THE UNIVERSITY OF WESTERN AUSTRALIA

Achieving International Excellence

Generating Malware Variants Using Machine Learning

Thesis Proposal

Semester 2 – 2018

Jason Seotis - 21877352

School of Computer Science and Software Engineering, University of Western Australia

Table of Contents

1. Introduction

Cyber-attacks are on the rise and costs companies millions of dollars each year. In a report published in 2017 by Accenture of 254 companies, the average cost annually to businesses due to malware attacks was $3.2 million AUD [9]. Malware is simply software coupled with malicious intent. This type of software is written specifically to cause harm to a computer or benefit an attacker at the expense of the victim. There are many different categories of malware including Ransomware, Spyware and Viruses. In a separate survey, it was identified that 91% of organisation surveyed experienced at least one cyber-attack in the last 12 months, with attacks commonly coming from phishing, spam and malware [7]. The implications for businesses are real and include loss of business-critical data, data breaches resulting in intellectual property being stolen, reputational damage and loss of money due to extortion. Antivirus and cybersecurity companies are trying to stay one step ahead of the bad guys by finding new ways to detect malware and any malware variants.

This proposal first explores how malware is detected and provides three techniques to assist with detection. Each technique is further broken down into three different approaches. Next, I explore techniques that malware uses to avoid detection. Four techniques are discussed including three different types of malware which can mutate. Obfuscation techniques are identified and then investigated to determine their effectiveness against both Ssdeep and TLSH. Finally, I discuss the algorithms of both Ssdeep and TLSH to better understand how they work. Even though Ssdeep is the industry standard, this project will focus on breaking TLSH as it's proven to be more robust against attacks [1]. The research gap is whether TLSH can be broken i.e. able to generate a file with identical functionality but appears different from TLSH. This is a growing field with mobiles devices such as smartphones and tablets gaining plenty of attention in recent years and can be the focus of future research to prevent these attacks.

My contribution towards this research area is to develop a web-based application that:

- Calculates the TLSH for a malware source provided by the user as input
- Uses machine learning to generate a malware variant such that the TLSH distance between the two files is greater than a user-specified threshold

2. Literature Review

2.1 Detecting Malware

The techniques for detecting malware can be categorised into two categories: anomaly-based detection and signature-based detection [10]. In addition to these two categories, there is a special type of anomaly-based detection known as specification-based detection. Each category is broken down into three different approaches. The taxonomy of malware detection techniques is shown in Figure 1

