PREDICTORS OF RANSOMWARE FROM BINARY ANALYSIS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Aaron Otis

June 2019

© 2019
Aaron Otis
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Predictors of Ransomware from Binary Analysis

AUTHOR: Aaron Otis

DATE SUBMITTED: June 2019

COMMITTEE CHAIR: Zachary N J Peterson, Ph.D.
Associate Professor of Computer Science

COMMITTEE MEMBER: John Bellardo, Ph.D.
Associate Professor of Computer Science

COMMITTEE MEMBER: Maria Pantoja, Ph.D.
Assistant Professor of Computer Science

ABSTRACT

Predictors of Ransomware from Binary Analysis

Aaron Otis

Ransomware, a type of malware that extorts payment from a victim by encrypting her data, is a growing threat that is becoming more sophisticated with each generation. Attackers have shifted from targeting individuals to entire organizations, raising extortions from hundreds of dollars to hundreds of thousands of dollars. In this work, we analyze a variety of ransomware and benign software binaries in order to identify indicators that may be used to detect ransomware. We find that several combinations of strings, cryptographic constants, and a large number loops are key indicators useful for detecting ransomware.

ACKNOWLEDGMENTS

Thanks to:

- Shibani, who I owe immense gratitude for all her support.
- The Singh family, who has offered their complete support and accepted me into their family.
- D.r DeBruhl for all of his continuous academic support and fun projects.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . viii

LIST OF FIGURES . . . x

CHAPTER

1 Introduction . . . 1

2 Background and Related Work . . . 3

2.1 Ransomware . . . 3

2.2 Cryptography . . . 10

2.2.1 Symmetric Cryptography . . . 11

2.2.2 Asymmetric Cryptography . . . 12

2.3 Malware Analysis . . . 16

2.3.1 Binary Analysis . . . 17

2.3.2 Control Flow Analysis . . . 20

2.3.3 Loop Detection . . . 22

2.4 Statistics . . . 25

3 Methodology . . . 28

3.1 Disassembly . . . 28

3.2 Binary Statistics . . . 30

3.3 Loop Detection . . . 31

3.3.1 Control Flow Graph . . . 31

3.3.2 Dominator Tree . . . 32

3.3.3 DJ Graph . . . 32

3.4 Common Strings . . . 34

3.5 Cryptographic Constants 35
4 Results 37
4.1 Binary Statistics 37
4.1.1 Sample Type Statistics 38
4.2 Model Predictors 45
4.2.1 Benign Vs Cryptographic 46
4.2.2 Benign Vs. Ransomware 52
4.2.3 Cryptographic Vs. Ransomware 56
5 Conclusion 61
5.1 Future Work 61
BIBLIOGRAPHY 63
APPENDICES

LIST OF TABLES

Table Page

3.1 Bitwise instructions. 30

3.2 Various ransomware in which ransom messages were transcribed. . . 35

4.1 Summary of sample types (benign, cryptographic, and ransomware) of all binary samples based on platform (Unix and Windows). . . . 37

4.2 Summary of means of quantitative statistics for all samples with outliers removed. 38

4.3 Summary of statistics of all binary samples based on type (benign, cryptographic, and ransomware). 40

4.4 The variables used in the full multiple logistic regression model. Note that the qualitative variables have binary values that represent whether the item was found (1) in a binary sample or not (0). 47

4.5 Summary of multiple logistic regression on the reduced model for the benign and cryptographic subset of observations. Note that the intercept (b_0) for this model was found to be -1.660290. 48

4.6 Probabilities of success ($\pi(x)$) and estimated odds ratio (EOR) for each of the statistically significant variables corresponding to the presence of the listed string in the begin vs. cryptographic sample dataset. Note that negative EOR indicates a decrease in probability of success when the variable is 1. 50

4.7 Summary of multiple logistic regression on the reduced model for the benign and ransomware subset of observations. Note that the intercept (b_0) for this model was found to be -3.872675. 53

4.8 Probabilities of success ($\pi(x)$) and estimated odds ratio (EOR) for each of the variables corresponding to the presence of the listed nominal variable in the begin vs. ransomware sample dataset. Note that negative EOR indicates a decrease in probability of success when the variable is 1. 54

4.9 Summary of multiple logistic regression on the reduced model for the cryptographic and ransomware subset of observations. Note that the intercept (b_0) for this model was found to be -3.119408. 57

4.10 Probabilities of success ($\pi(x)$) and estimated odds ratio (EOR) for each of the variables corresponding to the presence of the listed nominal variable in the cryptographic vs. ransomware sample dataset. Note that negative EOR indicates a decrease in probability of success when the variable is 1. 59

LIST OF FIGURES

Figure		Page
4.1	Boxplot of each quantitative statistic for all samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.	39
4.2	Boxplot of each quantitative statistic for benign samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.	42
4.3	Boxplot of each quantitative statistic for cryptographic samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.	43
4.4	Boxplot of each quantitative statistic for ransomware samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.	44
4.5	Logistic regression predictions from the benign vs. cryptographic dataset for each variable listed.	49
4.6	Logistic regression predictions from the benign vs. ransomware dataset for each variable listed.	58
4.6	Continued logistic regression predictions from the benign vs. ransomware dataset for each variable listed.	59
4.7	Logistic regression predictions from the cryptographic vs. ransomware dataset for each variable listed.	60

Chapter 1

INTRODUCTION

Ransomware is a class of malware that encrypts victims' data, rendering it unaccessable to victims. In order to decrypt files, the ransomware authors demand a payment to be made, usually in the form of cryptocurrency, such as Bitcoin [17, 40]. Many researcher not that ransomware is on the rise [29, 35] and becoming more sophisticated [40, 46].

Researchers are reproting that ransomware authors are changing tactics to focus on larger targets, such as governmental bodies and large organizations, which cybercriminals have termed *big game hunting* [42]. A trend of targeting local governments is emerging [11].

Eurofins Scientific, the largest provider of forensic services in the UK, was infected by ransomware [18]. This has led to delays in forensic science provision, which led to some court hearing being postponed. Eurofins Scientific refused to comment on whether the ransom was paid or not. However, quick optimistic updates of returning to normal business prompted many to believe the ransom was paid sometime between June 10 and 24, 2019 [18].

Jackson County, Georgia was infected with ransomware that forced most of the local government's IT systems offline[13]. The county hired a consultant to negotiate with the ransomware authors and settled on a payment of $400,000. County officials noted it cost them less to pay the ransom than to rebuild their IT systems and cited officials from Atlanta, Georgia who spent millions rebuilt their systems after a ransomware attack.

Lake City, Florida paid $490,000 in Bitcoin to recover affected systems, including email and telephone services for the city [11]. More that 20 cities have been targest of ransomware in the first half of 2019 [11]. By targeting municipal organizations, ransomware authors can affect a larger number of people and demand a higher payment than with individuals.

In this work, samples of ransomware are compared to nonmalicous, or *benign*, software in order to determine properties that may be used to predict whether or not a given sample is ransomware. Additionally, we explore similar predictors to differentiate between benign softare and cyrptographic library code. We find that the presence of the string "payment", the cryptographic constant AES powx, or at least three of the four strings "AES", "Crypt", "txt", and "TXT" are good predictors that a sample is ransomware. Additionally, we show that the number of loops and bitwise operations in a sample are not good predictors for detecting if a sample is cryptographic or ransomware.

The contributions of this work to the greater community are as follows:

- Provide insight into claims about cryptographic code in previous work.
- Provide several statistically significant predictors of ransomware that can be obtained via static analysis without executing any potential ransomware samples.

The remainder of this work is oranized as follows. Chapter 2 gives an overview of the background and related work on various topics presented in this work. Chapter 3 presents the methodology used to acheive the goals of this work. The results of all experiments are reported in Chapter 4. We conclude in Chapter 5 as well as present future research areas.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter conatins the necessary background and related work in order to understand the goals and implementation of this work. The information contained within is organized in a way to guide the reader from problem to solution. Section 2.1 gives an overview of ransomware, its orgin, and current analyses from literature. Section 2.2 gives enough of an overview of cryptography to enable the reader to understand certain details about ransomware and this work. Section 2.3 give a general overview of malware analysis and covers three important topics to this work: binary analysis in Section 2.3.1, control flow analysis in Section 2.3.2, and loop detection in Section 2.3.3. This chaper ends with a description of statistics used in Section 2.4, along with several important definitions.

2.1 Ransomware

Crypto ransomware is a form of *cryptoviral extortion*, which was first explored by Young and Yung in 1996 when they developed the first know cryptoviral attack [61]. Cryptoviral attacks are carried out by a *cryptovirus* [61].

Definition 2.1.1. A *cryptovirus* is a computer virus that uses a public key generated by the author to encrypt data D that resides on the host system, in such a way that D can only be recovered by the author of the virus (assuming no fresh backups exist).

Young and Yung define cryptoviral extortion [61] as:

Definition 2.1.2. A *cryptovirol extortion* attack is an attack where the virus writer is

able to for the victim to exchange information in return for the necessary information required to decrypt data D, and in addition provides a mechanism for verifying the authenticity of the data being extorted.

Since encryption via public key cryptography is less efficient that of symmetric cryptosystems, a *hybrid cryptosystem* is used. These cryptosystems use symmetric cryptography for bulk encryption of data and encrypt the symmetric key with the virus author's public key [61]. Further details about cryptography are covered in Section 2.2.

Young and Yung have pioneered the field of Crytpovirology, developing various cryptoviral attacks, including extortion [61, 68], espionage [62], and modified cryptographic protocols that securely leak sensitive data [63, 65, 66, 67]. In 2006, Young modified a cryptoviral extortion attack to use the Microsoft Cryptographic API to perform all cryptographic operations in 72 lines of C code [68].

Crytoviral attacks remained dormant until recently in which sophistication and diversity has given rise to many different families of ransomware. Several high profile ransomware families have been the focus of recent literature. Lemmou and Souidi provide both static and dynamic analysis of the GandCrab family of ransomware using a Windows 7 host inside of a VirtualBox virtutal machine [35]. The authors note the operations GandCrab performs leading up to encryption and reveal several interesting actions. First, it creates a mutex to prevent multiple copies of itself from running, then copies itself using a peculiar method that the ransomware authors believe (without evidence) will evade antivirus detection. The copy is run only on a reboot after the appropriate registry key has been created. It then enumerates running processes and closes them in order to ensure encryption of files does not fail due to a sharing violation. Next, GandCrab generates an RSA keypair for encryption.

GandCrab gathers system information that it encrypts with RC4 using the key `aeriedjD#shasj` before sending it to the C&C server. A very interesting note is that GandCrab will not encrypt files until it receives a response from the C&C server. Upon receiving a response from the C&C server, GandCrab will terminate if no '{' character occurs or if an unequal number of '{' and '}' occur. After encryption, GandCrab will delete shadow copies of files to prevent restoration of encrypted files. The authors conclude with several possible detection methods [35].

Cabaj et al. provide an analysis of network activity of the CryptoWall family ransomware, using a combination of honeypots and the Maltester malware analytical system [10]. Maltester utilizes Xen on a Debian system to create a virtual network that can prevent the infected host from communicating to the internet. After exchanging several messages with a C&C server, the authors block the infected host's access to the internet and instead emulate the C&C server. The authors also noticed that encryption of files will not occur until after CryptoWall has recieved 2048-bit RSA keypair from a C&C server.

The authors also discovered that several compromised WordPress websites were being used as a proxy for their C&C servers. After contacting the administrator of a compromised site, they were able to obtain PhP scripts that revealed that communications with the C&C server utilized RC4 encryption with a random key for each session. This enabled them to create their own proxy to intercept communications with the infected host. The authors also noticed that public key was delivered by the C&C server, no matter what version of CryptoWall was making requests [10].

A static [29] and dynamic [30] analysis of the WannaCry ransomware is given by Kao and Hsiao. Hsiao reported that WannaCry is modular and uses separate binaries for separate phases of the attack. WannaCry utilizes a *dropper* (`mssecsrv.exe`), which installs another service for persistence and downloads the main ransomware binary

`tasksche.exe`. `mssecsrv` is also capable of infection using the `EternalBlue` exploit. WannaCry contains the RSA public key of its author (k_a) and generates its own RSA-2048 keypair, (k_s, k_p). WannaCry promptly encrypts the generated private key, k_s, with the author's public key, k_a, and deletes k_s. Encryption is performed via AES using a randomly generated key for each file. Each AES key is encrypted with the public key k_p.

The second most interesting thing about WannaCry is that it searches for a specific domain before encryption and, if it resolves to a valid domain, it terminates without encrypting anything. This "kill switch" was first discovered by malware researcher Marcus Hutchins who promptly registered the domain and enabled the kill switch [1]. Another interesting note is that WannaCry allows up to ten files to be decrypted without payment as a demonstration of its decryption capabilities [29].

Craciun et al. provide an analysis of how ransomware has developed over several years, highlighting methods used for infection, as well as mistakes made by ransomware authors [17]. They note that 80% of ransomware used standard libraries for cryptography over a proprietary implementation. The authors also discover that ransomware typically uses one of five key management strategies: 1. Download a secret key/keystream, 2. create a random key, 3. use an embedded symmetric key, 4. use an embedded RSA public key to encrypt randomly generated symmetric keys, or use ECDH. They also note that ransomware authors tend to have poor knowledge of cryptography.

Savage et al. from Symantec give an overview and history of ransomware in lengthy a technical report [46]. The authors discuss both locker ransomware and crypto ransomware. They also give a brief history of ransomware and discuss shifts in trends of ransomware from misleading applications to crypto ransomware. Additionally, the authors also cover psychological aspects, such as social engineering, designed to coerce

victims into paying ransoms.

Gonzalez and Hayajneh discuss methods of infection and the technology behind ransomware as observed from several prominent families of ransomware, as well as provide recommendations to prevent infections [25]. The authors discuss methods of infection and ways ransomware may hide. They also discuss several families of ransomware, including Dirty decrypt, CryptoLocker, CryptoWall/CryptoDefense, Critroni/CTB Locker, and TorrentLocker. The authors note that CryptoLocker and CryptoWall both retrieve public keys from a command and control server before encryption and use Microsoft's Crypto API.

