MEDS
Malware Evolution Discovery System

by

Antonio César Vargas Valiente

May, 2014

MEDS

Malware Evolution Discovery System

Antonio César Vargas Valiente

This thesis has been presented to and accepted by the Office of Graduate Studies, John Jay College of Criminal Justice in partial fulfillment of the requirements for the degree of Master of Science in Digital Forensics and Cybersecurity.

Bilal Khan

Thesis Advisor	Signature	Date

Spiros Bakiras

Second Reader	Signature	Date

Anne Lopes

Dean of Graduate Studies	Signature	Date

Acknowledgements

I would like to dedicate this work to my family and the faculty of the Digital Forensics and Cybersecurity program at John Jay College of Criminal Justice, especially Professors Bilal Khan and Richard Lovely. I am also grateful to Jamie Levy for her constant advice and guidance since I decided to pursue a career and graduate degree in digital forensics and cybersecurity. I also would like to acknowledge Joy Song and Aidan Booth for their patience and advice when proofreading this document. Finally, I would like to express my gratitude to the open source community for its constant contribution to human knowledge, especially Candice Quates from the sdhash development team for providing me with a Python binding of sdhash for this project. The open source community has helped me view my academic and professional careers as more than completing papers or projects. It is about contributing back to the communities that one has benefited from. I hope this work accomplishes part of that goal.

Abstract

Malware, or malicious software, affects every computing device at our disposal, including personal computers, dedicated servers and more recently, mobile devices such as smart phones and tablets. The information stored on these devices makes them attractive targets for illegal financial gain by cybercriminals, and corporate espionage or even strategic operations by government agencies. Yet, traditional detection measures are increasingly ineffective at detecting the extensive number of malware variants. To make matters worse, these variants are becoming commodity products whose manufacture is facilitated by an underground industry that seeks to meet demands for products that can bypass current anti-malware technologies. Consequently, most present malware is not new since the development of new "from-scratch" malware is not economically viable for the underground malware industry. Instead, most malware found in the wild is a modification or feature upgrade of previously created malware.

This thesis attempts to frame the malware problem from the perspective of the evolutionary production of malware. The goal is to design an architecture that allows researchers to discover generative malware, and develop an initial implementation of the Malware Evolution Discovery System (MEDS). MEDS supports the creation of phylogenetic trees of malware, and attempts to make predictions of generative malware by applying two models of supervised regression analysis on malware samples and their corresponding phylogenetic tree. Finally, the MEDS framework is made available as an open source project, thus providing an innovative tool that has been previously unavailable for the cybersecurity and digital forensics communities.

Contents

Acknowledgements ii

Abstract iii

List of Figures v

List of Tables vi

1 Introduction 1

2 Previous Work and Background 5
- 2.1 Previous Work . . . 5
- 2.2 Approximate Matching . . . 9
- 2.3 Android . . . 12
- 2.4 Supervised Regression Analysis . . . 16

3 MEDS Framework 21
- 3.1 Limitations . . . 21
- 3.2 Framework Design . . . 23
- 3.3 Phylogenetic Tree Creation . . . 25

4 Predicting Generativeness 31
- 4.0.1 Training . . . 31
- 4.0.2 Evaluation of Prediction Parameters . . . 43

5 Conclusion and Future Work 54

Bibliography 56

List of Figures

1.1 5 Malware Samples May 14-August 2, 2010 4
1.2 10 Malware Samples Before and After August 2, 2010 4

2.1 MEDS Training 17
2.2 Univariate regression training and prediction phases. 18
2.3 Univariate linear regression cost function 18
2.4 Univariate linear regression prediction function 18
2.5 Sigmoid function 19
2.6 Logistic regression cost function. 19
2.7 Minimizing cost function to obtain prediction values in Octave. 19

3.1 APKDirectory Factory and RamResientMC classes. 25
3.2 Fingerints implemented in MEDS. 26
3.3 Approximate matching metrics. 27
3.4 RAMResidentGraph drives the creation of rooted phylogenetic tree. . . . 27
3.5 Rooted phylogeny creation code. 28
3.6 Phylogeny of 10 malware samples. 29
3.7 Phylogeny with approximate creation dates. 30

4.1 UML Diagrams of features used to calculate prediction parameters. 33
4.2 Number of dangerous permissions and actual generativeness values. 35
4.3 Number of receivers and actual generativeness. 37
4.4 Number of DEX classes and actual generativeness 39
4.5 Number of Children actual generativeness 40
4.6 XY scatter plot of approximate matching value and actual generativeness 42
4.7 Actual and predicted generativeness using number of dangerous permissions feature 45
4.8 Actual and predicted generativeness based on number of receivers 47
4.9 Actual and predicted generativeness values of evaluation set 1 and trial 1 based on number of DEX classes 49
4.10 Evaluation set 2 and trial 2 actual and predicted generativeness values based on number of children 51
4.11 Evaluation set 1 and trial 1 actual and predicted generativeness values based on approximate matching feature values 53

List of Tables

3.1 Malware hashes, creation date and name given by F-Secure Anti Virus . . 29

4.1 Dangerous permissions linear regression prediction parameters 34
4.2 Dangerous permissions logistic regression prediction parameters 35
4.3 Number of receivers linear regression prediction parameters 37
4.4 Number of receivers logistic regression prediction parameters 38
4.5 Linear regression prediction parameters calculated for the number of DEX classes 38
4.6 Logistic regression prediction parameters for the number of DEX classes . 38
4.7 Linear regression prediction parameters for the number of children 39
4.8 Logistic regression parameters for the number of children 40
4.9 Linear regression prediction parameters for approximate matching values . 41
4.10 Logistic regression prediction parameters for approximate matching values 41
4.11 Dangerous permissions error values for linear regression on random evaluation sets 44
4.12 Dangerous permissions accuracy values for logistic regression on evaluation sets 44
4.13 Error values for linear regression based on number of receivers 46
4.14 Number of receivers accuracy values for logistic regression on evaluation sets 46
4.15 Error values for linear regression based on number of DEX classes 48
4.16 Number of DEX classes accuracy values for logistic regression on evaluation sets 48
4.17 Error values for linear regression based on number of children 50
4.18 Accuracy values for logistic regression based on number of children. 50
4.19 Error values for linear regression based on approximate matching feature values 52
4.20 Accuracy values for logistic regression based on approximate matching value 52

Chapter 1

Introduction

Malicious software threatens the safety of computer systems on the Internet. The quantity and range of digital devices that populate the Internet makes them attractive targets for cybercriminals. Why commit crimes via physical means when digital crime requires only a computer and an Internet connection? Especially since, in the case of digital crimes, the perpetrator faces lower risk of being apprehended. Harris and Perlroth (2014) reported that during the data breach suffered by the retail company Target, in late 2013, the personal information stolen affected 70 million to 110 million people in the United States. Krebs (2014) explained that during Target's data breach incident the attack vector utilized a malicious application that targeted the point-of-sale (POS) computer systems used in the Target stores. This malicious application, known as BlackPOS, was able to extract credit card information about Target's customers, which was in turn sold through black market websites. The number of people affected by this type of attack shows that compromising the security of digital devices that store important information, such as financial or personal records, can be lucrative to criminals and damaging to the reputation of businesses and individuals. Our lives depend on the security of such computing systems, whether we like it or not.

Smart phones are everywhere, including where we shop, exercise, obtain medical care, work, live, etc. These devices are interconnected by a massive, insecure, communication infrastructure known as the Internet. The potential to access a large quantity of personal information stored by these systems makes them attractive targets for cybercriminals. DeGusta (2012) emphasized that smart phones took only 7 years to have a 50 percent penetration among U.S households. These days, it is difficult to imagine ourselves

without having access to the benefits of using a smart phone. We can check email, get weather information, watch our favorite movies, listen to our music collection, and execute financial transactions with the help of a smart phone.

Mobile malware has become common on mobile devices, especially on Android devices. Cybercriminals are aware of how important smart phones and tablets are to our every day lives, and so, not surprisingly, it took them only a few years to infiltrate the smart phone platform with "mobile malware." The year 2014 marked the tenth anniversary of mobile malware. Wueest (2014) tracked the mobile malware history with the first versions of Cabir, malware designed for the Symbian platform, all the way to the latest samples of Rootcager that targets Android devices. As of April 2014, Android is the primary mobile system attacked by malicious software in the smart phone market. Nmawston (2014) from StrategyAnalytics, a business analysis company, reported that for the year 2013, the total share of Android in the smart phone market reached 79 percent. Malicious software creators "follow the money," and in the case of computing devices, monetary gain is obtained by attacking systems that are most commonly used by end-users. On the other hand, effective solutions to the malware problem are quite limited.

Cohen (1984) formally proved that **antivirus programs** are not the solution to the malware problem. The reason is very simple: There is not one program that will adapt to detect all future malware variants because minimum modifications in malware code cause antiviral code to become inadequate at detecting new malware variants. This explains the large number of companies that specialize in creating their own antivirus products to detect malicious software. However, the products created by the antivirus industry have been found to be ineffective against newly created malware. Perlroth (2012) reported that "these programs rarely, if ever, block freshly minted computer viruses, expert say, because virus creators move too quickly"(para. 2). Nonetheless, antivirus software is currently the most popular tool for combating malware. Szor (2005) reminds us that, even though it has been formally proven that antivirus software is not the solution to the malware problem, the reality is that "without antivirus programs, the Internet would be brought to a standstill because of the traffic undetected computer viruses would generate"(p. 35). The natural question that comes to mind is whether there might be a different way to approach the malware problem.

The main **contribution** of this thesis is to look at the malware problem from a different point of view. The approach taken is based on the idea that as malware evolves, new malware samples are influenced by previous generations. Additionally, as malware progresses through time, there are external and internal factors that influence the evolution or extinction of malware families. While external evolutionary factors are outside of the scope of this research, we will see that internal information can be extracted from real malware, allowing us to understand their evolution over time. Toward this, we present the design and implementation of the Malware Evolution Discovery System (MEDS). This system takes large quantities of real malware and infers a probable evolutionary tree.

Phylogenetic/Evolutionary trees provide a visual approximation of the complex process behind malware evolution. The phylogenetic trees created by the MEDS framework are an important investigative asset. To validate the effectiveness of MEDS at reconstructing plausible phylogenetic trees, real malware samples that target the Android platform were used. These samples were obtained from the archive `VirusShare_Android_20130506.zip` provided by the `VirusShare.com` malware repository. Previous work has been done about malware phylogenis; however, none of the previous studies have made available an open source framework for the digital forensics and cybersecurity communities. This thesis attempts to fill that gap.

Generativeness is a numeric value that quantifies the extent to which present malware gives rise to future malware variants. To illustrate this definition consider Figure 1.1 which shows a hypothetical phylogenetic tree of five malware samples with creation dates ranging from May 14, 2010 through August 2, 2010. Figure 1.2 extends this tree by including new malware variants discovered after August 2, 2010. This figure shows that malware B was responsible for generating 40 percent of new variants created after August 2, 2010. On the other hand, 60 percent of new variants were generated from malware D. Malware samples A, C and E did not account for any malware strain created. Given this, we define the generativeness of A, B, C, D, E to be 0, 0.4, 0, 0.6, 0 respectively.

Malware features and phylogenetic trees created from malware samples can be used to make **predictions** about the future generativeness of present malware. This thesis presents an initial pursuit at this endeavor by applying supervised regression models. The results of applying supervised regression analysis models are inconclusive at this

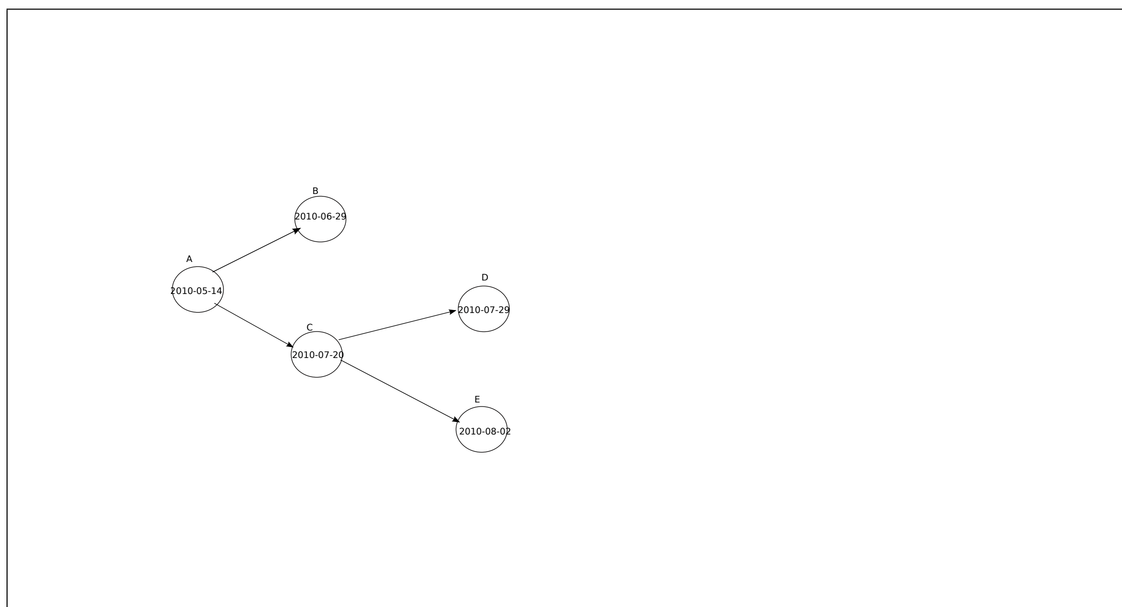

Figure 1.1: 5 Malware Samples May 14-August 2, 2010

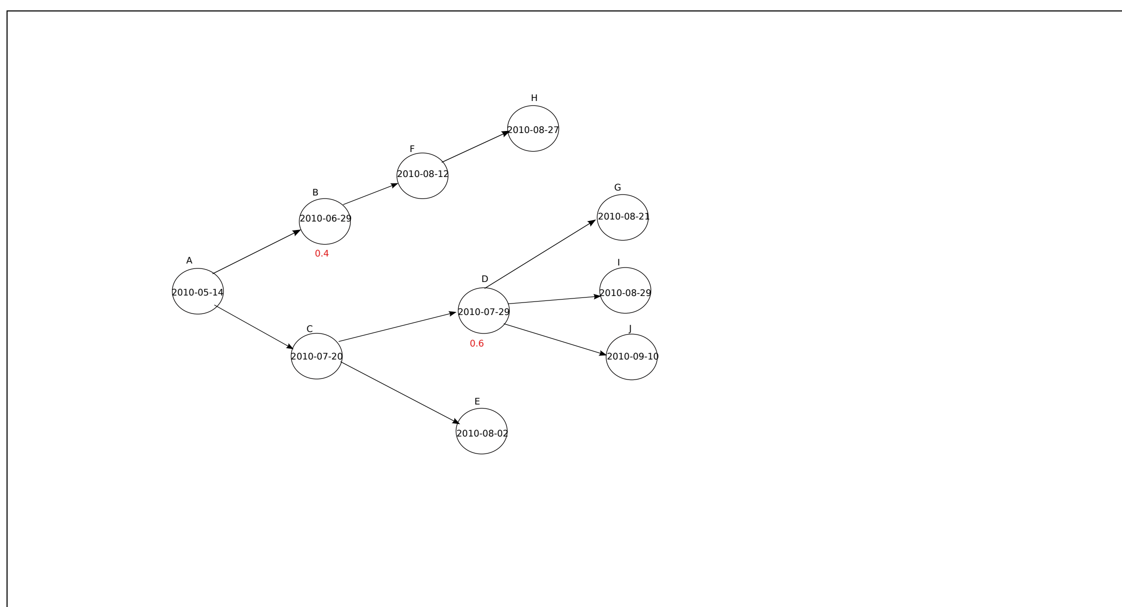

Figure 1.2: 10 Malware Samples Before and After August 2, 2010

point. However, future work will consider advanced machine learning approaches to improve accuracy and performance.

Chapter 2

Previous Work and Background

2.1 Previous Work

The **evolution** of any organism is a valuable lens through which to examine and learn about important generic information changes of a subject or group of subjects. The phenotype —the composite observable features— of living organisms "is always the result of the generic information that they carry and pass on to the next generation and its interaction with the environment"(Salemi and Vandamme, 2009, p. 31). This also applies to the evolution of malicious software. Cohen (1984) was the first person to formally define the infection property of a computer virus and investigate that, in the worst case, the infection can spread to what is formally known as the transitive closure of shared information. Informally, this means that if system A can infect system B, and system B can infect system C, then a computer virus originating in system A can propagate to system C. Although this is now well acknowledged, back in the early 1980s there were no formal experiments that examined this behavior in viral computer code. Another important contribution by Cohen is the importance of transitivity, communication via connection channels, and information sharing of computer systems for a computer virus to obtain high infection rates.

Kephart and White (1991) took the ideas put forth by Cohen on **computer virus spread infection** and developed an epidemiological model to analyze the theoretical propagation of viral code. This model was based on a directed graph, where each node represented a computer system, and the edges from each of these nodes represented the possible computer systems that could be infected by viral code. Their experiments

determined that while anti-virus programs were not the answer to the malicious software problem, they could be effective enough to control the spread of malicious code if the infection rate did not exceed a critical epidemic threshold value. Furthermore, they emphasized that defense systems would be sufficient if their detection and removal rate were high enough with regards to the viral infection rate among computer systems. The limitations of this prior research was that it was carried out only in simulations (100 node graphs and no real world malware). Additionally, this research concentrated on computer viruses, but today's malicious software includes a wide range of binary code in addition to computer viruses.

Earlier malware research focused on understanding the **phenomenon of malware** and its epidemiological model. Goldberg et al. (1998) noted that previous algorithm research on reconstructing evolutionary history of biological species could be applied to the evolutionary history of computer viruses. This was the first research paper that emphasized the use of phylogenies to understand malicious software evolution. Antivirus companies were also attracted to the idea of using phylogenies to model malware evolution. Carrera and Erdelyi (2004), malware analysts from the anti-virus company F-Secure, demonstrated that by using phylogenies, the analysis of complex malware could result in reducing the time required by malware examiners.