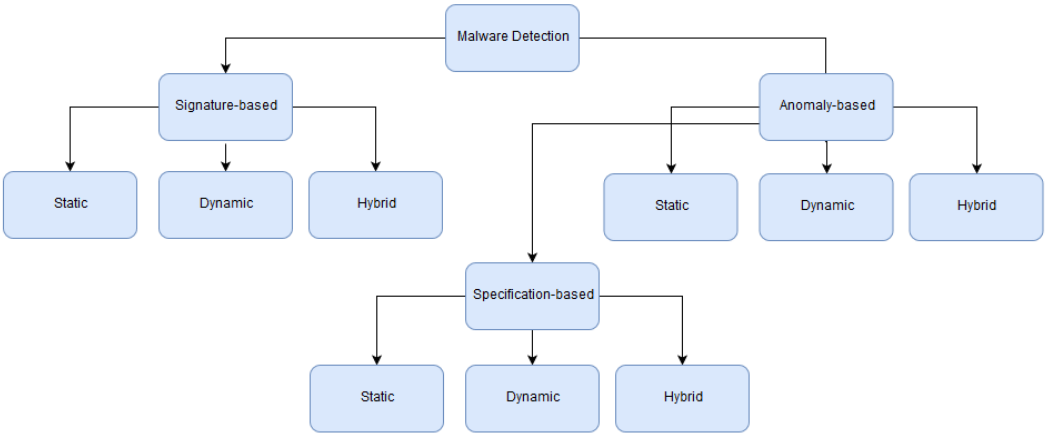

Figure 1: *Malware Detection Taxonomy [10]*

Signature-based detection uses what it knows to be malicious and compares it with the program being analysed. This technique requires a signature database to be regularly updated otherwise new malware signatures cannot be detected. The signature database is stored locally but will connect to a central server to download new signatures. An issue with this technique is that it cannot detect zero-day attacks. Anomaly-based detection uses its knowledge of what constitutes normal behaviour to decide the maliciousness of a program under inspection [10]. In other words, it creates a baseline of what the expected behaviour is for a program and when unusual behaviour is detected, its flagged as an anomaly. An advantage of this technique is that it can detect zero-day attacks however it's more complex due to deciding what should be learnt during the training phase. Specification-based detection is a type of anomaly-based technique in which a specification defines the valid behaviour and programs that violate the specification are considered anomalous and usually, malicious [10]. For each technique, the approach that is used is determined by how it gathers information to detect malware. Static analysis uses the syntax of the program and attempts to detect malware before its executed. This is an advantage over dynamic analysis as malware can be detected without being executed. In dynamic analysis, malware is detected during execution of the process using runtime information. A hybrid technique combines both static and dynamic approaches. For this project, in order to determine whether the generated malware variant is considerably different from the original malware, a signature-based detection technique using a static approach will be used where the TLSH is the signature.

Recent literature focuses on mobile malware detection as mobile devices are increasingly being used for day-to-day tasks and therefore being a popular target platform. Stowaway [27], ANASTASIA [28]

and DREBIN [29] are three techniques that statically analyse the behaviour of Android applications. Stowaway analyses API calls to detect overprivileged applications whereas ANASTASIA and DREBIN extract features from the tested application and compare whether they're indicative of malware. STREAM [30] is a technique which dynamic analyses application by detecting system calls, network traffic and system component usage e.g. RAM and CPU, to detect anomalies.

2.2 Avoiding Detection

Code obfuscation is the practice of making code hard to understand by applying transformations which change the physical appearance of the code while preserving the black-box specification of the program [17]. Code obfuscation can be used for good to protect intellectual property however, it can also be applied to malicious code to change its signature and avoid detection by signature-based detection mechanisms. One approach is to use encryption. In this approach, the encrypted malware consists of the decryptor and the encrypted main body [16]. However, the issue with this approach is that the decryptor remains the same from one generation to the next making it possible to detect based on the unmodified decryptor. The issues associated with the encryption approach have been addressed through oligomorphic and polymorphic malware by allowing the decryptor to mutate. Oligomorphic malware can generate at most a few hundred different decryptors whereas polymorphic malware can create countless distinct decryptors by using obfuscation techniques [16]. Polymorphic malware is more robust to detection over oligomorphic as it's able to create a unique decryptor for each generation, however, as the malware body remains the same in each generation after it's been decrypted and loaded into memory, signature-based detection can be applied [16]. Metamorphic malware uses obfuscation techniques to mutate its body for each generation and address the limitation of polymorphic malware. Although the code looks different, its functionality remains unchanged.

Various obfuscation techniques can be applied to both polymorphic and metamorphic malware to avoid detection and are described below.

Technique	Description
Dead-Code Insertion	Adds operations that effectively do nothing e.g. NOP or increment a counter not used etc.
Register Reassignment	Changing the registers used by live variables [17]
Function Reordering	Change the order of the functions defined within the code
Instruction Substitution	Replace instructions with other that logically do the same thing
Code Transposition	Reorder instructions without having an impact on the logic
Code Integration	Combining malware into a target/unrelated program [17]

Table 1: Obfuscation Techniques

These techniques have proven effective in breaking similarity hashing algorithms such as Ssdeep and Sdhash however, TLSH appears more robust. In fact, the effect of obfuscation techniques can vary significantly from one algorithm to another [1]. Further investigation will be done to determine whether they're suitable to evade TLSH.

2.2.1 Examples of Obfuscation

The following modification examples are related to obfuscation techniques described in Table 1 and are typical changes performed by polymorphic and metamorphic malware. They have been applied to both Ssdeep and TLSH with their effectiveness measured [1]. Although other examples for images, text and HTML are mentioned, the ones that apply to source code are most suitable for this project. In fact, the methods applied to images are significantly different to the others which include changes to height, width, font-size, rotations and background colour.

Table 2 summarises the experiments performed by Trend Micro [1] in measuring the effectiveness of each modification against both TLSH and Ssdeep where one is most effective.