Additionally, Gonzalez and Havaineh outline the typical scenario of a ransomware infection and the different types of behavior observed. The authors note that typical applications will read from many files but write to few and that the deletion of many files may be an indicator of ransomware infection [25]. They suggest monitoring API calls and filesystem activity to detect suspicious behavior and using decoy files as mitigation techniques. The authors conclude with recommendations for protection against infection, which are the same general recommendations that have been promoted for the last twenty years.

O'Kane et al. give an overview of the evolution of cybercrime from early-day scams to sophisticated ransomware attack campaigns [40]. The authors first discuss attack vectors, noting that social engineering and exploit kits (malicious applications that scan for vulnerable software on users' machines) are a very prominent way to infect victims. The authors then give an overview of several families of ransomware and their evolution over time. They explain that CTB-Locker and CryptoWall have similarities, but are believed to have been created by different authors. Additionally, the authors mention that CryptoLocker, CryptoWall, CTB-Locker, and TeslaCrypt all use Tor to obfuscate their network traffic and use RSA public keys to encrypt AES symmetric

keys. Several more recent families of ransomware including CryptXXX, Locky, Samas, Jigsaw, and Cerber were noted to follow similar trends.

O'Kane et al. also note that early ransomware suffered from poor cryptographic implementations that allowed for decryption without payment, but mentioned several families improved to become more sophisticated over time. Next, the authors discuss different payment methods used by ransomware and show a trend where over time ransomware authors shift to preferring Bitcoin over other forms of payment. They conclude with strategies for data recovery in the event of a ransomware infection.

Bajpai, Sood, and Enbody provide a ransomware taxonomy based upon an analysis of key management of 25 samples of ransomware [6]. The authors describe the shift from less secure key management schemes to a hybrid scheme that utilizes AES with a randomly generated key for encryption and RSA or ECDH to encrypt the AES key. They also define several key management strategies that range from: 1. None, 2. decryption essentials stored in the user domain (i.e. (a) a decryption key stored somewhere on the host or (b) distributed among peers), 3. decryption essentials stored in the attacker domain (i.e. (a) a single decryption key stored on a command and control (C&C) server or (b) a C&C-hybrid system). The first two strategies have relatively simple methods for decryption without payment, but the third method would prove impossible as long as the cryptographic implementation was correct.

CryptoLocker, CryptoDefense, and Cerber were observed to follow strategy 3(a), where the ransomware would need to obtain an asymmetric key pair from a C&C server before encryption could occur. CryptoDefense was observed to not securely delete decryption keys, but Cerber was [6]. WannaCry was observed to utilize strategy 3(b) in which it would generate its own asymmetric key pair (k_s, k_p), encrypt the private key, k_s, with the ransomware authors public key, and then delete k_s. It would then generate an AES key from a Cryptographically Secure Pseudorandom Generator

(CSRNG) for each file being encrypted, After each file encryption. WannaCry would encrypt the corresponding AES key with its own public key k_p. The only way to recover the AES keys is for the ransomware author to decrypt k_s (after payment, of course).

Genç et al. discuss techniques used by ransomware to derive encryption keys and analyze the security of each [22]. The authors noted that ransomware typically used one of four key management strategies: 1. Derive keys from a CSPRNG, 2. fetch keys from a C&C server, 3. generate keys from a non-cryptographic RNG, or 4. embed secrets into the binary. The authors also suggest several mitigations such as: hooking into cryptographic API calls to escrow keys or store copies of CSPRNG output, backdoor CSPRNGs, and whitelisting application access to CSPRNGs.

Andronio, Zanero, and Maggi describe common characteristics of mobile ransomware families. They also developed *HelDroid*, a framework to detect Android ransomware. HelDroid differs from signature based detection methods by detecting whether an application attempts to lock a device, encrypt data, or both with negligible false positive rate [5].

Continella et al. developed *ShieldFS*, a filesystem that monitors low-level activity to create a set of adaptive models of user behavior. ShieldFS dynamically toggles a copy-on-write protection mechanism if it detects a process violating any model. This gives the operating system a self-healing capability to respond to the threat faster than a pure-detection based method. The authors discovered that ShieldFS is best suited for protecting short-term file changes, while traditional backups are better suited for long-term file changes [14].

Ahmadian, Shahriari, and Ghaffarian provide a taxonomy of ransomware and develop the *Connection -Monitor & Connection-Breaker* framework for detecting high surviv-

able ransomware. This framework relies on detecting the moment when ransomware attempts to perform a key exchange with a command and control server and blocking the exchange, which prevents the ransomware from encrypting files on the host computer [3].

Ahmadian and Shahriari develop the 2entFOX framework for detecting high survivable malware that uses a combination of static and dynamic analysis to extract behavioral information using a Bayesian relief network [2]. 2entFOX can reliably detect high survivable malware, but does not reliably detect low survivable malware.

In summary, early ransomware was developed with poor choices in cryptographic algorithms, insecure implementations of cryptographic algorithms, poor key management techniques, and a wide variety of payment methods. Recent literature has shown that modern variants of ransomware have evolved to use more secure cryptographic algorithms, often using secure implementations from cryptographic libraries instead of custom implementations, and utilize more sophisticated key management techniques. Ransomware authors have also shown a tend to prefer cryptocurrency, such as Bitcoin, over other forms of payment.

2.2 Cryptography

The goal of cryptography is to allow two entities, commonly referred to as Alice and Bob, to communicate securely over an insecure channel such that any adversary cannot recover the information being conveyed. This hidden information is commonly referred to as *plaintext*, and can take any arbitrary form, not just human language. The form that hidden plaintext takes on is referred to as *ciphertext* [31, 36, 52]. The algorithms that allow transformations between plaintext and ciphertext are known as *cryptosystems*[36, 52].

Definition 2.2.1. A cryptosystem consists of the following:

1. A finite set of all possible plaintexts $\mathcal{P}$.
2. A finite set of all possible ciphertexts $\mathcal{C}$.
3. A finite set of all possible encryption keys, $\mathcal{K}$, known as the *encryption keyspace*.
4. A finite set of all possible decryption keys, $\mathcal{K}'$, known as the *decryption keyspace*.
5. An efficient key generation algorithm $\mathcal{G} : \mathbb{N} \to \mathcal{K} \times \mathcal{K}'$.
6. An efficient encryption algorithm $\mathcal{E} : \mathcal{P} \times \mathcal{K} \to \mathcal{C}$.
7. An efficient decryption algorithm $\mathcal{D} : \mathcal{C} \times \mathcal{K}' \to \mathcal{P}$.

This section is intended to give the reader adequate background on cryptographic topics covered in this work, including AES, RSA, and ECDH, and is by no means a complete cryptography reference. For additional details not covered in this work, refer to Katz and Lindell [31], Mao [36], Stinson [52], Young and Yung [64], or Ferguson, Schneier, and Kohno [21].

2.2.1 Symmetric Cryptography

In symmetric cryptosystems, encryption and decryption utilize the same key [36, 31, 52], thus $\mathcal{K} = \mathcal{K}'$.

AES

AES is defined in FIPS 197 [44], which specifies the Rijndael symmetric block cipher [36] for processing block sizes of 128 bits using keys of lengths 128-, 192-, or

256-bits in length [44]. AES stores an 128-bit message internally as a 4×4 byte matrix [36]. AES is based on the Rijndael cipher, which performs a multiple number of iterations of a basic unit of transformation, referred to as a *round* [36]. Typically, each round consists of the following four transformations [36]:

SubBytes Provides a nonlinear substitution on each byte of state.

ShiftRows Provides a transposition on each row i of state by cyclic shifting of each byte by $4 - i$.

MixColumns Provides a transformation on each column of state similar to a polyalphabetic substitution using a fixed key.

AddRoundKey Adds a round key to bytes of state. Each round key is unique and is derived from the encryption key using a key schedule scheme.

Each of these operations are invertible [36], which allows for decryption.

The important thing to note is that the SubBytes operation is often implemented using a lookup table known as an *s-box*[36, 52]. This increases performance and prevents timing analysis attacks [36]. S-boxes can be stored in applications, which means it will then be present in the resulting binary. In this work, binaries are analyzed for the presence of several possible constants as discussed in Section 3.5 withe results reported in Section 4.2.

2.2.2 Asymmetric Cryptography

In asymmetric cryptosystems, $\mathcal{K} \neq \mathcal{K}'$. This removes the requirement for encryption keys to be kept secret; only the decryption key must be kept secret [36]. This is made possible with the use of *one-way trapdoor functions* [36, 31, 52]:

Definition 2.2.2. A one-way trapdoor function $f_t(x) : \mathcal{D} \to \mathcal{R}$ is a function in which it is easy to evaluate $f_t(x)$ for all $x \in \mathcal{D}$, but is difficult to invert for almost all $y \in \mathcal{R}$, unless trapdoor information t is used.

The asymmetry in encryption and decryption is what gives ransomware authors the leverage for extortion [64].

RSA

RSA is a cryptosystem developed by Ronald Rivest, Adi Shamir, and Leonard Adleman and first published in 1978 [21] and utilizes the following algorithms [36]:

Definition 2.2.3. RSA:

Key Setup:

1. Compute $n = pq$ where p and q are two large primes of relatively same size.
2. Compute $\phi(n) = (p-1)(q-1)$.
3. Choose a random $e < \phi(n)$ such that $\gcd(e, \phi(n)) = 1$ and compute d such that

$$ed \equiv 1 \mod \phi(n)$$

4. Ouput (n, e) as public key and d as the private key.

Encryption:

- Compute ciphertext message c from plaintext message m as $m^e \mod n$.

Decryption:

- Compute m from c as $c^e \mod n$.

The notation $x \mod n$ refers to the remainder of integer division of x by n. The security in RSA lies in the fact that factoring large integers is difficult [36, 52]. The encryption algorithm of RSA is considered a one-way trapdoor function since decryption requires secret knowledge of the factorization of n in order to compute the decryption exponent d [52].

Contrary to Bajpai et al.'s claim that there is no difference in security between RSA and ECDH [6], the RSA cryptosystem is fragile [39]. The description of RSA presented here is dangerous to use in practice [21] and instead one should rely on more robust public key cryptosystems, such as those utilizing Elliptic Curve Cryptography (ECC).

ECDH

ECDH is a version of the Diffie-Hellman key exchange protocol (DH) that obtains security through the difficulty of the computing logarithms for elliptic curve groups instead of integer groups [31, 52]. A *group* is a mathematical structure with certain algebraic properties with a formal definition that is outside of the scope of this work. The interested reader is referred to Mao [36], Stinson [52], or Katz et al. [31] for a more rigorous explanation.

Even without a formal definition of groups, the problem of computing logarithms, known as the *discrete log problem*, can be illustrated with an example. Let a, b, and i be integers such that $b = a^i$. If i is large enough, 4096 bits for example, it is infeasible to determine i from a and b [52].

The original Diffie-Hellman protocol [36] can be described (in a simplified manner) between two parties, Alice and Bob, as

Definition 2.2.4. Diffie-Hellman:

1. Both parties agree on a large prime p and an integer g where $0 < g < p$.
2. Alice picks integer a such that $0 < a < p - 1$, computes $g_a = g^a \mod p$, and sends g_a to Bob.
3. Bob picks integer b such that $0 < b < p - 1$, computes $g_b = g^b \mod p$, and sends g_b to Alice.
4. Alice computes $k = g_b^a \mod p$.
5. Bob computes $k = g_a^b \mod p$.

At the end of this protocol, both Alice and Bob will have a shared secret $k = g^{ab} \mod p$, which they can use as a key, or use to derive a key, for encryption.

Elliptic curves are described by the set of solutions to certain equations in two variables [52]. The interesting thing about groups consisting of points on on elliptic curve is that there is no known sub-exponential time algorithm for solving the discrete log problem on them (when chosen properly) [31]. This makes algorithms based on the discrete log problem more efficient when using elliptic curve groups [31].

ECDH is similar to DH described above. Alice and Bob must first agree on several domain parameters, including a reference point G on their chosen elliptic curve. They then generate points $Q_a = aG$ and $Q_b = bG$ as their respective public keys and send those to each other. Afterwards, they can each calculate the same point on their elliptic curve $aQ_b = (x_k, y_k) = bQ_a$, which they can use as their shared secret. Since (EC)DH is a key exchange protocol, it is not used for encryption directly. Instead, the shared secret is typically used to derive a symmetric key for encryption.

2.3 Malware Analysis

The goal of malware analysis is typically to respond to a network intrusion [49]. There are two fundamental techniques for malware analysis: *static analysis* and *dynamic analysis*.

Static analysis involved examining the binary (or binaries) of suspected malware. This may involve reverse engineering the binary to discover the instructions it executes in order to determine exactly what it does [49]. Dynamic analysis involves running suspected malware in a safe environment to discover its behavior. In many cases, this involves using a debugger on the executable to recover detailed information that is difficult to acquire with other methods [49].

Static analysis has good code coverage and is scalable, but has poor precision due to indirect branch resolution. Dynamic analysis, on the other hand, suffers from poor code coverage and scalability [60]. Additionally, different types of analyses have different trade-offs. For example, dynamic analysis of some program may be able to trace code execution but not know *why* it was executed, while a static analysis using symbolic execution may be able to determine specific bytes of input that trigger execution, but not understand the higher semantic meaning of those bytes [47].

A general, formal definition of malware is given by Kramer and Bradfield. They note that malware has been recognized as difficult to define and that previous attempts to define malware are not adequate. They use the language of a modal fixpoint logic to define malware in the single sentence "A software system s is malware by definition if and only if s damages non-damaging software systems (the civil population to say) or software systems that damage malware" [33].

