Global Information Assurance Certification Paper

Leveraging the PE Rich Header for Static Malware Detection and Linking

GIAC (GREM) Gold Certification

Author: Maksim Dubyk, modubyk@gmail.com
Advisor: Hamed Khiabani, Ph.D.
Accepted: June 25th, 2019

Abstract

An ever-increasing number of malware samples are identified and assessed daily. Malware researchers have the difficult mission of classifying and grouping these malware specimens. Defenders must not only judge if a file is malicious or benign, but also determine how a file may relate to other groupings of known samples. The static comparison of file and file-format based properties are often utilized to execute this objective at scale. This paper builds upon previously identified Windows' portable executable (PE) static comparison techniques through the exploration of the undocumented PE Rich header. The Rich header is a PE section that serves as a fingerprint of a Windows' executable's build environment. This under-utilized wealth of information can provide value to defenders in support of classifying and associating PE-based malware. This paper explores how to extract the details hidden in the Rich header and how they might be exploited to link and classify malware samples. In addition, this paper evaluates how the static linking of PE rich header sections compare to traditional static PE linking techniques.

1. Introduction

The cyber threat landscape is comprised of a diverse set of adversaries, each with varied objectives and capabilities. While the specifics of an adversary's tactics, techniques, and procedures will differ greatly, the reliance on a set of malicious tools used to achieve a desired aim is commonplace. According to Carbon Black, in 2016, the percentage of non-malware based cyber attacks did not rise above 14% for any single month (Carbon Black, 2016). While living off the land techniques are rising, malware is still heavily relied upon to achieve the actions and objectives of cyber intrusions. That same year, AVTest reported that Windows portable executables comprised 37.90% of all malware file types; the highest of any single file format (AVTest, 2016). Defenders are faced with the steep challenge of not only classifying these samples as malicious, but also determining how malware samples relate to one another.

Malware analysts often attempt to level the playing field by adopting streamlined static malware analysis techniques. A wide range of analysis techniques have exploited static properties of portable executable files for malware classification pursuits (Hassen, Carvalho, & Chan, 2017; Kozachok & Kozachok, 2017). Static malware analysis techniques provide opportunities for scalable automation that enable defenders to drastically increase detection and analysis capacity to counter the high volume of malware samples. In 2015, Lockheed Martin open-sourced LaikaBOSS; a file-centric malware analysis and intrusion detection system that demonstrated the utility found in adopting scalable frameworks for statically analyzing malware (Arnao, Smutz, Zollman, Richardson, & Hutchins, 2015). Since then, a number of other similarly purposed platforms have arisen; namely File Scanning Framework (Emerson, 2019) and Strelka (Target, 2019). Such tools have originated as a result of the challenges in performing static malware triage and analysis at scale in large computer networks.

This work explores the undocumented PE Rich header section and how it can be leveraged for the static analysis and comparison of PE-based malware samples. The Rich header is seldom explored, and most frequently through unstructured, one off, blogging. This PE header section is a product of the Microsoft Linker and is observed in 71% of

malicious PE samples (Webster et al., 2017). The information stored in this header section provides a window into the environment where an executable was built (Pistelli, 2010). This work first explores what information is contained within the Rich header, how it can be extracted, and what sense can be made from the raw bytes. From this information, two checksum techniques are investigated that utilize the Rich header for similarity comparisons between PE samples. These techniques were then evaluated on a sample set of PE malware specimens using common static cryptographic hashing techniques as benchmarks for which to compare to. The results indicate the Rich header provides a robust series of data points that can be leveraged for the static detection and linking of PE samples.

2. Breaking Down the Rich Header

The Rich header is an undocumented section in the portable executable header that exists as a result of the compilation and build process of Microsoft-produced executables. The Rich section is comprised of an array of metadata for each step in the build process of a portable executable (Webster et al., 2017). In order to create a PE two high-level exercises typically take place. First, the compiler will translate high-level programming language code to machine code. The result of the compiler is low-level code that is able to be executed by the target computer's architecture. Then, the resulting machine code objects are typically combined into a single executable through a linking process (Dean, 1997). Within Microsoft products, the process of combining multiple compiled objects into a single executable is responsible for the Rich header's creation. Portable executables built with utilities other than the Microsoft linker (link.exe) will not have a Rich header (.Net, GCC, etc.). This header section exists in plain sight and directly follows the MZ file header and DOS Stub. It has received its name due to its defining marker, the presence of the hexadecimal bytes 0x52696368 or the ascii text "Rich." The Rich section begins at offset 0x80 and extends until the "Rich" string. This byte sequence acts as an anchor and marks the end of the section.

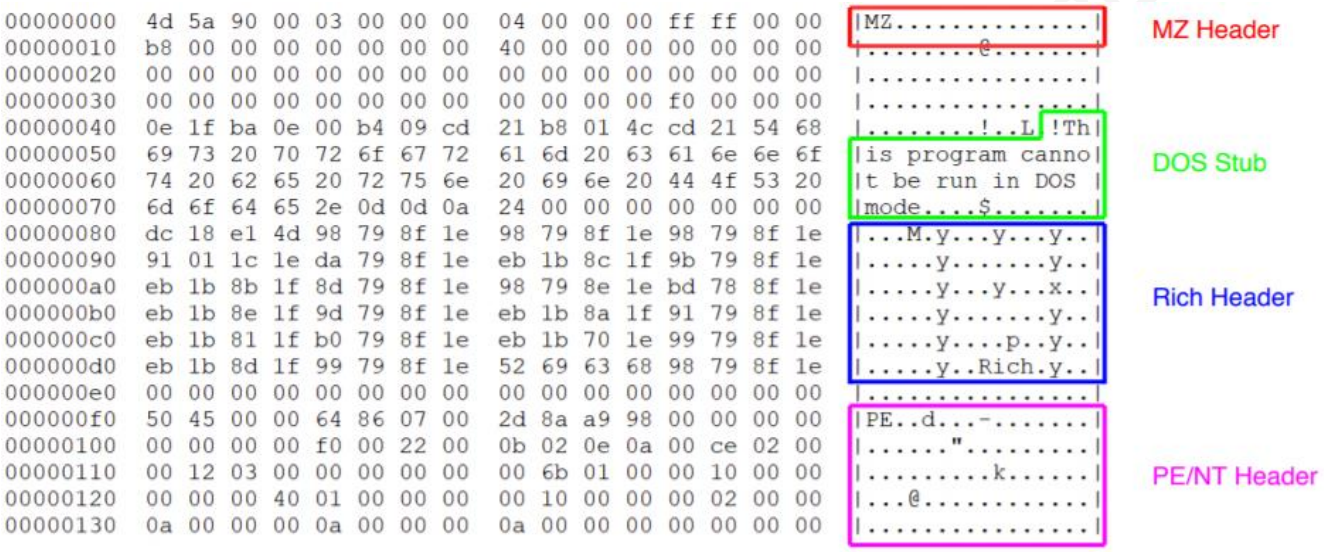

Figure 1. Cmd.exe PE header.

The content of the header itself is encrypted and obfuscated. Located directly after "Rich" is a 32-bit checksum and decryption key. This byte sequence confirms the validity of the Rich header and provides a means to decrypt the section's contents. In cmd.exe (e08fe2de3ddd22123247d49a11b4f53d), the checksum and decryption key are 0x98798f1e (Figure 2). Following this key exists 16 bytes of padding before the PE header begins at offset 0xf0.

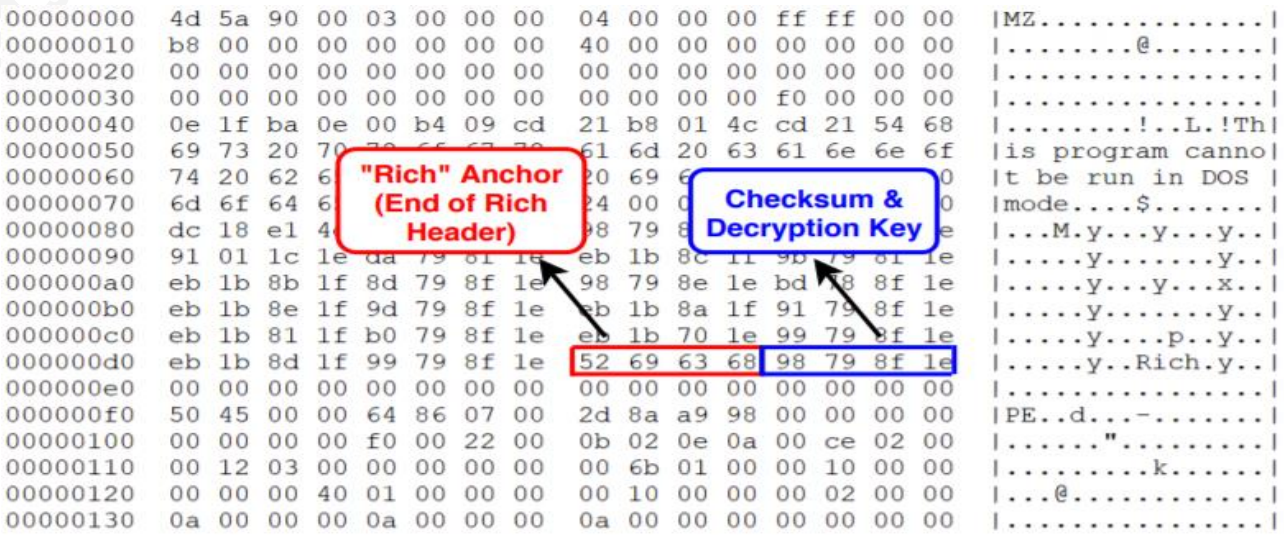

Figure 2. Cmd.exe Rich section footer.

This decryption key is computed by adding the values of two different checksum algorithms. The first checksum is derived from the bytes that constitute the DOS header where e_lfanew or the PE header offset field is zeroed out. The second checksum is the combined value of each array entry in the body of the Rich header (Case studies in Rich Header analysis and hunting, 2018). The two checksums are summed and then masked with 0xFFFFFFFF to result in the final checksum observed after the "Rich" ascii string. During the creation of the Rich header this checksum is derived and then XORed with each 4-byte chunk, providing an elementary means of encrypting the section's contents (Case studies in Rich Header analysis and hunting, 2018).