Modification	TLSH	Ssdeep
New String Variables	1	1
OR-Reordering	2	1
If-Then-Else	3	2
Add NOPs	3	3
AND-Reordering	3	5
Split Strings	3	6
New Integer Variables	4	4

Table 2: *Obfuscation Effectiveness*

2.3 Similarity Algorithms

Traditionally, cryptographic hashing algorithms such as MD5 and SHA-1 have been used in computer forensics due to a property that if a single bit of the input is changed, the output hash will be totally different [3]. This is useful in forensics as unknown files can be identified by comparing them with the hashes of a set of known files. This reduces the amount of work required by the examiner as unknown files don't need to be further investigated if their hash matches the hash of a known file [3]. The issue with cryptographic hashing algorithms is that they can only give a yes/no answer as to whether files are identical and therefore can't be used for file similarity testing as a single change will produce a totally different hash. An alternative to cryptographic hashing is similarity hashing which attempt to solve the nearest neighbour problem. There are different schemes for implementing a similarity algorithm which include feature extraction, Locality Sensitive Hashing (LSH) and Context Triggered Piecewise Hashing (CTPH).

Two similarity algorithms which have been researched are Ssdeep [18] and TLSH [2]. Both algorithms compute the distance between two hashes to determine how similar those files are to each other. Ssdeep uses the CTPH approach and is based off spamsum which compared whether two emails were similar [13]. TLSH was introduced by Trend Micro as a LSH and will be the focus of this project due to research suggesting it's more robust [2].The technical details for both Ssdeep and TLSH can be found under Appendix A and B respectively.

3. Problem Identification

Similarity digest schemes have the property that small changes in the file being hashed result in a small change to the hash [1]. Three schemes often compared together are Ssdeep [18], Sdhash [31] and TLSH [2] with Ssdeep being the de facto standard. In fact, Ssdeep is used by security companies such as Virus Total [5] for malware analysis. However, previous literature suggests that Sdhash outperforms Ssdeep in many instances [12] with TLSH outperforming both in terms of robustness to randomisation attacks [1]. Given that there are known ways of exploiting both Ssdeep and Sdhash, security companies are likely to change to a more robust hashing algorithm such as TLSH.

As TLSH is considered to be more robust over other similarity digest schemes, the contributions of this project are to validate the robustness of TLSH and determine whether the digest scheme can be broken. The problem will be divided into two parts: (1) malware detection and (2) malware variant generation with the overall goal to develop a web-based application that achieves the following:

- Allows users to upload any malware source into the application
- Ssdeep, Sdhash and TLSH digests are computed against the uploaded malware source
- Uses machine learning to generate a malware variant that provides the same functionality but is syntactically different
- Malware variants are compared with the source to determine whether the digest has been broken
- Provides analytics on the number of generations required to 'break' the hash

In TLSH, the hash is 'broken' when the distance between the two hashes exceeds the user-defined threshold. For Ssdeep and Sdhash, it's when the distance is 0. The last point is important as it gives a robustness comparison against the digest schemes.

Determining the robustness of TLSH is important as it provides an indication of how good it is against randomisation attacks which impact whether it becomes widely adopted. If any flaws in the digests are identified, future work could attempt to address these through static or dynamic anomaly-based detection algorithms. Alternatively, a new digest could be developed which improves on TLSH.

4. Methodology

This project will be delivered in two phases: 1) malware detection and 2) malware variant generation. In the first phase, the web application will be developed allowing users to upload any malware and it will compute its TLSH. In the second phase, a machine learning algorithm will be developed and integrated into the application allowing malware variants to be generated. In the following section, the decisions made and any resources required have been stated.

4.1 Project Choices and Resources Required

For this project, decisions need to be made on the following criteria:

I. Application type
II. Infrastructure
III. Programming language
IV. TLSH source language

The application type defines the type of application that will be developed with choices including Windows, UNIX-based, mobile and web. As we would like the application to be accessible from any device, a web-based application is an ideal choice. Web allows anyone with a browser to access it, and when deployed in a production environment, allows crowdsourcing to build up the signature database (for prevention).