De Carli et al. note that ALIGOT, the state of the art for encryption fingerprint-

ing in dynamic execution traces of obfuscated malware, is not scalable and present an enhanced algorithm based on ALIGOT named *KALI*. The authors noted that KALI had orders of magnitude reduction in execution time and memory utilization compared to ALIGOT for large traces [12].

2.3.1 Binary Analysis

Static analysis techniques may be performed on source code or compiled binaries. In order to analyze binaries, their bytes must be disassembled into assembly [28, 4]. Unlike source code, binaries are not centered around functions and variables that utilize human readable names. These human readable names may be stored in a *symbol table* inside a binary, but are often stripped from a binary to reduce file size or make reverse engineering more difficult [4, 28], and, consequently, making binary analysis more challenging.

Binaries contain more than the instructions the source code they were compiled from produce. Binaries typically store both code and data in different regions of the binary, called *sections*, but this separation is not required. Mixing of code in data presents an additional challenge that may lead to incorrect disassembly [27].

Practical Binary Analysis by Andriesse gives a good overview of what binary analysis is, its challenges, and walkthroughs for creating binary analysis tools [4]. This is a great resource for those who which to learn more about binary analysis and was especially helpful in helping me create my own tools.

Bilar discusses polymorphic and metamorphic malware and how they can evade structural fingerprints [8]. Bilar used statistics on binaries' opcode from both malware and benign software called "goodware" in order detect opcode predictors of malware. Bilar noted that malware samples contained rarely used opcodes which caused

the frequency distribution for opcodes in malware to be significantly different from goodware.

Gröbert et al. present several automated methods for identifying cryptographic primitives in binaries using fine-grained dynamic binary instrumentation. The authors noted that their technique improved state-of-the-art approaches in analysis and they were able to successfully extract cryptographic keys from binaries [26]. The authors also noted that encryption routines use a high number of bitwise arithmetic instructions, loops are core components of cryptographic algorithms, and the decryption process decreases information entropy of tainted memory. The first two of these three observations are explored further in this work.

Shoshitaishvili, et al. discuss the challenges of binary analysis, but note the importance of binary analysis to prove or disprove properties of the code that is actually executed [47]. The authors explain that many techniques developed to find flaws in binary programs suffer from one of two problems: many techniques begin and end with a research prototype and that the work required to reproduce these systems makes replication of result impractical. The authors also note that binary analysis must balance *replayability*, which is necessary to reduce false positives in analysis, and *semantic insight*, the ability to reason about programs in semantically meaningful ways [47]. Analyses that attempt to achieve both suffer from poor scalability.

The authors attempt to mitigate the issue of short-lived binary analysis research prototypes by creating the open source binary analysis framework *angr* [48]. angr provides analysis engines for both static and dynamic methods, while being architecture and platform independent and supporting multiple analysis paradigms [47]. angr also provide accurate control flow graph (CFG) recovery, using a combination of forced execution, backwards slicing, symbolic execution, and value set analysis. Forced Execution is a technique that ensures both paths of a conditional branch

are executed [47]. Value-Set Analysis (VSA) is a technique that combines numerical analysis and pointer analysis for binary programs[47].

Brumley, et al. develop *BAP*, a publicly available binary analysis platform [9]. BAP provides a front end for lifting binaries into an intermediate language (IL), which is then processed by the BAP's back end for analysis. Unlike previous binary analysis frameworks, BAP makes instruction side effect explicit in the IL. BAP is also capable of creating verification conditions (VCs) that can test whether certain properties hold under specific inputs during program execution. BAP performs linear disassembly when lifting binaries and expects users to interact with the backend to direct control flow [9].

Di Federico et al. develop *REV.NG*, a unified binary analysis framework [20]. REV.NG utilizes QEMU as its front end and the LLVM intermediate representation (IR) to enable analysis agnostic to the 17 architectures supported by QEMU. The authors note several challenges for CFG recovery, including that it is "impossible to enumerate the exact set of possible jump targets for indirect control transfers" [20]. Additionally, several challenges exist in recovering function boundaries, call thunks (calls to the next instruction for the purpose of recovering the value of the program counter), noreturn functions (such as `exit` in C), shared code, calls to the middle of a function, and tail calls (calls implemented as unconditional jumps). REV.NG performs an iterative CFG recovery, exploring new jump targets as they become available, until no unresolved jump targets exist. After the CFG has been recovered, it is analyzed for function boundary recovery using a five step process.

2.3.2 Control Flow Analysis

Although processors in computers appear to execute instructions sequentially, several instructions exist to cause the processor to *branch* from the current sequence of instruction to a difference sequence. Several variations of branch instructions that cause a branch to occur on a variety of conditions exist and are aptly name *conditional branches*. Conditional branches are why control flow analysis is essential to binary analysis.

The relationship between transfers of control flow are captured in a *control flow graph*. Xu et al. [60] describe a control flow graph as

Definition 2.3.1. A *control flow graph (CFG)* is a data structure representing all the control flow paths of a program that might be traversed during execution.

The vertices in a CFG are *basic blocks* and the edges are the possible transfers of control flow from one basic block to another. A basic block is a sequence of instructions that are entered at their first instruction and executed without control flow entering or exiting except at the last instruction [58, 60].

Harris and Miller give a practical analysis of stripped binaries [28]. They describe a model that allows for function and CFG detection without the use of symbols. Their process utilizes two steps iteratively: 1. Look for function prologues and 2. build a CFG and check for conflicts. Harris and Miller also identify indirection jumps via jump tables by backtracking along the control path leading to the jump and identifying the instructions setting up jump table access, which reveals the base address to the jump table as well as the number of entries in the table [28].

Hanov describes the challenges of applying static analysis techniques to binary programs, especially in CFG recovery [27]. In addition to CFG recovery, analysis typically

involves searching for groups of instructions that, called *idioms*, have no individual meaning, but when taken as a whole represent larger operations. Hanov notes one technique used to recover targets of indirect jumps called *program slicing*. Program slicing eliminates statement that do not involve a problem being analyzed in order to reduce complexity of analysis [27]. Utilizing program slicing yield more accurate CFGs, which can be used to derive information about higher level constructs, such as procedures.

Kinder and Veith describe *Jakstab*, a tool designed to enable static analysis and model checking on binaries [32]. They noted that compiler optimizations and obfuscation mangle control flow structure of a program. To overcome this, Jakstab iteratively builds a CFG and performs data flow analysis to update the CFG each iteration. Jakstab performs live variable analysis, dead code removal, and propagates and folds constants and memory cells through registers to resolve indirect branches. Newly discovered indirect branches are then disassembled during the next iteration [32].

Theiling also describes an iterative method to create CFGs from binaries [58]. Using a bottom up approach to avoid confusing code and data, Theiling's method approximates a CFG, performs constant propagation, then updates the CFG with new information from constant propagation. Theiling also describes *Inter-Procedural Control Flow Graphs* (ICFGs), which consist of a call graph and CFG for each routine, which is used to approximate a CFG for the program.

Xu, et al. develop a method of recovering a CFG from binaries using forced execution to explore both paths of a conditional branch, saving program state before exploration to aid in indirect branch resolution, and using backwards reachability analysis to avoid re-execution [60]. Their algorithm creates a partial CFG and performs analysis on it to resolve additional targets of indirect jumps and updates the CFG. Using this iterative process, the authors show that their algorithm produces a nearly ideal CFG.

2.3.3 Loop Detection

Loop detection may assist in certain types of analyses. Loop identification is essential to this work because it attempts to confirm findings by Gröbert et al. that loops are core components of cryptographic code [26]. A loop is defined [50] as follows:

Definition 2.3.2. A *loop* is a strongly connected subgraph of a flow graph.

Unfortunately, this definition introduces two more terms, which we must now define.

Definition 2.3.3. A *flowgraph* is a connected, directed graph $G = (V, E, Start, End)$.

In this definition, $Start$ is the entry point into the flowgraph and End is the terminal vertex in which paths from all other vertices will flow. A graph is *connected* if there exists a path from $Start$ to each $v \in V$. Tarjan provides a good definition of what it means for a graph to be *strongly connected* [53].

Definition 2.3.4. A graph $G = (V, E)$ is *strongly connected* if for each pair vertices $v, u \in V$, there exists paths from v to u and u to v.

This means that every vertex in G has a path that flows to every other vertex.

Xie describes the importance of loops analysis in significantly improving program analysis [59]. Xie also outlines several challenges, such as variable updates from multiple paths, the undecidability of analyzing variables that do not have a constant change each loop iteration, and nested loops. Xie then purposes a technique labeled path dependency automation (PDA) to model loop execution.

In this work, we desire a count of the number of loops in a binary sample, thus may use a simpler analysis than that purposed by Xie. Sreedhar et al. describe a method for detecting loops a special type of graph called a *DJ Graph*. DJ Graphs have a structure

that aides in loop identification [50]. Since this is the method employed in this work for detecting loops, we will first introduce concepts related to the construction of DJ Graphs in order to reveal the structure of DJ Graphs.

Definition 2.3.5. A vertex v *dominates* vertex u if and only if all paths from the start of the flow graph to u pass through v, and is denoted $v\,dom\,u$. If $v\,dom\,u$, then v is referred to as a *dominator* of u.

It is important to note that every vertex v in a flowgraph dominates itself.

Definition 2.3.6. A vertex v *strictly dominates* vertex u if and only if $v\,dom\,u$ and $v \neq u$. Strict domination is denoted $v\,stdom\,u$. If $v\,stdom\,u$, v is said to *strictly dominate* u.

Definition 2.3.7. A vertex v *immediately dominates* vertex u if $v\,stdom\,u$ and no $w \in V$ exists such that $v\,stdom\,w\,stdom\,u$. Immediate domination is denoted $v = idom(u)$.

Georgiadis et al. describe several methods for computing dominators [23]. An iterative algorithm [23] can be used to find the sets of dominators for each vertex v, $Dom(v)$, which are each the maximal solution to

$$Dom'(v) = \left(\bigcap_{u \in pred(v)} Dom'(u) \right) \cup \{v\},\ \forall v \in V \tag{2.1}$$

where $pred(v)$ is the set of all predecessors of vertex v in a flowgraph G. This algorithm is illustrated in Algorithm 1 and its implementation is described in Section 3.3.2.

Once dominators are identified, we can capture the dominance relation between all vertices in a flowgraph using a *dominator tree*. Cooper et al. define a dominator tree [16] as:

```
1  Dom'(r) ← {r};
2  foreach vertex v ∈ V where v ≠ r do Dom'(v) ← V;
3  repeat
4      updated ← false;
5      foreach vertex v ∈ V do
6          if Dom'(v) ≠ ∩_{u∈pred(v)} (Dom'(u)) ∪ {v} then
7              Dom'(v) ← ∩_{u∈pred(v)} (Dom'(u)) ∪ {v};
8              updated ← true;
9          end
10     end
11 until updated = false;
```

Algorithm 1: Iterative algorithm for discovering dominators.

Definition 2.3.8. A *dominator tree* is a tree with the same vertices as flowgraph G and edges exist only between a vertex and its immediate dominator.

With the knowledge of dominance, the formal definition of a DJ Graph becomes more clear.

Definition 2.3.9. A *DJ Graph* $H = (V, \overline{E} = D \cup J)$ is a graph where V is the same as in a flowgraph G and $\overline{E}$ is the union of the sets D of edges in the dominator tree and J of edges in $E - D$, where E is from flowgraph G.

Two different varieties of J edges exist and their distinction is important for some loop detection algorithms [50]. They are defined as follows:

Definition 2.3.10. A *Back J (BJ) edge* is an edge $(v, u) \in J$ where $u \, dom \, v$. A *Cross J (CJ) edge* $(v, u) \in J$ is one in which u does not dominate v.

Sreedhar et al. describe the following two-step method for constructing a DJ Graph [50]:

1. Build a dominator tree
2. Insert each $(v, u) \in J$ into the dominator tree

DJ Graphs capture the progression of flow from *Start* to *End* through D edges, as well as capture the presence of loops with J edges. Sreedhar et al. note that previous work was only able to detect *reducible* loops.

Definition 2.3.11. A flowgraph is *reducible* if and only if the set of edges can be partitioned into two disjoint sets of forward and backwards edges.

Using DJ Graphs, Sreedhar, et al. describe a method for identifying both reducible and irreducible loops [50]. The implementation of their method in this work is described in Section 3.3.3.

2.4 Statistics

Several statistical methods are used in this work.

Hypothesis Testing

A test of hypotheses is a method for using sample data to choose one of two competing hypotheses [19]. The *null hypothesis*, H_0, is the assertion initially assumed to be true. The *alternative hypothesis*, H_a, is the contradictory assertion. The null hypothesis will be rejected in favor of the alternative if and only if sample evidence strongly contradicts the null hypothesis. Otherwise, we fail to reject the null hypothesis and trust its truth [19].

Hypothesis tests are conducted using a *test statistic*, which is a function computed on the data to determine whether or not to reject the null hypothesis [19]. The *p-value* is

the probability, assuming that the null hypothesis is true, of obtaining a test statistic at least as contradictory as the value actually resulted. The smaller the p-value, the more contradictory the data is, and the null hypothesis should be rejected if the p-value is sufficiently small [19].

Multiple Logistic Regression

Logistic regression is a statistical technique used when the outcome of a problem is binary [34]. The probability that the outcome is 1 is known as the *probability of success* $\pi(x)$ and is defined by the *logit* function

$$\pi(x) = \frac{e^{b_0+b_1x}}{1+e^{b_0+b_1x}} \tag{2.2}$$

where x is referred to as a *predictor*. *Multiple logistic regression* is an extension of Equations 2.2 where multiple predictors exist and can be written as

$$\pi(x_1, x_2, \ldots, x_n) = \frac{e^{b_0+b_1x_1+b_2x_2+\cdots+b_nx_n}}{1+e^{b_0+b_1x_1+b_2x_2+\cdots+b_nx_n}}. \tag{2.3}$$

for n predictor variables. The *estimated odds ratio* (EOR) is a measure of association between a predictor and the outcome of a logistic regression model. An EOR greater than one indicates a higher odds of a successful outcome, while an EOR less than one indicates lower odds of a successful outcome [19, 34]. EOR is calculated as the exponential function of the regression coefficient, e^{b_i}.