Fortunately, it is trivial to extract the checksum and decrypt the rest of the Rich header. In order to derive the decrypted contents of the Rich header the key identified after the "Rich" string is XORed with each proceeding DWORD until the ascii string "DanS" is observed. Just as "Rich" identifies the end of the Rich header, "DanS" or 0x44616e53 is the marker of its beginning. Dan Spalding and Richard Shupak were Microsoft employees who worked on the Visual C++ linker and library code base in the late 1990s and are believed to be the source of the section's ascii header and footer markers ("Undocumented Microsoft "Rich" Header," 2017). Figure 3 shows the decrypted Rich header in cmd.exe. The existence of the ascii string "DanS" at offset 0x80 confirms successful decoding of the Rich header.

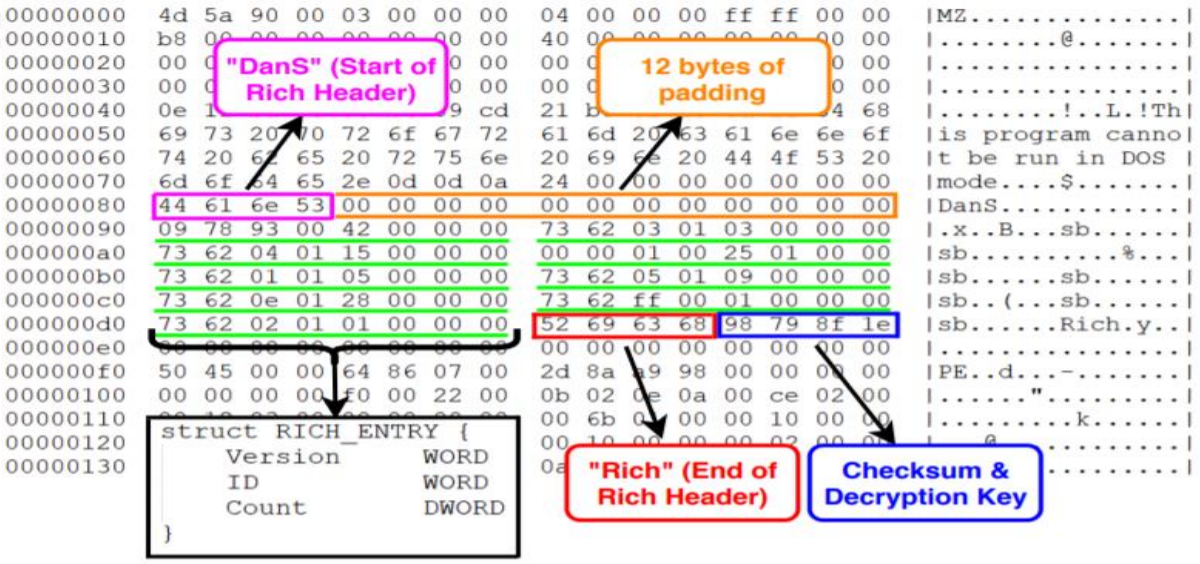

Figure 3. Decrypted cmd.exe Rich header.

The body of the Rich header is comprised of single array that stores metadata on each step of the build chain during the linking of objects into a single executable. The array is variable in length and made up of an 8-byte structure. The structure contains the product identification (pID), product version (pV), and a count (pC) of each time the product was used; the total number of source files compiled (Webster et al., 2017). The first two bytes account for the pV, the second two for the pID, and the final four for the pC (Figure 3). Each element in the array is representative of a step during the executable build chain. The values exist in little-endian byte order and must be reversed to big-endian in order to be understood correctly. Due to the nature of the Rich header being an unofficial and undocumented PE section, there is no official comprehensive mapping of pID to Microsoft products. However, several researchers have partially mapped out common products (Dishather, 2019). Figure 4 presents the extracted array from the Rich header of cmd.exe.

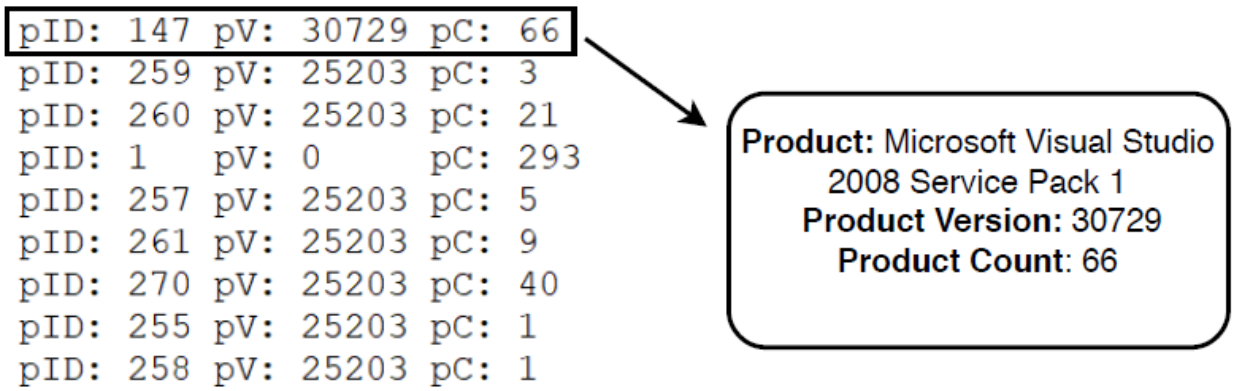

Figure 4. Rich header array extracted from cmd.exe.

Given the simplistic obfuscation method employed to conceal the Rich header, it is trivial to programmatically extract the section's body. Conveniently, the pefile Python module supports extraction and XOR decoding of the Rich header. Appendix A contains Python code capable of extracting the complete content stored in a PE's Rich header and outputting the values in JSON. For example, cmd.exe produces the following results.

```
analyst@LAB:~/samples$ python parseRich.py e08fe2de3ddd22123247d49a11b4f53d |
jq '."Rich Header"'
[
  {
    "Product_Count": 66,
    "Product_Version": 30729,
    "Product_ID": 147
  },
  {
    "Product_Count": 3,
    "Product_Version": 25203,
    "Product_ID": 259
  },
  {
    "Product_Count": 21,
    "Product_Version": 25203,
    "Product_ID": 260
  },
  ...
  ...
  ...
  {
    "Product_Count": 5,
    "Product_Version": 25203,
    "Product_ID": 257
  }
]
```

3. Assessing the Rich Header's Detection and Linking Value

3.1. Capturing the Rich Header with Rich and RichPV Hashes

In order to evaluate the Rich header's utility in support of malware triage and classification pursuits, two hashing techniques were utilized. These techniques allow for the static comparison and linking of portable executables based on equivalent hash values. Both methods fingerprint the Rich header's content by performing cryptographic hash computations on different segments of the Rich section. While this study uses the MD5 algorithm, any other cryptographic hashing function can be used in its place with the same underlying methodology. The first hashing technique will use the entire decoded content of the Rich header as an input. This paper will refer to this computation as the Rich Hash. The decrypted values were chosen to ensure the Rich Hash will match

the content of Rich header in other samples should the XOR key change. The Rich Hash will be calculated by the algorithm presented in Figure 5. First, beginning at offset 0x80, the Rich header section will be extracted until the start of the PE header. Next, the XOR key will be located by identifying the 4 bytes directly following the anchor to the Rich section (0x52696368 or "Rich" in ascii). The body of the section is then decoded by taking the identified key and performing XOR operations with each proceeding DWORD until 0x44616e53, or "DanS" is observed. Lastly, this newly decoded content is the input for a md5sum computation.

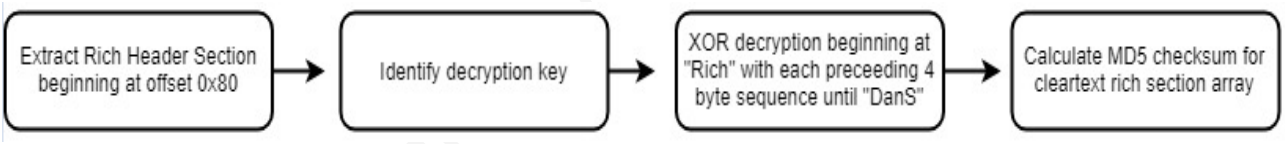

Figure 5. Rich hash algorithm

The second technique attempts to create a fingerprint that profiles the Rich header's content while also having greater resistance to changes in an executable's build environment compared to the Rich hash. This is accomplished by excluding pC, the most volatile of the three fields in the Rich header body. For each pID and pV pair, pC measures the number of source files referenced by the PE. As a result, the pC field has the potential to change across different PEs as the number of source files increase and decrease even if the products and their versions remain constant. Therefore, this technique will result in a checksum that will not be impacted by any changes to pC fields. This second hash is computed by the same method as the Rich hash with a modification to exclude the last four bytes of each element in the array that correspond to pC. This second technique will be referred to as the RichPV hash.

In addition to extracting the contents of the Rich header, the Python code included in Appendix A computes a PE's Rich and RichPV hashes. For example, the cmd.exe sample referenced several times (e08fe2de3ddd22123247d49a11b4f53d) has the following Rich and RichPV hashes.

```
analyst@LAB:~/samples$ python parseRich.py e08fe2de3ddd22123247d49a11b4f53d |
jq '."Rich Hashes"'
{
  "Rich Hash": "3d75441fa2dca655f337ee83519d34dc",
  "Rich PV": "dc083eb68efdb8840ddfaee612a2755d"
}
```

3.2. Rich and RichPV Detection with Yara Rules

While Rich and RichPV hashes can be computed and compared amongst PE samples by a variety of mechanisms, Yara rules provide one method of identifying specific Rich header content. Given the widespread use of Yara in many static and dynamic malware analysis and detection platforms, it is a formidable means of exploiting the information stored in the PE Rich header for sample identification. The current Yara build version 3.10.0 includes several modules that extend its standard rule capabilities. The PE module exposes properties specific to the portable executable file format and includes several Rich header content and metadata attributes. According to the PE module documentation those properties specific to the Rich header are as follows ("PE module — yara 3.10.0 documentation," 2019).