Infrastructure relates to the environment in which the web app will be deployed on. Local and cloud are two different infrastructure approaches. For local, the web service is running on the same machine as the user of the system. For cloud, the web service is running on a separate machine hosted on a cloud provider such as Amazon AWS or Microsoft Azure. Cloud infrastructure was chosen as it can scale at ease and has high availability, both of which are important for web applications. As the web service will be exposed publically, anyone can access it. AWS was chosen over other cloud providers due to familiarity with the platform.

There are many different programming languages out there that can be used to develop a web app. Two stacks considered were Python using Flask and MEAN, both of which allow web apps to be developed quickly. The decision was therefore based on the machine learning capability of the language. As a result, Python was chosen due to extensive machine learning libraries being available.

TLSH has been implemented in C++, Golang, Java, JavaScript, Python and Ruby with all languages except Python providing a full implementation. For the Python port, it's essentially a wrapper calling C++ code. Python was strongly considered as it's the same language used to develop the web app however, I decided against it as TLSH doesn't provide a full Python implementation. Instead, JavaScript was chosen as it's a web language and therefore minimal changes would be required to integrate it into the application. The downside is that the TLSH processing will need to be done on the client-side.

As the application will be hosted on cloud infrastructure, an AWS account is required that has permission to provision the following services:

AWS Service	Description
EC2	VM required to investigate malware binaries
S3	Cloud storage that will host the malware variants generated
Lambda	Required by Zappa to create a serverless web app
API Gateway	Required by Zappa to create a serverless web app
RDS	Database service that will store name/hash pairs of computed files/binaries

Table 3: *AWS Services*

4.2 Malware Detection

In the detection phase, a web-based application will be developed that allows users to upload a file into the application which will then compute the TLSH. The application will compare the uploaded file with all others files hosted on the server and report any similarities i.e. within a threshold distance. The application will be hosted on AWS and accessible through a web browser. The diagram below describes a typical interaction with the application

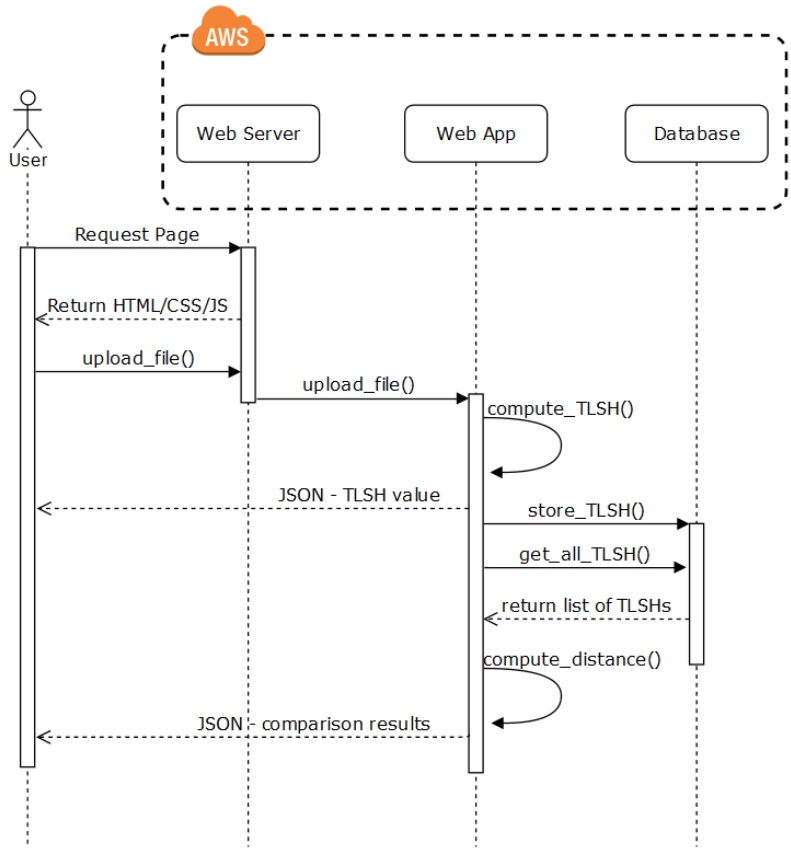

Figure 3: *Phase 1 Sequence Diagram*

The following languages, services and frameworks will be used:

Languages	Python 3 (Back-end), JavaScript (TLSH source)
Frameworks	Bootstrap (Front-end), Flask (Web), Zappa [18]
AWS Services	EC2, S3, Lambda, API Gateway, Relational Database Service (PostgreSQL)

Table 4: *Infrastructure & Software Stack*

Python using the Flask web framework has been chosen over the MEAN stack to create a web-based application as machine learning libraries are well supported and readily available in Python. Zappa will be used to create a server-less Python web application by leveraging both AWS Lambda and AWS API Gateway. PostgreSQL will be used as the database engine to store the filename and TLSH mappings. A file hosted on S3 could have been used instead of a database however, I wanted something that could scale. The collection of malware obtained from TheZoo [11] will be hosted on S3. Finally, EC2 will be used as a VM to investigate both the malware source and binaries.

4.3 Malware Variant Generation

In the generation phase, a machine learning algorithm will be developed. It will be written in Python due to the extensive machine learning libraries being available for the language. The algorithm will

be implemented as a Learning Classifier System (LCS) which is a rule-based method of machine learning. This type of machine learning is classified as reinforced learning and is based on rewards. Rewards will be distributed by the fitness function and proportional to the effectiveness of the action. A reinforced learning paradigm learns about good actions through trial and error. This differs from the supervised learning paradigm which attempts to solve classification problems by training the algorithm using a pre-labelled training dataset and then use a different unlabelled dataset to predict the labels. As this is not a classification problem, supervised learning is not suitable.

Basically, a number of obfuscation modifications will be templated and available to the algorithm. The decision of which modification to apply in the current generation will be determined by roulette wheel selection. In early generations, the algorithm will be in exploration as the fitness of each modification will be similar and therefore selection will effectively be random. However, as the generations increase, modifications with a higher fitness will more likely be selected. The fitness of a modification is updated after its run and its effectiveness has been measured by the fitness function. The algorithm will continue to execute until the threshold distance between the two hashes has been met or until N generations have been run, both of which are user configurable. A parser will also be developed as it's required to parse the source file and identify the appropriate placement for each type of modification. The following diagram describes the workflow of the algorithm.