In the academic world, Karim et al. (2005) began exploring the idea of **constructing phylogenies** using permutation of code found on malicious software with the intention of improving analysis of new malware. Ma et al. (2006) began formulating questions about the malware ecosystem, that is, the internal or external pressures that drive the structural and functional evolution of malware, and the human factor behind malware creation. The authors advocated the use of phylogenetic trees to better understand the evolution of malicious software of such environments. The malware ecosystem, from their point of view, is anything related to the relationships among malware samples. Although their research concentrated in analyzing phylogenies of malicious remote code injection exploits, also known as shellcodes, they had a fundamentally new approach to the malware problem. For example, through analysis of the phylogenies constructed from malicious shellcode, they were able to locate important shared features among different shellcode samples. Furthermore, having better knowledge about such shared features allowed them to create detection signatures that were robust at identifying new shellcode variants. This work was important because it provided concrete evidence that the

malware industry has a tendency to use previously written code, and that few malware instances are created from scratch. In a similar study, Wehner (2007) explored the idea of constructing phylogenies using the Kolmogorov Complexity by compressing computer worms and network traffic. Her results demonstrated that the similarity patterns found by analyzing phylogenies of computer worms could be used to create efficient network signatures that were able to identify attacks of new computer worm variants. Khoo and Lio (2013) took this research further by defining a framework based on the study of phylogenies to perform malware reverse engineering and detection. Their experiments found that 100% of memory management procedures used in the malware families FakeAV-DO and "Skyhoo" were identified by their framework. The main limitation of their research is that their experiments were executed on a small number of samples.

Some of the problems with malware research is the lack of **accessible real malware** sets, the absence of information regarding the collection process of malware, and the unavailability of "ground truth" about malware creation and evolution. These factors make it very difficult to create tools and methods for accurate analysis of malware research. This was clearly identified by Dumitras and Neamtiu (2011) in their research on phylogeny trees of open source projects as models of phylogenies of malware. This research captivated the attention of the Defense Advanced Research Project Agency (DARPA) and led to the creation of the Cyber Genome program with a funding of $43 million to investigate and develop techniques to automatically discover, identify, and describe malware variants.

As of April 2014, only two research papers have been published as the results of the **Cyber Genome program** . Jang et al. (2013) focused in designing algorithms to construct phylogenies, and evaluation metrics to determine the quality of the constructed phylogenies. Darmetko et al. (2013) modeled the evolution process by extracting malware features that are used as evidence to determine if a malware sample is an ancestor or descendant of another malware sample. The resulting relationships are used to create malware phylogenies. Both of these research projects verified their findings by using a malware set of 128 samples provided by the Cyber Genome evaluation team where the ground truth evolution history is known.

The main limitation of **all previous research,** including the two projects funded through the Cyber Genome project, is that their frameworks, and even their data sets, are not available to the general public. There is still a big taboo about sharing malware

samples and information between researchers, even though the malware problem affects everyone who uses a computing system. The problem of having access to real malware has been partly solved by the malware sets provided by VX Heavens and VirusShare. However, wider accessibility to the frameworks and malware sets will allow other researchers to verify and improve findings of previous research.

Having covered the long list of important historical work about malware and its phylogenetic studies, the rest of this thesis project will focus in the **design of MEDS**, the evaluation process of this system for creating malware phylogenies of real mobile malware and the evaluation process of making predictions of generative malware. The major contribution of this thesis is to provide an open source framework that other researchers can use, examine and enhance for the benefit of the general public.

2.2 Approximate Matching

Approximate matching is the process of identifying how similar, or dissimilar, two digital objects are to each other. These digital objects can be anything that can be stored on a computing system in binary format, including pictures, text-files and malware. The problem of finding how similar two digital artifacts are has been explored by the digital forensics community for quite a few years, but only recently has this community emphasized the need for more approximate matching research in algorithms and tools. In fact, one of the panel discussions during the 2013 Digital Forensics Research Workshop (DFRWS) was titled "Approximate Matching of Digital Artifacts." The moderator of this panel was Barbara Guttman, who is one of the main contributors of NIST's Special Publication 800-168, from the National Institute of Standards and Technology (NIST). In this panel, the topic of approximate matching was discussed among end-users, digital forensic practitioners, and developers of approximate matching tools. Consequently, in January 2014 NIST released a draft of Special Publication 800-168, laying the foundation for the requirements and considerations for end-users of approximate matching algorithms and tools. Breitinger et al. (2014) define approximate matching in NIST's Special Publication 800-168 as follows:

> Approximate matching is a promising technology for designed [*sic*] to **identify similarities** between two digital artifacts. It is used to find objects that resemble each other or to find objects that are contained in another object. This can be very useful for filtering data for security monitoring, digital forensics, or other applications. (p. 1)

Resemblance and containment are the two main ideas behind approximate matching. Resemblance is the process of finding an approximate matching value between two objects that are very similar in size. On the other hand, containment deals with finding an approximate matching value of two objects with different sizes. Furthermore, the approximate matching value is calculated based on the concept of whether the larger object contains the smaller one. Approximate matching algorithms can be divided into three main categories: **Bytewise, syntactic and semantic**.

Bytewise approximate matching algorithms depend on the order of the byte sequences of a digital objects. This order is not influenced by any meaning that those bytes might have to a computer system. This is very important because bytewise algorithms can be applied to any kind of digital data.

Syntactic approximate matching algorithms consider the internal structures of the digital object. For example, a syntactic algorithm can be influenced by the order sequence of letters found in a one-dimensional string.

Semantic approximate matching algorithms are the most computationally expensive of all three approximate matching algorithms because they take into account contextual attributes of digital objects; however, they are the closest to the human perception process. For more information on approximate matching, please refer to Breitinger et al. (2014).

The MEDS framework utilizes a **bytewise approximate matching** implementation called **sdhash.** Sdhash breaks binary data into data blocks of 64 bytes, then calculates the entropy values, i.e. the extent of randomness, found within each data block. It eliminates data blocks with low entropy (which has been determined through experiments done by the creators of sdhash). The rationale behind eliminating data blocks with a low threshold entropy value is to "pick object features that are least likely to occur in other data objects by chance" (Roussev, 2010, p. 113). Here "object features" are data blocks since sdhash is a bytewise approximate matching algorithm. Once the appropriate high entropy data blocks have been selected, each of these data blocks are processed by using a series of hash functions. Each of these functions' results are stored in a data structure known as a Bloom filter. The idea behind a Bloom filter is to represent large quantities of data in one dimensional vectors. Sdhash implements 256-byte Bloom filters, with a maximum of 128 data blocks, that can be represented by a single Bloom filter. The complete set of Bloom filters represent the whole binary object. To measure the similarity between two binary objects, a comparison is carried out between their respective Bloom filters. The objective behind calculating similarity in the MEDS framework is to construct phylogenetic trees and locate similar malware samples. The advantage of using sdhash is that we can calculate the approximate matching value or "distance" between pairs of malware without concern for their internal structure. For more information on sdhash, please refer to Roussev (2010).

In addition to bytewise approximate matching algorithm, MEDS also utilizes **SequenceMatcher**, an optimized implementation of a syntactic approximate string matching algorithm. SequenceMatcher is included in the Python library difflib. Additionally, SequenceMatcher is based on the **Ratcliff/Obership** algorithm, which works by analyzing two strings and extracting the largest block of text that is common between the two. This block of common characters is used as a point of mutual reference. The algorithm proceeds by analyzing text to the right and left of the point of reference; furthermore,

these blocks of letters are placed into a stack and the point of reference step is applied to them. The approximate matching value is calculated as twice the total number of characters found in common between the two strings, and divided by the total number of characters of the two strings. The resulting value is a percentage match between 0 and 1. Ratcliff and Metzener (1988) emphasized that the worst case runtime for the Ratcliff/Obership algorithm is $O(n^2)$, quadratic runtime, where n is the length of the strings being compared. On the other hand, the main advantage of using the optimized Ratcliff/Obeship implementation in difflib is its best run time, O(n). However, it can only be applied to strings. In MEDS, the AndroidManifest.xml file is extracted from an Android package, then is converted into a string. When comparing the approximate matching values of two Android malware samples, an efficient way to achieve this is by comparing their AndroidManifest.xml files. Fore more information on Racliff/Obership, please refer to Ratcliff and Metzener (1988).

2.3 Android

Android is the most **popular mobile operating system** for smartphones and tablet devices. Android was initially designed by Android, Inc. and was acquired by Google, Inc. as reported by Elgin (2005). Android's core architecture is formed by the following: *Kernel*, *Android Runtime*, *Libraries*, *Application Framework* and *Applications*. The current Android *Kernel* is based on the version 2.6 of the Linux Kernel which provides memory, process management, and a network stack. The *Android Runtime* is also known as the *Dalvik Virtual Machine*; the task of *Dalvik* is to provide the core libraries of the Java language and runtime environment. Furthermore, Android applications can be separated by running inside of their own *Dalvik* instance. *Dalvik* is a major component of the Android ecosystem because it provides a customized Java runtime environment for mobile devices, which is needed by all Android applications since they are written using the Java programming language. Android, like most current operating systems, takes advantage of third party's libraries that provide additional features to Android applications. The following are some of the libraries provided with the Android platform: *SQLite*, *LibC*, *OpenCORE*, *LibWebCore*, and *OpenGL*. The *Application Framework* is the playground, Android's developers manual argues that this is a control playground "subject to security constraints enforced by the framework"(Application Framework, 2013, para. 2), where Android programs can share information and communicate with other programs. Some of the components that applications can interact with in the application framework are *Activities*, *Notifications*, *Resource Managers*, *Content Providers* and *Views*. *Applications* are the final core entity of the Android framework. Some applications are available through the default Android installation, but the majority of them are only a click away through third party repositories or "application distribution platforms," such as the Google Play marketplace (formerly known as the Android Market), where users can search for applications based on specific interests. It is evident that Android is a complex system, with many segments interacting with each other, in small devices that fit on the palm of our hands.

The **Android package** , also known as APK, is the file format used to package, distribute and install applications for Android systems. An APK file is a compressed file, commonly ending with the ".apk" file extension, that includes the application's code, resource data and metadata, such as the application's certificate, assets and the AndroidManifest.xml file. Certificates are primarily used to distinguish application authors/entities. AndroidManifest.xml is extremely important because it must be present

on all Android packages. Furthermore, it "presents essential information about the application to the Android system, information the system must have before it can run any of the application's code"(Android Manifest, 2013, para. 1). The application's compiled code is stored in the file "classes.dex." Classes.dex provides us with the compiled byte code of an Android application, including the java classes and their respective methods. As mentioned previously, the APK package includes an application certificate. The application's certificate identifies the author/entity accountable for an application. If this certificate is not present, an Android device will not install the application in question. However, it is not required to be signed by a certificate authority. Research by Vidas and Christin (2013) found that, in a sample of 76480 Android applications 99%, used self-signed certificates.

The **AndroidManifest.xml** file is an essential part of an Android application. This file is composed of elements tags and their respective attributes. Elements and attributes, configured in the manifest file, present us with a unique identity and behavior of the application being analyzed. For example, the *package* attribute within the *<manifest>* element must be unique for each application because it defines the name of the application. The Android developer's manual suggests that this package name be constructed by using Internet domain names, in reverse order, associated with the entity responsible for the application. The purpose of this naming scheme is to avoid naming conflicts of applications from other developers. Thus, if a company *example.com* wants to publish a new Android app called *newapp*, the attribute package must be *com.example.newapp*. The *package* attribute is also used during the runtime of an application as it becomes the process name presented by the Android system. Furthermore, once an application has been set with a specific package name and submitted to the Google Play Marketplace, it can not be changed. Attacks have been discovered by Vidas and Christin (2013) where non-malicious applications are repackaged with malware and submitted to Application marketplaces with different package names and falsified certificates.

A **Permission** is a restriction that limits access of applications to the Android system. *Intents* are messages between different software components, whose delivery is controlled by permissions. It is essential to re-emphasize that intercommunication between Android components is controlled by the flow of Intents. Yet, *Permissions* are at the core of this flow. An Android application without the necessary *Permissions* would simply not work. Likewise, *Permissions* need to be declared in the AndroidManifest.xml file of an application. Moreover, *Permissions* are only requested at install

time and not at run time. Once a set of *Permissions* has been granted, they can not be taken away unless the application is uninstalled. *Permissions* requests are configured by using the *<uses-permission>* element tag. Each of the permission values within a *<uses-permission>* tag is identified by a unique label, and "often the label indicates the action that's restricted"(Android Permissions, 2013, para. 1). For example, the *android.permission.CALL_EMERGENCY_NUMBERS* label would indicate that an application needs the permission *CALL_EMERGENCY_NUMBERS* from the *android.permission* entity. Many of these permissions are defined by the Android base platform, but custom *Permissions* can also be created by an application. The creation of custom *Permissions* is done by the <permission> label. *Permissions* are activated at install time by prompting end-users for access to a list of required permissions. Early versions of Android's permission prompts were difficult for end-users to understand, which caused them to be ignored, making it easier for users to enable the permissions required by malicious applications. Research by Vidas et al. (2011) on Android's Permission model notes that "unprivileged application attacks can take advantage of the complexity of the Android permission model and the Android market to persuade users to install malicious software and grant applications the permissions required to deliver harmful payload"(p.2). Recent versions of Android have changed the *Permission* prompts to include messages that are easier to understand. Nonetheless, it is still very common for end-users to ignore many of these message prompts, especially when attackers utilize ingenious social engineering tactics to trick users into installing malicious applications.

In summary, the *AndroidManifest.xml* file provides abundant information about Android applications. An investigator can obtain *Activities* , *Services* and *Broadcast Receivers* components just by analyzing this file. Additionally, *Intents* are the communication channels that make interaction possible between all Android components. *Intents* can also be located in the manifest file; consequently, one can find what communication channels are needed by an application. Finally, *Permissions* are required for *Intents* to reach specific components of the Android system. These *Permissions* are also defined in the AndroidManifest.xml file, allowing malware analysts to have a better idea about malicious permissions that can increase the vulnerability of an Android system. However, while the *AndroidManifest.xml* file provides the core configurations and permissions needed by an application, it does not contain any executable code.

Executable Android code is fundamental to the operation of an Android application. The executable code is stored in a file named classes.dex. The content of classes.dex

interacts with the Dalvik system which, as of April, 2014, is the most commonly used runtime environment on the Android platform. Mobile malware for Android systems stores executable code in the classes.dex file as well; therefore, it is logical to believe that analyzing the contents of DEX files will allow us to observe trends in malware samples that are more prominent to create new offspring. For more information on Android internals, please refer to citation Application Framework (2013), Android Manifest (2013) and Android Permissions (2013).

2.4 Supervised Regression Analysis

Supervised regression analysis is a mathematical technique used to model relationships between input variables and their corresponding output values. The term "supervised" implies the availability of "ground truth data," or accurate input information, about input variables modeling a specific regression problem and their known results. Such a set of ground truth data is referred to as a **training set**. For example, if we wanted to predict a good estimate of housing prices for a specific city over time, the regression model would use both past and present housing market data. The process of incorporating ground truth information into a regression model is known as "training the model." Training a regression model uncovers potential relationships between the input (independent) variables, e.g., size of a house, number of rooms, and the output (dependent) variable, e.g., price of a house. Determining this relationship allows us to make predictions about future as yet unseen cases.

Figure 2.1 highlights the training phase of the **regression analysis process**. The regression process starts by training the regression model with a training set. MEDS creates a phylogenetic tree of the malware samples in the training set. Simultaneously, it extracts feature values from each of the malware samples in the training set. True generativeness values are obtained based on the phylogenetic tree previously constructed. The true generativeness values and the feature values obtained from the training set are used to the model coefficients which are know as θ_0 and θ_1. The θ values are calculated by a function called the cost function.

The job of the **cost function** is to calculate the model coefficients that can be passed on to a regression function, also known as the hypothesis function, denoted by the symbol h. The function h is a mathematical formula that takes as input a feature value, or set of features values, extracted from new samples, not previously analyzed. Additionally, the h function utilizes prediction parameters calculated by the cost function. In return, the h function produces a predicted generative value. In the case of the MEDS framework, the input features extracted during the training and prediction phases are features specific to malware samples and their phylogenetic trees. Consequently, the result of h is a value that predicts how generative that malware will be in the near future. The two regression models implemented in MEDS are univariate linear and logistic regression.

Univariate linear regression predicts continuous output values. The term continuous indicates the tendency of linear regression to calculate values that fall in a straight

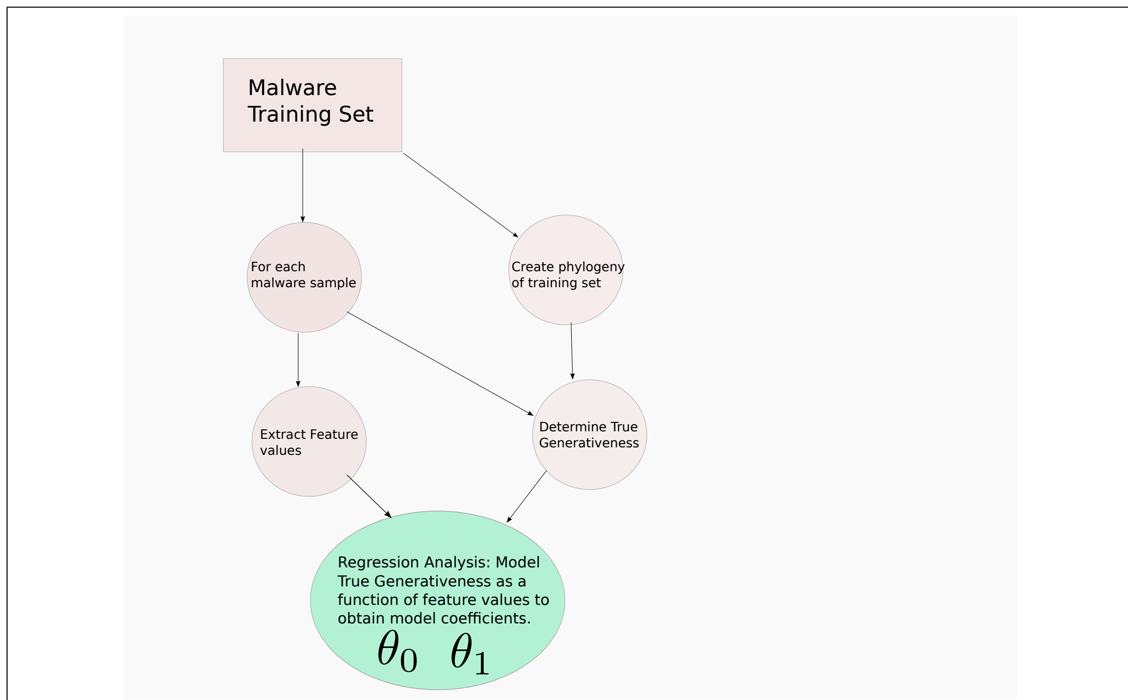

FIGURE 2.1: MEDS Training

line. There are a few terms that are important to emphasize for linear regression's prediction phase (shown in Figure 2.2). To begin calculating predictions, the h function of the predictive model needs prediction parameters. These prediction parameters are obtained during the training phase, and they are denoted by the symbols θ_0 and θ_1. Additionally, the predictive model also needs feature values extracted from new or unknown malware samples. The feature values are denoted by the symbols x_0 and x_1. In the MEDS framework, only x_1 contains feature values extracted from malware samples, and x_0 is always set to 1. The x_0 feature parameter is also known as the bias feature. Ultimately, the h function creates prediction values that are denoted by the symbol y. Keep in mind that that one of the objectives of MEDS is to make predictions about which malware samples will have a sufficient generative value and influence new malware variations in the future. Furthermore, the end-user indicates to the MEDS system what specific feature to extract to make these predictions. Some of the features currently implemented in MEDS will be discussed in-depth in the section "MEDS Framework Design."