Akaike's information criterion (AIC) is a good choice for comparing models to each other. It measures error in the model as well as penalizes for adding additional

predictors [34]. AIC is given by

$$AIC = n \ln SSE - n \ln n + 2p$$

where n is the sample size in an experiment, p is the number of predictors in a model and SSE is the error sum of squares, which is the sum of the squares of the difference of each predicted and actual values [19]. The model with the lowest AIC is often the best model. Including the number of predictors in the calculation of the criterion prevents overfitting of regression models with predictors that can occur in other model selection techniques, such as coefficient of determination (R^2).

Chapter 3

METHODOLOGY

Data was collected using a custom disassembly tool, VBD [41], created specifically for this project and described in Section 3.1. Samples where analyzed on a PC with an eight core AMD FX-8150 processor and 16 GB of memory running Debian 9.9 as the operating system with Linux kernel `4.19.0--0.bpo.4-amd`. ELF binaries and ransomware samples were analyzed in the host operating system Debian and PE binaries were collected from `Windows` 10 running in `VirtualBox` 5.2.24_Debian r128122 with 8 GB of memory and two processors provisioned. In addition to disassembly, VBD also collects statistics on binaries, detects loops, and specific strings and cryptographic constants, which will be covered in the remaining sections of this chaper. Results from data collection were stored in an instance of MongoDB with version 3.2.11. Ransomware samples were obtained from Virushare [45] and Fabrizio Monaco's malware sample Github repository [37].

3.1 Disassembly

VBD, was written in `Rust` and compiled with version `1.37.0-nightly` of `rustc`. VBD uses the Binary File Descriptor (BFD) library [24] `libbfd` to load both ELF and PE binaries. Disassembly is accomplished using Capstone (`libcapstone`) [43]. VBD utilizes the host operating system's versions of both `libbfd` and `libcapstone` via Rust's Foreign Function Interface (FFI) [55] with Rust bindings gererated with `bindgen` [54].

In addition to performing tasks specific to the goal of this work, VBD provices sev-

eral options to perform specific disassembly operations similar to a subset of options offered by `objdump`. VBD can display all symbols, sections, a single user specified section, and dump section contents in a hexadecimal format similar to `xxd`. VBD can also perform and output individual tasks of analysis, such as disassembly, loop detection, and statistics collection, when the corresponding option is given at the command line.

VBD can also read an write to a MongoDB instance and has options to override the default server, port, database and collection. VBD only utilizes a database after analysis to store results or when exporting results to a CSV file. The `--analyze` option analyzes a given binary, or or multiple binaries in a given directory, by performing each of the following tasks: gather statistics, loop detection, a string search, and search for cryptographic constants. Each of these tasks are discussed individually in Section 3.2, Section 3.3, Section 3.4, and Section 3.5 respectively.

With the `--disassemble` option (or `--analyze` option which automatically performs disassembly), VBD performs linear disassembly to look for function prototypes and then performs recurisve disassembly [4] on addresses discovered during linear disassembly, as well as targets of direct jumps discovered during recursive disassembly. Currently, VBD is unable to resolve indirect jumps. During the disassembly process, VBD groups instructions into basic blocks, splitting them as needed if an address inside the basic block is later discovered to be the target of a jump. VBD also tracks cross references to basic blocks which are displayed with the output of disassembly when the `--disassemble` option is used. Currently, VBD only supports the 32- and 64-bit versions of the `x86` ISA.

Table 3.1: Bitwise instructions.

• and	• not	• rol	• vpsrld
• andn	• or	• ror	• vpsrldq
• andps	• orps	• rorx	• vpxor
• andnps	• orpd	• shl	• xor
• andnps	• pandn	• shld	• xorpd
• andpd	• por	• shr	• xorps
• andnpd	• pxor	• vpor	

3.2 Binary Statistics

Several attributes were stored in each database document. The name of the binary served as the ID of the document. All instruction opcodes discovered via disassembly and their number of occurances are recorded. From these, the quantity of bitwise instructions was calculated by summing the number of occurances of each instruction listed in Table 3.1. Additionally, the binary type (ELF/PE), architecture (x86), bits (32/64), address of the entry point into the binary, sample type (benign/cryptographic/ransomware), elapsed time (with nanosecond precision), and number of vertices and edges of the corresponding CFG were also stored.

3.3 Loop Detection

Loop detection is performed using the method developed by Sreedhar, et al. using DJ graphs [50]. Recall that a DJ graph is created by first creating a dominator tree from a CFG, then adding J edges to the dominator tree. Loop detection thus requires the following steps in order:

1. Create a CFG,
2. Create a dominator tree,
3. Create a DJ graph, and
4. Use the algorithm outlined by Sreedhar, et al. [50] to discover loops.

3.3.1 Control Flow Graph

A CFG for a binary is created after it has been disassembled. Vertices consist of basic blocks and edges denote the possible blocks control flow may transfer to, and, thus, are directed. The edge set is constructed iteratively; first by adding edges for each basic block depending on all possible targets it could transfer control flow to within the `.text` section of the binary, then replacing edges derived from function calls to a target address outside of the `.text` section of the binary with a fall-through edge to the next basic block. Due to the fact that some targets of calls may not be resolved, a fall-through edge is added from the basic block containing the call to the next basic block immediately following it. Basic blocks that contain return instructions are considered terminating vertices and will not be the origin of any edge.

3.3.2 Dominator Tree

The first step in creating a dominator tree is to discover the dominators $Dom(v)$ for each vertex $v \in V$. Several algorithms exist for discovering dominators of a flowgraph. In this work, the iterative algorithm [23] was chosen due to its simplicity and is illustrated in Algorithm 1 in Section 2.3.3. Recall that the sets $Dom(v)$ are the maximal solution to

$$Dom'(v) = \left(\bigcap_{u \in pred(v)} Dom'(u) \right) \cup \{v\},\ \forall v \in V$$

where $pred(v)$ is the set of all predecessors of vertex v in the CFG.

By definition, each vertex is its own dominator [50] and since the root r of the CFG has no predecessors, we assign $Dom'(r) = \{r\}$. Next, we assign $Dom'(v) = V$ for all $v \in V$. We then iterate through each $v \in V$ and evaluate $Dom'(v)$ to the above equation. If it is false, we replace $Dom'(v)$ with the expression on the right of the above equation. We repeatedly iterate through each $v \in V$ updating $Dom'(v)$ until no change is made to $Dom'(v)$ for every $v \in V$.

A dominator tree is constructed using the vertices of the CFG with edges that exist only between each vertex v and its immediate dominator $idom(v)$ [16]. Edges are added to the dominator tree by iterating over each $v \in V$ and adding edge $(idom(v), v)$ to the tree. The immediate dominator $idom(v)$ is calculated by searching for a $u \in V$ such that $u \neq v$ and $Dom(u) = Dom(v) - \{v\}$.

3.3.3 DJ Graph

Once a dominator tree is constructed, a DJ graph can be easily constructed from it and the CFG. D edges are those found in the dominator tree, which are added to the

DJ graph. J edges are those that exist in the CFG, but not in the dominator tree [50]. Recall that there are two types of J edges, BJ and CJ, which can be identified by iterating over each $(u, v) \in J$. If v *dom* u, then it (u, v) is added to the DJ graph as a BJ edge. If not, it is added to the DJ graph as a CJ edge. It is important to note that DJ graphs maintain level information from the dominator tree. This is important for loop identification, as will be shown later in this section.

Once the DJ graph has been constructed, loop identification can begin. VBD implements the algorithm purposed by Sreedhar et al. [50]. First a DFS is performed on the DJ graph to create a spanning tree. The implementation of spanning trees in VBD includes sp-back edges, which are essential for identifying irreducible loops [50]. For each level in the spanning tree, starting from the lowest level and working toward the root, each vertex on the current level has all incoming edges examined.

Loops can be identified by the type of incoming edge into a vertex. A BJ edge identifies a reducible loop. Using an implementation of `reach_under` described by Sreedhar et al. [50], all vertices in the loop are identified and collapsed, removing all vertices except the head of the loop (the vertex being examined), removing all edges between vertices in the body of the loop, and replacing edges that connect vertices in the body of the loop to an outside vertex u with any edge that connects the head of the loop to u. A CJ edge identifies an irreducible loop if it is also an sp-back edge in the spanning tree. When encountered, a boolean flag is set to true so that the irreducible loop can be processed after all reducible loops on the current level are identified and collapsed. Irreducible loop bodies are identified using Tarjan's Strongly Connect Components algorithm [50, 53, 38] and then collapsed in the same manner as reducible loops.

Loop bodies are copied into a Rust `struct`, which encapsulates all pertinent information about loops, and stored in a vector that is returned after the after the algorithm

completes. It is important to note that this algorithm only works on a connected CFG. If a CFG is disconnected, this algorithm may be used on its connected components and the resulting loops may be combined into a single set. However, there will likely be a loss of information that could lead to incomplete loop detection.

3.4 Common Strings

Common strings in ransomware notes were identified from ransom notes from the 33 varieties of ransomware listed in Table 3.2. While ransom notes are readily available [15], they are typically only available as screenshots and not text that can be parsed, requiring a human to transcribe them. After transcription, word frequency was conducted using a Python script. Captialization was perserverd, but all non-alphanumeric characters were removed in order to increase word frequency. This was done in order to count words such as “encrypted”, “encrypted.” and “encrypted!!!” was the same word. A subset of the most frequent words that were not words common in all english text (“the”, “a”, etc.) were selected for search parameters.

The Rust library `regex` [56] was used to search for the selected strings in each binary. `regex` provides the ability to search for multiple regular expression in one pass using an optimized version of the Aho-Corasick algorithm [56]. Each string was converted into a literal regular expression stored in a static array. To increase performance, the regular expression was compiled once and stored in a static variable for use among multiple binaries. Upon completion, the algorithm returns a vector of indices into the array of strings for identification.

Table 3.2: Various ransomware in which ransom messages were transcribed.

- 7ev3n
- ALFA Ransomware
- Alma Locker
- Anubis
- ASN1
- BadBlock
- BadEncript
- BadRabbit
- Bandarchor2
- BandarChor
- BlackShades
- Bucbi
- BuyUnlockCode
- Cerber2
- Cerber
- Chimera
- CHIP
- CoinVault
- CryptFile2
- CryptoLocker
- CryptoWall
- CryptXXX
- GoldenEye
- Jigsaw
- LockerGoga
- Locky
- MIRCOP
- NotPetya
- Petya
- Ryuk
- SamSam
- TeslaCrypt
- WannaCry

3.5 Cryptographic Constants

Other than the AES sbox defined in the FIPS 197 specification [44], it was difficult to find many cryptographic constants. The majority of cryptographic constants were discovered via source code of AES implementations. AES RCON and AES ltable were obtained from a Java implementation of AES by Johan Stenberg [51], the inverse AES sbox, POWX, and AES atable were obtained from an AES implementation by the Go Project [57], and the AES constant for the poly1305aes MAC was obtained directly

from its source code [7].

Similarly to the strings discussed in Section 3.4, each crptographic constant was encoded as a literal regular expression and stored in a static array. The regular expression to search for all constants was complied once and stored in a static variable for use on multiple binaries. The algorithm also returned a vector of indices into the array of constants for identification of all discovered constants.

Chapter 4

RESULTS

In this chapter, results of statistical analysis on binary samples is presented. Section 4.1 provides general statistics on all binary samples collected. Identified predictors from multiple logistic regression models are presented in Section 4.2.

4.1 Binary Statistics

A total of 2887 samples were analyzed. Of these samples, 159 samples contained less than 10 instructions and were not considered for further analysis due to possible errors in disassembly, leaving a total of 2728 samples. Table 4.1 gives a breakdown of the number of samples by type and platform. Note that most benign samples and all ransomware samples are PE binaries, while the majority of cryptographic samples are ELF binaries.

Table 4.2 lists means of each quantitative statistic for all sample types across both platforms. On average, each sample contained 35.04 loops, 791.92 bitwise arith-

Table 4.1: Summary of sample types (benign, cryptographic, and ransomware) of all binary samples based on platform (Unix and Windows).

	Sample Type			
Platform	**Benign**	**Cryptographic**	**Ransomware**	**All**
Unix	782	712	0	1494
Windows	1151	16	67	1234
Both	1933	728	67	2728

Table 4.2: Summary of means of quantitative statistics for all samples with outliers removed.

	Loops	Bitops	Strings	Constants	Instructions
Mean	35.04	791.92	2.05	0.05	11843.24
	Size (b)	**Runtime**	**Vertices**	**Edges**	
Mean	395023.06	30.82	2965.95	3255.80	

metic instructions, 2.05 potentially identifying strings, and 0.05 cryptographic constants. Additionally, the following statistics were collected in order to evaluate performance. On average, each sample contained 11843.24 total instructions, had a CFG with 2965.95 vertices and 3255.80 edges, had a runtime of 30.82 seconds, and were 395023.06 bytes in size.

Figure 4.1 provides boxplots for each statistic. Note that, even after removing outliers, the Inter-Quartile Range (IQR) is much lower than the maximum. It is also important to note that many of the outliers were much larger than the maximums shown in Figure 4.1. This is likely due to the fact that many of these outliers where large programs or libraries and cryptographic and ransomware samples. Cryptographic and ransomware samples are explored in greater detail in Section 4.1.1.