- offset
- length
- key
- raw_data
- clear_data
- version(version, [toolid])
- toolid(toolid, [version])

These exposed features can be leveraged in order to create Yara rules that identify PEs based on Rich and RichPV matches. A Yara rule can be crafted to identify a PE sample with a specific Rich hash by combining the PE module's Rich Signature clear_data attribute and the Hashlib module's capacity to perform hash computations. To do so, the output of the Rich Signature clear_data attribute can be directed as the input for

the Hashlib module. The following Yara rule will identify any PEs that have a Rich hash equivalent to 3d75441fa2dca655f337ee83519d34dc.

```
import "hash"
import "pe"
rule RichHash_3d75441fa2dca655f337ee83519d34dc
{
    meta:
        description ="Matches a Rich Hash of 3d75441fa2dca655f337ee83519d34dc"
    condition:
        hash.md5(pe.rich_signature.clear_data) ==
"3d75441fa2dca655f337ee83519d34dc"
}
```

Creating a Yara rule to identify a Rich hash is easily supported with the PE and Hashlib modules. However, identifying a RichPV match with Yara requires additional work. The PE module's Rich Signature clear_data and raw_data fields expose the content of the Rich header, but Yara is limited in its capacity to selectively parse the resulting content with the aim of excluding the pC field. As an alternative, the PE module provides two functions that return boolean values if a specific pID and pV pair exist in a single Rich header entry. The functions do not capture the pC field of each Rich header entry, which provides a means of capturing the same data as the RichPV hash; albeit in a different form. However, this prevents a Yara rule from identifying PEs with an exact RichPV hash match. Instead, the Yara rule will check whether each pID and pV exist in the Rich header structure. A precursor to crafting such a rule is identifying the integer values for all pID and pV Rich header entries (these values can be identified with the Python script found in Appendix A). The following Yara rule will match all of the pID and pV values that comprise the RichPV hash value of dc083eb68efdb8840ddfaee612a2755d.

```
import "pe"
rule RichPVHash_8948f5950ed099bf499a6ceb09c30559
{
    meta:
        description="Matches a RichPV Hash of
8948f5950ed099bf499a6ceb09c30559"
    condition:
        pe.rich_signature.toolid(147, 30729) and
        pe.rich_signature.toolid(259, 25203) and
        pe.rich_signature.toolid(260, 25203) and
        pe.rich_signature.toolid(1, 0)       and
```

```
        pe.rich_signature.toolid(257, 25203) and
        pe.rich_signature.toolid(261, 25203) and
        pe.rich_signature.toolid(270, 25203) and
        pe.rich_signature.toolid(255, 25203) and
        pe.rich_signature.toolid(258, 25203)
}
```

Included in Appendix B is Python code that creates each of the two aforementioned Yara rules that identify Rich and RichPV matches in portable executables. The Python program takes a PE as input and uses the yara_tools Python library to dynamically generate Rich and RichPV Yara rules that identify the input sample's Rich header content.

3.3. Malware Samples used in the Evaluation

In this experiment, a total of 350 malware samples were used to evaluate the utility of the Rich header for malware classification. The samples are a combination of 10 different groupings of malware. Each grouping has a total of 35 malware samples that correspond to a known threat actor or malware family. The malware samples were acquired from a combination of public malware repositories and reporting on advanced threat actors. These groups of malware were chosen with the intent of evaluating the Rich header's value in classifying both opportunistic and targeted forms of malware. Furthermore, this paper sought to understand the relationship between the Rich header's classification utility across groups of malware families and threat actors given an adversary can compile different families of malware in the same build environment. See Appendix C for sample MD5 hashes.

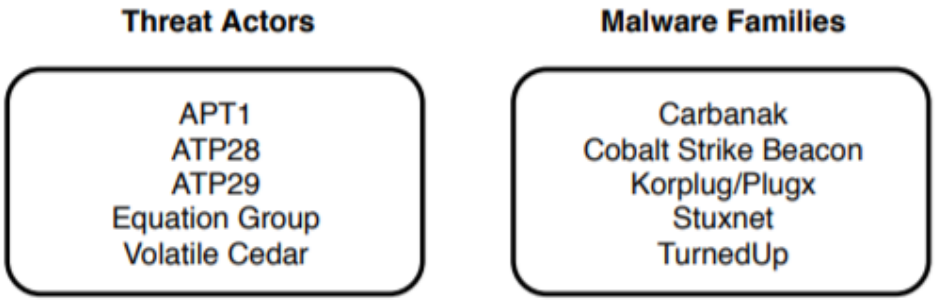

Figure 6. Malware samples used in evaluation grouped by family and threat actor.

3.4. Other Static Techniques for Benchmarks

In the evaluation of the Rich header, three common static comparison techniques were used to provide benchmarks for which to compare the proposed Rich and RichPV hashes. These three techniques include ssdeep, import hash, and import fuzzy hash. Ssdeep is a widely used fuzzy hashing algorithm that provides a means of identifying similar files (Jianguo, Jiuming, Qian, Kunying, & Chao, 2016). It is a context triggered peacewise hashing algorithm and a cyber security industry standard for comparing malware samples. The second static comparison method used in this study is the PE import hash or Imphash. Proposed by Mandiant in 2013, the Imphash is the fingerprint for a PE's import address table (Tracking Malware with Import Hashing, 2014). Since becoming adopted in common practice, the Imphash has become a high fidelity means of identifying similar malware families (Choi et al., 2013). With a sufficiently large import address table, the Imphash can be a unique indicator to cluster and group PE-based malware. Lastly, Import Fuzzy hash or Impfuzzy takes the ssdeep algorithm and applies it to the import address table of a PE (Tomonaga, 2016). This measure allows for the identification of similar import tables even if they are not fully equivalent. Whereas Imphash will only identify a relationship with an equal checksum, Impfuzzy can provide greater flexibility and linking capacity.

3.5. Evaluating the Rich and RichPV Hashes

After acquiring the malware samples, the next step involved the creation of the desired static analysis products. A custom Python script was developed to calculate each of these cryptographic hashes. The script took a PE as input and produced a JSON dictionary with the sample's MD5, ssdeep, impfuzzy, imphash, Rich, and RichPV hashes. The script ingested all malware samples to produce the desired hashes for each portable executable. Next, a Neo4j graph database was utilized to create relationships amongst the samples and evaluate each static technique's linking strength. Neo4j was chosen due to its free community licensing model and native strength in performing link analysis metrics.

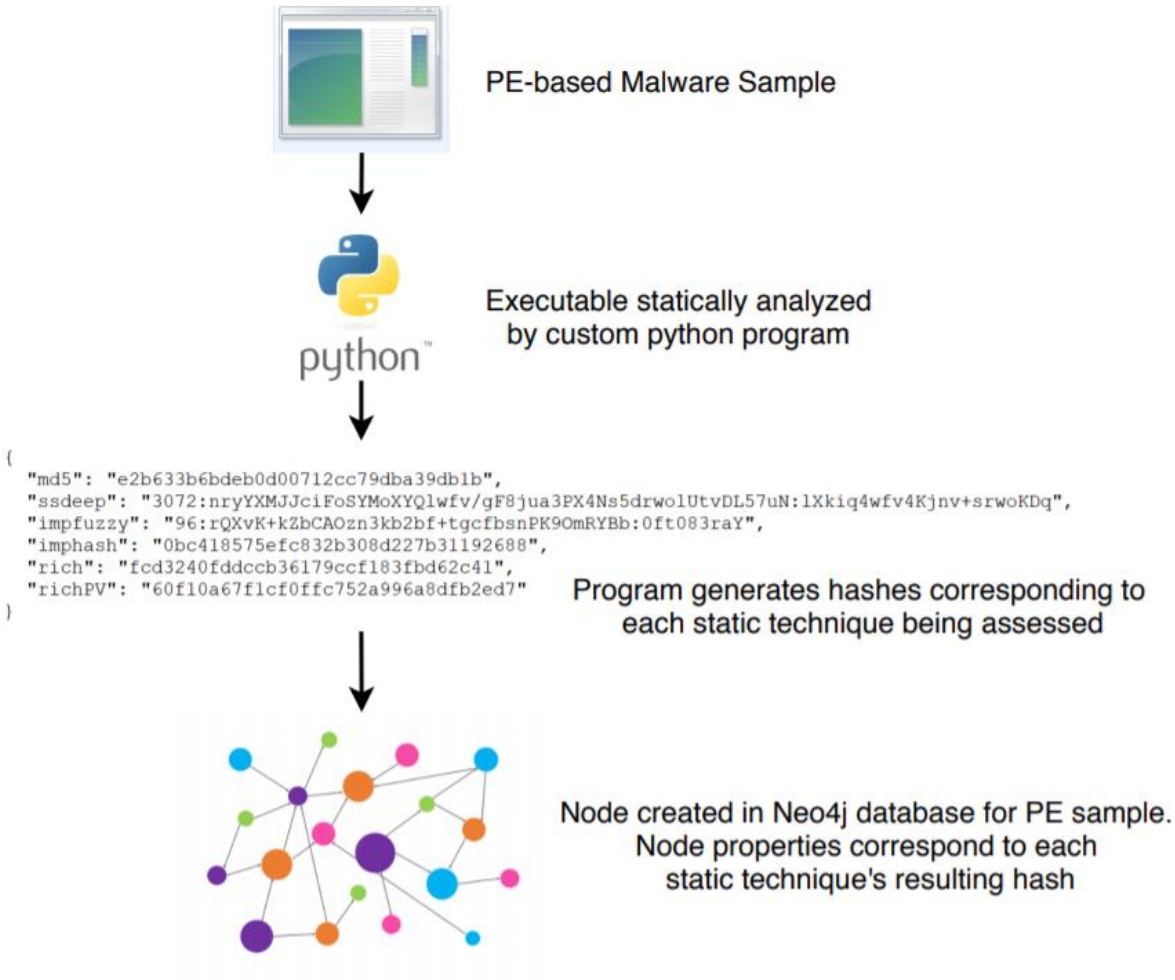

Figure 7. A PE was analyzed by a custom Python program that generated each static technique's product for linking evaluation in the graph database.

Each malware sample existed in the graph database as a single node. The properties of a node included each technique's hash product that the Python script performed. A total of five relationships were defined that correspond to each static technique. Imphash, Rich, and RichPV relationships were established between any two nodes if the samples had an exact hash match. Ssdeep and impfuzzy relationships were created between samples where the values had a similarity match of greater than or equal to 80%. This threshold was chosen as a result of previous studies that determined 80% to be an effective threshold for high fidelity similarity comparisons with the ssdeep algorithm (Oliver, Cheng, & Chen, n.d.). Figure 8 presents a subset of this graph focusing on 25 Korplug samples. In the graph, each Korplug sample is represented by 1 of the 25

nodes. Between Korplug nodes exist none or several of the defined edges that correspond to the static hashing techniques being evaluated.