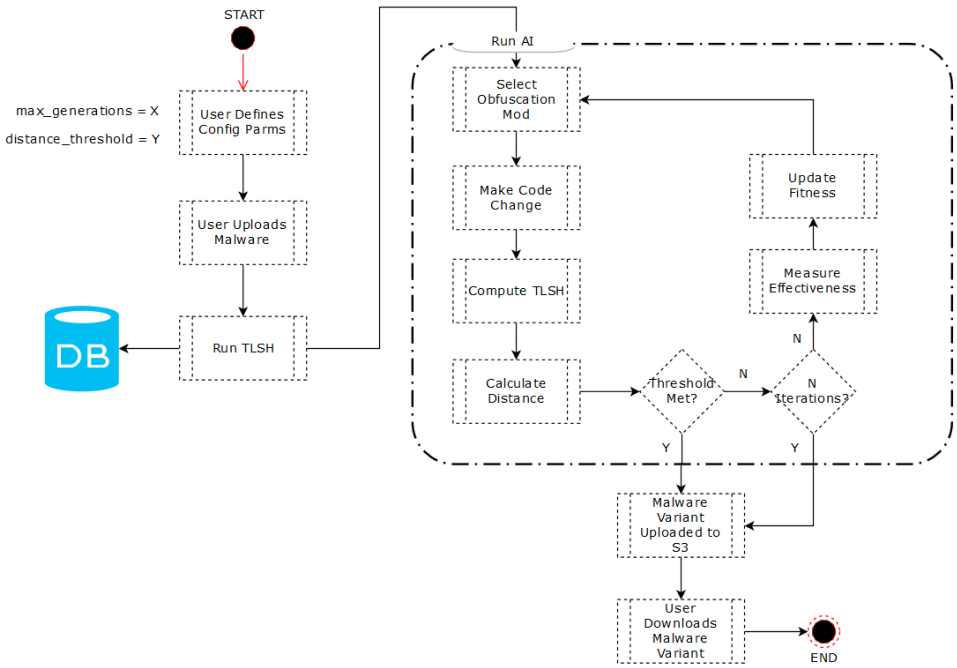

Figure 4: *Malware Variant Generation Workflow*

5. Timeline

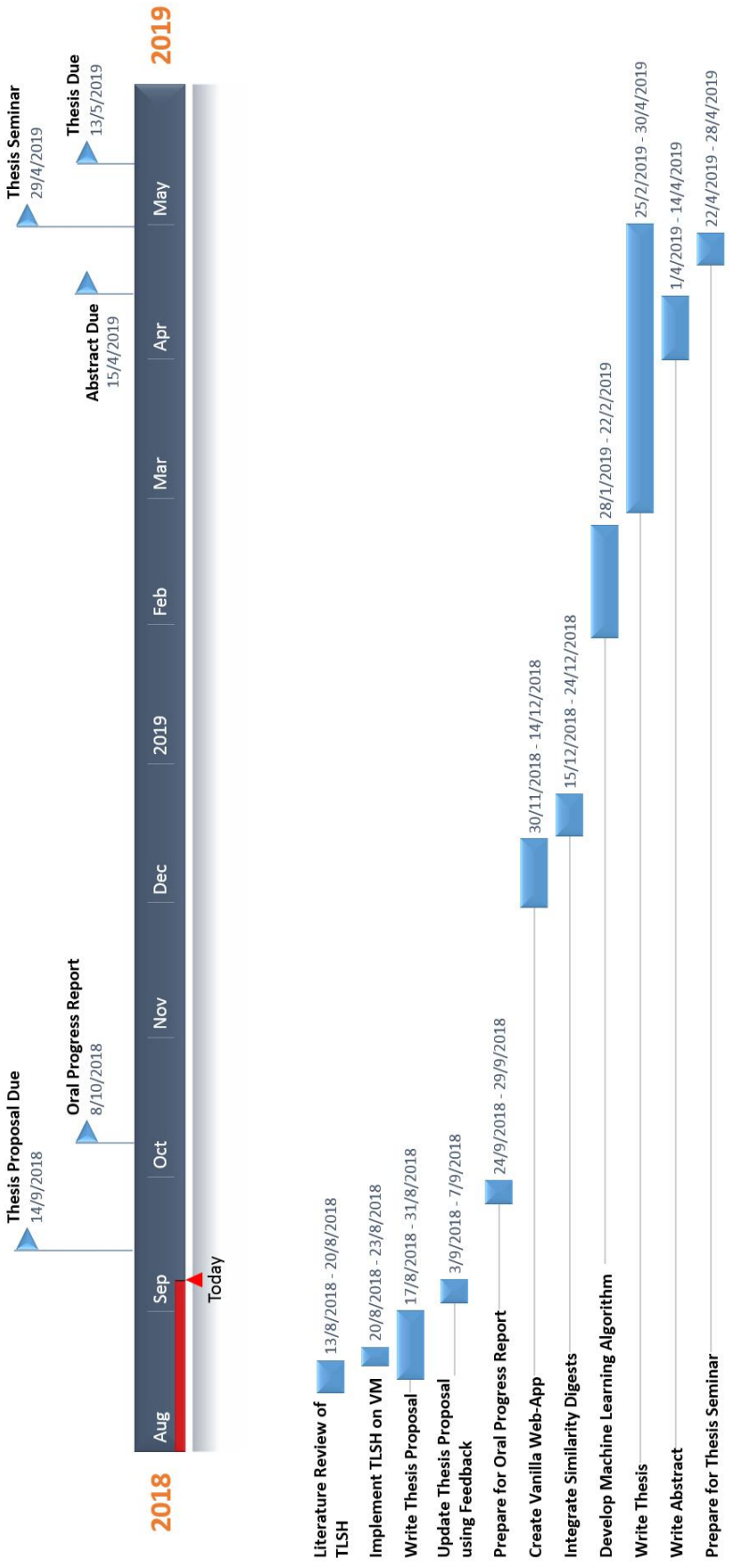

Figure 5: *Gantt Chart*

6. References

[1] Oliver, J, Forman, S & Cheng, C 2014, 'Using Randomization to Attack Similarity Digests', in, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 199-210.