The linear and logistic regression models are implemented by way of cost and predictions functions, as is typical in the machine learning framework. The **cost function** used in MEDS is known as the **normal equation** (See Figure 2.3). The normal equation is implemented in Octave, an open source software designed for numerical computations,

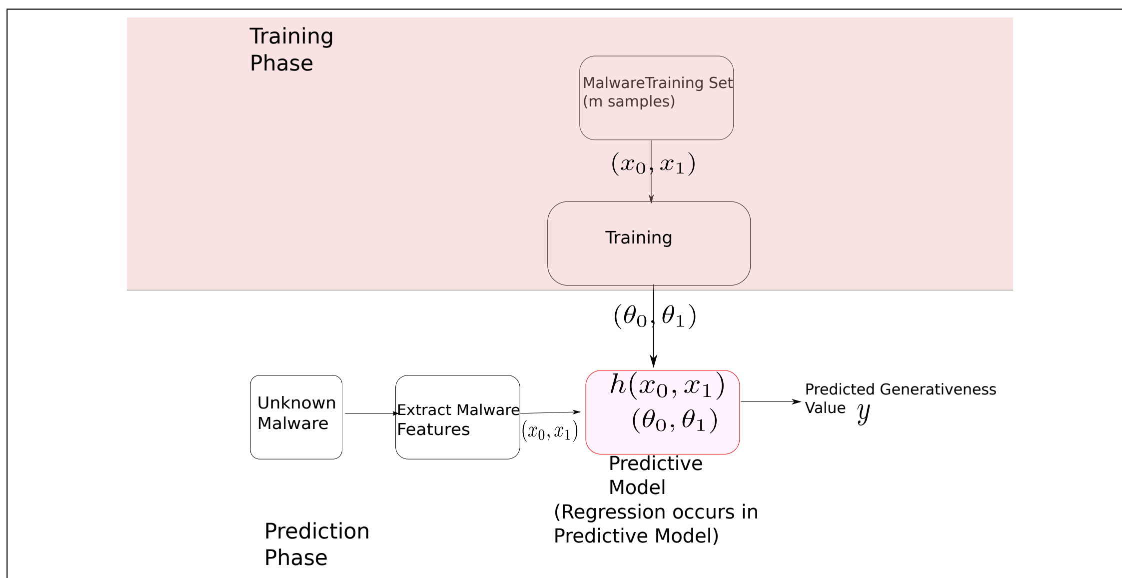

FIGURE 2.2: Univariate regression training and prediction phases.

in vectorized form. Features values are stored in a matrix X, while dependent variables are stored in a vector y. The normal equation allows us to calculate the optimum values of θ in a single matrix computation. Note that the symbol θ represents the vector pair values of θ_0 and θ_1.

$$\theta = (X^T X)^{-1} X^T y$$

FIGURE 2.3: Univariate linear regression cost function

The **prediction function** h is shown in Figure 2.4. As described before, the parameters θ_0 and θ_1 are determined from the feature values x_1, which are extracted from the malware training set via the cost function (See Figure 2.3).

$$h_\theta(x) = \theta_0 + \theta_1 x_1$$

FIGURE 2.4: Univariate linear regression prediction function

Univariate logistic regression models categorical data. Logistic regression is also known as classification, since its main objective is to determine whether the object being analyzed is or is not a member of a class. In the implementation of the MEDS framework, malware samples are classified as either generative or not. Logistic regression follows the same steps layout outlined in Figures 2.1 and 2.2. The major differences are in the cost and prediction functions.

The cost and prediction functions depend on the **sigmoid function**. The sigmoid function outputs values between 0 and 1, which is what we want for our logistic regression classifier. The form of the sigmoid function is shown in Figure 2.5. The value of z shown in Figure 2.5 is a simple vectorized representation of Figure 2.4. The cost function of the sigmoid function is presented in Figure 2.6. The θ parameters are obtained by minimizing the value of $J(\theta)$.

$$z = \theta^T x$$
$$g(z) = \frac{1}{1 + e^{-z}}$$

FIGURE 2.5: Sigmoid function

$$J(\theta) = -\frac{1}{m}\left(\log\left(g\left(X\theta\right)\right)^T y + \log\left(1 - g\left(X\theta\right)\right)^T (1 - y)\right)$$

FIGURE 2.6: Logistic regression cost function.

This is better explained by the Octave code in Figure 2.7. Line 4 in Figure 2.7 shows that a vector initial_theta is created, with size based on the number of elements found in matrix X, which contains the feature values extracted from the training set. Line 5 sets optimization parameters for the fminunc Octave function. The fminunc function is an optimization function that determines the minimum value of the cost functions. Finally, Octave is asked to return the theta values, which are stored in the vector theta.

```
1 function [theta] = logistic(X, y)
2 %% Initialization
3 [m, n] = size(X);
4 initial_theta = zeros(n, 1);
5 options = optimset('GradObj', 'on', 'MaxIter', 400)
6 [theta, cost] = ...
7         fminunc(@(t)(costFunction(t, X, y)), initial_theta, options);
8
9 end
```

FIGURE 2.7: Minimizing cost function to obtain prediction values in Octave.

Once the optimum θ values are calculated by Octave, these values are used in the sigmoid function to act on feature values extracted the new unclassified malware samples. Since the sigmoid function generates a continuous value between 0 and 1, which represents a

probability of belonging to a generative class, one needs a threshold value to produce a concrete classification. In the MEDS framework, if the likelihood value for a new malware sample is greater than 0.5, the sample is classified as generative. Otherwise, it is labeled as non-generative.

Chapter 3

MEDS Framework

3.1 Limitations

To the author's knowledge the MEDS framework is the first open source framework for creating phylogenetic trees and making predictions of generative malware, there are a **few limitations** to keep in mind.

At present, this framework only **creates phylogenetic trees**. This means that there will be always one root from which all other samples originate from. However, the main intention is to emphasize that even with only a single rooted phylogenetic tree, one can still find malware families within that tree. Additionally, for the duration of this project, phylogentic trees provided satisfactory results.

At present, the framework only **extracts single features** from malware samples during the training and evaluation phase of supervised linear and logistic regression. The idea of this project was to find the information that could be provided with single features. As with everything else in real life, one has to test the basic case and find preliminary results before moving forward with more advanced algorithms. However, the MEDS framework has code implemented to deal with multiple features, but due to the time constraints of this project results are not available to be presented.

At present, the framework only has **two approximate matching algorithms**. However, the design of this framework allows others to create their own approximate matching algorithms. With that in mind, the following sections go into the MEDS framework in greater detail.

3.2 Framework Design

The **Malware Evolution Discovery System** is divided in modules.

The **evolution** module deals with the creation of phylogenies from malware collections. The resulting phylogenetic trees represent the evolution of malware over time. These phylogenies are a hypothesis of the evolution taking place in each of the malware samples being analyzed. Each of the transformations occurring within a malware sample has significant meaning. For instance, cyber criminals constantly need to update their malware inventory to avoid detection by antivirus software. Obviously, new malware samples are not created from scratch, but some modifications need to be applied to cause them to appear completely different. Investigators usually calculate fingerprints, i.e., mathematical hash values, to determine if two binary files are different. Hashes, by design, do a very good job at determining if two items are exactly the same or completely different. However, with modified malware, better known as polymorphic malware, investigators are primarily concerned about the level of similarities between two malware samples.

If we can obtain a close approximation of the similarity value between two objects, phylogenies are very good at **visually exhibiting evolution** modifications in malware. This tool has the advantage of detecting similar samples and forming clusters of them. The benefit of grouping similar objects may provide guidance to investigators because the analysis of one sample, from a specific cluster, can be applied to other members of the same family cluster, therefore, reducing the amount of time needed to analyze a large number of similar malware samples. However, determining how similar two objects are can be problematic.

There are a few difficulties with **detecting similarities** of two digital objects, and they are covered in greater detail on the section “Approximate Matching” of the background section of this document. Once an approximate matching value, or distance metric, is obtained from each malware sample, the MEDS framework proceeds to create a phylogeny tree. The information provided by phylogenies is full of important knowledge that can help us trace important changes over time in malware. Furthermore, this paper will explore the concept of making predictions of future generative malware based on current knowledge.

The **discovery module** is concerned with malware generativeness. It is also the second core section of the MEDS framework. The discovery component makes predictions about future malware trends based on models of presently known malware samples. It is well known that most malware is created through iterative mutations of previous malware. Therefore, the natural question that comes to mind is whether we can predict which of our presently known malware might experience mutations in the future based on the malware we currently have. In MEDS, this is accomplished by creating linear and logistic regression models towards such predictions. The aim is to be able to predict the potential of a malware sample to create new malware variations in the future. In this thesis, the property of a malware sample to produce new offspring is its **generativeness**. Seeking to know which malware is likely to be generative of new species is a new approach with the significant potential consequence. However, it is challenging to make accurate predictions. Preliminary results show the difficulty in making good predictions of malware generativeness. On the other hand, it is important to emphasize that this is a new concept and will require additional research efforts to improve upon the initial results in this thesis.

3.3 Phylogenetic Tree Creation

MEDS takes as input a directory of **malware samples**. Each malware sample in this directory is processed by creating an APKDirectoryFactory object. The final product created by the directory factory class is an object called RamResidentMC, which at its core contains a list data structure to store each malware sample. Figure 3.1 shows the UML, i.e., Unified Modeling Language, diagram for the directory factory class and the RamResidentMC class. The job of the malware corpus class is to create a sorted list based on the creation date of each malware. The creation date approximation depends on the binary file being analyzed. For instance, the creation date on APK files is calculated by analyzing the creation date of the AndroidManifest.xml file of each APK file.

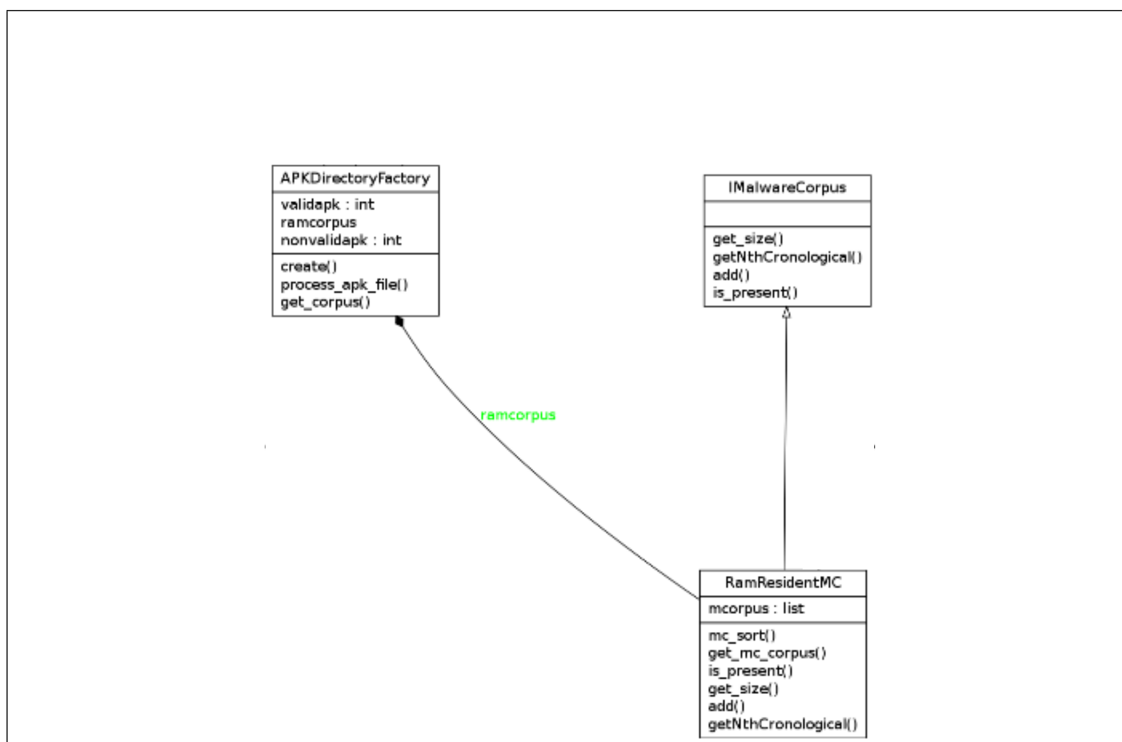

FIGURE 3.1: APKDirectory Factory and RamResientMC classes.

Information such as creation date is controlled by the **fingerprint implementation**. A fingerprint defines how a malware sample being analyzed will be represented within MEDS, and it allows us to extract information from each malware sample that can be used to calculate information (e.g., creation dates). At the moment, there are two types of fingerprints implemented in MEDS: the "lossless" fingerprint and the "Android manifest" fingerprint. The lossless fingerprint loads the complete object into memory and uses the complete object as it resides on disk as the fingerprint of the malware.

This can be expensive in terms of load time and memory consumption, but it provides the benefit of allowing us to work with complete binary objects. The Android Manifest fingerprint, on the other hand, uses metadata in the Manifest XML file as the fingerprint of the malware. The metadata used for this fingerprint is the AndroidManifest.xml file, which is present in all Android applications. Figure 3.2 shows the UML diagram for fingerprints implemented in MEDS and their respective factories to create them. Future extension of MEDS may consider additional fingerprint definitions.

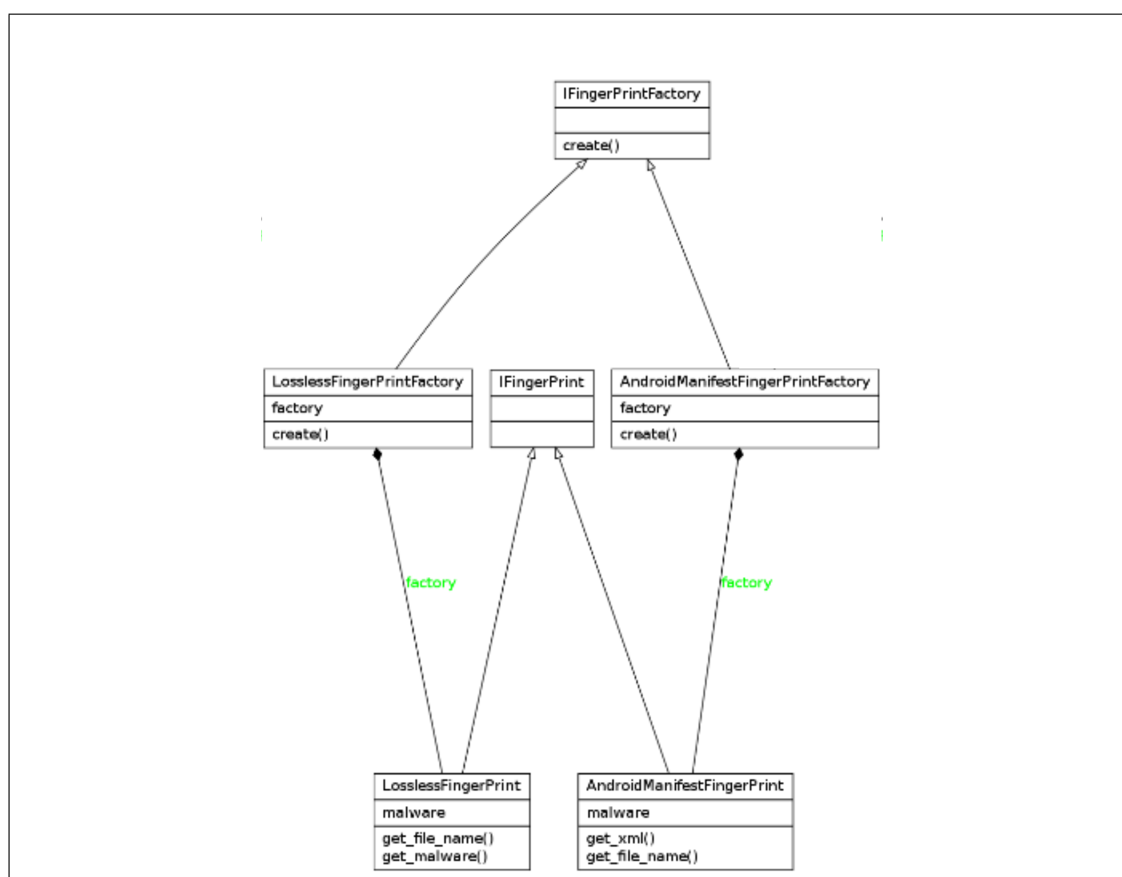

FIGURE 3.2: Fingerints implemented in MEDS.

The advantage of this class design is that **new fingerprint objects** can be easily incorporated for the analysis of different binary executables. Analysts can create their own custom fingerprint definitions to suit their needs. Fingerprints are the basis from which MEDS calculates pairwise distance metrics (approximate matching values). Approximate matching values help us determine how different two malware samples are. The bigger the difference, the less likely the samples are to have been derived from each other, or have a recent common ancestor. Figure 3.3 shows the UML diagram of all the implemented approximate matching algorithms. Of the implemented approximate

matching algorithms only ratcliff and sdhash are presently used. The ratcliff algorithm calculates an approximate matching value based on the edit distance between a pair of Android Manifest fingerprints. On the other hand, the sdhash metric utilizes fuzzy hashing to calculate the distance between a pair of lossless fingerprints.