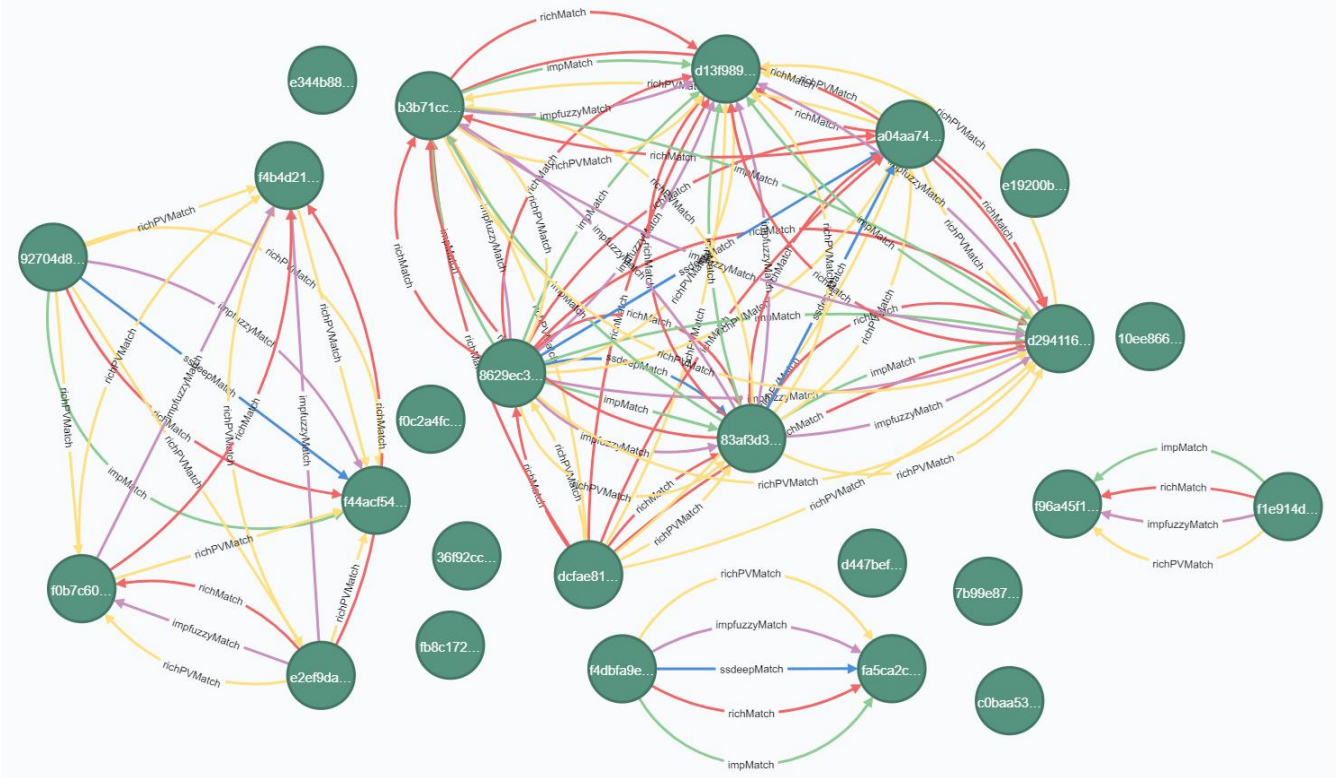

Figure 8. Sample of 25 Korplug nodes & their relationships in Neo4j database.

After all malware samples and the edges between them were defined in the database, clusters of each malware group were immediately recognizable. Using visualization software Gephi, a Fruchterman Reingold representation of the dataset clearly depicted groupings of malware due to Rich, RichPV, ssdeep, import, and impfuzzy hash defined relationships.

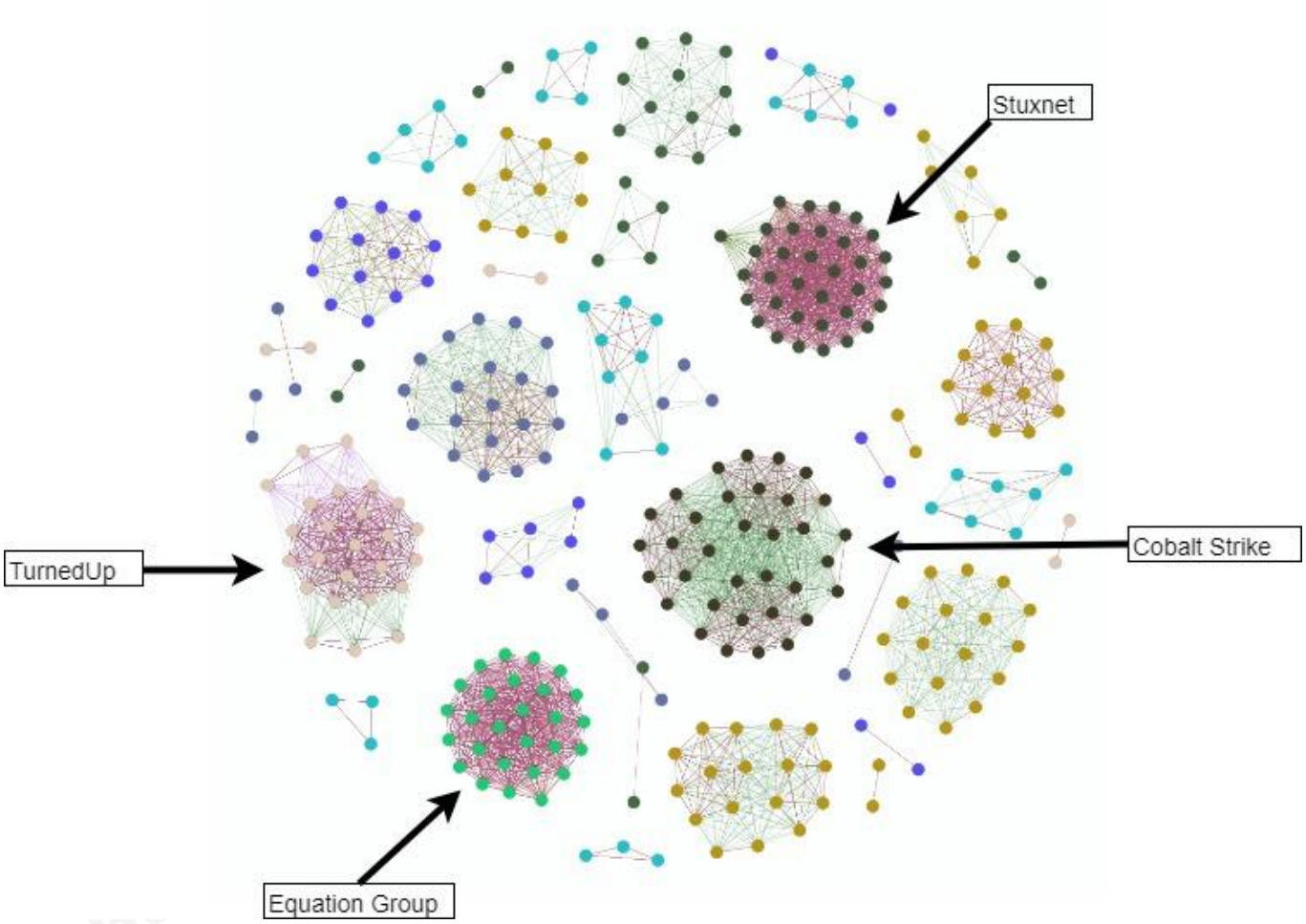

Figure 9. Gephi Fruchterman Reingold visualization of relationships between malware samples according to defined edge types.

The Rich and RichPV hashes were evaluated with two graph theory Link-based Object Classification (LOC) techniques. First, the graph's density was considered. The network's Density (D) was measured as function of total number of edges (E) over the total number of possible edges (V) multiplied by itself minus one (Figure 10). The resulting density calculation spans from zero to one, where one is a dense graph and zero is considered sparse (Samatova, Hendrix, Jenkins, Padmanabhan, & Chakraborty, 2013). A measure of network density was chosen in order to evaluate how each static method impacted the connectedness of the network of malware samples. Density calculations were computed for each edge type according to both the individual malware groups and the entire evaluation's set of malware samples.

$$D = \frac{|E|}{|V|\,(|V| - 1)}$$

Figure 10. Density (D) computation (Samatova, Hendrix, Jenkins, Padmanabhan, & Chakraborty, 2013).

Using Neo4j's Cypher query language the below query computed density calculations according to each static technique's defined relationship type.

```
//Density by Relationship Type
MATCH (c) WITH tofloat(COUNT(DISTINCT(c))) AS numNodes
MATCH (s)-[r]->()
RETURN type(r), COUNT(type(r)) AS numEdges, numNodes, COUNT(type(r)) /
(numNodes * (numNodes - 1)) AS Density
ORDER BY Density DESC
```

Figure 11 shows the subsequent density calculations across the 350 malware samples. Out of the five hashing techniques, the RichPV hash led to the greatest density value of 0.021. The Rich hash resulted in the second highest density value and also out-performed imphash, impfuzzy, and ssdeep.

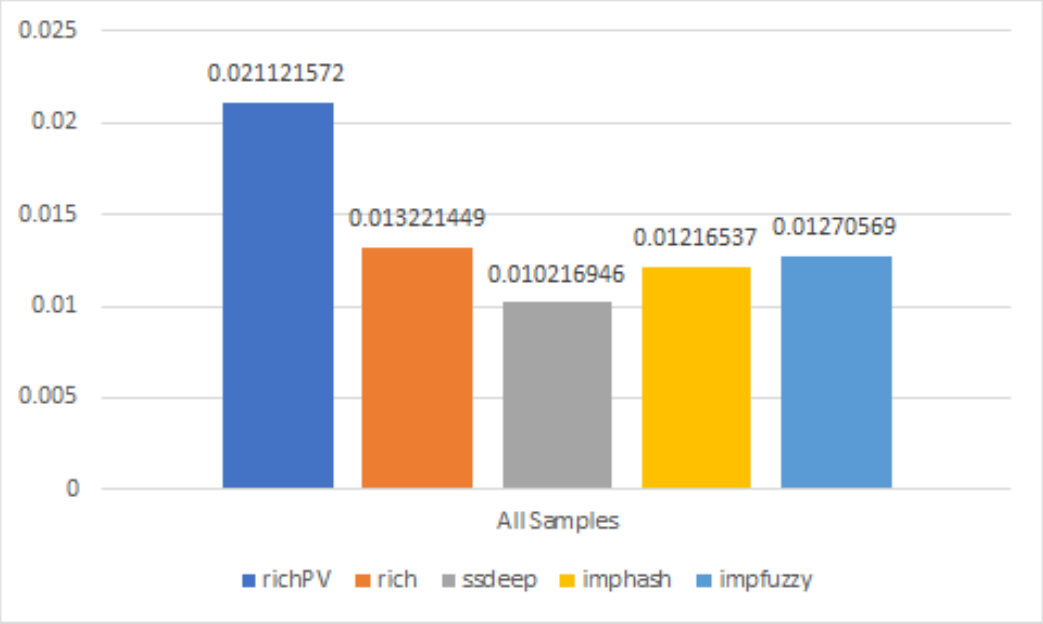

Figure 11. Density calculations for each relationship type across all 350 samples.