4.1.1 Sample Type Statistics

When examining statistics on observations grouped by sample type, several differences appear. Table 4.3 provides the means of each of the statistics presented in Section 4.1 for each sample type. Both benign and cryptographic samples were found to have similar means of number of potential strings (1.97 and 2.08 respectively), while ransomware samples had a mean number of potential strings more than double each

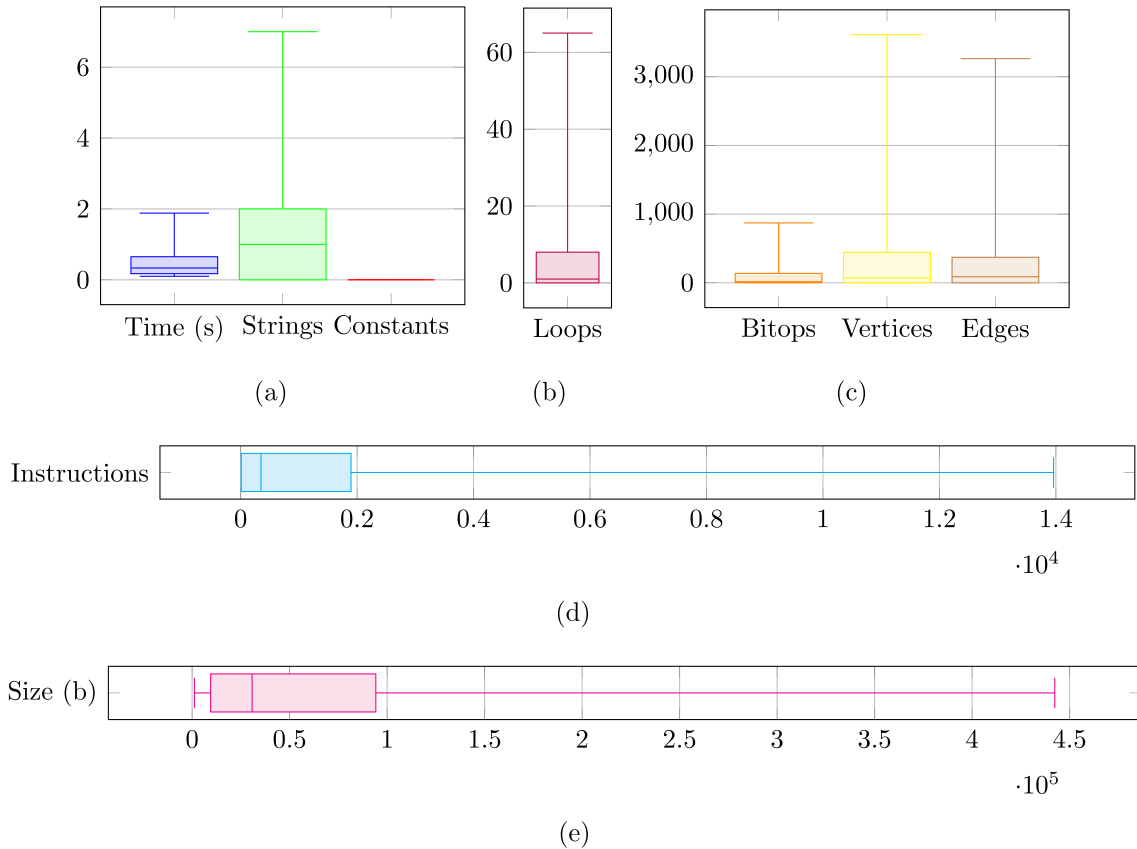


Figure 4.1: Boxplot of each quantitative statistic for all samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.

of the previous (4.23). Similarly, ransomware smaples were found to have a higher mean number of cryptographic constants detected (0.60) than both benign (0.04) and cryptographic (0.01) samples. One would expect the cryptographic samples to have a higher mean number of potential strings and cryptographic constants than benign samples. The fact that this is not reflected suggests that the majority of cryptographic libraries examined are not encryption routines. Since both potential strings and cryptographic constants are discovered by searching for regular expressions on whole binaries, finding either does not depend on the quality of the disassembler; they simply are not present.

Table 4.3: Summary of statistics of all binary samples based on type (benign, cryptographic, and ransomware).

	Sample Type		
Statistic	**Benign**	**Cryptographic**	**Ransomware**
Samples	1933	728	67
Loops	40.41	14.64	101.82
Bitops	915.38	444.09	1009.45
Strings	1.97	2.08	4.23
Constants	0.04	0.01	0.60
Instructions	14477.00	4242.96	18439.43
Size (b)	477532.02	60731.39	1646881.54
Runtime (s)	17.30	54.53	163.42
Vertices	3718.05	861.31	4135.54
Edges	3927.40	1185.51	6374.63

Cryptographic samples also had the lowest mean number of instructions (4242.96) and mean size in bytes (60731.39) compared to benign samples (14477 and 477532.02 respectively) while ransomware samples had the largest mean number of instructions (18439.43) and mean size in bytes (1646881.54). Unsurprisingly, cryptographic samples had the lowest mean number of vertices (861.31) and edges (1185.51), ransomware samples had the highest mean number of vertices (4135.54) and edges (6374.63), while benign samples fell somewhere in between (3718.05 and 3927.40 respectively). Counterintuitively, cryptographic samples had a larger mean runtime in seconds (54.53) than benign samples (17.30), but ransomware samples maintained the larges mean runtime in seconds (163.42). Figures figs. 4.2 to 4.4 showbox plots similar to Figure 4.1, but for benign, cryptographic, and ransomware samples, respectively.

Interestingly, cryptographic samples have lower means for both quantities of loops (14.64) and bitwise instructions (444.09), than benign samples (40.41 and 915.38 respectively), while ransomware samples have higher means for both (101.82 and 1009.45 respectively). According to Gröbert, et al., encryption routines have a high number of bitwise arithmetic and loops are a core components [26]. Thus, we expect to observe a higher number of bitwise instructions and loops in cryptographic binaries than non-cryptographic ones. In order to test whether or not the differences in means are statistically different, t-tests are performed using `R`. Although the sets of observations produce skewed distributions, the sample sizes are all over thirty. Thus, the Central Limit Theorem implies that the normality assumption for t-tests is no longer needed [19]. The following hypothesis tests are used to determine whether or not the difference in means for both number of loops and number of bitwise instructions are significant:

H_0**:** *The mean number of loops for cryptographic samples is equal to the mean number of loops for benign samples.*

H_A**:** *The mean number of loops for cryptographic samples is greater than the mean number of loops for benign samples.*

Using `R` to perform a t-test with a 95% confidence interval reveals a p-value of 1.0000. As a result, we fail to reject the null hypothesis, H_0, and show that there is clearly no statistical evidence that the mean number of loops in cryptographic samples are greater than the mean number of loops in benign samples. It is important to note, however, that Gröbert, et al. state that loops are a core component of cryptographic algorithms, which may not make up a significant portion of the cryptographic libraries examined or that the disassembler could not accurately disassemble those binaries.

Similarly to the hypothesis test for loops, we may perform a hypothesis test on the mean number of bitwise instructions for both benign and cryptographic samples.

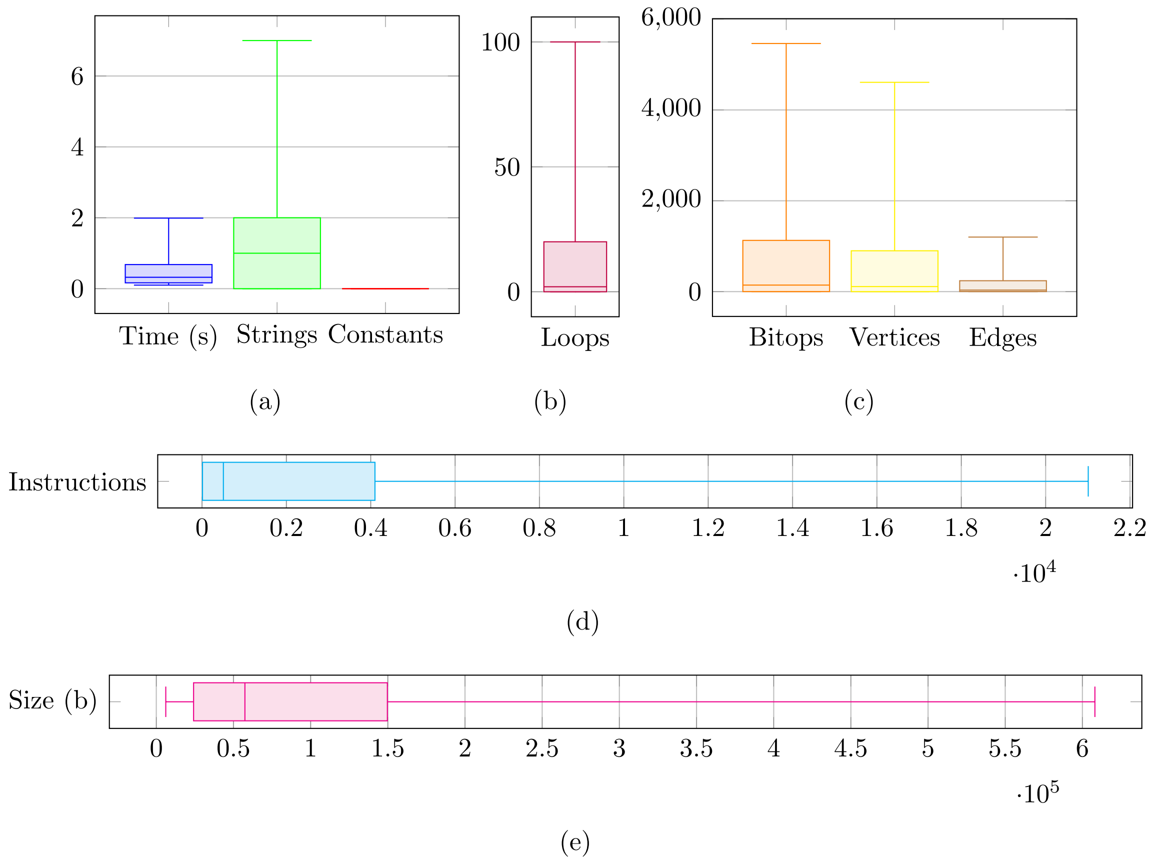


Figure 4.2: Boxplot of each quantitative statistic for benign samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.

H_0: *The mean number of bitwise instructions for cryptographic samples is equal to the mean number of bitwise instructions for benign samples.*

H_A: *The mean number of bitwise instructions for cryptographic samples is greater than the mean number of bitwise instructions for benign samples.*

With a p-value of 1.0000, we again fail to reject the null hypothesis, showing that there is no statistical evidence that the mean number of bitwise instructions for cryptographic samples is greater than the mean number of bitwise instructions in benign samples. Next, hypotheses between benign and ransomware samples are tested.

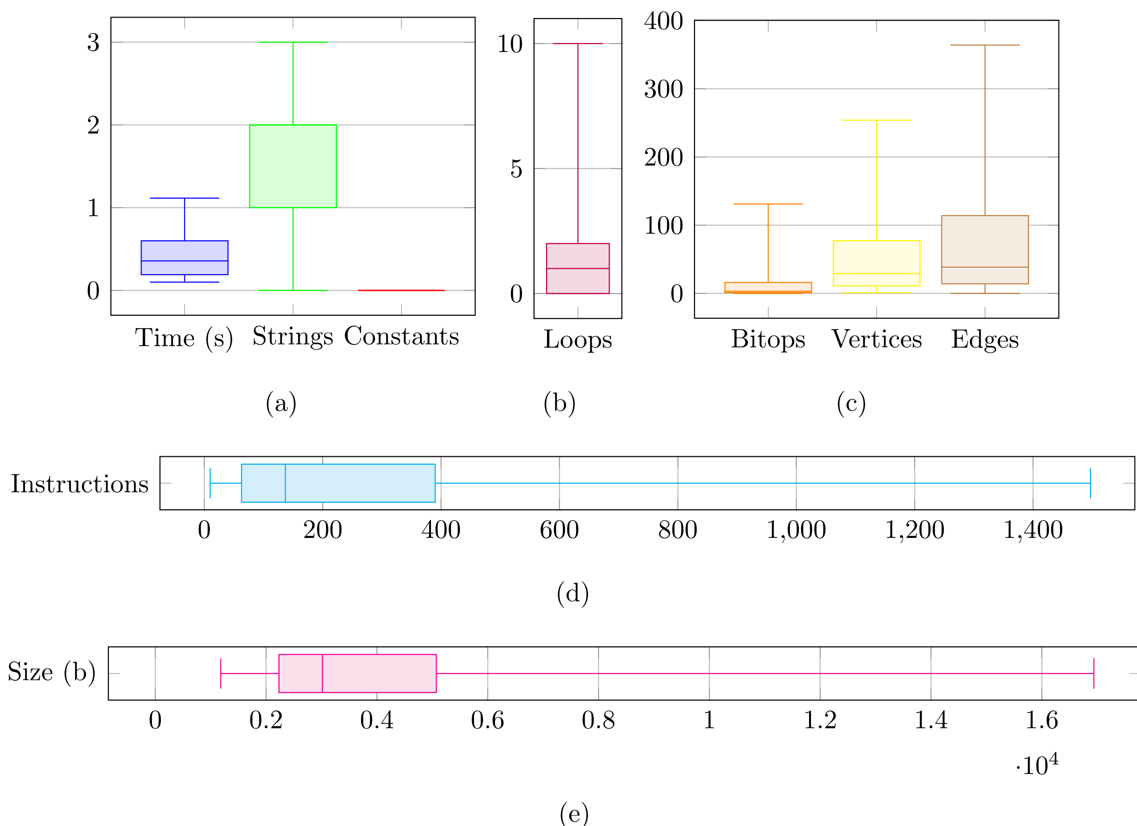


Figure 4.3: Boxplot of each quantitative statistic for cryptographic samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.