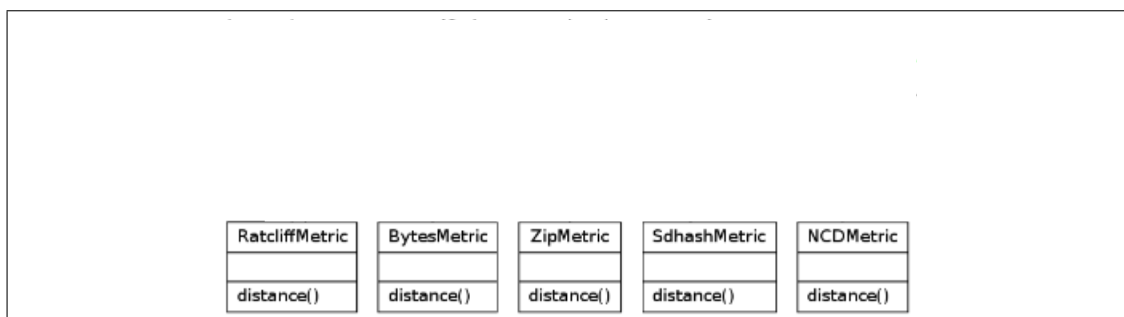

FIGURE 3.3: Approximate matching metrics.

The MEDS framework creates **phylogenetic trees** based on approximate matching values and creation dates of each malware sample. Figure 3.4 shows the relationship between the tree model that drives the creation of rooted phylogenetic trees and the RAMResidentGraph object.

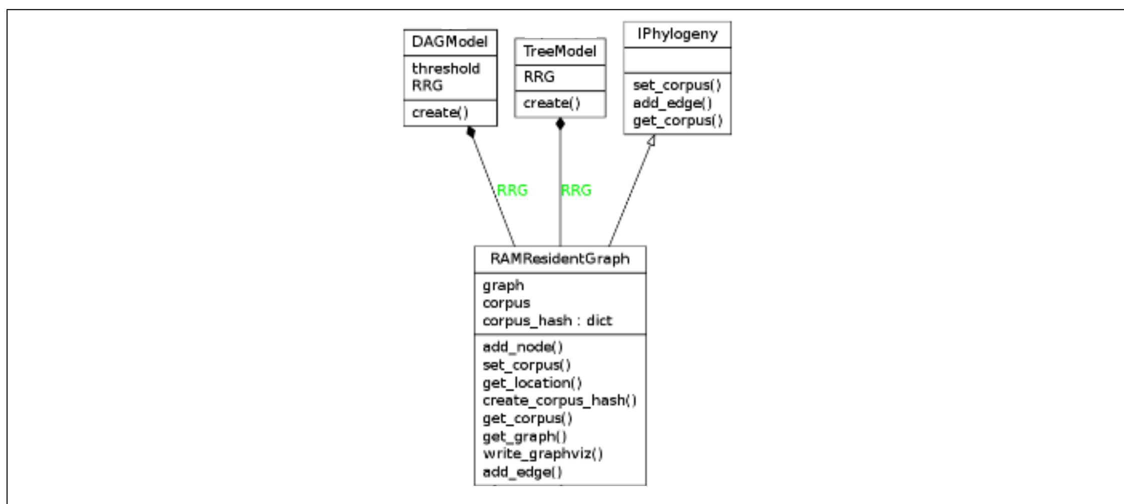

FIGURE 3.4: RAMResidentGraph drives the creation of rooted phylogenetic tree.

The actual creation of the phylogeny is accomplished by the code shown in Figure 3.5.

The for loop on line 9 iterates over the sorted malware corpus starting from the second malware sample. During each iteration, the malware sample is first extracted (line 10 of Figure 3.5) from the corpus and gets stored into a variable called m representing malware1. The second loop, line 12, iterates over all the malware samples. During

```
 5     def create(self,malwarecorpus,fingerprintfactory,distancemetric):
 6         self.RRG = RAMResidentGraph()
 7         self.RRG.set_corpus(malwarecorpus)
 8         self.RRG.create_corpus_hash()
 9         for i in range(1,malwarecorpus.get_size()):
10             m = malwarecorpus.getNthCronological(i)
11             mins = float('infinity')
12             for j in range(i):
13                 m2 = malwarecorpus.getNthCronological(j)
14                 s = distancemetric.distance(fingerprintfactory.create(m),
15                                             fingerprintfactory.create(m2))
16                 if s < mins:
17                     mins = s
18                     x = m2
19             self.RRG.add_edge(x,m,mins)
20         return self.RRG
```

FIGURE 3.5: Rooted phylogeny creation code.

each iteration, a malware sample gets extracted based on chronological order and stored into the variable $m2$ (which stands for malware 2). At each iteration m and $m2$ are compared based and an approximate matching value is calculated. If this value is less than a threshold specified in the variable *mins* (initially set to infinity) replace *mins* with the approximate matching value, such that with each subsequent iteration of the [inner, second] loop *mins* keeps track of the minimum approximate matching value. After the two loops have been completed, all approximate matching values have been calculated and each malware sample has been inserted into the rooted phylogeny tree through its association with the most similar predecessor. The final product is a rooted phylogenetic tree, such as the ones in Figures 3.6 and 3.7.

Figures 3.6 and 3.7 show two examples of phylogenetic trees, each of which were created from 10 malware samples. For each figure, the approximate matching value of sample 0 and 1 is greater than 70 percent. Therefore, sample 2 is branching away from sample 0. On the other hand, samples 2,3 and 4 have an approximate matching value of 0 when comparing them with node 1. Consequently, these nodes are very similar to node 0. We can also see that nodes 2, 3 and 4 were created with a minimum creation date difference. For example, nodes 2, 3 and 4 were created at the same hour on the same day but only a few minutes apart (See Figure 3.7). The insight, gained from examining these phylogenies presented is of great value because we know now that the malware cluster of nodes 1, 2, 3 and 4 can be dealt with by analyzing only node 1. On the other hand, Table 1 shows us the problem with traditional digital hash signatures. There, we see that each sample has a very different MD5 signature. Traditionally, an investigator who guides his/her investigation based only on MD5 hashes would conclude that malware samples 1-4 were different, when in reality they are very similar. Phylogenetic analysis also

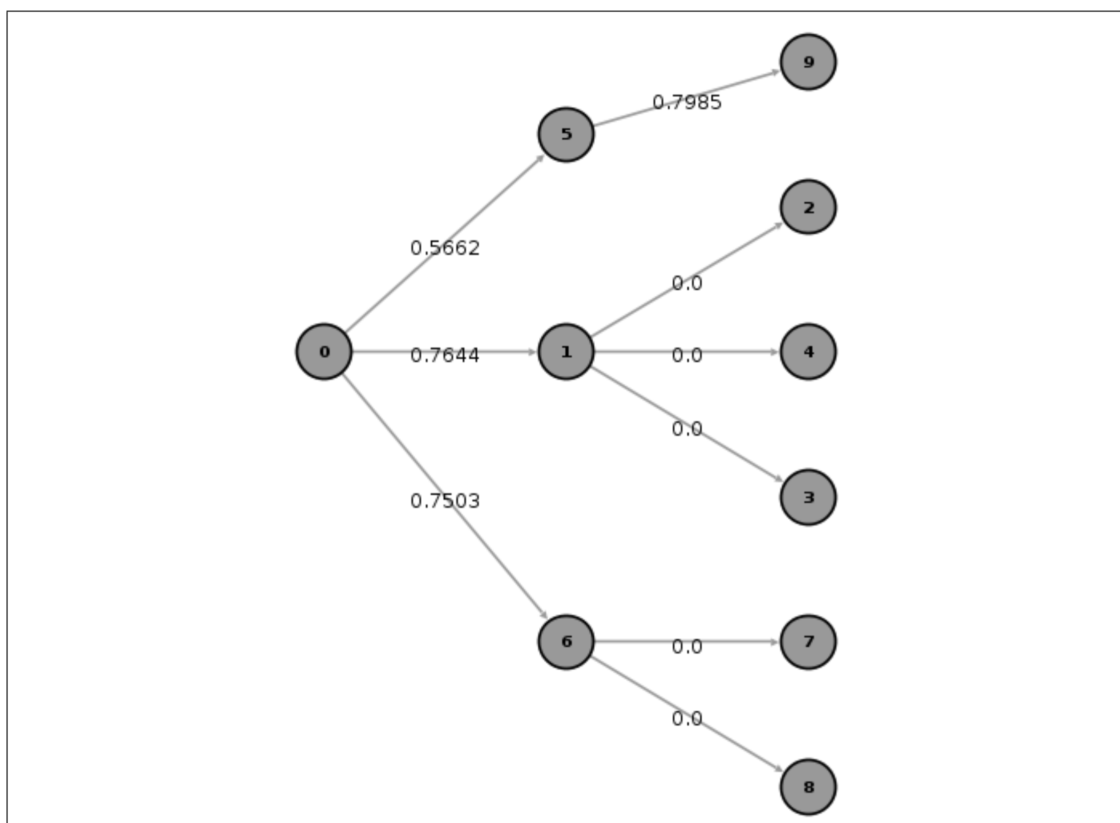

FIGURE 3.6: Phylogeny of 10 malware samples.

highlights issues with naming schemes adopted by antivirus companies. For example, based on the names (Andr.Exploit.Ratc) for nodes 5 , 6, 7 and 8, one would think that these malware samples would be clustered together. However, they are not, since nodes 5 and 9 are on a different cluster than nodes 6, 7 and 8.

#	MD5 Hash	Creation Date	Name
0	5b087aef1247591b1efe78032476bde7	2010-10-08 16:46:58	Andr.FakePlayer-11
1	ffb376be1e8d8311d320f7a107caee9a	2010-10-15 17:03:18	Andr.Kmin-1
2	5d22de7163fdd4d91a2722930410e950	2010-10-27 11:27:24	Andr.Kmin-1
3	1a1e517c6c5bf03704da790294d83533	2010-10-27 11:27:36	Andr.Kmin-1
4	2bf71a51b9879bd88a0f8b5d1c415cc8	2010-10-27 11:28:22	Andr.Kmin-1
5	abc88d92524a6eeebda1f8908f1e0725	2010-11-19 11:58:26	Andr.Exploit.Ratc
6	46965bd41dac0e4988515aa2f9f95b19	2010-11-21 14:04:06	Andr.Exploit.Ratc
7	d034ea7167c906d73c243332d3e4c9c2	2010-12-13 01:43:06	Andr.Exploit.Ratc
8	fc6adccbbef6e5394d0c2a1f3ac0f8a0	2010-12-24 17:12:22	Andr.Exploit.Ratc
9	83a98eabf044826622db7c211764cdf4	2010-12-31 09:56:00	Andr.Nickspy

TABLE 3.1: Malware hashes, creation date and name given by F-Secure Anti Virus

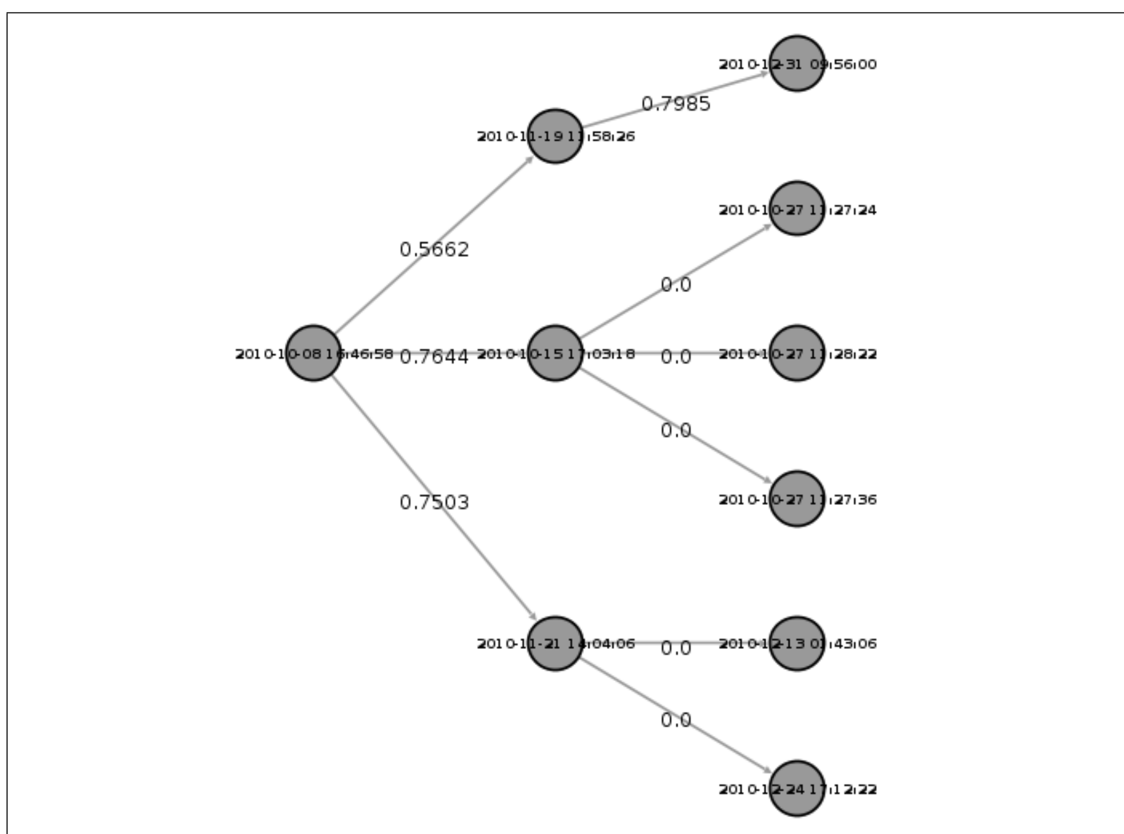

Figure 3.7: Phylogeny with approximate creation dates.

Chapter 4

Predicting Generativeness

The creation of generativeness predictions involves two steps: (I) model generativeness from a training set, and (II) application of the model to predict the generativeness of new malware, which is then compared to the actual generativeness (computable from phylogenies).

4.0.1 Training

Calculating prediction parameters involves creating a phylogeny tree that includes all currently known malware in a training set. The training set consists of a pair of phylogenectic trees T_1 and T_2. T_1 consists of 250 malware that emerge in a 4 month period, XX-YY. T_2 consists of 500 malware: the 250 that are in T_1, plus an additional 250 samples that appeared in the interval YY-ZZ. This pair of trees is used to compute the actual generativeness estimates for malware in T_1. These estimates are then correlated with features of the malware in T_1.

Features can be any information obtained from the malware or the actual phylogenetic tree where the malware sample is located. The list below are features that were extracted from the malware samples and the phylogenetic tree:

1. The number of dangerous permissions for each APK malware sample.
2. The number of receivers configured on each APK malware sample.
3. The total number of DEX classes found on the DEX binary file of each APK malware sample.

4. The total number of malicious children originating from a malware sample.
5. The approximate matching value of a malware sample to its parent.
6. The age (in seconds) of a malware sample from its parent.
7. The age (in seconds) of a malware sample from the root of the phylogenetic tree.
8. The age (in seconds) of a malware sample from the last malware sample in the phylogentic tree.
9. The age (in seconds) of the latest child found for each malware sample.
10. The age (in seconds) of the latest descendant of each malware sample.

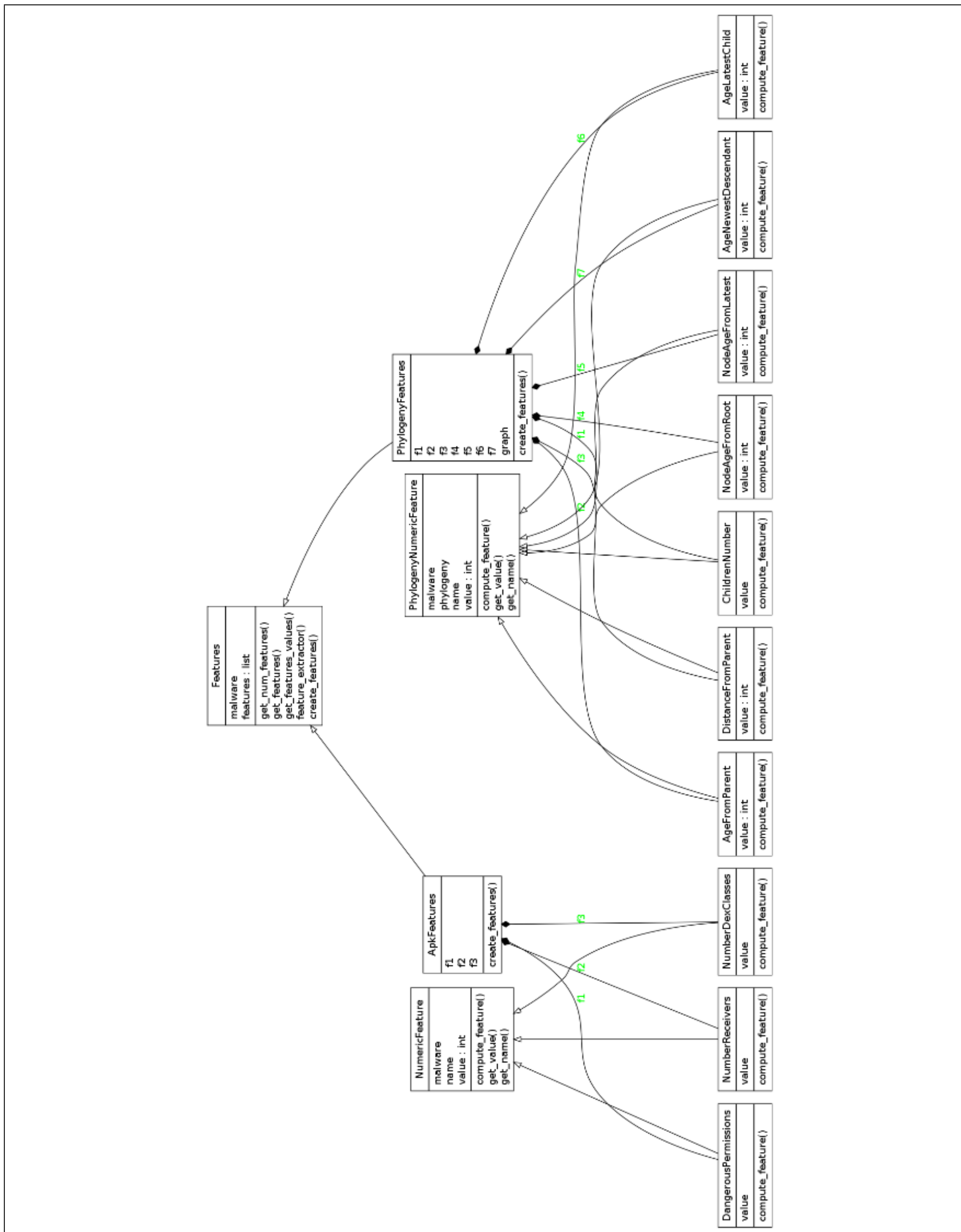

FIGURE 4.1: UML Diagrams of features used to calculate prediction parameters.