In addition to calculating density for the entire data set, graph density was assessed for each individual malware grouping. While the rate of classification for each technique varied between malware groups, Rich and RichPV hashes had strong density performance relative to the other three methods. The RichPV generated a significantly greater number of connections in the Cobalt Strike Beacon, APT28, APT29, and Carbanak malware groups. However, with Stuxnet, Equation Group, and TurnedUp malware groups, the Rich header techniques only performed slightly better.

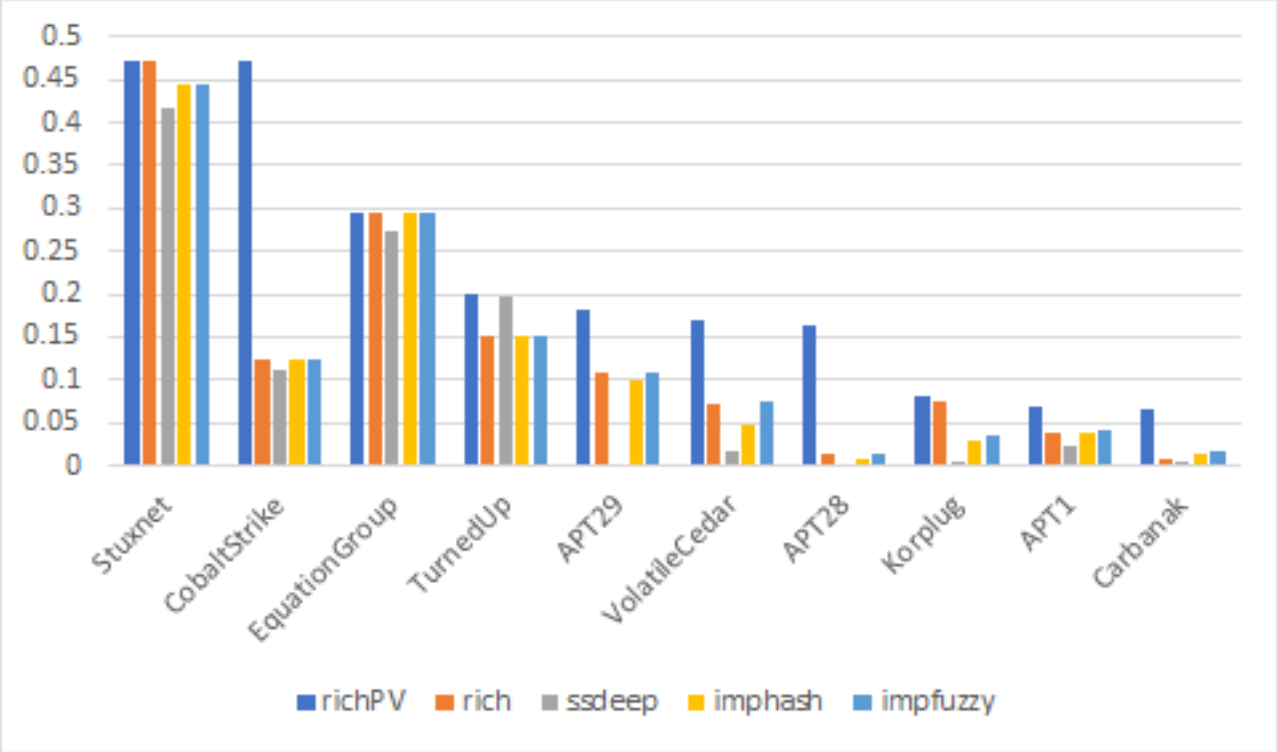

Figure 12. Density calculations for each static technique by malware type.

Closeness Centrality (CLC) provided a second metric for which to evaluate the five static techniques. CLC quantifies how central a node is to the rest of the graph and its ability to traverse relationships to reach other nodes with the shortest possible distance. A CLC metric provides utility by quantifying how easily a network can be traversed with the shortest distance from a single starting point. In the context of this evaluation, a CLC computation provided a measure of which static technique resulted in the shortest path to

all other nodes in the network. CLC was derived by measuring average farness or inverse distance to all other nodes in the graph. The CLC for any given node (v) is defined as the ratio of total nodes (V) minus one to the shortest distance of all other nodes in the graph (Samatova, Hendrix, Jenkins, Padmanabhan, & Chakraborty, 2013). As with density, the output CLC calculation spans between zero and one. A CLC value closer to one indicates a node as central to the graph with minimal distance to all other nodes.

$$CLC(v) = \frac{|V| - 1}{\sum_{i, v \neq v_i} \text{distance}(v, v_i)}$$

Figure 13. Closeness Centrality (CLC) computation (Samatova, Hendrix, Jenkins, Padmanabhan, & Chakraborty, 2013).

Average CLC computations were calculated for each edge type with the following Neo4j Cypher query.

```
//Avg Closeness Centrality by Relationship Type
MATCH (s)-[r]->()
WITH DISTINCT(type(r)) AS Relationship_Type
CALL algo.closeness.stream('', Relationship_Type) YIELD nodeId, centrality
RETURN Relationship_Type, avg(centrality) AS Avg_Closeness_Centrality
ORDER BY Avg_Closeness_Centrality DESC
```

The average CLC was computed across all nodes and grouped by each static technique. As with graph density evaluations, RichPV again outperformed the other methods with an average CLC of 0.882. With the exception of ssdeep, the others had a CLC just below 0.8. The Rich hash did not perform better than the other benchmark techniques.

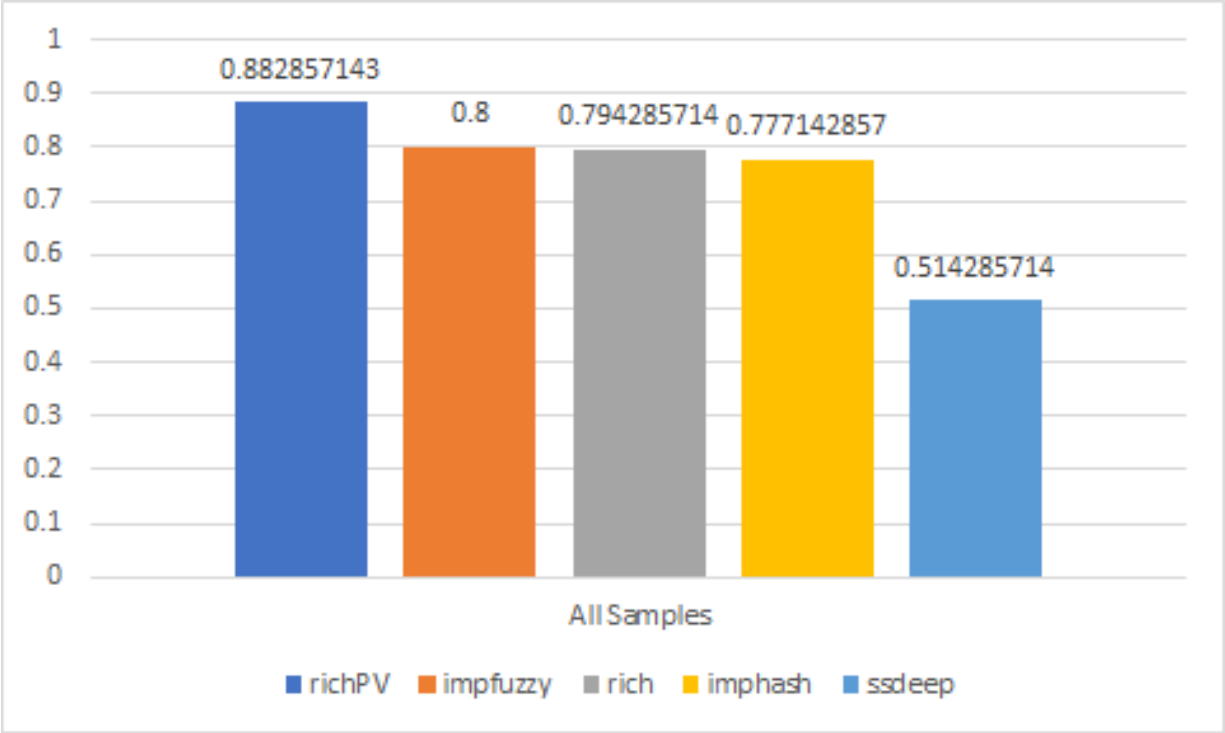

Figure 14. Average CLC for 350 samples by static comparison technique.

Both density and average CLC metrics demonstrate the Rich and RichPV hashing techniques capable of linking and classifying samples. The RichPV hash outperformed all other assessed techniques, resulting in higher sample identification rates. Furthermore, the Rich and RichPV hashes did not result in any false positive relationships created between malware samples from the different malware groups. The results indicate the Rich header to be a valuable asset in identifying and linking PE-basedmalware samples. The Richer header acts as a unique fingerprint for an executable's build environment and can be leveraged for the classification of Microsoft-produced portable executables.

4. Limitations

As with ssdeep, imphash, impfuzzy, and other traditional static techniques, the Rich header has several limitations. While this evaluation did not encounter any false positive links between unrelated malware samples, future work is needed to better investigate how likely false positive conclusions will be made a result of Rich and RichPV hash comparisons. Furthermore, in order for the Rich and RichPV hashes to be

unique, the rich header must be sufficiently long. Similar to classification based on the import address table, if the section is minimal in length it will not provide a high fidelity and unique fingerprint of the executable. In addition, the Rich header itself is not required for any aspect of the executable to function. Because of this, it can be outright removed or overwritten and have no impact on a malware's ability to function. In 2018, Kaspersky presented research on the Olympic Destroyer malware. Kaspersky observed a sample that had a manipulated Rich header (Kaspersky GReAT, 2018). The Rich header was modified and meant to be to be a false flag with the intent of having the malware incorrectly attributed to a different adversary. While the Rich header provides a formidable means of classifying malware, sophisticated adversaries will continue to thwart evolving detection and classification mechanisms malware defenders utilize.