H_0: *The mean number of loops for ransomware samples is equal to the mean number of loops for benign samples.*

H_A: *The mean number of loops for ransomware samples is greater than the mean number of loops for benign samples.*

Using `R` to perform a t-test, a p-value of 0.0002 is discovered. We thus reject the null hypothesis and show that, with 95% confidence, the mean number of loops in ransomware samples is greater than the mean number of loops in benign samples. A comparison of bitwise instructions are tested with the following hypotheses:

H_0: *The mean number of bitwise instructions for ransomware samples is equal to the*

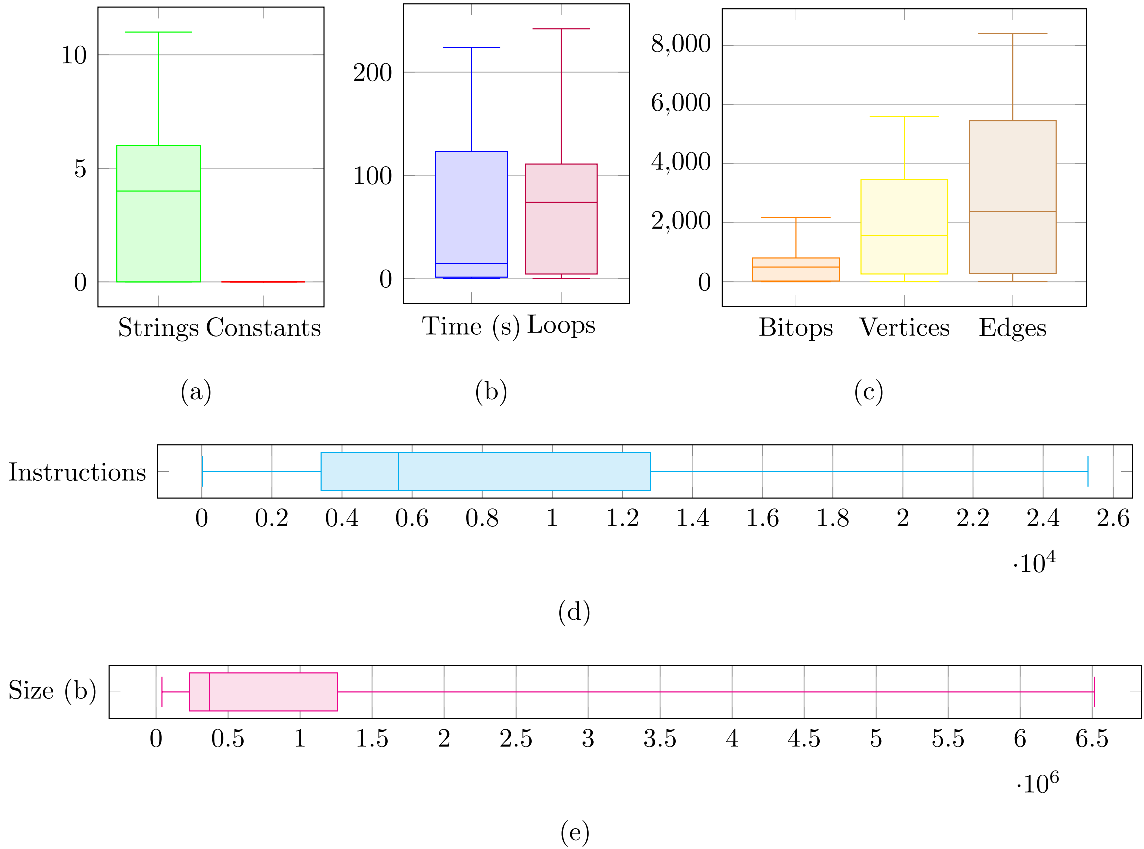


Figure 4.4: Boxplot of each quantitative statistic for ransomware samples. Upper and lower whiskers represent minimum and maximum values, while upper and lower edges of each box repectively represent the third and first quartiles. For clarity, outliers have been removed and plots with different scales are used.

mean number of bitwise instructions for benign samples.

H_A: *The mean number of bitwise instructions for ransomware samples is greater than the mean number of bitwise instructions for benign samples.*

A p-value of 0.3757 prevents a rejection of the null hypothesis. Therefore, we can conclude with 95% confidence that the mean number of bitwise instructions in ransomware samples is not greater than the mean number of bitwise instructions in benign samples.

The final hypothesis tests are between cryptographic and ransomware samples. First,

we test the following hypotheses about loops:

H_0**:** *The mean number of loops for ransomware samples is equal to the mean number of loops for cryptographic samples.*

H_A**:** *The mean number of loops for ransomware samples is not equal to the mean number of loops for cryptographic samples.*

A p-value of approximately 0.0000 allows us to reject the null hypothesis and show with 95% confidence that the mean number of loops in ransomware samples is not equal to the mean number of loops in cryptographic samples. Next, we examine the following hypotheses:

H_0**:** *The mean number of bitwise instructions for ransomware samples is equal to the mean number of bitwise instructions for cryptographic samples.*

H_A**:** *The mean number of bitwise instructions for ransomware samples is not equal to the mean number of bitwise instructions for cryptographic samples.*

With a p-value of 0.0659, we fail to reject the null hypothesis. Thus, with 95% confidence, we note that there is no statistical difference between the mean number of bitwise instructions in ransomware samples and the mean number of bitwise instructions in cryptographic samples.

4.2 Model Predictors

Observations were divided into three subsets containing only two sample types each in order to determine predictors via multiple logistic regression. The dataset was divided into the followin subsets: benign and cryptographic samples, benign and ransomware samples, and cryptographic and ransomware samples. For each subset of observations,

the reduced multiple logistic regression model was selected by eliminating variables from the full model until the lowest AIC and residual deviance was obtained. The full multiple logis regr model consists of the following form with the type code (i.e. benign, cryptographic, ransomare) as the response variable, written as a function of the variables listed in Table 4.4:

$$e^{b_0+b_1x_1+b_2x_2+\cdots+b_kx_k}.$$

Each b_i coefficient was calculated by `R`.

4.2.1 Benign Vs Cryptographic

The comparison between benign and cryptographic samples accomplishes the goal of examining unproven claims from previous work [26]. The reduced model found for the subset consisting of benign and cryptographic samples was found to have a minimized AIC of 1200.75, a residual deviance of 1172.75 on 2647 degrees of freedom, and a null deviance of 3122.89 on 2660 degrees of freedom when using a reduced model with variables listed in Table 4.5. Although several variables are not statistically significant, removing them would result in a model with both a higher AIC and a higher residual deviance.

Recall that the probability that a logistic regression model produces the value 1 is defined by the logit function

$$\pi(x) = \frac{e^{b_0+b_1x}}{1+e^{b_0+b_1x}}$$

which can be rewritten as

$$\frac{\pi(x)}{1-\pi(x)} = e^{b_0+b_1x}$$

and represents the odds of success.

Table 4.4: The variables used in the full multiple logistic regression model. Note that the qualitative variables have binary values that represent whether the item was found (1) in a binary sample or not (0).

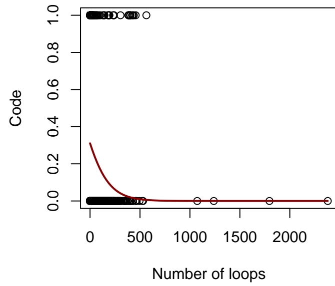
Quantitative				
Quantity	• Loops • Bitwise instructions			
Qualitative				
Strings	• “aes” • “AES” • “rsa” • “RSA” • “des” • “DES”	• “your” • “Your” • “YOUR” • “ransom” • “key” • “.onion”	• “chacha” • “bitcoin” • “crypt” • “Crypt” • “CRYPT” • “HELP”	• “files” • “FILES” • “txt” • “TXT” • “payment”
Constants	• AES sbox • Inverse AES sbox • AES powx • AES atable		• AES ltable • AES rcon • poly1305AES	

Table 4.5: Summary of multiple logistic regression on the reduced model for the benign and cryptographic subset of observations. Note that the intercept (b_0) for this model was found to be -1.660290.

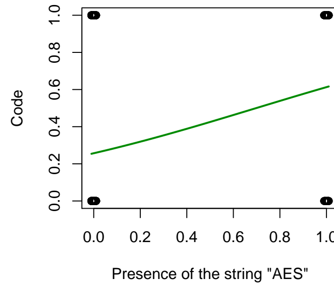
Model Quality				
Null Dev	**Null DF**	**Resid Dev**	**Resid DF**	**AIC**
3122.89	2660	1172.75	2647	1200.75

Quantitative Variables			
Variable	**Coefficient**	**p-value**	**Significant?**
Loops	-0.019443	1.17e-11	Yes

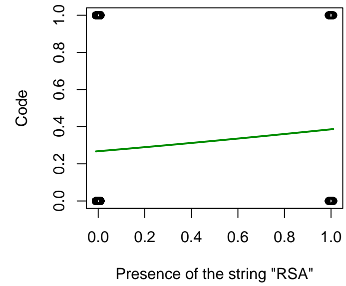
Nominal Variables			
Variable	**Coefficient**	**p-value**	**Significant?**
"aes"	2.591519	0.000194	Yes
"AES"	1.448235	0.053676	No
"RSA"	1.993096	7.29e-06	Yes
"des"	-1.058622	5.57e-08	Yes
"DES"	-1.214891	0.010120	Yes
"chacha"	8.868145	1.51e-07	Yes
"crypt"	4.613125	$< 2e-16$	Yes
"Crypt"	-3.818068	$< 2e-16$	Yes
"CRYPT"	-0.637244	0.091241	No
"your"	-24.541630	0.961825	No
"files"	-4.958511	5.43e-11	Yes
"FILES"	-6.873966	0.280569	No
"payment"	4.768951	0.455382	No



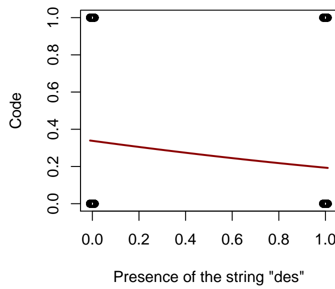
(a) Loops.



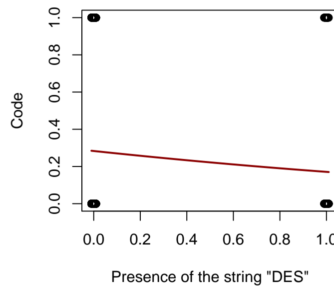
(b) "aes".



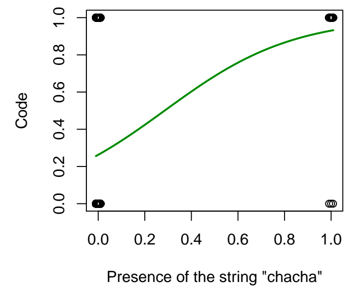
(c) "RSA".



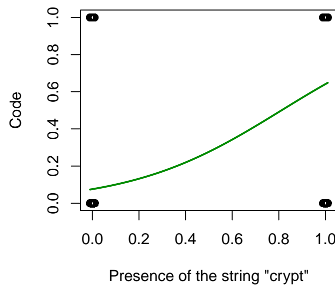
(d) "des".



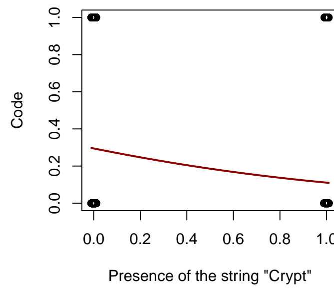
(e) "DES".



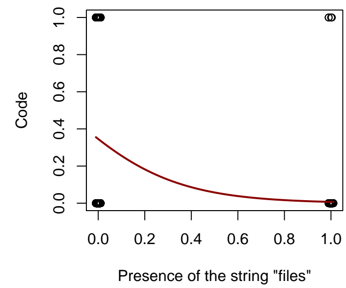
(f) "chacha".



(g) "crypt".



(h) "Crypt".



(i) "files".

Figure 4.5: Logistic regression predictions from the benign vs. cryptographic dataset for each variable listed.

Table 4.6: Probabilities of success ($\pi(x)$) and estimated odds ratio (EOR) for each of the statistically significant variables corresponding to the presence of the listed string in the begin vs. cryptographic sample dataset. Note that negative EOR indicates a decrease in probability of success when the variable is 1.

Variable	π(x)	EOR	Variable	π(x)	EOR
"aes"	0.7167	12.3500	"crypt"	0.9503	99.7987
"RSA"	0.5817	6.3382	"Crypt"	0.0041	-0.9780
"des"	0.0617	-0.6531	"chacha"	0.9993	7101.094
"DES"	0.0533	-0.7033	"files"	0.0013	-0.9930

Upon examining the affect that the number of loops have in this model, we discover that the odds ratio

$$e^{b_0} = e^{-1.663227} = 0.9807448 < 1$$

which is interpreted as a decrease of approximately 1.93% in probability of success for each additional loop discovered in a sample. Since an increase in the quantity of loops causes a decrease in the probability of success, the maximum positive effect loops will have on this model is when there are no loops, in which

$$\pi(0) = \frac{e^{b_0 + b_1 \cdot 0}}{1 + e^{b_0 + b_1 \cdot 0}} = \frac{e^{-1.663227}}{1 + e^{-1.663227}} = 0.1593293$$

which yields probability of approximately 15.93%. The probabilities of success when the mean number of loops for both benign and cryptographic samples yield were found to be 12.45% and 7.87% respectively. Figure 4.5(a) depicts the negative effect an increasing number of loops have on the probability of success.

The remainder of the variables in this model were nominal, which means they only take on values of zero or one. We calculate the effect each variable has on the model alone by setting each other variable to zero and setting the variable in question to one. Table 4.6 shows the probability and estimated odds ratio for each of the nominal

variables in this model, which happen to all represent the presence of strings in a sample binary.

To determine the effect of the presence of a string in a binary, without any other variables, we calculate the probability of success $\pi(x)$ on that single variable as above. We will only consider nominal variables which are statistically significant. We discover that the presence of the string "aes" alone has a 71.67% probability of successfully identifying the sample as cryptographic. The presence of the string "RSA" was found to have slightly lower probability of 58.32%.