Figure 4.1 shows the UML diagram of each of the features implemented to create **generativeness predictions**. Each of these features provides important information about each malware's internal structure and its place in the phylogenetic tree. For example, permissions are essential to the Android ecosystem and malicious applications. Malicious applications need to use specific permissions for their malicious intentions. These permissions are called "dangerous" because their activation can impact the confidentiality and integrity of the Android system; therefore, it is important to known about these permissions and how current Android malware is applying them towards nefarious ends.

The MEDS framework utilizes a defined set of **malicious permissions** that have been found to be commonly used by Android malicious software. Examples of these dangerous permissions include the authority to read private information from the phone, and the ability to send SMS messages without using the dialer. Each malware's permissions are examined and compared to the "dangerous permissions" list. A counter is kept of the number of dangerous permissions requested by each malware sample. This count is provided as a measure of how dangerous the malware is compared to other samples. Since this information is hard-coded in each malware's manifest file, it is possible for MEDS to extract this information during training. In fact, the number of dangerous permissions is a specific attribute that can be used to train the linear and logistic regression algorithms to make generativeness predictions.

Figure 4.2 shows the XY scatter plot of the training set. One important observation about Figure 4.2 is that the actual number of dangerous permissions has not correlation at predicting generative malware. For example, the XY scatter plot shows that the most generative Android malware samples required 4 or more dangerous permissions. However, a large cluster of malware samples which required 4 or 5 permissions turned out to not be generative.

The **prediction parameters**, pparam1 and pparam2, are computed by running 3 different training sets, each containing 250 random malware samples from a larger set. The prediction parameters obtained for linear and logistic regression are presented in Tables 4.1 and 4.2.

StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Error
2011-09-01	2012-01-01	636	0.0047	-0.0001	0.33
2012-01-01	2012-05-01	2764	0.0030	0.0001	0.59
2012-05-01	2012-09-26	1685	0.0021	0.0002	0.57

TABLE 4.1: Dangerous permissions linear regression prediction parameters

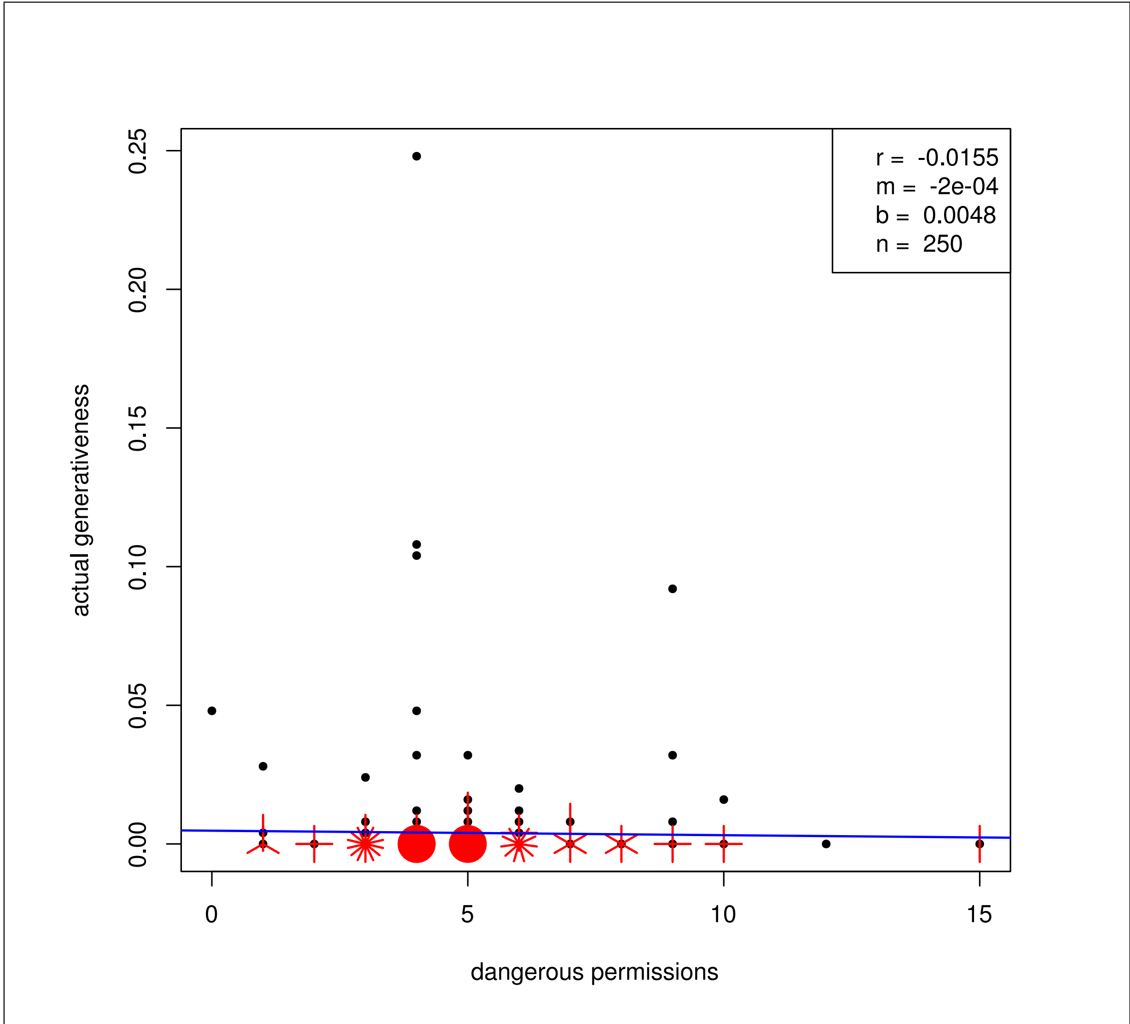

FIGURE 4.2: Number of dangerous permissions and actual generativeness values.

StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Accuracy
2011-09-01	2012-01-01	636	-0.0211	-0.0977	81.60%
2012-01-01	2012-05-01	2764	-0.0192	-0.0981	90.00%
2012-05-01	2012-09-26	1685	-0.0160	-0.0987	88.40%

TABLE 4.2: Dangerous permissions logistic regression prediction parameters

The prediction parameters for **linear and logistic univariate regression** are used to obtain generative prediction values of each malware, these values and their usage to calculate generativeness predictions will explained in greater detail in the evaluation section. Meanwhile, Table 4.1 shows that the error percentage of linear regression increases on each successive training set. In addition, there is a correlation between the malware size pool and the error percentage on each training set. For example, the error rate is 59 percent for the set with 2764 malware samples. In comparison, the training set with

636 samples has an error rate of 33 percent; therefore, as the malware set size increases the error rate increases as well. On the other hand, logistic regression (See Figure 4.2) achieved high accuracy percentages. For example, the training set with 2768 samples obtained an accuracy of 90 percent.

APK broadcast receivers, better known as “receivers,” allow an Android application to be notified about specific system-wide messages created by the internal system or other programs installed on the mobile device. Receivers are a feature extracted from the APK malware files. The main purpose of receivers is to pass information about specific events that can only be processed by recipients/applications whose broadcast receivers are registered with the Android system to accept such messages. Examples include notifications of events, such as receiving or making a call, receiving or sending an SMS message, or a notification that a mobile phone has been restarted. Receivers can be configured statically by defining them in the *AndroidManifest.xml* file. Figure 4.3 shows the XY plot of a training set used to calculate the prediction parameters based on the number of receivers feature.

Figure 4.3 shows that as the number of receivers increases the **generative value** of a malware sample decreases. This was an unexpected finding because applications with more receivers are able to extract more data from a mobile device, therefore, becoming more dangerous. Figure 4.3 also shows that there is a a large amount of malware samples with zero generative value that require 0 to 15 receivers. Finally, the malware sample that was most prolific in Figure 4.3 has zero number of receivers and a generative value of 20 percent. These results show that there is no correlation between the number of receivers and the generativeness of a malware sample. Tables 4.3 and 4.4 show the prediction parameters calculated by using the number of receivers as a generativeness prediction feature. Table 4.4 shows that logistic regression achieves an accuracy level of 80 percent or more. On the other hand, the error values for linear regression are from 31 to 56 percent. This is a similar behavior previously observed with the number of dangerous permissions feature.

The **DEX file** *classes.dex* contains the executable code of an Android applications. Some of the information that can be obtained from a DEX file is the number of classes that a malware sample needs to execute. Furthermore, the content of *classes.dex* interacts with the Dalvik system which, as of April, 2014, is the most commonly used

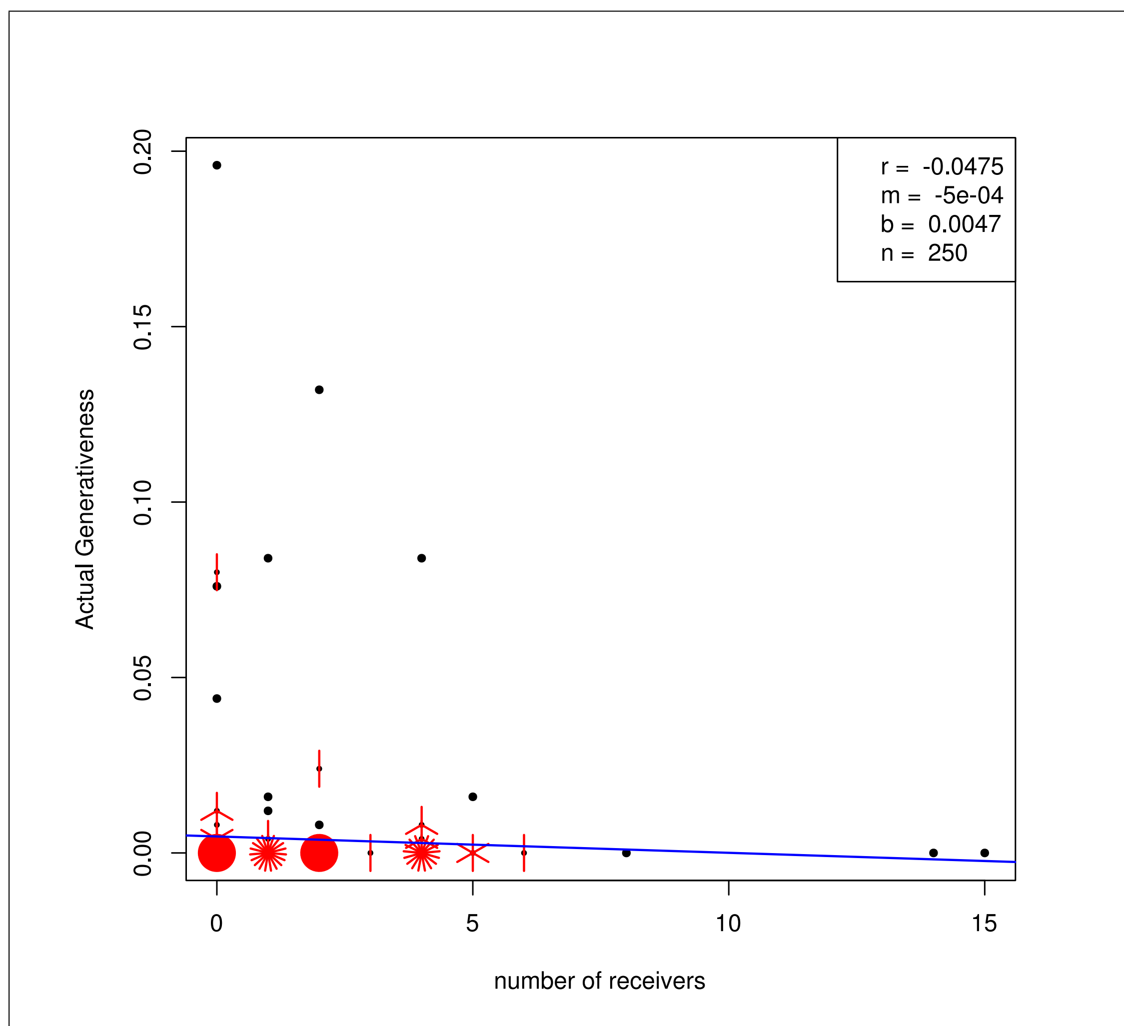

FIGURE 4.3: Number of receivers and actual generativeness.

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Error
1	2011-09-01	2012-05-01	636	0.0046	-0.0004	0.31
2	2012-01-01	2012-09-01	2764	0.0020	0.0012	0.56
3	2012-05-01	2013-01-01	1685	0.0101	-0.00213	0.53

TABLE 4.3: Number of receivers linear regression prediction parameters

runtime environment on the Android platform, therefore, it is compelling to extract the total number of classes in the DEX file to predict malware generativeness. Figure 4.4 shows that all malware samples with generative values greater than 1 percent have less than 1000 DEX classes in their code, and that the majority of the malware samples have zero generative value but they can have anything from 0 to 2000 DEX classes. In other words, the regression line plotted over the data set in Figure 4.4 is influenced by the non-generative malware samples, therefore, providing a weak correlation of a malware

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Accuracy
1	2011-09-01	2012-05-01	636	-0.0560	-0.0828	82.80%
2	2012-01-01	2012-09-01	2764	-0.0584	-0.0811	88.40%
3	2012-05-01	2013-01-01	1685	-0.0281	-0.0959	89.20%

Table 4.4: Number of receivers logistic regression prediction parameters

generativeness based on its number of DEX classes. Tables 4.5 and 4.6 show the prediction parameters calculated by linear and logistic regression base on the number of DEX classes. In this case Table 4.5 shows that the error value increases with each subsequent training sets for linear regression. Table 4.6 shows that the accuracy of logistic regression on the training set is above 82 percent.

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Error
1	2011-09-01	2012-05-01	636	0.0041	-1.1477e-06	0.349
2	2012-01-01	2012-09-01	2764	0.0050	-2.5375e-06	0.607
3	2012-05-01	2013-01-01	1685	0.0031	4.5524e-06	0.566

Table 4.5: Linear regression prediction parameters calculated for the number of DEX classes

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Accuracy
1	2011-09-01	2012-05-01	636	-0.0038	-0.0018	82.00%
2	2012-01-01	2012-09-01	2764	-8.1274e-06	-0.0011	88.40%
3	2012-05-01	2013-01-01	1685	-5.1093-06	-0.0006	90.00%

Table 4.6: Logistic regression prediction parameters for the number of DEX classes

The feature **number of children** takes advantage of the information obtained through a phylogenetic tree. Phylogenies have the advantage of visually showing clusters of similar malware samples, furthermore, when a cluster is dense the intuition is that the parent of that malware cluster is generative. With this in mind, the number of children feature was implemented in the MEDS framework. Figure 4.5 shows the XY scatter plot of the number of children of each malware sample and their respective generativeness values. This figure shows that samples with a high number of children have very low generativeness. This graph also shows that there are few malware samples with significant generativeness that have 0 to 15 number of children.

Tables 4.7 and 4.8 show the prediction parameters obtained from the 3 training sets of linear and logistic regression with the number of children as a feature. Table 4.7

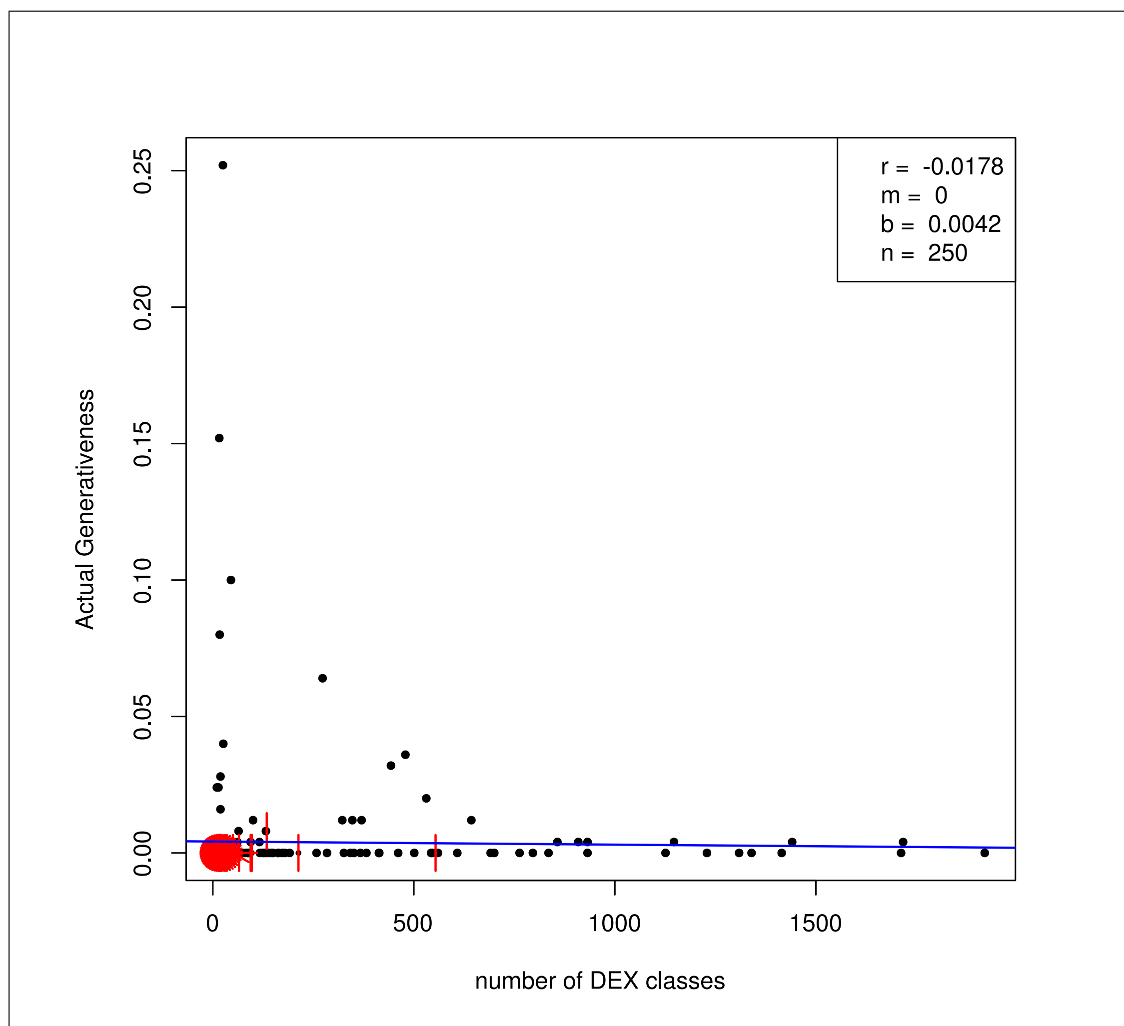

FIGURE 4.4: Number of DEX classes and actual generativeness

shows the error rate increasing as the number of total number of samples increases in each training set of linear regression. Table 4.8 shows the accuracy level for logistic regression to be above 84 percent. Table 4.7 also shows that the training with 2768 malware obtained an accuracy value of 88 percent, yet its predictions parameters are zero. This result will be examined in more detail in the evaluation phase of linear and logistic regression.

