al., 2013).

REFERENCES

Aafer Y, Du W, Yin H. Droidapiminer: mining api-level features for robust malware detection in android. In: Security and privacy in communication networks. Springer; 2013. p. 86–103.

Aquilino B, Aquiano K, Bejerasco C, Cajucom E, Goh SG, Hilyati A, et al. Mobile threat report h2. Report. F-Secure Labs; 2014.

Crussell J, Gibler C, Chen H. Andarwin: scalable detection of semantically similar android applications. In: Computer security–ESORICS. Springer; 2013. p. 182–99.

Deshotels L, Notani V, Lakhotia A. Droidlegacy: automated familial classification of android malware. In: Proc. of ACM SIGPLAN on program protection and reverse engineering workshop. ACM; 2014. p. 3.

Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. of the 9th USENIX conference on operating systems design and implementation (OSDI). USENIX Association; 2010. p. 1–6.

Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proc. of the 16th ACM conference

on computer and communications security. ACM; 2009. p. 235–45.
F-Secure Labs. Mobile threat report Q1. Report. 2014.
Faruki P, Ganmoor V, Laxmi V, Gaur MS, Bharmal A. Androsimilar: robust statistical feature signature for android malware detection. In: Proc. of the 6th international conference on security of information and networks. ACM; 2013a. p. 152–9.
Faruki P, Laxmi V, Ganmoor V, Gaur MS, Bharmal A. Droidolytics: robust feature signature for repackaged android apps on official and third party android markets. In: Proc. of the 2nd international conference on Advanced Computing, Networking and Security (ADCONS). IEEE; 2013b. p. 247–52.
Gascon H, Yamaguchi F, Arp D, Rieck K. Structural detection of android malware using embedded call graphs. In: Proc. of the ACM workshop on artificial intelligence and security. ACM; 2013. p. 45–54.
Gilbert P, Chun B-G, Cox LP, Jung J. Vision: automated security validation of mobile apps at app markets. In: Proc. of the second international workshop on mobile cloud computing and services (MCS). ACM; 2011. p. 21–6.
Kwon J, Lee H. Bingraph: discovering mutant malware using hierarchical semantic signatures. In: Proc. of 7th international conference on malicious and unwanted software (MALWARE). IEEE; 2012. p. 104–11.
Lee J, Jeong K, Lee H. Detecting metamorphic malwares using code graphs. In: Proc. of the ACM symposium on applied computing (SAC). ACM; 2010. p. 1970–7.
Lee S, Lee J, Lee H. Screening smartphone applications using behavioral signatures. In: Security and privacy protection in information processing systems (IFIP SEC). Springer; 2013. p. 14–27.
Mawston N. Android captures record 85% share of global smartphone shipments in Q2 2014. Report. Strategy Analytics; 2014.
Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proc. of the 8th ACM SIGSAC symposium on information, computer and communications security. ACM; 2013. p. 329–34.
Sanz B, Santos I, Nieves J, Laorden C, Alonso-Gonzalez I, Bringas PG. Mads: malicious android applications detection through string analysis. In: Network and system security. Springer; 2013a. p. 178–91.
Sanz B, Santos I, Ugarte-Pedrero X, Laorden C, Nieves J, Bringas PG. Instance-based anomaly method for android malware detection. In: SECRYPT; 2013. p. 387–94.
Sanz B, Santos I, Ugarte-Pedrero X, Laorden C, Nieves J, Bringas PG. Anomaly detection using string analysis for android malware detection. In: International joint conference SOCO-CISIS-ICEUTE. Springer; 2014. p. 469–78.
Suarez-Tangil G, Tapiador JE, Peris-Lopez P, Blasco J. Dendroid: a text mining approach to analyzing and classifying code structures in android malware families. Expert Syst Appl 2014;41(4):1104–17.
Wood P, Nahorney B, Chandrasekar K, Wallace S, Haley K. Internet security threat report. Report. Symantec Corporation; 2014.
Yan L-K, Yin H. Droidscope: seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In: USENIX security symposium; 2012. p. 569–84.
Zheng M, Sun M, Lui J. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Proc. of the 12th IEEE international conference on trust, security and privacy in computing and communications (TrustCom). IEEE; 2013. p. 163–71.
Zhu J, Guan Z, Yang Y, Yu L, Sun H, Chen Z. Permission-based abnormal application detection for android. In: Information and communications security. Springer; 2012. p. 228–39.

Jehyun Lee received the B.S. and M.S. degrees in Computer Science and Engineering from Korea University, Korea, in 2007 and 2009. Currently, he is a Ph.D. candidate in Department of Computer Science and Radio Communication Engineering, Korea University. His research interests include network security and malware.

Suyeon Lee received the B.S. degree from Dongduk Women's University in 2010, and M.E. degree in Computer Science and Engineering from Korea University, Korea, in 2012. Currently, she is a research engineer in LG Electronics CTO. Her research interests include dynamic taint analysis and smartphone security.

Heejo Lee is a professor at the Dept. of Computer Science and Engineering, Korea University, Seoul, Korea. Before joining Korea University, he was at AhnLab, Inc. as a CTO from 2001 to 2003. From 2000 to 2001, he was a postdoctorate at the Department of Computer Sciences and the security center CERIAS, Purdue University. Dr. Lee received his B.S., M.S., Ph.D. degrees in Computer Science and Engineering from POSTECH, Pohang, Korea. Dr. Lee serves as an editor of the Journal of Communications and Networks. He has been an advisory member of Korea Internet Security Agency and Korea Supreme Prosecutor's Office.

Please cite this article in press as: Lee J, et al., Screening smartphone applications using malware family signatures, Computers & Security (2015), http://dx.doi.org/10.1016/j.cose.2015.02.003