The strings that provided the highest probabilities of success when present were "chacha" and "crypt", with probabilities of 99.95% and 95.03% respectively. The remainder of strings had much lower probabilities of success, with the string "des" being the highest with a probability of 6.18% and the string "your" being the lowest with a probability of success of 0%.

When combined with mean number of loops found in either benign or cryptographic samples, we find different results. Recall from Section 4.1 that the mean number of loops for benign samples was found to be 40.41 and the mean number of loops for cryptographic samples was found to be 14.64. The probability of success when the strings "aes" and "RSA" are present is

$$\pi(\mathbf{x}) = \pi(x_1, x_2, x_3) = \frac{e^{b_0+b_1x_1+b_2x_2+b_3x_3}}{1+e^{b_0+b_1x_1+b_2x_2+b_3x_3}}$$

$$\pi(14.64, 1, 1) = \frac{e^{-1.660290+-0.0198\cdot 14.64+1.4165+1.9960}}{1+e^{-1.660290+-0.0198\cdot 14.64+1.4165+1.9960}} = 0.9995$$

and

$$\pi(40.41, 1, 1) = \frac{e^{-1.660290+-0.0198\cdot 40.41+1.4165+1.9960}}{1+e^{-1.660290+-0.0198\cdot 14.64+1.4165+1.9960}} = 0.8976$$

when the mean number of loops for benign and cryptographic samples are used re-

spectively. Thus, the strings “aes” and “RSA” are good predictors of cryptographic code when they are present at the same time,

Using a similar method, we find that the probability of success when the strings “des”, “DES”, “Crypt”, “your”, “files”, and “FILES” are simultaneously present with the mean number of loops for benign samples is 8.80%. Similarly, when paired with the mean number of loops of cryptographic samples, the probability of success is 13.85%. Even when no loops are present, these variables provide a mere 17.68% probability of success. Note from Table 4.6 that each of these variables have a decreasing EOR, which is one reason they are poor.

In summary, an increase in loops causes a decrease in probability of success. More importantly, the presence of the strings “chacha” and “crypt” are good indicators that the sample in question is cryptographic. The presence of both strings “aes” and “RSA” at the same time forms another good indicator that the sample in question is cryptographic. The remaining variables are not good indicators at all.

4.2.2 Benign Vs. Ransomware

The best reduced model found for the subset of data consisting of benign and ransomware samples was found to have an AIC of 438.13, a residual deviance of 414.13 on 1988 degrees of freedom, and a null deviance of 586.82 on 1999 degrees of freedom, using the variables listed in Table 4.7. The number of loops was the only quantitative variable that was statistically significant. The EOR for the number of loops was 1.0023, which is greater than 1 and is interpreted as there being a 0.2804% increase in probability of success for each additional loop found in a sample. This relationship can be seen in Figure 4.6 (a). In order to obtain a 95% probability of success, at least 2472 loops must be present.

Table 4.7: Summary of multiple logistic regression on the reduced model for the benign and ransomware subset of observations. Note that the intercept (b_0) for this model was found to be -3.872675.

Model Quality

Null Dev	Null DF	Resid Dev	Resid DF	AIC
586.82	1999	414.13	1988	438.13

Quantitative Variables

Variable	Coefficient	p-value	Significant?
Loops	0.0028	0.0000	Yes

Nominal Variables

Variable	Coefficient	p-value	Significant?
"aes"	-4.8899	0.0003	Yes
"AES"	1.7500	0.0037	Yes
"Crypt"	1.1275	0.0071	Yes
"CRYPT"	-1.0612	0.0447	Yes
"files"	-1.8640	0.0002	Yes
"txt"	2.1011	0.0000	Yes
"TXT"	1.7594	0.0092	Yes
".onion"	-4.1656	0.0219	Yes
"payment"	7.1336	0.0000	Yes
AES POWX	7.0682	0.0000	Yes

Table 4.8: Probabilities of success ($\pi(x)$) and estimated odds ratio (EOR) for each of the variables corresponding to the presence of the listed nominal variable in the begin vs. ransomware sample dataset. Note that negative EOR indicates a decrease in probability of success when the variable is 1.

Variable	$\pi(\mathbf{x})$	EOR	Variable	$\pi(\mathbf{x})$	EOR
"aes"	0.0002	-0.9925	"Crypt"	0.0604	2.0880
"AES"	0.1069	4.7546	"CRYPT"	0.0071	-0.6540
".onion"	0.0003	-0.9845	"txt"	0.1454	7.1755
"payment"	0.9631	1252.3608	"TXT"	0.1078	4.8092
"files"	0.0032	-0.8450	AES powx	0.9607	1172.9802

The remainder of variables in the model are all nominal. Only one nominal variable represents the presence of cryptographic constants. The remainder represent the presence of strings. Table 4.8 lists the probabilities ($\pi(x)$) and EOR for each. Interestingly, both strings "aes" and "AES" were found to have lower probabilities of success than the previous subset of data. Each produced probabilities of 0.02% and 10.69% respectively.

Similarly, the presence of the strings "txt" and "TXT" provided 14.54% and 10.78% probabilities of success, respectively. The strings "CRYPT", "files", and ".onion" provided very low probabilities of success 0.71%, 0.32%, and 0.03% respectively when present. The variables with the largest probabilities of success were the presence of the string "payment" and the cryptographic constant AES powx with respective probabilities of success 96.31% and 96.07%.

The probability of success when every nominal variable, except "payment" and AES powx, are present is 0.01%. However, this result is misleading. If we instead consider the variables "AES", "Crypt", "txt", and "TXT", which all have increasing EORs, a probability of success of 94.61% is produced. When two of each of these variables

are paired together, we observe the following. When "AES" and "Crypt" are both present, a probability of success of 26.99% occurs. When "AES" and "txt" are both present, the probability of success produced is 49.46%. When "AES" and "TXT" are both present, a probability of success of 41.02% can be seen. When "Crypt" and "txt" are seen simultaneously, a probability of success of 34.43% can be found. When "Crypt" and "TXT" are both present, the probability of success is 27.17%. Finally, when "txt" and "TXT" are both present, a probability of success of 49.70% will be observed.

If instead we consider groups of three out of the four strings listed above, we obtain different results. When the strings "AES", "Crypt", and "txt" are all present, the probability of success is calculated to be 75.14%. When "AES", "Crypt", and "TXT" are all present, the probability of success is found to be 68.23%. When "AES", "txt", and "TXT" are all present, the probability of success is 85.04%. Finally, when "Crypt", "txt", and "TXT" are simultaneously present, the probability of success is 75.31%.

Again, recall that the mean number of loops for benign samples was found to be 40.41 and the mean number of loops for ransomware samples was found to be 101.82. Each of these on their own yield 2.28% and 2.69% probabilities of success, respectively. When combined with any of the above combinations or any single variable, a marginal improvement of less than 4% is observed for all. Out of the 16 samples that were observed to contain the cryptographic constant AES powx, 12 of them were WannaCry samples. The remaining were: `mongoimport`, `mongotop`, `mongostat`, and `pandoc`.

In summary, the presence of the string "payment", the cryptographic constant AES powx are, or the simultaneous presence of the strings "AES", "Crypt", "txt", and "TXT" good indicators that a sample is ransomware. Additionally, the presence of

any combination of three of these four strings is a moderate indicator that a sample is ransomware. Quantities of 2472 loops or higher constitute a good indicator that a sample is ransomware.

4.2.3 Cryptographic Vs. Ransomware

For completeness, we also compared cryptographic samples to ransomware. The reduced model for the subset of data consisting of cryptographic and ransomware samples had an AIC of 117.85, a residual deviance of 89.85 on 781 degrees of freedom, and a null deviance of 459.66 on 794 degrees of freedom. Although including bitwise instruction counts helped to reduce this model's AIC, only the number of loops was statistically significant. In order to achieve a 95% or higher probability of success, at least 90 loops must be present. All of the variables required to produce the model with the best fit are listed in Table 4.9.

Table 4.10 provides the probabilities of success and EORs for each of the statistically significant nominal variables in this model. Note that when AES sbox is present, the probability of success is 100%. This constant was found to be present in WannaCry and LockerGoga samples, as well as `libbotan`, `libcrypto`, `libcrypto++`, and `tiny AES`. The presence of strings "aes" and "crypt" result in extremely low probabilities of success 0% and 0.33%, respectively. The presence of the srings "rsa" and "Crypt" fall somewhere in between these two groups, providing 64.04% and 35.52% respective probabilities of success. When both "rsa" and "Crypt" are present, the probability of success is calculated to be 95.69%.

In Summary, the presence of the cryptographic constant AES sbox and the simultaneous presence of the strings "rsa" and "Crypt" are good indicators that a sample is ransomware and not cryptographic. While not particularly useful alone, these re-

Table 4.9: Summary of multiple logistic regression on the reduced model for the cryptographic and ransomware subset of observations. Note that the intercept (b_0) for this model was found to be -3.119408.

Model Quality

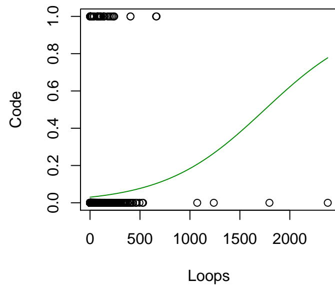
Null Dev	**Null DF**	**Resid Dev**	**Resid DF**	**AIC**
459.66	794	89.85	781	117.85

Quantitative Variables

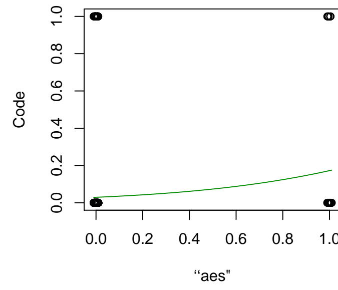
Variable	**Coefficient**	**p-value**	**Significant?**
Loops	0.0675	0.0000	Yes
Bitops	-0.0013	0.0632	No

Nominal Variables

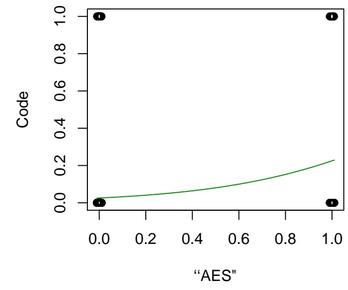
Variable	**Coefficient**	**p-value**	**Significant?**
“aes”	-26.3772	0.0359	Yes
“rsa”	3.6964	0.0003	Yes
“RSA”	-3.5480	0.1787	No
“crypt”	-2.5837	0.0018	Yes
“Crypt”	2.5233	0.0458	Yes
“CRYPT”	-3.7832	0.0600	No
“your”	24.5651	0.9932	No
“key”	-1.6280	0.1094	No
“txt”	4.9.233	0.0543	No
“TXT”	43.1524	0.9847	No
AES sbox	30.8642	0.0140	Yes



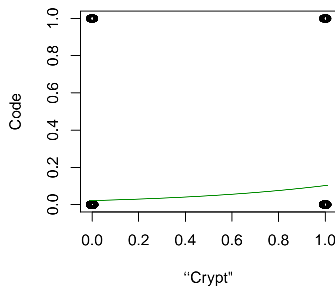
(a) Loops.



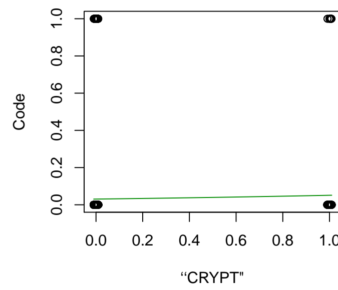
(b) “aes”.



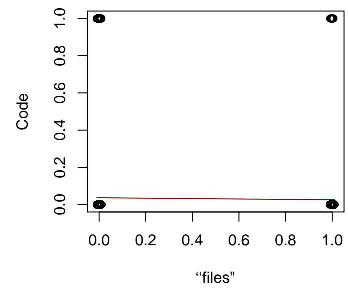
(c) “AES”.



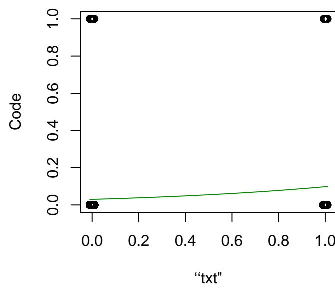
(d) “Crypt”.



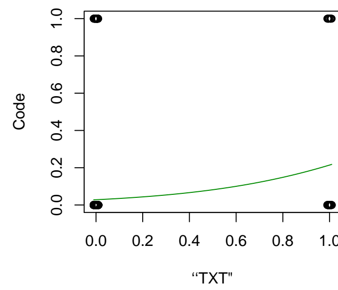
(e) “CRYPT”.



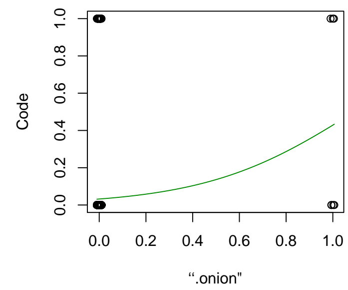
(f) “files”.



(g) “txt”.

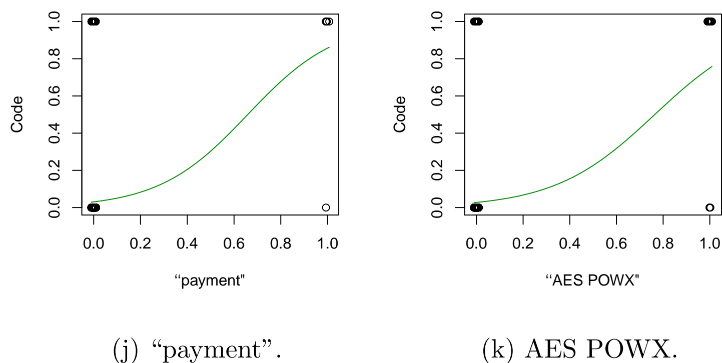


(h) “TXT”.



(i) “.onion”.

Figure 4.6: Logistic regression predictions from the benign vs. ransomware dataset for each variable listed.