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Error
1	2011-09-01	2012-05-01	636	0.0036	0.0003	0.312
2	2012-01-01	2012-09-01	2764	0.0030	0.0009	0.626
3	2012-05-01	2013-01-01	1685	0.0037	0.0002	0.572

TABLE 4.7: Linear regression prediction parameters for the number of children

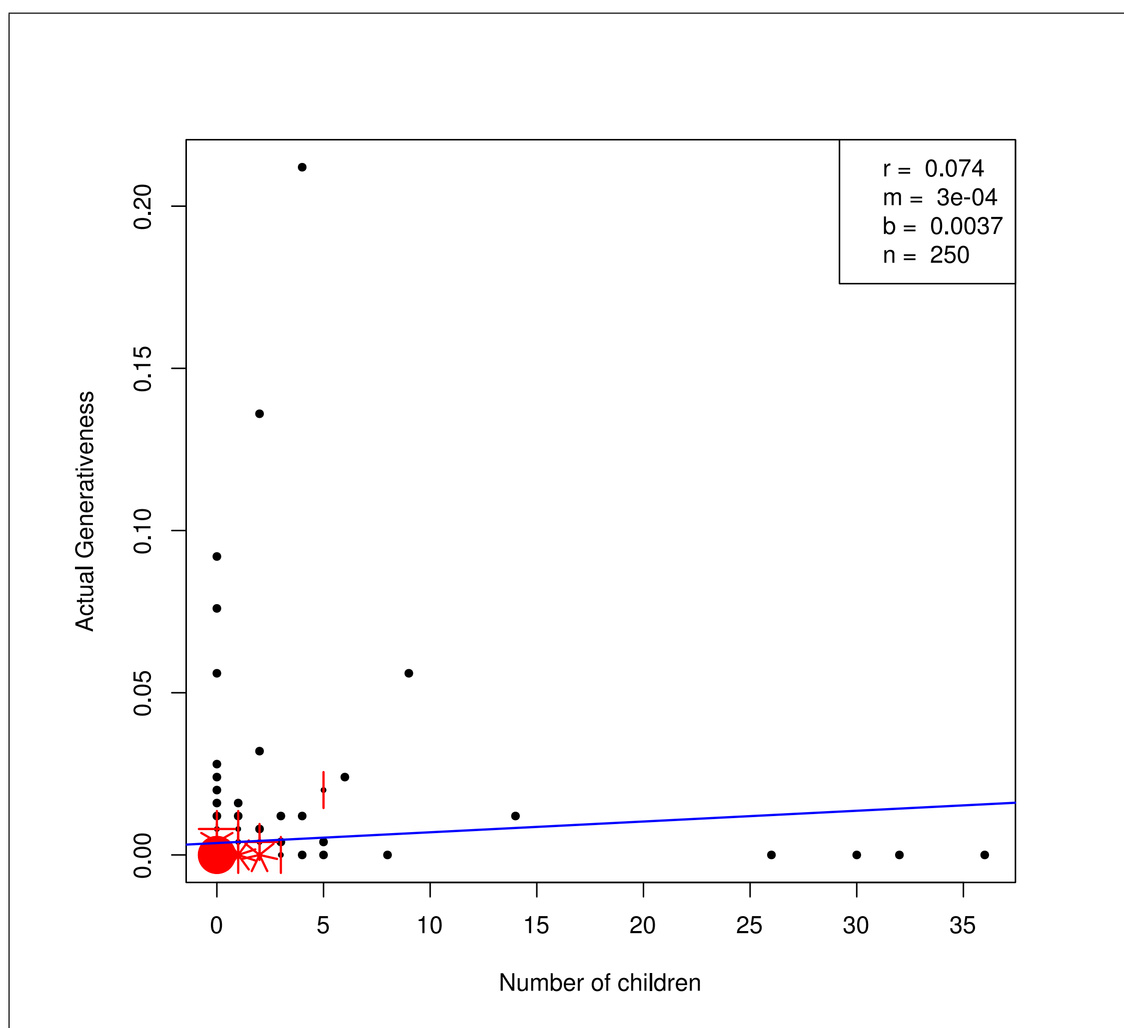

FIGURE 4.5: Number of Children actual generativeness

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Accuracy
1	2011-09-01	2012-05-01	636	-0.0832	-0.0553	84.00%
2	2012-01-01	2012-09-01	2764	0	0	88.00%
3	2012-05-01	2013-01-01	1685	-0.0868	-0.0495	89.20%

TABLE 4.8: Logistic regression parameters for the number of children

The next generative prediction feature is related to the **approximate matching value** between a malware sample and its predecessor. During the phylogenetic tree creation this value gets stored as the edge value between a parent and its descendant, therefore, making it very trivial to extract the approximate matching value from a phylogenetic tree. The intuition behind this feature is that as malware evolves it will branch out from its parent and create new malware samples, hence, becoming generative. Figure 4.6 shows that the most generative malware samples have an approximate matching

distance to their predecessor of 21 to 60 percent. It also that only one sample has become degenerated, i.e., completely different, with a very small generativeness. Tables 4.9 and 4.10 show the prediction parameters calculated for the approximate matching malware feature.

Table 4.9 shows that the first training set has the smallest value of all the **linear regression error** training samples. However, it increases for subsequent training sets. As mentioned previously, the increase of the error is correlated with the total number of malware samples in the training sets. Table 4.10 shows that the accuracy percentages for logistic regression are greater than 82 percent. This is a similar pattern found on the previous 4 features discussed. However, in this case all the prediction parameters are set to zero. Yet, the accuracy percentages for all the logistic regression training sets is above 83 percent. This will be analyzed in more detail on the evaluation phase.

TS	StartDate	Completion Date	Malware Set Size	pparam1	pparam2	Error
1	2011-09-01	2012-05-01	636	0.0029	0.0044	0.287
2	2012-01-01	2012-09-01	2764	0.0055	-0.0049	0.607
3	2012-05-01	2013-01-01	1685	0.0036	0.0024	0.532

Table 4.9: Linear regression prediction parameters for approximate matching values

TS	StartDate	Completion Date	total sample size	pparam1	pparam2	Accuracy
1	2011-09-01	2012-05-01	636	0	0	82.60%
2	2012-01-01	2012-09-01	2764	0	0	88.00%
3	2012-05-01	2013-01-01	1685	0	0	90.00%

Table 4.10: Logistic regression prediction parameters for approximate matching values

The additional 5 features implemented in MEDS **obtained similar results**, therefore, they will not be discussed. Instead, the prediction parameters obtained from linear and logistic regression for each of the features analyzed previously were used to calculate generativeness predictions of malware variants. The next section goes into greater detail about the results obtained when using such prediction parameters.

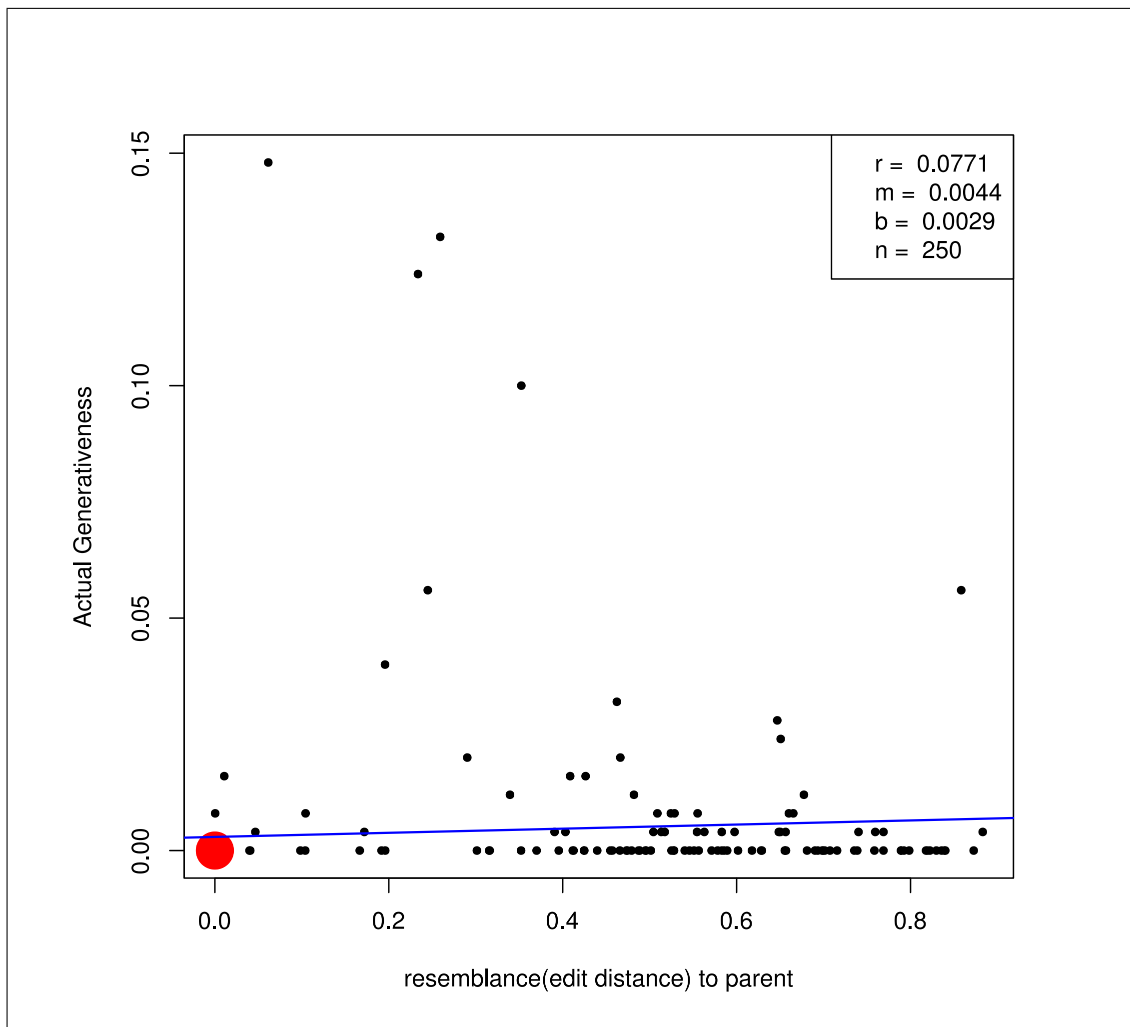

FIGURE 4.6: XY scatter plot of approximate matching value and actual generativeness

4.0.2 Evaluation of Prediction Parameters

Linear regression in the training sets demonstrated that as the number of malware samples increased the error value increased. Linear regression also showed that the regression line plotted was determined by the large number of non-generative samples. However, this information was obtained by testing the prediction parameters on the training sets. This section analyzes the results obtained by applying the prediction parameters on randomly chosen evaluation sets. These evaluations sets were obtained by randomly choosing 250 malware samples with a creation date XX-YY, then choosing another 250 samples with creation date YY-ZZ. The samples with creation dates XX-YY were sorted and stored in a list called $m1$. Additionally, a second list $m2$ contained the malware samples with creation dates XX-YY and YY-ZZ. This was done four times for each time periond XX-ZZ. The results obtained are for the features: number of dangerous permissions, number of receivers, number of DEX classes, number of children and approximate matching value.

Table 4.11 shows that the error values calculated on the evaluation sets match the behavior seen of linear regression on the training sets by applying the number of dangerous permissions feature. For example, each of the trials samples on evaluation set 1 have error values between 0.25 and 0.34 percent. On the other hand, the error values for the evaluation sets 2 and 3 show an increase from set 1. Figure 4.7 shows the XY scatter plot of the actual and predicted generativeness values for trial 1 of evaluation set 1. This plot implies that most malware samples have 0 value as their actual generativeness. However, linear regression is calculating values between 0.003 to 0.005, but it never calculates 0. This was expected because linear regression tries to fit a line that can describe the data provided which clearly shows that most malware is not generative.

Set	Trial	sdate	date1	date2	r	m	b	m1	m2	Error
1	1	2011-09-01	2011-09-01	2012-05-01	0.0885	0.0014	0.004	250	250	0.29
1	2	2011-09-01	2011-09-01	2012-05-01	0.0459	0.0007	0.0039	250	250	0.34
1	3	2011-09-01	2011-09-01	2012-05-01	0.026	0.0005	0.004	250	250	0.25
1	4	2011-09-01	2011-09-01	2012-05-01	0.0261	0.0005	0.0039	250	250	0.3
2	1	2012-01-01	2012-01-01	2012-09-01	-0.0082	-0.0001	0.004	250	250	0.57
2	2	2012-01-01	2012-01-01	2012-09-01	-0.0106	-0.0001	0.004	250	250	0.6
2	3	2012-01-01	2012-01-01	2012-09-01	-0.0144	-0.0002	0.0041	250	250	0.58
2	4	2012-01-01	2012-01-01	2012-09-01	-0.0083	-0.0001	0.0041	250	250	0.46
3	1	2012-05-01	2012-05-01	2013-01-01	-0.0596	-0.001	0.0041	250	250	0.46
3	2	2012-05-01	2012-05-01	2013-01-01	-0.0301	-0.0005	0.0041	250	250	0.48
3	3	2012-05-01	2012-05-01	2013-01-01	-0.0094	-0.0002	0.004	250	250	0.46
3	4	2012-05-01	2012-05-01	2013-01-01	-0.0128	-0.0003	0.004	250	250	0.48

Table 4.11: Dangerous permissions error values for linear regression on random evaluation sets

Table 4.12 shows the accuracy percentage obtained from logistic regression using the number of dangerous permissions feature on the evaluation sets. The highlights of table 4.12 are that the accuracy percentages for all the evaluations sets are above 83 percent. Additionally, by analyzing the number of negatives and false negatives, we can conclude that the accuracy of logistic regression is above 80% logistic regression which confirms that most malware is non-generative.

Set	Trial	sdate	date1	date2	Positives	Negatives	False Positives	False Negatives	Accuracy
1	1	2011-09-01	2012-01-01	2012-05-01	0	209	0	41	83.60%
1	2	2011-09-01	2012-01-01	2012-05-01	0	208	0	42	83.20%
1	3	2011-09-01	2012-01-01	2012-05-01	0	213	0	37	85.20%
1	4	2011-09-01	2012-01-01	2012-05-01	0	211	0	39	84.40%
2	1	2012-01-01	2012-05-01	2012-09-01	0	227	0	23	90.80%
2	2	2012-01-01	2012-05-01	2012-09-01	0	226	0	24	90.40%
2	3	2012-01-01	2012-05-01	2012-09-01	0	225	0	25	90.00%
2	4	2012-01-01	2012-05-01	2012-09-01	0	225	0	25	90.00%
3	1	2012-05-01	2012-09-01	2013-01-01	0	226	0	24	90.40%
3	2	2012-05-01	2012-09-01	2013-01-01	0	228	0	22	91.20%
3	3	2012-05-01	2012-09-01	2013-01-01	0	230	0	20	92.00%
3	4	2012-05-01	2012-09-01	2013-01-01	0	232	0	18	92.80%

Table 4.12: Dangerous permissions accuracy values for logistic regression on evaluation sets

Number of receivers had a similar error value on the evaluation sets. Table 4.13 shows that the lowest error values are found in all the trials of evaluation set 1. The error values increased for trials of evaluation sets 2 and 3. This is the same behavior seen on the training set using the number of receivers feature. Figure 4.8 shows the XY scatter plot

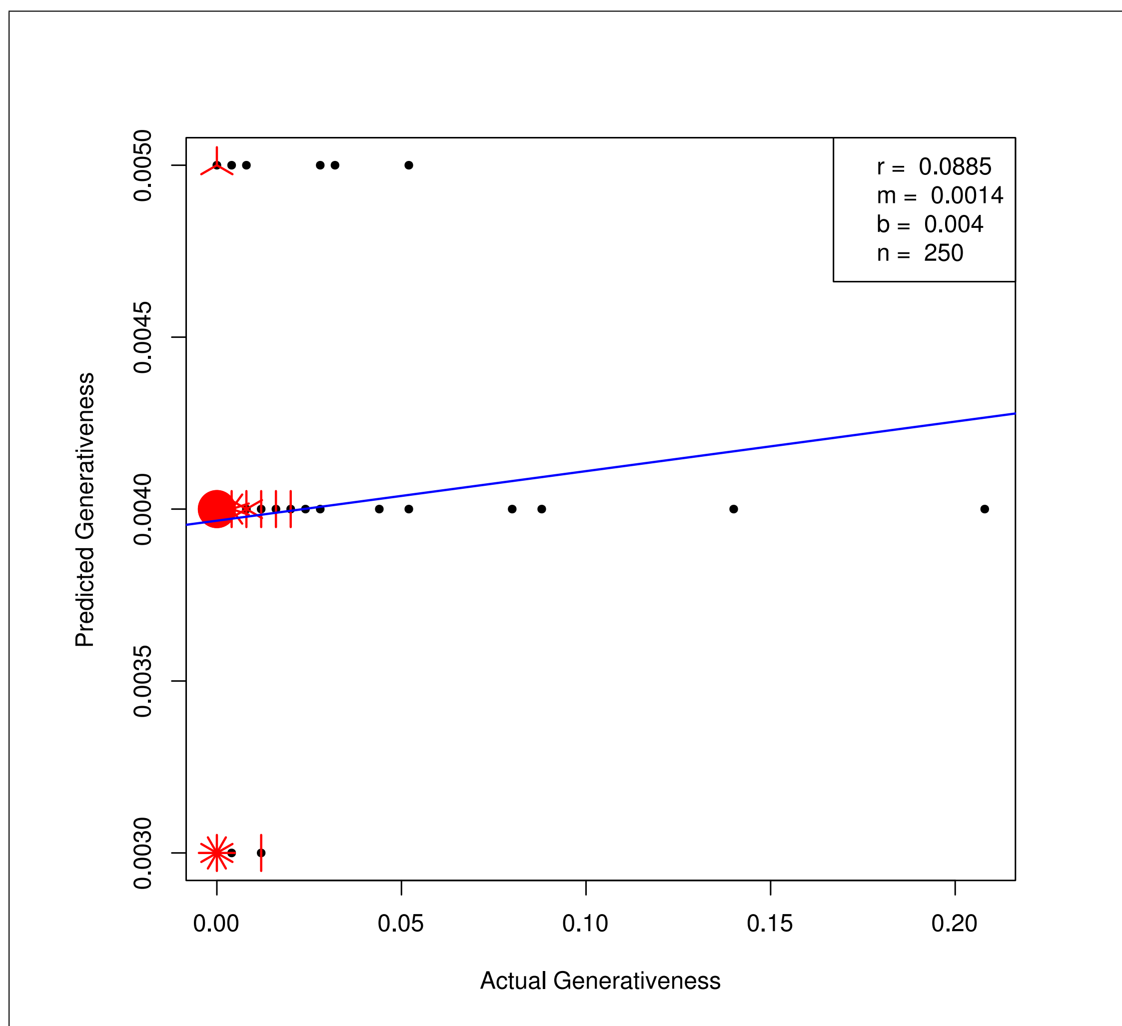

FIGURE 4.7: Actual and predicted generativeness using number of dangerous permissions feature

of the actual and predicted generativeness values of the number of receivers feature. This graphs shows that the linear regression model predicts low generative values for samples that had actual generativeness of 0. As mentioned before, this is expected because linear regression tries to fit a straight line on the data presented. However, this graph also shows that 4 samples with an actual significant generativeness values have predictions values from 0.002 through 0.006. For example, the malware sample on the far right of Figure 4.8 has an actual generative value of 0.6 but linear regression predicted a value of 0.006.