5. Conclusion

This paper investigated the portable executable Rich header section. The Rich header stores metadata about steps in an executable's linking process that effectively serves as a profile of an adversary's build environment. This paper explored what data is stored in the Rich header, how it can be extracted, and what information can be understood from this PE header section. The Rich and RichPV hashes were proposed as two methods to leverage the information stored in the Rich header for the linking and classification of malware samples. These proposed methods were evaluated with 350 malware samples. The results establish the Rich header to be a powerful series of data points capable of classifying malware and outperforming other standard static comparison techniques. While the PE file format has been well-investigated in the study of malware analysis and detection, there has been minimal research into the hidden details of the Rich header. Furthermore, little progress has been made to understand how the Rich header can be leveraged in the pursuit of malware classification and linking. This paper extends current capabilities defenders have available to classify and link malware samples through the investigation of the Rich and RichPV hashes.

References

Arnao, M., Smutz, C., Zollman, A., Richardson, A., & Hutchins, E. (2015). *Laika BOSS: Scalable File-Centric Malware Analysis and Intrusion Detection System*. Retrieved from https://github.com/lmco/laikaboss/raw/master/LaikaBOSS_Whitepaper.pdf

AVTest. (2016, November 28). NUMBERS AND STATISTICS SECURITY REPORT 2015/16. Retrieved from https://www.av-test.org/fileadmin/pdf/publications/security_report/AV-TEST_Security_Report_2015-2016.pdf

Carbon Black. (2016, December 15). Carbon Black Threat Report: Non-Malware Attacks and Ransomware Take Center Stage in 2016. Retrieved from https://www.carbonblack.com/wp-content/uploads/2017/04/Carbon_Black_Threat_Report_Non-Malware_Attacks_and_Ransomware_FINAL.pdf

Case studies in Rich Header analysis and hunting [Blog post]. (2018, August 9). Retrieved from http://ropgadget.com/posts/richheader_hunting.html

Choi, J., Han, Y., Cho, S., Yoo, H., Woo, J., Park, M., … Chung, L. (2013). A Static Birthmark for MS Windows Applications Using Import Address Table. *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. doi:10.1109/imis.2013.159

Dean, D. (1997). The security of static typing with dynamic linking. *Proceedings of the 4th ACM conference on Computer and communications security - CCS '97*. doi:10.1145/266420.266428

Dishather. (2019, April 6). richprint. Retrieved May 19, 2019, from https://github.com/dishather/richprint

Emerson. (2019, January 28). File Scanning Framework. Retrieved May 19, 2019, from https://github.com/EmersonElectricCo/fsf

Hassen, M., Carvalho, M. M., & Chan, P. K. (2017). Malware classification using static analysis based features. *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. doi:10.1109/ssci.2017.8285426

Jianguo, J., Jiuming, C., Qian, Y., Kunying, L., & Chao, L. (2016). Research and design of similar file forensics system based on fuzzy hash. *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*. doi:10.1109/itnec.2016.7560378

Kaspersky GReAT. (2018, March 8). The devil's in the Rich header. Retrieved from https://securelist.com/the-devils-in-the-rich-header/84348/

Kozachok, A. V., & Kozachok, V. I. (2017). Construction and evaluation of the new heuristic malware detection mechanism based on executable files static analysis. *Journal of Computer Virology and Hacking Techniques, 14*(3), 225-231. doi:10.1007/s11416-017-0309-3

Oliver, J., Cheng, C., & Chen, Y. (n.d.). TLSH - A Locality Sensitive Hash. Retrieved from https://documents.trendmicro.com/assets/wp/wp-locality-sensitive-hash.pdf

PE module — yara 3.10.0 documentation. (2019, May 1). Retrieved May 19, 2019, from https://yara.readthedocs.io/en/v3.10.0/modules/pe.html

Pistelli, D. (2010). Microsoft's Rich Signature (undocumented) [Blog post]. Retrieved from https://www.ntcore.com/files/richsign.htm

Samatova, N. F., Hendrix, W., Jenkins, J., Padmanabhan, K., & Chakraborty, A. (2013). *Practical Graph Mining with R*. Boca Raton, FL: CRC Press.

Target. (2019, May 16). Strelka. Retrieved May 19, 2019, from https://github.com/target/strelka

Tomonaga, S. (2016, May 25). Classifying Malware using Import API and Fuzzy Hashing ? impfuzzy ? [Blog post]. Retrieved from https://blogs.jpcert.or.jp/en/2016/05/classifying-mal-a988.html

Tracking Malware with Import Hashing [Blog post]. (2014, January 23). Retrieved from https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html

The Undocumented Microsoft "Rich" Header. (2017, March 12). Retrieved from http://bytepointer.com/articles/the_microsoft_rich_header.htm

Webster, G. D., Kolosnjaji, B., Von Pentz, C., Kirsch, J., Hanif, Z. D., Zarras, A., & Eckert, C. (2017). Finding the Needle: A Study of the PE32 Rich Header and

Respective Malware Triage. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 119-138. doi:10.1007/978-3-319-60876-1_6

6. Appendix A – Parsing the Rich Header in Python

6.1. Code Sample

```
#!/usr/bin/env python
import hashlib
import pefile
import sys
import struct
from collections import OrderedDict
import json

def main():
    results = OrderedDict()
    pe = pefile.PE(sys.argv[1])

    rich_data = pe.get_data(0x80)
    data = list(struct.unpack('<%sI' % str(len(rich_data)/4), rich_data))
    checksum = data[1]
    try:
        rich_end = data.index(0x68636952)
    except ValueError:
        print "PE does not have Rich Header"
        sys.exit()

    #Compute Rich Hash
    Rich_Hasher = hashlib.md5()
    for i in range(rich_end):
        Rich_Hasher.update(struct.pack('<I', (data[i] ^ checksum)))
    richHash = Rich_Hasher.hexdigest()

    #Compute RichPV Hash
    RichPV_Hasher = hashlib.md5()
    for i in range(rich_end):
        if i > 3:
            if i % 2: continue
            else:
                RichPV_Hasher.update(struct.pack('<I', (data[i] ^ checksum)))
    richPV = RichPV_Hasher.hexdigest()

    #Parse elements of Rich header
    richArray = []
    for richElement in range(0, len(pe.RICH_HEADER.values), 2):
        productID = pe.RICH_HEADER.values[richElement] >> 16
        productVersion = pe.RICH_HEADER.values[richElement] & 0xffff
        productCount = pe.RICH_HEADER.values[richElement + 1]
        richArray.append({
            "Product_ID": productID,
            "Product_Version": productVersion,
            "Product_Count": productCount})

    results['Rich Header'] = richArray
    results['Rich Hashes'] = {'Rich Hash': richHash, 'Rich PV': richPV}
```

```
    print json.dumps(results)

if __name__ == "__main__":
    main()
```

6.2. Example Usage

```
analyst@LAB:~/samples$ python parseRich.py 06cd99f0f9f152655469156059a8ea25 |
jq .
{
  "Rich Header": [
    {
      "Product_Count": 62,
      "Product_Version": 9044,
      "Product_ID": 48
    },
    {
      "Product_Count": 6,
      "Product_Version": 7291,
      "Product_ID": 12
    },
    {
      "Product_Count": 11,
      "Product_Version": 8047,
      "Product_ID": 10
    },
    {
      "Product_Count": 2,
      "Product_Version": 8047,
      "Product_ID": 4
    },
    {
      "Product_Count": 10,
      "Product_Version": 7299,
      "Product_ID": 14
    },
    {
      "Product_Count": 21,
      "Product_Version": 4035,
      "Product_ID": 93
    },
    {
      "Product_Count": 243,
      "Product_Version": 0,
      "Product_ID": 1
    },
    {
      "Product_Count": 61,
      "Product_Version": 9782,
      "Product_ID": 10
    },
    {
      "Product_Count": 12,
```

```
      "Product_Version": 9782,
      "Product_ID": 11
    }
  ],
  "Rich Hashes": {
    "Rich Hash": "e06d109762445c31877c719587e9127e",
    "Rich PV": "d19ad1ceca9c94f238dac4ab0e9cb30c"
  }
}
```

7. Appendix B – Dynamically Creating Rich & RichPV Yara Rules in Python

7.1. Code Sample

```
#!/usr/bin/env python
import hashlib
import pefile
import sys
import struct
import yara_tools

def Rich_Yara(sample_md5,rich_end,data,checksum):

    Rich_Hasher = hashlib.md5()
    for i in range(rich_end):
        Rich_Hasher.update(struct.pack('<I', (data[i] ^ checksum)))
    richHash = Rich_Hasher.hexdigest()

    rule = yara_tools.create_rule(name="RichHash")
    rule.add_import(name="pe")
    rule.add_import(name="hash")
    rule.add_meta(key="description", value="Ref: " + sample_md5)
    rule.set_default_boolean(value='and')
    rule.add_condition(condition="uint16(0x00) == 0x5a4d")
    rule.add_condition(condition='hash.md5(pe.rich_signature.clear_data) == "'
+ richHash + '"')
    compiled_rule = rule.build_rule()
    return compiled_rule

def RichPV_Yara(sample_md5,pe):

    rule = yara_tools.create_rule(name="RichPV")
    rule.add_import(name="pe")
    rule.add_meta(key="description", value="Ref: " + sample_md5)
    rule.set_default_boolean(value='and')
    rule.add_condition(condition="uint16(0x00) == 0x5a4d")

    for richElement in range(0, len(pe.RICH_HEADER.values), 2):
        productID = pe.RICH_HEADER.values[richElement] >> 16
```

```
        productVersion = pe.RICH_HEADER.values[richElement] & 0xffff
        rule.add_condition(condition='pe.rich_signature.toolid(%s, %s)' %
(productID,productVersion))

    compiled_rule = rule.build_rule()
    return compiled_rule

def main():

    md5_hasher = hashlib.md5()
    with open(sys.argv[1]) as f: md5_hasher.update(f.read())
    sample_md5 = md5_hasher.hexdigest()

    pe = pefile.PE(sys.argv[1])

    rich_data = pe.get_data(0x80)
    data = list(struct.unpack('<%sI' % str(len(rich_data)/4), rich_data))
    checksum = data[1]
    try:
        rich_end = data.index(0x68636952)
    except ValueError:
        print "PE does not have Rich Header"
        sys.exit()

    print Rich_Yara(sample_md5,rich_end,data,checksum)
    print RichPV_Yara(sample_md5,pe)

if __name__ == "__main__":
    main()
```