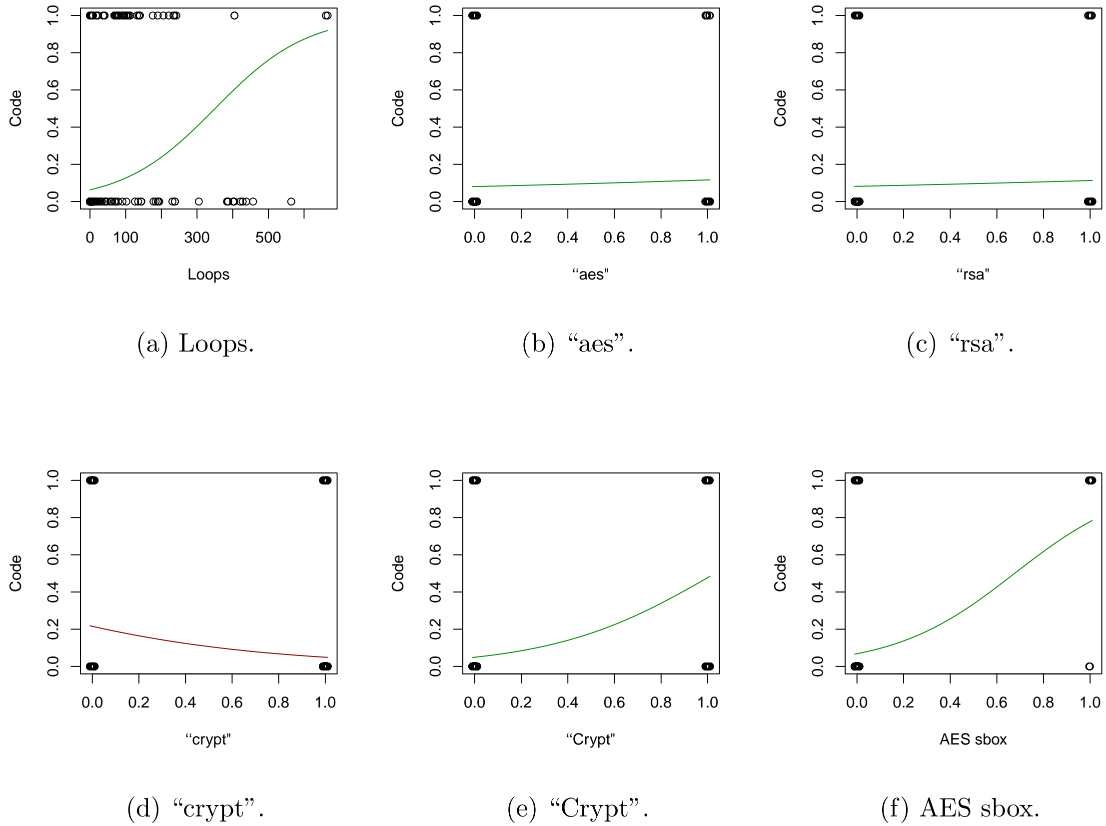
(j) "payment".   (k) AES POWX.

Figure 4.6: Continued logistic regression predictions from the benign vs. ransomware dataset for each variable listed.

Table 4.10: Probabilities of success ($\pi(x)$) and estimated odds ratio (EOR) for each of the variables corresponding to the presence of the listed nominal variable in the cryptographic vs. ransomware sample dataset. Note that negative EOR indicates a decrease in probability of success when the variable is 1.

Variable	$\pi(\mathbf{x})$	**EOR**	**Variable**	$\pi(\mathbf{x})$	**EOR**
"aes"	0.0000	-1.0000	"crypt"	0.0033	-0.9245
"rsa"	0.6404	39.3022	"Crypt"	0.3552	11.4700
AES sbox	1.0000	2.536e14			

sults provide some insight into the difference of cryptographic and ransomare samples obtained in this study.



(a) Loops. (b) "aes". (c) "rsa".

(d) "crypt". (e) "Crypt". (f) AES sbox.

Figure 4.7: Logistic regression predictions from the cryptographic vs. ransomware dataset for each variable listed.

Chapter 5

CONCLUSION

In this work, we describe statistics on binary samples obtained and analysis on logistic regression models in order to derive statistically significant predictors of ransomware. We find that the presence of each of the string “payment” and the cyrpographic constant AES powx are significantly statistical predictors that a sample is ransomware. Additionally, the presence of at least three out of the four strings “AES”, “Crypt”, “txt”, or “TXT” is also a moderately good indicator that a sample is ransomware.

We have also shown that the presence of either of the strings “chacha” or “crypt”, or the simultaneous presence of the strings “aes” and “RSA” are statistically significant predictors that a sample is cryptographic and not benign. We have also shown that a large number of loops suggests that a sample is benign and not cyrpographic, but may also predict that a sample is ransomware and not benign software. The number of bitwise instructions was shown to not be a statistically significant predictor in any of the three experiments. Thus, the number of loops and number of bitwise operations are not reliable predictors of either cryptographic or ransomware samples.

5.1 Future Work

Several areas for improvement exist for future work. While thousands of samples were collected in this work, millions of samples would be better. Therefore, in the future more samples of each type of software should be collected. In particular, it may be useful to identify and isolate cryptographic primitives from known cryptographic libraries to more precisely study their properties. Additionally, incorporating an

existing framework for disassembling, CFG reconstruction, and loop detection should be utilized. Not only will this enable more accurate results, it will allow for new possible predictors to be explored.

BIBLIOGRAPHY

[1] `https://www.malwaretech.com/2017/05/how-to-accidentally-stop-a-global-cyber-attacks.html`.

[2] M. M. Ahmadian and H. R. Shahriari. 2entfox: A framework for high survivable ransomwares detection. In *2016 13th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*, pages 79–84, Sept 2016.

[3] M. M. Ahmadian, H. R. Shahriari, and S. M. Ghaffarian. Connection-monitor & connection-breaker: A novel approach for prevention and detection of high survivable ransomwares. In *Information Security and Cryptology (ISCISC), 2015 12th International Iranian Society of Cryptology Conference on*, pages 79–84. IEEE, 2015.

[4] D. Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. No Starch Press, Incorporated, 2018.

[5] N. Andronio, S. Zanero, and F. Maggi. Heldroid: Dissecting and detecting mobile ransomware. In H. Bos, F. Monrose, and G. Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, pages 382–404, Cham, 2015. Springer International Publishing.

[6] P. Bajpai, A. K. Sood, and R. Enbody. A key-management-based taxonomy for ransomware. In *2018 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–12. IEEE, 2018.

[7] D. J. Bernstein. A state-of-the-art message-authentication code. `https://cr.yp.to/mac.html`.

[8] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[10] K. Cabaj, P. Gawkowski, K. Grochowski, and D. Osojca. Network activity analysis of cryptowall ransomware. *Przeglad Elektrotechniczny*, 91(11):201–204, 2015.

[11] A. Caplan. Hacker paid $490,000 in bitcoin for ransomware attack on lake city. `https://www.gainesville.com/news/20190625/hacker-paid-490000-in-bitcoin-for-ransomware-attack-on-lake-city`.

[12] L. D. Carli, R. Torres, G. Modelo-Howard, A. Tongaonkar, and S. Jha. Kali: Scalable encryption fingerprinting in dynamic malware traces. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 3–10, Oct 2017.

[13] C. Cimpanu. Georgia county pays a whopping $400,000 to get rid of a ransomware infection. `https://www.zdnet.com/article/georgia-county-pays-a-whopping-400000-to-get-rid-of-a-ransomware-infection/`.

[14] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi. Shieldfs: A self-healing, ransomware-aware filesystem. In *Proceedings of the 32Nd Annual Conference on Computer*

Security Applications, ACSAC '16, pages 336–347, New York, NY, USA, 2016. ACM.

[15] V. contributors. Ransomware overview. `https://docs.google.com/spreadsheets/d/1TWS238xacAto-fLKh1n5uTsdijWdCEsGIM0Y0Hvmc5g/pubhtml`.

[16] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[17] V. C. Craciun, A. Mogage, and E. Simion. Trends in design of ransomware viruses. In *International Conference on Security for Information Technology and Communications*, pages 259–272. Springer, 2018.

[18] B. N. Danny Shaw. Eurofins scientific: Forensic services firm paid ransom after cyber-attack. `https://www.bbc.com/news/uk-48881959`.

[19] J. Devore, N. Farnum, and J. Doi. *Applied Statistics for Engineers and Scientists*. Cengage Learning, 2013.

[20] A. Di Federico, M. Payer, and G. Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141. ACM, 2017.

[21] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, 2011.

[22] Z. A. Genç, G. Lenzini, and P. Y. Ryan. Security analysis of key acquiring strategies used by cryptographic ransomware. In *Proceedings of the Central European Cybersecurity Conference 2018*, page 7. ACM, 2018.

[23] L. Georgiadis, R. E. Tarjan, and R. F. Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications*, 10(1):69–94, 2006.

[24] GNU. Documentation for binutils 2.32. `https://sourceware.org/binutils/docs-2.32/bfd/index.html`.

[25] D. Gonzalez and T. Hayajneh. Detection and prevention of crypto-ransomware. In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 472–478. IEEE, 2017.

[26] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In R. Sommer, D. Balzarotti, and G. Maier, editors, *Recent Advances in Intrusion Detection*, pages 41–60, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[27] S. Hanov. Static analysis of binary executables. *University of Waterloo*, 2009.

[28] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[29] S. Hsiao and D. Kao. The static analysis of wannacry ransomware. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 153–158, Feb 2018.

[30] D.-Y. Kao and S.-C. Hsiao. The dynamic analysis of wannacry ransomware. In *Advanced Communication Technology (ICACT), 2018 20th International Conference on*, pages 159–166. IEEE, 2018.

[31] J. Katz and Y. Lindell. *Introduction to modern cryptography.* Chapman and Hall/CRC, 2014.

[32] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In

International Conference on Computer Aided Verification, pages 423–427. Springer, 2008.

[33] S. Kramer and J. C. Bradfield. A general definition of malware. *Journal in computer virology*, 6(2):105–114, 2010.

[34] M. Kutner. *Applied Linear Statistical Models*. McGrwa-Hill international edition. McGraw-Hill Irwin, 2005.

[35] Y. Lemmou and E. M. Souidi. Inside gandcrab ransomware. In J. Camenisch and P. Papadimitratos, editors, *Cryptology and Network Security*, pages 154–174, Cham, 2018. Springer International Publishing.

[36] W. Mao. *Modern Cryptography: Theory and Practice*. HP Professional Series. Prentice Hall PTR, 2004.

[37] F. Monaco. malware-samples. `https://github.com/fabrimagic72/malware-samples/tree/master/Ransomware`.

[38] E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[39] T. of Bits. Seriously, stop using rsa. `https://blog.trailofbits.com/2019/07/08/fuck-rsa/`.

[40] P. O'Kane, S. Sezer, and D. Carlin. Evolution of ransomware. *IET Networks*, 7(5):321–327, 2018.

[41] A. Otis. Vbd. `https://github.com/aaron-otis/VBD`.

[42] D. Palmer. Ransomware warning: The gang behind this virulent malware just changed tactics again.

https://www.zdnet.com/article/ransomware-warning-the-gang-behind-this-virulent-malware-just-changed-tactics-again/.

[43] N. A. Quynh. Capstone engine. https://github.com/aquynh/capstone.

[44] V. Rijmen and J. Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[45] J.-M. Roberts. Virusshare.com - because sharing is caring. https://virusshare.com/.

[46] K. Savage, P. Coogan, and H. Lau. The evolution of ransomware. *Symantec, Mountain View*, 2015.

[47] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[48] Y. Shoshitaishvili, R. F. Wang, A. Dutcher, L. Dresel, E. Gustafson, N. Redini, P. Grosen, C. Unger, C. Salls, N. Stephens, C. Hauser, and J. Grosen. angr. http://angr.io/.

[49] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press Series. No Starch Press, 2012.

[50] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, 1996.

[51] J. Stenberg. Aes/src/org/johanstenberg/aes/constants.java. https://github.com/johanstenberg92/AES/blob/master/src/org/johanstenberg/aes/Constants.java.

[52] D. R. Stinson. *Cryptography: theory and practice*. CRC press, 2005.

[53] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[54] T. R. Team. The ‘bindgen‘ user guide. https://rust-lang.github.io/rust-bindgen/.

[55] T. R. Team. Foreign function interface. https://doc.rust-lang.org/nomicon/ffi.html.

[56] T. R. Team. regex crate. https://github.com/rust-lang/regex.

[57] G. I. The Go Project. Source file src/crypto/aes/const.go. https://golang.org/src/crypto/aes/const.go.

[58] H. Theiling. Extracting safe and precise control flow from binaries. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, pages 23–30. IEEE, 2000.

[59] X. Xiaofei. Static loop analysis and its applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1130–1132, New York, NY, USA, 2016. ACM.

[60] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep*, 2009.

[61] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *sp*, page 0129. IEEE, 1996.

[62] A. Young and M. Yung. Deniable password snatching: On the possibility of evasive electronic espionage. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 224–235. IEEE, 1997.

[63] A. Young and M. Yung. Kleptography: Using cryptography against cryptography. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 62–74. Springer, 1997.

[64] A. Young and M. Yung. *Malicious cryptography: Exposing cryptovirology*. John Wiley & Sons, 2004.

[65] A. Young and M. Yung. Kleptography from standard assumptions and applications. In *International Conference on Security and Cryptography for Networks*, pages 271–290. Springer, 2010.

[66] A. Young and M. Yung. Cryptography as an attack technology: proving the rsa/factoring kleptographic attack. In *The New Codebreakers*, pages 243–255. Springer, 2016.

[67] A. L. Young. *Kleptography: using cryptography against cryptography*. Columbia University, 2002.

[68] A. L. Young and M. M. Yung. An implementation of cryptoviral extortion using microsoft's crypto api. 2005.