Logistic regression for the **number of receivers** predictions parameters has an accuracy of 80 percent and above. Logistic regression is influenced by non-generative malaware,

Set	Trial	date1	date2	r	m	b	m1	m2	Error
1	1	2012-01-01	2012-05-01	-0.0397	-0.0021	0.0042	250	250	0.296
1	2	2012-01-01	2012-05-01	0.045	0.0021	0.0042	250	250	0.294
1	3	2012-01-01	2012-05-01	0.0181	0.0008	0.0042	250	250	0.336
1	4	2012-01-01	2012-05-01	0.0351	0.0018	0.0041	250	250	0.305
2	1	2012-05-01	2012-09-01	0.0829	0.0048	0.0039	250	250	0.631
2	2	2012-05-01	2012-09-01	0.0777	0.0049	0.0039	250	250	0.607
2	3	2012-05-01	2012-09-01	0.0694	0.0042	0.004	250	250	0.613
2	4	2012-05-01	2012-09-01	0.0767	0.0044	0.0039	250	250	0.649
3	1	2012-09-01	2013-01-01	0.124	0.0174	0.0041	250	250	0.488
3	2	2012-09-01	2013-01-01	0.1352	0.0158	0.0041	250	250	0.488
3	3	2012-09-01	2013-01-01	0.1511	0.0208	0.0041	250	250	0.438
3	4	2012-09-01	2013-01-01	0.1222	0.0152	0.0042	250	250	0.479

Table 4.13: Error values for linear regression based on number of receivers

therefore, it predicts that all new malware will be noo-generative. As explained before, this is not completely wrong because this model will be correct 80 percent or more of the time.

Set	Test	sdate	date1	date2	Positives	Negatives	False Positives	False Negatives	Accuracy
1	1	2011-09-01	2012-01-01	2012-05-01	0	214	0	36	85.60%
1	2	2011-09-01	2012-01-01	2012-05-01	0	210	0	40	84.00%
1	3	2011-09-01	2012-01-01	2012-05-01	0	211	0	39	84.40%
1	4	2011-09-01	2012-01-01	2012-05-01	0	203	0	47	81.20%
2	1	2012-01-01	2012-05-01	2012-09-01	0	220	0	30	88.00%
2	2	2012-01-01	2012-05-01	2012-09-01	0	224	0	26	89.60%
2	3	2012-01-01	2012-05-01	2012-09-01	0	218	0	32	87.20%
2	4	2012-01-01	2012-05-01	2012-09-01	0	229	0	21	91.60%
3	1	2012-05-01	2012-09-01	2013-01-01	0	224	0	26	89.60%
3	2	2012-05-01	2012-09-01	2013-01-01	0	229	0	21	91.60%
3	3	2012-05-01	2012-09-01	2013-01-01	0	223	0	27	89.20%
3	4	2012-05-01	2012-09-01	2013-01-01	0	222	0	28	88.80%

Table 4.14: Number of receivers accuracy values for logistic regression on evaluation sets

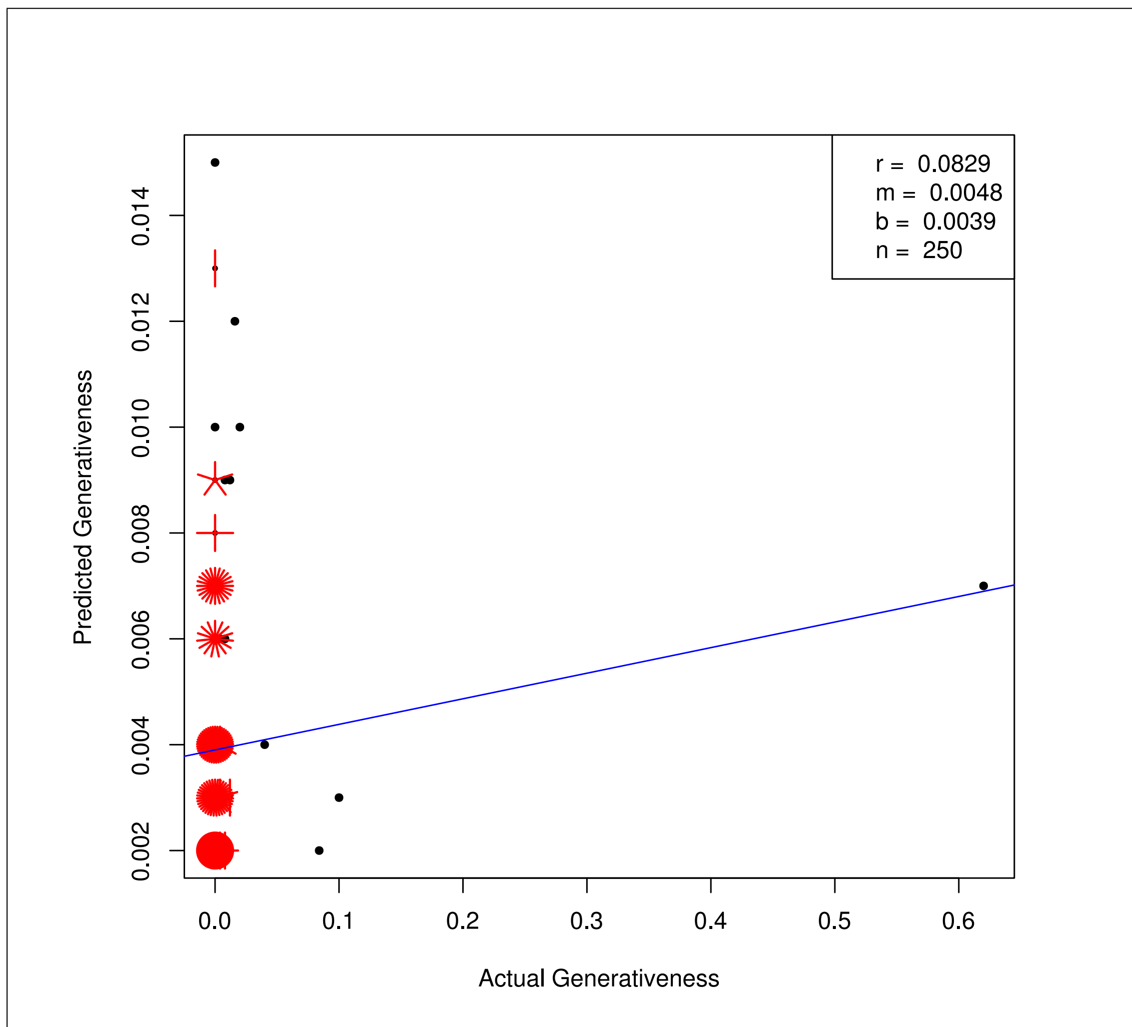

FIGURE 4.8: Actual and predicted generativeness based on number of receivers

Table 4.15 shows the error values obtained when applying the prediction parameters based on the number of DEX classes. In this case the error values start small and increased with each subsequent set. As mentioned before this is due to the fact that the complete size of the set increases as well.

Set	Test	sdate	date1	date2	r	m	b	m1	m2	Error
1	1	2011-09-01	2012-01-01	2012-05-01	-0.0059	-0.0001	0.0039	250	250	0.273
1	2	2011-09-01	2012-01-01	2012-05-01	0.014	0.0002	0.0039	250	250	0.33
1	3	2011-09-01	2012-01-01	2012-05-01	-0.1279	-0.002	0.0039	250	250	0.309
1	4	2011-09-01	2012-01-01	2012-05-01	0.0284	0.0004	0.0039	250	250	0.305
2	1	2012-01-01	2012-05-01	2012-09-01	0.0364	0.0014	0.004	250	250	0.629
2	2	2012-01-01	2012-05-01	2012-09-01	0.0359	0.0016	0.0039	250	250	0.592
2	3	2012-01-01	2012-05-01	2012-09-01	0.0476	0.0021	0.004	250	250	0.563
2	4	2012-01-01	2012-05-01	2012-09-01	0.035	0.0014	0.0041	250	250	0.566
3	1	2012-05-01	2012-09-01	2013-01-01	0.071	0.0046	0.0038	250	250	0.464
3	2	2012-05-01	2012-09-01	2013-01-01	0.0222	0.0013	0.0038	250	250	0.5
3	3	2012-05-01	2012-09-01	2013-01-01	0.0613	0.0039	0.0038	250	250	0.471
3	4	2012-05-01	2012-09-01	2013-01-01	0.025	0.0014	0.0039	250	250	0.484

Table 4.15: Error values for linear regression based on number of DEX classes

Figure 4.9 shows that linear regression tends to predict 3 generative values. This graph looks very similar to the XY scatter plot analyzed for the number of dangerous permissions feature. The problem with this behavior is that the actual generative values are disperse. Table 4.16 shows that logistic regression on the evaluation sets predict that all walware samples are non-generative and it will be correct about 80 percent of the time or more.

Set	Test	sdate	date1	date2	Positives	Negatives	False Positives	False Negatives	Accuracy
1	1	2011-09-01	2012-01-01	2012-05-01	0	209	0	41	83.60%
1	2	2011-09-01	2012-01-01	2012-05-01	0	206	0	44	82.40%
1	3	2011-09-01	2012-01-01	2012-05-01	0	210	0	40	84.00%
1	4	2011-09-01	2012-01-01	2012-05-01	0	205	0	45	82.00%
2	1	2012-01-01	2012-05-01	2012-09-01	0	225	0	25	90.00%
2	2	2012-01-01	2012-05-01	2012-09-01	0	223	0	27	89.20%
2	3	2012-01-01	2012-05-01	2012-09-01	0	220	0	30	88.00%
2	4	2012-01-01	2012-05-01	2012-09-01	0	222	0	28	88.80%
3	1	2012-05-01	2012-09-01	2013-01-01	0	223	0	27	89.20%
3	2	2012-05-01	2012-09-01	2013-01-01	0	230	0	20	92.00%
3	3	2012-05-01	2012-09-01	2013-01-01	0	224	0	26	89.60%
3	4	2012-05-01	2012-09-01	2013-01-01	0	225	0	25	90.00%

Table 4.16: Number of DEX classes accuracy values for logistic regression on evaluation sets

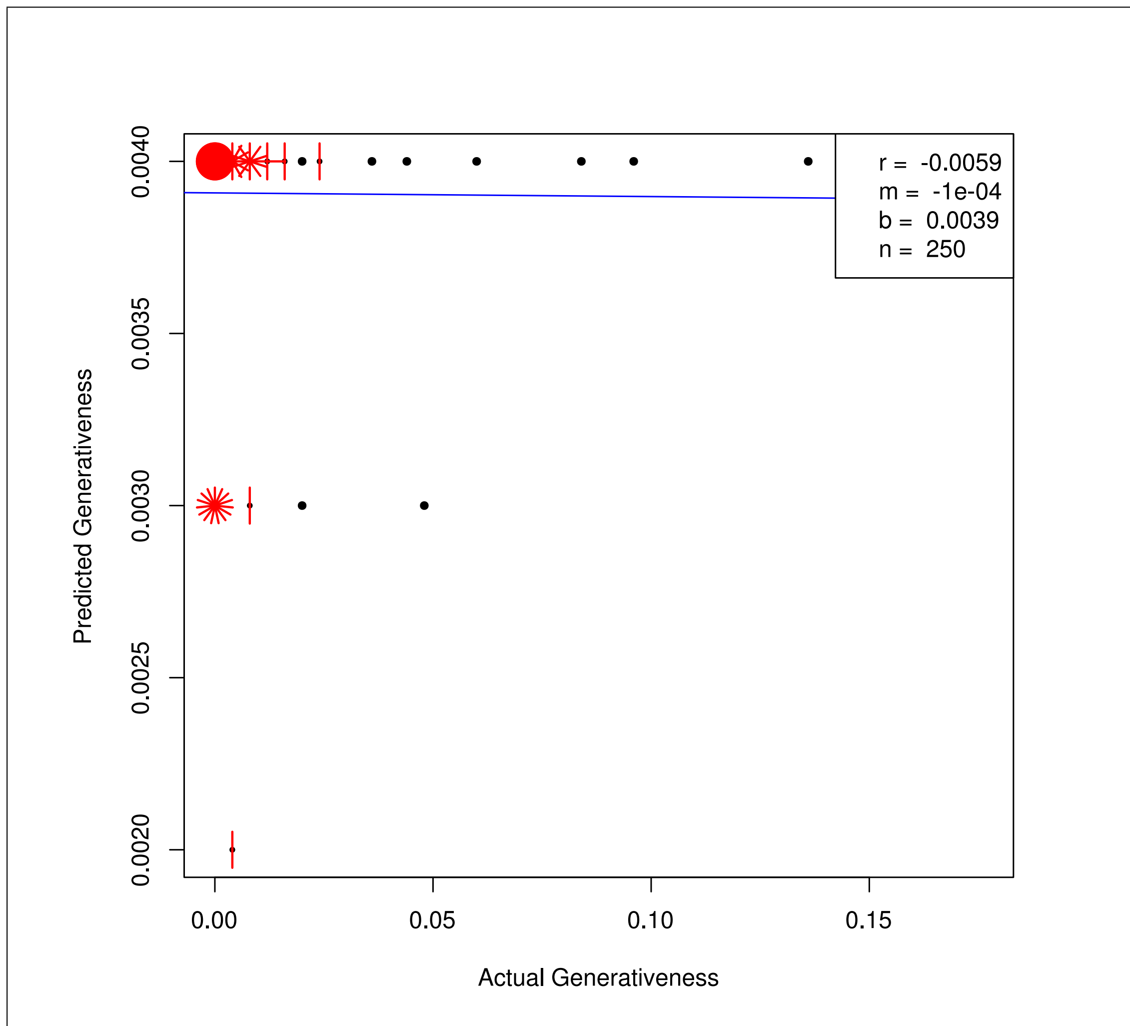

FIGURE 4.9: Actual and predicted generativeness values of evaluation set 1 and trial 1 based on number of DEX classes

Table 4.17 shows the error values for the number of children prediction parameters. In this case we also see that as the number of samples in a set increases, the error values increase as well. Figure 4.10 shows the XY scatter plot of the actual and predicted generativeness. Analyzing this plot shows that most malware samples are clustered on the lower left side of the graph. However, there are still very few samples that escape this cluster. For exaple, the malware sample on the lower right corner of the plot shows an actual generative value close to 0.6 but linear regression predicted a value very close to zero.

Set	Test	sdate	date1	date2	r	m	b	m1	m2	Error
1	1	2011-09-01	2012-01-01	2012-05-01	0.1477	0.0104	0.0042	250	250	0.29
1	2	2011-09-01	2012-01-01	2012-05-01	0.097	0.0072	0.0042	250	250	0.277
1	3	2011-09-01	2012-01-01	2012-05-01	0.0789	0.0048	0.0042	250	250	0.29
1	4	2011-09-01	2012-01-01	2012-05-01	0.0511	0.0036	0.0042	250	250	0.275
2	1	2012-01-01	2012-05-01	2012-09-01	0.0816	0.007	0.0039	250	250	0.634
2	2	2012-01-01	2012-05-01	2012-09-01	0.1701	0.0169	0.0039	250	250	0.593
2	3	2012-01-01	2012-05-01	2012-09-01	0.1778	0.0168	0.0039	250	250	0.619
2	4	2012-01-01	2012-05-01	2012-09-01	0.0772	0.0067	0.0039	250	250	0.619
3	1	2012-05-01	2012-09-01	2013-01-01	0.0552	0.0039	0.0042	250	250	0.456
3	2	2012-05-01	2012-09-01	2013-01-01	0.0378	0.0022	0.0042	250	250	0.494
3	3	2012-05-01	2012-09-01	2013-01-01	0.0252	0.0017	0.0042	250	250	0.491
3	4	2012-05-01	2012-09-01	2013-01-01	0.0332	0.0021	0.0042	250	250	0.49

TABLE 4.17: Error values for linear regression based on number of children

Table 4.18 shows the results of logistic regression based on the number of children. The accuracy percentages are high due to the fact that most malware samples are non-generative.

Set	Test	sdate	date1	date2	Positives	Negatives	False Positives	False Negatives	Accuracy
1	1	2011-09-01	2012-01-01	2012-05-01	0	212	0	38	84.80%
1	2	2011-09-01	2012-01-01	2012-05-01	0	210	0	40	84.00%
1	3	2011-09-01	2012-01-01	2012-05-01	0	211	0	39	84.40%
1	4	2011-09-01	2012-01-01	2012-05-01	0	208	0	42	83.20%
2	1	2012-01-01	2012-05-01	2012-09-01	25	0	225	0	10.00%
2	2	2012-01-01	2012-05-01	2012-09-01	29	0	221	0	11.60%
2	3	2012-01-01	2012-05-01	2012-09-01	28	0	222	0	11.20%
2	4	2012-01-01	2012-05-01	2012-09-01	28	0	222	0	11.20%
3	1	2012-05-01	2012-09-01	2013-01-01	0	226	0	24	90.40%
3	2	2012-05-01	2012-09-01	2013-01-01	0	226	0	24	90.40%
3	3	2012-05-01	2012-09-01	2013-01-01	0	221	0	29	88.40%
3	4	2012-05-01	2012-09-01	2013-01-01	0	223	0	27	89.20%

TABLE 4.18: Accuracy values for logistic regression based on number of children.