7.2. Example Usage

```
analyst@LAB:~/samples$ python richYara.py 06cd99f0f9f152655469156059a8ea25
import "hash"
import "pe"

rule RichHash
{
        meta:
                description="Ref: 06cd99f0f9f152655469156059a8ea25"


        condition:
                uint16(0x00) == 0x5a4d and
                hash.md5(pe.rich_signature.clear_data) ==
"e06d109762445c31877c719587e9127e"

}
import "pe"

rule RichPV
{
```

```
        meta:
                description="Ref: 06cd99f0f9f152655469156059a8ea25"


        condition:
                uint16(0x00) == 0x5a4d and
                pe.rich_signature.toolid(48, 9044) and
                pe.rich_signature.toolid(12, 7291) and
                pe.rich_signature.toolid(10, 8047) and
                pe.rich_signature.toolid(4, 8047) and
                pe.rich_signature.toolid(14, 7299) and
                pe.rich_signature.toolid(93, 4035) and
                pe.rich_signature.toolid(1, 0) and
                pe.rich_signature.toolid(10, 9782) and
                pe.rich_signature.toolid(11, 9782)

}
```

8. Appendix C – Malware MD5s in Rich Header Evaluation

APT1

```
8442ae37b91f279a9f06de4c60b286a3
c0a33a1b472a8c16123fd696a5ce5ebb
f8437e44748d2c3fcf84019766f4e6dc
8b75bcbff174c25a0161f30758509a44
a316d5aeca269ca865077e7fff356e7d
9ea3c16194ce354c244c1b74c46cd92e
33de5067a433a6ec5c328067dc18ec37
ca6fe7a1315af5afeac2961460a80569
55886d571c2a57984ea9659b57e1c63a
321d75c9990408db812e5a248a74f8c8
db2580f5675f04716481b24bb7af468e
57326cd78a56d26e349bbd4bcc5b9fa2
2f930d92dc5ebc9d53ad2a2b451ebf65
001dd76872d80801692ff942308c64e6
f3611c5c793f521f7ff2a69c22d4174e
543e03cc5872e9ed870b2d64363f518b
989b797c2a63fbfc8e1c6e8a8ccd6204
0c28ad34f90950bc784339ec9f50d288
0149b7bd7218aab4e257d28469fddb0d
36d5c8fc4b14559f73b6136d85b94198
7a670d13d4d014169c4080328b8feb86
4c9c9dbf388a8d81d8cfb4d3fc05f8e4
2b659d71ae168e774faaf38db30f4a84
1415eb8519d13328091cc5c76a624e3d
f7f85d7f628ce62d1d8f7b39d8940472
4c6bddcca2695d6202df38708e14fc7e
ef8e0fb20e7228c7492ccdc59d87c690
cc3a9a7b026bfe0e55ff219fd6aa7d94
1966b265272e1660e6f340b19a7e5567
c75d351d86de26718a3881f62fddde99
995b44ef8460836d9091a8b361fde489
f10d145684ba6c71ca2d2f7eb0d89343
43ce605b2584c27064febb0474a787a4
423a30c077b12354a4a5c31d4de99689
e66dd357a6dfa6ebd15358e565e8f00f
```

APT28

```
2d4eaa0331abbc6d867f5f979b2c890d
eda061c497ba73441994a30e36f55b1d
91381cd82cdd5f52bbc7b30d34cb8d83
c2a0344a2bbb29d9b56d378386afcbed
211b7100fd799e9eaabeb13cfa446231
c16b07f7590a8620a8f0f687b0bd8bd8
7c2b1de614a9664103b6ff7f3d73f83d
d535c3fc5f0f98e021bea0d6277d2559
18efc091b431c39d3e59be445429a7bc
ed601bbd4dd0e267afb0be840cb27c90
75f71713a429589e87cf2656107d2bfc
7764499bb1c4720d0f1d302f15be792c
```

```
07c8a0a792a5447daf08ac32d1e283e8
f7ee38ca49cd4ae35824ce5738b6e587
732fbf0a4ceb10e9a2254af59ae4f880
a579d53a1d29684de6d2c0cbabd525c5
800af1c9d341b846a856a1e686be6a3e
a3c757af9e7a9a60e235d08d54740fbc
c4ffab85d84b494e1c450819a0e9c7db
6159c094a663a171efd531b23a46716d
1219318522fa28252368f58f36820ac2
dffb22a1a6a757443ab403d61e760f0c
540e4a7a28ca1514e53c2564993d8d87
56e011137b9678f1fcc54f9372198bae
291af793767f5c5f2dc9c6d44f1bfb59
2dfc90375a09459033d430d046216d22
ac75fd7d79e64384b9c4053b37e5623f
078755389b98d17788eb5148e23109a6
072c692783c67ea56da9de0a53a60d11
607a7401962eaf78b93676c9f5ca6a26
8b6d824619e993f74973eedfaf18be78
991ffdbf860756a4589164de26dd7ccf
23ae20329174d44ebc8dbfa9891c6260
ea726d3e8f6516807366584f3c5b5e2a
9617f3948b1886ebc95689c02d2cf264
```

APT29

```
2ef51f1ca11ce73fa20b54a5886ad1dd
416db420e781c709bb71acee0b79282f
98a6484533fa12a9ba6b1bd9df1899dc
9f65e3b320ec91380ebc28d4fdff4895
f58a4369b8176edbde4396dc977c9008
08709ef0e3d467ce843af4deb77d74d5
5ebce6cbedfec82f1428c3409e3df0ef
eb22b99d44223866e24872d80a4ddefd
f0a6436ffee12558a434a0fc24b3b33f
3d3363598f87c78826c859077606e514
43c012086c1ae0a67c38b0926d6cba3f
330ed7549d50bdb56497a5577132610a
c79bf9a04913a5018ab8de65ffd1060f
c42bf27579eaadfa080134f3400a417b
62c4ce93050e48d623569c7dcc4d0278
acffb2823fc655637657dcbd25f35af8
66d2b5ed8646a0ef38eef822555b9828
a5d6ad8ad82c266fda96e076335a5080
181a88c911b10d0fcb4682ae552c0de3
52474b705610245f67bbd1c86ab8bd7b
83acacbd57997f6326817f709f857893
9ad55b83f2eec0c19873a770b0c86a2f
b4ae6966e65e47afa41610b1fb554607
90bd910ee161b71c7a37ac642f910059
d9703d014c5d4f55e2996f3573544476
01a2c13c42f1a0557421d341f4165423
fef254d6c46fdced294db44acef8d839
90674c3cca487fedbe77c4986d023296
1ff0ed11fc6a41db458a75ae71670f94
270ca8368cd4216b1813281d3efe485d
3a746f525877b3d006758def2957ddaf
bc626c8f11ed753f33ad1c0fe848d898
4121414c63079b7fa836be00f8d0a93b
91aaf47843a34a9d8d1bb715a6d4acec
f02da961eb7b87b41aee5fd9537022f0
```

Carbanak

```
6e564dadc344cd2d55374dbb00646d1b
972092cbe7791d27fc9ff6e9acc12cc3
55040dd42ccf19b5af7802cba91dbd7f
5443b81fbb439972de9e45d801ce907a
751d2771af1694c0d5db9d894bd134ca
5aeecb78181f95829b6eeeefb2ce4975
1e127b92f7102fbd7fa5375e4e5c67d1
0155738045b331f44d300f4a7d08cf21
0275585c3b871405dd299d458724db3d
1300432e537e7ba07840adecf38e543b
4f16b33c074f1c31d26d193ec74aaa56
36cdf98bc79b6997dd4e3a6bed035dca
2e2aa05a217aacf3105b4ba2288ad475
608bdeb4ce66c96b7a9289f8cf57ce02
0ad6da9e62a2c985156a9c53f8494171
643c0b9904b32004465b95321bb525eb
88c0af9266679e655298ce19e231dff1
aa55dedff7f5dbe2cc4a47f2f8d44f94
10e0699f20e31e89c3becfd8bf24cb4c
2c6112e1e60f083467dc159ffb1ceb6d
50f70e18fe0dedabefe9bf7679b6d56c
1b9b9c8db7735f1793f981d0be556d88
1e47e12d11580e935878b0ed78d2294f
4afafa81731f8f02ba1b58073b47abdf
b400bb2a2f9f0ce176368dc709359d3d
1f43a8803498482d360befc6dfab4218
a1979aa159e0c54212122fd8acb24383
551d41e2a4dd1497b3b27a91922d29cc
407795b49789c2f9ca6eca1fbab3c73e
6163103103cdacdc2770bd8e9081cfb4
1d1ed892f62559c3f8234c287cb3437c
3dc8c4af51c8c367fbe7c7feef4f6744
1fd4a01932df638a8c761abacffa0207
763e07083887ecb83a87c24542d70dc5
7d0bbdda98f44a5b73200a2c157077df
```

Cobalt Strike Beacon

```
e2b633b6bdeb0d00712cc79dba39db1b
```

74d6d45f027025a13ed6ae894d03c227
de6f57f6b10474c01fba8d573b1b1bfd
ca4036b961d333e1aa9ee5edbaa3b60a
37fa49bd3dad22ae7a9014d03570c37c
d8e7337bf3f2cca77f50b96736ead0e8
2ed3d0f44e84ab603781327d2b23bfb1
3a909966bd21446b9b05d732668051c6
73ea5db269f885d846baf5e6f0cc8b79
3f41c063b24159233ae8b51252f0c6cb
2ea81109828f19ec2590dfbd538931cf
bffa64a413c1e2b1c0662b17838c0701
38cdcda930fabb623a5ae01b6bbc8b8a
7aa7b969cd69585719178876f04c9092
6da3df05d500172371d21cbb1b5f2ae9
a500a1b92f0beddab283542bcca8c841
5b99f0dad7744eb4f8def14e0a2b1bca
523ec3b6a9bf630b2aa2f7d410a15403
ad59b9c7e4f668eb8476bfffcba82e62
5f2a942b6777265f4c3833b65ed6b011
9ed16b87af65120169f2c9e0a1eb0114
198ca0b909dd89c638cdeef4f9283466
e08f5d1720029b34ca95679f513a8224
0f63ed30ab1b2b2f045b43e25d560397
5190e11004dbab4edeeab60f00c66092
955f16849c59c70b851849811da1f231
4305909174235321fdd670aac1422d20
6390ed89df50bce6128af3725e42982f
a052db0685969e82eb3b5d2518ef89b1
d9d841e4e0e53d6fffce0da7c0240d07
25c4f0fa2f4c167ec68e9e81c5dbfc3b
db62d9de07966cd6ac2955f400363965
4f4da0137b176a9ccfc0a7bf6eb60bb0
044957b7bff481f46d392f88dc4b1acc
e569fb84b20c0da6052d3910a7c832ae