[2] Oliver, J, Cheng, C & Chen, Y 2013, 'TLSH--A Locality Sensitive Hash', in Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth, IEEE, pp. 7-13.

[3] Kornblum, J 2006, 'Identifying almost identical files using context triggered piecewise hashing', Digital Investigation, vol. 3, pp. 91-97.

[4] Breitinger, F, Baier, H & Beckingham, J 2012, 'Security and implementation analysis of the similarity digest sdhash', in First international baltic conference on network security & forensics (nesefo).

[5] Virus Total. Available from: <https://www.virustotal.com/>. [31 August 2018]

[6] Pagani, F, Dell'Amico, M & Balzarotti, D 2018, 'Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis', in Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, ACM, pp. 354-365.

[7] Iovan, S & Iovan, A-A 2016, 'From cyber threats to cyber-crime', Bucureşti: Editura Universitară, Journal of Information Systems & Operations Management, pp. 425-434.

[8] Symantec 2018, Internet Security Threat Report.

[9] Accenture 2017, Cost of Cyber Crime Study.

[10] Idika, N & Mathur, AP 2007, 'A survey of malware detection techniques', Purdue University, vol. 48.

[11] TheZoo. Available from: <https://github.com/ytisf/theZoo>. [31 August 2018]

[12] Roussev, V 2011, 'An evaluation of forensic similarity hashes', digital investigation, vol. 8, pp. S34-S41.

[13] Tridgell, A, Spamsum. Available from: <https://www.samba.org/ftp/unpacked/junkcode/spamsum/README>. [31 August 2018]

[14] Sharp, R 2009, An Introduction to Malware, Spring.

[15] You, I & Yim, K 2010, 'Malware obfuscation techniques: A brief survey', in Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on, IEEE, pp. 297-300.

[16] Balakrishnan, A & Schulze, C 2005, 'Code obfuscation literature survey', CS701 Construction of compilers, vol. 19.

[17] Zappa. Available from: < https://github.com/Miserlou/Zappa/>. [31 August 2018]

[18] Ssdeep. Available from: <https://github.com/ssdeep-project/ssdeep>. [31 August 2018]

[19] Damiani, E, di Vimercati, SDC, Paraboschi, S & Samarati, P 2004, 'An Open Digest-based Technique for Spam Detection', ISCA PDCS, vol. 2004, pp. 559-564.

[20] Pearson, PK 1990, 'Fast hashing of variable-length text strings', Communications of the ACM, vol. 33, no. 6, pp. 677-680.

[21] Chen, L & Wang, G 2008, 'An efficient piecewise hashing method for computer forensics', in Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on, IEEE, pp. 635-638.

[22] Christodorescu, M, Jha, S, Seshia, SA, Song, D & Bryant, RE 2005, 'Semantics-aware malware detection', in Security and Privacy, 2005 IEEE Symposium on, IEEE, pp. 32-46.

[23] Rabek, JC, Khazan, RI, Lewandowski, SM & Cunningham, RK 2003, 'Detection of injected, dynamically generated, and obfuscated malicious code', in Proceedings of the 2003 ACM workshop on Rapid malcode, ACM, pp. 76-82.

[24] Roussev, V 2010, 'Data fingerprinting with similarity digests', in IFIP International Conference on Digital Forensics, Springer, pp. 207-226.

[25] Vasudevan, A & Yerraballi, R 2006, 'Spike: engineering malware analysis tools using unobtrusive binary-instrumentation', in Proceedings of the 29th Australasian Computer Science Conference-Volume 48, Australian Computer Society, Inc., pp. 311-320.

[26] Roussev, V, Richard III, G & Marziale, L 2008, 'Classprints: Class-aware similarity hashes', Advances in Digital Forensics.