The last feature that is analyzed in this project is the **approximate matching** value between a malware sample and its predecessor. Table 4.19 shows that all the trials of evaluation set 1 have a small error value. However, this error increases with sets 2 and 3. Figure 4.11 shows that this feature assigns predicted generativeness values in five clusters. For example, by looking at the XY scatter plot we can see that the predicted generative values start at 0.03 and end at 0.07 with smallincrements. This means that there is a set of 5 predicted generative values that linear regression chooses from. For example, the sample on the lower right corner of the plot was assigned a predicted generativeness value of 0.03 percent but in reality had a generativeness value greater than 0.2 percent.

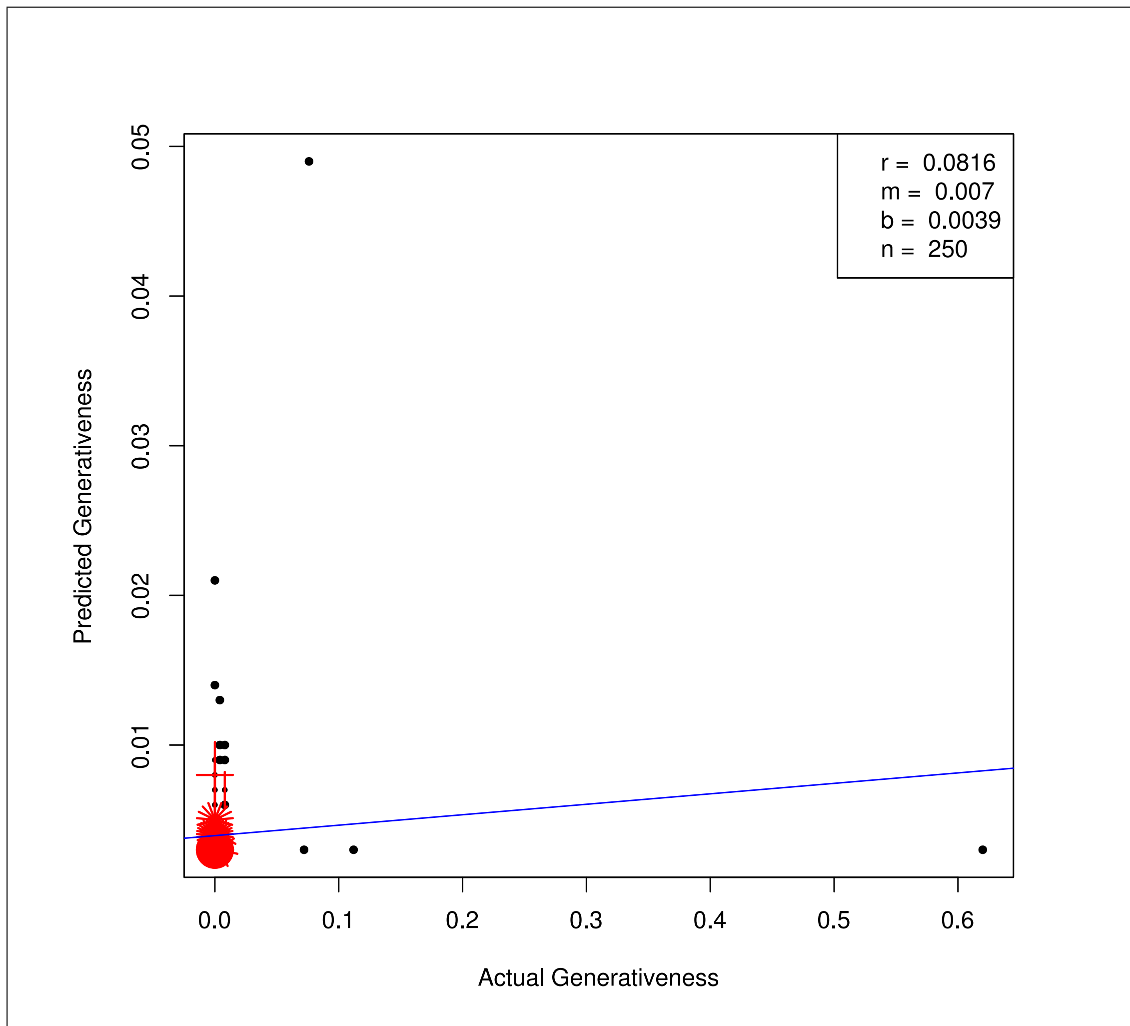

FIGURE 4.10: Evaluation set 2 and trial 2 actual and predicted generativeness values based on number of children

Table 4.20 show the results of logistic regression based on the **approximate matching distance** feature. This table shows that in this case logistic regression predicted that most malware is generative. This is incorrect because only very few malware samples are truly generative as we already seen in the previous 4 features analyzed.

Set	Test	sdate	date1	date2	r	m	b	m1	m2	Error
1	1	2011-09-01	2012-01-01	2012-05-01	0.0734	0.0053	0.004	250	250	0.291
1	2	2011-09-01	2012-01-01	2012-05-01	0.1666	0.0134	0.0039	250	250	0.268
1	3	2011-09-01	2012-01-01	2012-05-01	0.1118	0.0074	0.004	250	250	0.316
1	4	2011-09-01	2012-01-01	2012-05-01	0.1731	0.0123	0.004	250	250	0.298
2	1	2012-01-01	2012-05-01	2012-09-01	0.0249	0.0009	0.0039	250	250	0.605
2	2	2012-01-01	2012-05-01	2012-09-01	0.0027	0.0001	0.0039	250	250	0.606
2	3	2012-01-01	2012-05-01	2012-09-01	0.0211	0.0009	0.0038	250	250	0.551
2	4	2012-01-01	2012-05-01	2012-09-01	0.026	0.001	0.0038	250	250	0.586
3	1	2012-05-01	2012-09-01	2013-01-01	-0.009	-0.0002	0.0042	250	250	0.473
3	2	2012-05-01	2012-09-01	2013-01-01	0.0115	0.0002	0.0042	250	250	0.509
3	3	2012-05-01	2012-09-01	2013-01-01	-0.0189	-0.0003	0.0042	250	250	0.481
3	4	2012-05-01	2012-09-01	2013-01-01	0.0056	0.0001	0.0042	250	250	0.476

Table 4.19: Error values for linear regression based on approximate matching feature values

Set	Test	sdate	date1	date2	Positives	Negatives	False Positives	False Negatives	Accuracy
1	1	2011-09-01	2012-01-01	2012-05-01	41	0	209	0	16.40%
1	2	2011-09-01	2012-01-01	2012-05-01	43	0	207	0	17.20%
1	3	2011-09-01	2012-01-01	2012-05-01	35	0	215	0	14.00%
1	4	2011-09-01	2012-01-01	2012-05-01	33	0	217	0	13.20%
2	1	2012-01-01	2012-05-01	2012-09-01	25	0	225	0	10.00%
2	2	2012-01-01	2012-05-01	2012-09-01	24	0	226	0	9.60%
2	3	2012-01-01	2012-05-01	2012-09-01	25	0	225	0	10.00%
2	4	2012-01-01	2012-05-01	2012-09-01	26	0	224	0	10.40%
3	1	2012-05-01	2012-09-01	2013-01-01	23	0	227	0	9.20%
3	2	2012-05-01	2012-09-01	2013-01-01	27	0	223	0	10.80%
3	3	2012-05-01	2012-09-01	2013-01-01	25	0	225	0	10.00%
3	4	2012-05-01	2012-09-01	2013-01-01	26	0	224	0	10.40%

Table 4.20: Accuracy values for logistic regression based on approximate matching value

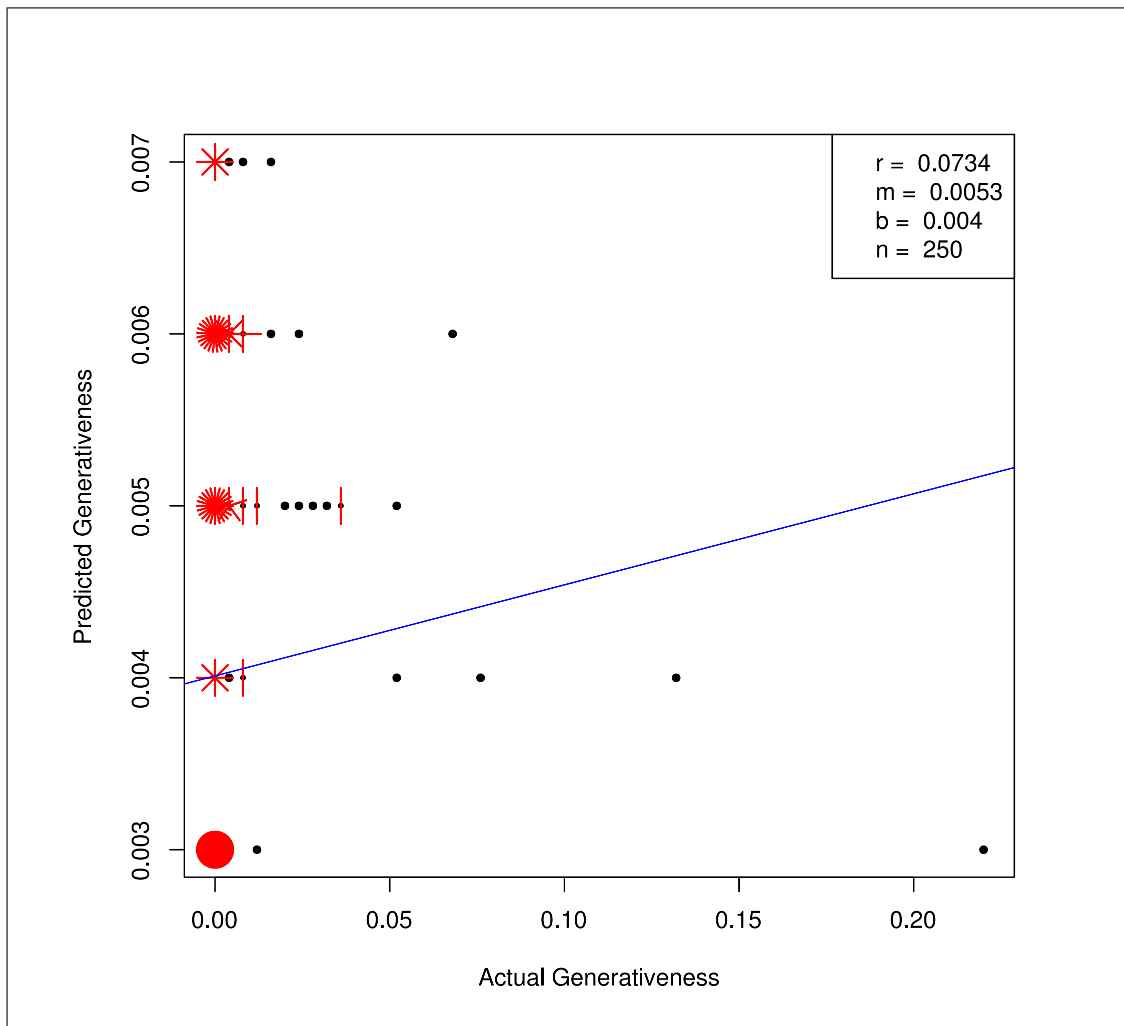

FIGURE 4.11: Evaluation set 1 and trial 1 actual and predicted generativeness values based on approximate matching feature values

Chapter 5

Conclusion and Future Work

Phylogenies provide **meaningful information** about malware evolution. They are powerful tools to visualize malware clusters and expansion. By construction phylogenies of real Android malware we visually located clusters of similar malware samples. The power of phylogenies for visualizing malware trends is attractive. At the core of creating good estimates of phylogenetic trees is the concept of approximate matching. Presently, approximate matching has been getting a lot of attention, and one of the future enhancements of MEDS will be to implement additional approximate matching algorithms.

Malware samples for most cases are modified copies of other malware. This is corroborated in this thesis since the number of **non-generative malware** surpasses the number of generative samples. It is non-lucrative for the malware industry to create "from scratch" malware samples. This benefits malware researchers because by analyzing previous malware there is evolutionary information that be obtained through phylogenetic trees. Phylogenetic analysis can help discover malware outbreaks and clustering of specific malware. In addition, they can show the evolution of malware samples over time which be used to predict generative malware.

Predicting generative malware will require further research and implementation of advanced machine learning methods. This thesis demonstrates the **power of supervised regression** analysis and the concept of predicting malware generativeness. At the moment, single malware features are extracted and used to make predictions of generative malware. This is a limitation, and the results obtained in the evaluation phase illustrate why it is important to look into advanced machine learning concepts, such as

neural networks. Furthermore, single malware features in supervised linear and logistic regression are influenced by non-generative malware. However, the option of using multiple features with quadratic function models to predict malware generativeness is another option for future work. The malware problem is not an easy problem. There is a complete industry behind creating malware and fighting it back. This thesis shows a different approach with the intention to benefit the digital forensics and cybersecurity communities. With that in mind, the MEDS framework is provided to anyone at the following URL https://www.nacolabs.com/meds.

Bibliography

Android Manifest (2013). Retrieved from http://developer.android.com/guide/topics/manifest/manifest-intro.html.

Android Permissions (2013). Retrieved from http://developer.android.com/guide/topics/manifest/manifest-intro.html.

Application Framework (2013). Retrieved from http://developer.android.com/about/versions/index.html.

Breitinger, F., Guttman, B., McCarrin, M., and Roussev, V. (2014). Approximate matching: definition and terminology. Retrieved from http://csrc.nist.gov/publications/drafts/800-168/sp800_168_draft.pdf.

Carrera, E. and Erdelyi, G. (2004). Digital genome mapping advanced binary malware analysis. In *15th Virus Bulletin International Conference*, pages 187–197.

Cohen, F. (1984). Computer viruses theory and experiments. *Computers and Security*, 6:22–35.

Darmetko, C., Jilcott, S., and Everett, J. (2013). Inferring accurate histories of malware evolution. In *Twenty-Sixth International FLAIRS Conference*.

DeGusta, M. (2012). Are smart phones spreading faster than any technology in human history? *MIT Technology Review*. Retrieved from http://www.technologyreview.com/news/427787/are-smart-phones-spreading-faster-than-any-technology-in-human-history/.

Dumitras, T. and Neamtiu, I. (2011). Experimental challenges in cybersecurity: a story of provenance and lineage for malware. In *4th conference on Cyber Security Experimentation and test*, page 9.

Elgin, B. (2005). Google buys android for its mobile arsenal. Retrieved from http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal.

Goldberg, L., Goldberg, P., Phillips, C., and Sorkin, G. (1998). Constructing computer virus phylogenies. *Journal of Algorithms*, 26(1):188–208.

Harris, A. E. and Perlroth, N. (2014). For target, the breach numbers grow. *New York Times*. Retrieved from http://www.nytimes.com/2014/01/11/business/target-breach-affected-70-million-customers.html.

Jang, J., Woo, M., and Brumley, D. (2013). Towards automatic software lineage inference. In *22nd USENIX conference on Security*, pages 81–96.

Karim, E., Walenstein, A., Lakhotia, A., and Parida, L. (2005). Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23.

Kephart, J. and White, S. (1991). Directed-graph epidemiological model of computer viruses. *Research in Security and Privacy*, pages 343–359.

Khoo, W. and Lio, P. (2013). Unity in diversity: phylogenetic-inspired techniques for reverse engineering and detection of malware families. In *SysSec Workshop*.

Krebs, B. (2014). A first look at the target intrusion, malware. Retrieved from http://krebsonsecurity.com/2014/01/a-first-look-at-the-target-intrusion-malware/.

Ma, J., Dunagan, J., Wang, H., Savage, S., and Voelkner, G. (2006). Finding diversity in remote code injection exploits. In *6th ACM SIGCOMM Conference on Internet Measurement*, pages 53–64.

Nmawston (2014). Android captured 79% share of global smartphone shipments in 2013. Retrieved from http://blogs.strategyanalytics.com/WSS/post/2014/01/29/Android-Captured-79-Share-of-Global-Smartphone-Shipments-in-2013.aspx.

Perlroth, N. (2012). Outmaneuvered at their own game, antivirus makers struggle to adapt. Retrieved from http://www.nytimes.com/2013/01/01/technology/antivirus-makers-work-on-software-to-catch-malware-more-effectively.html.

Ratcliff, J. W. and Metzener, D. E. (1988). Pattern matching: The gestalt approcah. Retrieved from http://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5.

Roussev, V. (2010). *Data Fingerprinting with Similarity Digests*, chapter 207-226. Springer.

Salemi, M. and Vandamme, A. (2009). *The Phylogenetic Handbook: A Practical Approach to Phylogenetic Analysis and Hypothesis Testing*. Cambridge University Press, New York, New York, 2nd edition.

Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, Upper Saddle River, NJ.

Vidas, T. and Christin, N. (2013). Sweetening android lemon markets: Measuring and curbing malware in application marketplaces. Retrieved from http://users.ece.cmu.edu/~tvidas/academics.html.

Vidas, T., Votipka, D., and Christin, N. (2011). All your droid are belong to us. Retrieved from http://users.ece.cmu.edu/~tvidas/academics.html.

Wehner, S. (2007). Analysing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320.

Wueest, C. (2014). The tenth anniversary of mobile malware. Retrieved from http://www.symantec.com/connect/blogs/tenth-anniversary-mobile-malware.

Index

Activities, 14
Android, 12
Android Manifest, 12
Android Package, 12
Android Permissions, 13
Antivirus Programs, 2
Approximate Matching, 9

Bloom Filter, 10
Broadcast Receiver, 14
Bytewise Approximate Matching, 9

Containment, 9
Cyber Genome Program, 7

Dalvik, 15

Evolution, 5

Generativeness, 3

Logistic Regression, 18

Malicious Software, 1
MEDS, 21
Mobile Malware, 2

Normal Equation, 17

Phylogenies, 23

Ratcliff/Obership Algorithm, 10
Regression Analysis, 16
Resemblance, 9

Sdhash, 10
Semantic Approximate Matching, 10
SequenceMatcher, 10
Services, 14
Smart Phones, 1
Syntactic Approximate Matching, 10

Univariate Linear Regression, 16