Equation Group

9180d5affe1e5df0717d7385e7f54386
6fe6c03b938580ebf9b82f3b9cd4c4aa
24a6ec8ebf9c0867ed1c097f4a653b8d
0a209ac0de4ac033f31d6ba9191a8f7a
2a12630ff976ba0994143ca93fecd17f
e2320f490cbb2e082e699ebeb0faa917
78b1ff3b04fac35c890462225c5fbc49
21a6959a33909e3cdf27a455064d4d4d
487e79347d92f44507200792a7795c7b
6d10eb87d57fc0b3eb1c41cccf0319f4
4810559ed364a18843178f1c4fca49fc
aaa06c8458f01bedcac5ec638c5c8b24
72312f1e2ae6900f169a2b7a88e14d93
bd7a693767de2eae08b4c63aaa84db43
dc7ad1008509d0a67dbafde8ecffb4be
380258de6e47749952b60e5307d22dc0
ddeff291518f4677c5fa7518f2a3d716
a5f2c5ca6b51a6bf48d795fb5ae63203
aff10dd15b2d39c18ae9ee96511a9d83
44bd4cf5e28d78cc66b828a57c99ca74
ea943c7cc83d853de678c58b838fbd65
02d5eb43f5fc03f7abc89c57b82c75f8
9a8def5ccee1b32f4d237c1dd1eba8c6
37085d946c77f521c3092f822bc3983f
4ad2f62ce2eb72eff45c61699bdcb1e3
c05255625bb00eb12eaf95cb41fcc7f5
199e39bda0af0a062ccc734faccf9213
c69dfb1302032d28df98ae70474809f2
752af597e6d9fd70396accc0b9013dbe
40fee20fe98995acbda82dbcde0b674b
11fb08b9126cdb4668b3f5135cf7a6c5
c303afe1648d3b70591feeffe78125ed
4556ce5eb007af1de5bd3b457f0b216d
ba39212c5b58b97bfc9f5bc431170827
17d287e868ab1dbafca87eb48b0f848f

Korplug/Plugx

c0baa532636ecca97de85439d7ae8cb3
92704d878ad23c11e527485e5f3cbf57
f0c2a4fc3a9731c83e48e1adc2679ff8
f1e914d7cd74323777e20ec73eee7601
7b99e87e3d5a03edd30430bf8b019242
e19200b9834a091263ec19a976c674b7
10ee8662a2382243c6e5f879413f6dc0
dcfae8107986b3596eb3b84432550caf
f96a45f1e6603591fe7b26aa768b8595
8629ec3d579795f1b9bf244a6c654e40
83af3d34f6458255014768b100ed5799
a04aa74cd3526024e8787d758494f383
e2ef9da322dca474fff073a0172e1843
f4dbfa9e5ab9ddbfc962172fc4d27401
f0b7c6008a64971c5ac9b761de92f8f6
fb8c172c964e6740963eb223407a917c
f4b4d217e62cd415ae2a87fa7eae8946
d447befee3ecbeeba8cbdd1ed196126f
fa5ca2cba0dab28fa0fb93a9bd7b1d83
f44acf54b6d91a43954cc0f99b0b1ba9
e344b888c4b4daafdc36e67a74f2da6a
b3b71cc20d12f538dd446d5aae8ea60d
d2941164132c12714e91e421d1556d21
d13f989b02f10670e4fbc31c54048b97
36f92cc6e98534b1234f7ff7714a4ab2
cdc8f1bd5798c4ab3345b55e54ed9b87
4e08ecebfa8bc7fba0c0b483029a2483
e49fd59a53109ed2a1ebf959f80f3147

f7df09d04109694e6dfe42e6753da1e1
f23ff8906dcc74f9a42ac84092ccf755
fb1cfb4a900088bbc5fb8100f8bc2bb7
5df100f2a3b91a277794ea5e0a456752
4dfa2f7a5138b9b158ecf9bbc65906e0
584b871bd675208840e3f79d87ee63c6
69a8d99359dd86bb384bcb2232d01277

Stuxnet
0eb67a36a30dd07a3a9dfbcc2ebbbbb4
13dbce0ab05c47e3d965b89b80406956
100e5489f2257b23dcdfa2dbf8538246
15275445c62bbada68995de21a700a07
447dc04020227a5b8bfff600448c424f
265adfbfbe0815e9e583478d27592b43
2a8191c401ae3eeea92351f48a4a95b8
13c3a16b8273b4145f36e9482a8d94f2
1d682fcb624bf3bb96d93a9a217ec066
1b1dd89f6b0962b410bf22449446dbb7
2c37c9f77a89c80f09d205b5e95e548b
359f9f98bfaf355a2a2755989a09503a
7d9ee95f41f5a305e3c42914bc26d4f4
8583fdd26c95ddf158969a447bffbd53
4b8579280b1262055574cdd2a2ef7da0
53d6a9707e6be689a79ed2ffe8c9bf70
5bb7fa235037ed4044f70cd617e4ad24
5c7e60775cbe74a79ab4c05407e68f69
58c02be6190bd82100cbe0dfeb5e8cbc
664c9c4c175a46d941628a2a380c3197
67062fa7da5ae65957753d95290d7a15
6db9effe78a9ba10593023e12361febe
855a26bec09ca067895679e88f9dfd17
91a49c4f97291ca35faabc3a472d8d94
9b015fdbaf3f148295dc17fbc9915f56
9ca1d37cb2f61c4e6a3ebbaec3e56329
a304d1e439cd234d0b8db312e999d4a0
ad502dbdf5acb80f3a5d8b604dc69344
ba67c6b1fa4b29636f098fd30d291d22
c6a3b81f606561f0f75cc5a1cc0e43f7
e32f8be18d06af2465620b142c1bae8c
d6ef4e76f27a593b8e788231586f80e9
ed5d272c0d7981a97309d98d690ba875
fc800fe9644e2622500bbdf8c7c58271
f8303b19349f1b06edb8ac8a3b3b4111

TurnedUp
a272326cb5f0b73eb9a42c9e629a0fd8
6b41980aa6966dda6c3f68aeeb9ae2e0
8e67f4c98754a2373a49eaf53425d79a
6f1d5c57b3b415edc3767b079999dd50
fb21f3cea1aa051ba2a45e75d46b98b8
0ccc9ec82f1d44c243329014b82d3125
1381148d543c0de493b13ba8ca17c14f
c02689449a4ce73ec79a52595ab590f6
3f5329cf2a829f8840ba6a903f17a1bf
d01781f1246fd1b64e09170bd6600fe1
6a0f07e322d3b7bc88e2468f9e4b861b
59d0d27360c9534d55596891049eb3ef
a80c7ce33769ada7b4d56733d02afbe5
bbdd6bb2e8827e64cd1a440e05c0d537
b3d73364995815d78f6d66101e718837
32a9a9aa9a81be6186937b99e04ad4be
8e6d5ef3f6912a7c49f8eb6a71e18ee2
c2d472bdb8b98ed83cc8ded68a79c425
c57c5529d91cffef3ec8dadf61c5ffb2
ae47d53fe8ced620e9969cea58e87d9a
c6f2f502ad268248d6c0087a2538cad0
b189b21aafd206625e6c4e4a42c8ba76
aa63b16b6bf326dd3b4e82ffad4c1338
ae870c46f3b8f44e576ffa1528c3ea37
b12faab84e2140dfa5852411c91a3474
a813dd6b81db331f10efaf1173f1da5d
c2fbb3ac76b0839e0a744ad8bdddba0e
a2af2e6bbb6551ddf09f0a7204b5952e
de7a44518d67b13cda535474ffedf36b
c66422d3a9ebe5f323d29a7be76bc57a
b681aa600be5e3ca550d4ff4c884dc3d
b5f69841bf4e0e96a99aa811b52d0e90
0753857710dcf96b950e07df9cdf7911
797bc06d3e0f5891591b68885d99b4e1
c55b002ae9db4dbb2992f7ef0fbc86cb

Volatile Cedar
981234d969a4c5e6edea50df009efedd
29eca6286a01c0b684f7d5f0bfe0c0e6
22872f40f5aad3354bbf641fe90f2fd6
740c47c663f5205365ae9fb08adfb127
c19e91a91a2fa55e869c42a70da9a506
edaca6fb1896a120237b2ce13f6bc3e6
08c988d6cebdd55f3b123f2d9d5507a6
7031426fb851e93965a72902842b7c2c
6f11a67803e1299a22c77c8e24072b82
c898aed0ab4173cc3ac7d4849d06e7fa
d2074d6273f41c34e8ba370aa9af46ad
e6f874b7629b11a2f5ed3cc2c123f8b6
5b505d0286378efcca4df38ed4a26c90
306d243745ba53d09353b3b722d471b8
44b5a3af895f31e22f6bc4eb66bd3eb7
eb7042ad32f41c0e577b5b504c7558ea
1d4b0fc476b7d20f1ef590bcaa78dc5d

1dcac3178a1b85d5179ce75eace04d10	034e4c62965f8d5dd5d5a2ce34a53ba9
66e2adf710261e925db588b5fac98ad8	3f35c97e9e87472030b84ae1bc932ffc
61b11b9e6baae4f764722a808119ed0c	5ca3ac2949022e5c77335f7e228db1d8
c9a4317f1002fefcc7a250c3d76d4b01	4f8b989bc424a39649805b5b93318295
7dbc46559efafe8ec8446b836129598c	9a5a99def615966ea05e3067057d6b37
7cd87c4976f1b34a0b060a23faddbd19	826b772c81f41505f96fc18e666b1acd
c7ac6193245b76cc8cebc2835ee13532	5d437eb2a22ec8f37139788f2087d45d
ea53e618432ca0c823fafc06dc60b726	184320a057e455555e3be22e67663722
ab3d0c748ced69557f78b7071879e50a	2b9106e8df3aa98c3654a4e0733d83e7