[27] Felt, A, Chin, E, Hanna, S, Song, D & Wagner, D 2011, 'Android permissions demystified' In Proceedings of the 18th ACM conference on Computer and communications security, ACM, pp. 627-638

[28] Fereidooni, H, Conti, M, Yao, D & Sperduti, A 2016, 'ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications' In New Technologies, Mobility and Security (NTMS), 2016 8th IFIP International Conference, IEEE, pp. 1-5

[29] Arp, D, Spreitzenbarth, M, Hubner, M, Gascon, H, Rieck, K & Siemens, C.E.R.T 2014 'DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket' In Ndss, vol. 14, pp. 23-26

[30] Amos, B, Turner, H & White, J 2013 'Applying machine learning classifiers to dynamic android malware detection at scale' In Wireless communications and mobile computing conference (iwcmc), 2013 9th international, IEEE, pp. 1666-1671

[31] Sdhash. Available from: <https://github.com/sdhash/sdhash>. [07 September 2018]

[32] Feizollah, A, Anuar, NB, Salleh, R. and Wahab, AWA 2015. 'A review on feature selection in mobile malware detection', Digital Investigation, vol. 13, pp.22-37

7. Appendices

7.1 Appendix A – Ssdeep Technical Details

CTPH algorithm used within Ssdeep consists of two separate hashing algorithms. One is the piecewise hashing algorithm and the other is the rolling hash algorithm. The piecewise algorithm uses an arbitrary hashing algorithm e.g. Fowler/Noll/Vo (FNV) to create many checksums for the file. This works by having a hash generated for each fixed-sized segments of the file. The rolling hash algorithm works by maintaining a state based solely on the last few bytes from the input. Each byte is added to the state, processed and then removed after a set number of other bytes have been processed. When the output from the rolling hash algorithm matches a specific trigger value, the current piecewise hash value is added to the CTPH signature and the piecewise hash value is reset. As each hash value added to the signature only depends on part of the input, changes made to the file are localised to that portion of the hash with the rest of the signature staying the same. Two Ssdeep/spamsum signatures are compared by applying the weighted edit distance between them. The edit distance is defined as the minimum number of point mutations required to change one hash into the other hash where a point mutation is either changing, inserting or deleting a character. The edit distance, $e(s_1, s_2)$, is used in the formula below to calculate the match score which is a measure of how many of the bits of the two signatures are identical and in the same order. The match score ranges from 0-100 where the higher the score, the more likely the files are similar.

$$M = 100 - \left(\frac{100Se(s_1,s_2)}{64(l_1+l_2)}\right) \quad (1)$$

7.2 Appendix B – TLSH Technical Details

A TLSH digest is represented as either a 70 or 134-character hexadecimal string where the length depends on the bucket size parameter configured. A bucket size of 128 generates a 70-character TLSH digest whereas a bucket size of 256 generates a 134-character digest. A TLSH digest is constructed from the concatenation of a 3-byte header and either 32 or 64-byte body.

The first step in constructing the body portion of the digest is to process the input data as a byte string using a sliding window of size 5 (i.e. 5 bytes) to populate an array of bucket counts. A sliding window size of 5 was chosen due to it being used in the Nilsimsa hash which proved to be effective. Each sliding window produces 6 triplets i.e. 6 different 3 bytes combinations, called trigrams. Each trigram is input into a hashing function which maps the trigram to a bucket. Once the bucket has been identified, it's incremented by one. The hashing function to map trigrams to buckets is known as the Pearson hash. Once an array of bucket counts has been determined, the Q1, Q2 and Q3 quartiles can be calculated. Q1 corresponds to the lower-quartile i.e. where 75% of bucket counts are greater than it. Similarly, Q2 corresponds to the median (50%) and Q3 corresponds to the upper-quartile (25%). Once the quartiles values have been calculated, the digest can be constructed by loop through each bucket and based on its count, determines the two binary digits associated with that bucket with the logic shown in Figure 2. The header portion of the digest is calculated from a checksum (1-byte), L value (1-byte) and Q1 and Q2 ratios (one hex-digit each).

```
For bi = 0 to 127 {
    if bucket[bi] <= q1      Emit(00)
    else if bucket[bi] <= q2 Emit(01)
    else if bucket[bi] <= q3 Emit(10)
    else                     Emit(11)
}
```

Figure 6: *Bucket Count to Digest Mapping*

To compute how similar two hashes are to each other, TLSH uses a distance score. A score of 0 indicates that two files are identical. The higher the score, the more difference between the two. A threshold distance score is chosen to decide when two files are considered non-similar. There are two functions used to calculate distance, one processes the header portion and the other processes the body. The function that processes the body calculates an approximation to the hamming distance between the two hash bodies.