Malware Attribution Using the Rich Header

Robert J. Joyce
Dept. of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD
joyce8@umbc.edu

Kevin Bilzer
Dept. of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD
kbilzer1@umbc.edu

Seamus Burke
Dept. of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore, MD
sburke1@umbc.edu

***Abstract*—Modern malware analysis is faced with the difficult issue of attribution. There are a variety of ways to misdirect an analyst's efforts towards attribution, from tampering with data to make attribution impossible, to attempting to frame other threat actors. We propose a new metadata hash called RichPE that takes into account the undocumented, but extremely useful Microsoft Rich header as well as elements of the PE header to fingerprint malware based on the environment it was compiled in. To achieve this goal we dug deep into the Rich header's format, reversed parts of the Microsoft compiler toolchain, and experimented with a variety of different hashing approaches to get the best results. Our proposed hash is efficient and scalable across large datasets of malware. We describe the implementation of our hash and and evaluate its effectiveness and accuracy against the industry state-of-the-art metadata hashes Imphash and pehash.**

Keywords—Malware Analysis, Cyber Attack Attribution, File Format Metadata, Metadata Hashing, Compilers/Linkers

I. Malware Classification and Attribution

VirusTotal, an online malware aggregation and analysis service, observes hundreds of thousands of unique, new files submitted to it daily [1]. Given the sheer scale of file submissions, the task of categorizing them is extremely challenging. There are three distinct ways that these files must be categorized: detection, classification, and attribution. The most basic of these three is detection; whether the file is benign or malicious. If a file is determined to be malicious, the next step is to classify it into a family or at least a broad category of malware. Finally, attribution is the process of determining the identity of the person who used the malware in a cyber attack. In this paper, we will be focusing primarily on methods of performing classification and attribution.

Classification can be performed in a variety of ways. The slowest but surest method is to have an analyst manually reverse-engineer the malware sample in question. There are also ways to use signatures to automatically classify malware. Automatic malware signaturing is subdivided into two categories: static and behavior-based.

One of the most common methods for performing static signaturing is with the open-source malware scanning tool YARA. YARA rules can contain the strings, byte sequences, and metadata expected to be present in samples from specific malware families. It is common for analysts to release YARA rules to the public along with other indicators of compromise. Unfortunately, if an actor finds that a YARA rule for their malware has been released, they can simply modify new versions of their malware so that it is no longer detected. Many antivirus engines also perform static signaturing in a similar way as YARA, although most do not publish their rules. Even so, the use of packers to obfuscate malware has made static signaturing increasingly difficult.

Behavior-based signaturing involves running malware samples in a sandbox and classifying them into a family based on the actions they perform. Snort is one of the most common open-source behavior-based methods for signaturing malware. Snort rules classify malware samples based on their network traffic. Many antivirus engines also perform behavior-based classification. They consider many features when classifying a running malware sample, such as the ways that it interacts with the filesystem, registry, and network. Although behavior-based signaturing is not hindered by packing, it is faced with other challenges. Running malware in a sandbox is time-consuming, especially at a large scale. In addition, malware may behave differently based on conditions such as the input it receives or the environment it is run in. Finally, some kinds of malware can detect if they are being run in a sandbox and will not display any malicious behavior if they are.

The ultimate goal of attribution is identify which actor performed a cyber attack so that they can be held accountable for their actions. Unfortunately, cyber attack attribution is an even more challenging problem than family classification. Analysts must consider evidence from two broad categories when performing attribution: indicators of compromise (IOCs) and tactics, techniques, and procedures (TTPs). IOCs are the evidence left behind by a cyber attack, such as the files left on infected systems and the IP addresses and domain names used by the attackers. On the other hand, TTPs are a description of the typical behavior of an actor, such as the infection vectors and malicious payloads they commonly use, the victims they target, and the goals they want to achieve. Only with sufficient TTPs

and IOCs can an analyst confidently attribute a cyber attack to a malicious actor.

Malicious actors have a vested interest in evading attribution and have become much more cunning about how they do so. They have reduced the value host-based IOCs by using techniques such as polymorphism as well as the value of network-based IOCs by frequently changing their network infrastructure [2]. In addition, rather than using custom malware payloads, actors are beginning to use commodity malware as their focus has shifted from evading detection to evading attribution [2]. Some advanced actors are taking this notion a step further and performing "false-flag" operations by using the tactics, techniques, and procedures that are typical of another actor to point attribution elsewhere [2].

II. PE File Metadata

The malware payloads used in cyber attacks are among the most valuable host-based IOCs. Analyzing an executable payload can yield a lot of information about the attacker, such as the goal of the cyber attack and the command-and-control infrastructure used during the attack. Executable files also contain a lot of metadata that can be useful for attribution, such as when the file was compiled, the actor's language, and which library functions were imported. Our research primarily focuses on using malware metadata to aid classification and attribution, specifically the metadata that remains as an artifact in an executable file as a result of the software development lifecycle. The scope of our research was limited to malware payloads that are in the Portable Executable, or PE, format. Files in this format are executable files that can run on the Windows operating system, such as .exe and .dll files.

PE files are laid out in a predictable structure and contain a variety of different tables and headers. All PE files begin with the MS-DOS header. In PE files compiled and linked with the Microsoft toolchain, a header called the Rich header follows the MS-DOS header. The Rich header is the primary focus of our research and it will be discussed in detail in a later section. Following the DOS header (and the Rich header, if it is present) is a series of fields known collectively as the PE header. One major field in the PE header is the *IMAGE_FILE_HEADER*, which contains basic metadata such as the architecture the file is intended to run on, the number of PE sections, and the compilation timestamp. Another header, the *IMAGE_OPTIONAL_HEADER*, includes an even larger amount of metadata. Each PE file contains some number of sections that include different types of data. Some common sections include .text, which contains executable code, .bss, .rdata, and .data, which include uninitialized, read-only, and global data respectively, and .idata, which includes import data [3]. Each section in the PE file has its own *IMAGE_SECTION_HEADER*, which includes metadata such as the section's name, virtual address, and size. The .idata section is of particular note, as it contains the import address table, or IAT. The IAT is a table that contains a pointer to each imported library function.

III. Metadata Hashing

A technique called metadata hashing is extremely useful for clustering malware samples with similar metadata. To compute the metadata hash of a malware sample, specific metadata fields are parsed from the file and then used as input to a cryptographic hash function. The resulting digest is the metadata hash of that malware sample. A database of malware samples can be indexed on a metadata hash, allowing for efficient querying. Two metadata hashes are particularly popular in the malware analysis community: Imphash and pehash.

The Imphash of a malware sample is computed by hashing its imports in the order that they appear in the IAT. The linker generates the IAT based on the order that imported functions appear in the source code [4]. Two malware samples that share the same imported functions in the same order in their source code will share the same import hash. While two unrelated files can have this property, as the number of imported functions increases, the likelihood that two files with the same imphash are related also increases. The security company Mandiant popularized Imphash by using it to great success when tracking the alleged Chinese threat group APT1 [4].

Polymorphism is a technique that malicious actors use to reduce the value of host-based IOCs. Analysts track the files involved in cyber attacks by computing their cryptographic hashes. However, if a file changes by a single byte, its file hash changes entirely. Polymorphism allows files to have the same behavior but different executable code, defeating file hashing and inhibiting static signaturing. A metadata hash called pehash was developed with the goal of clustering self-modifying polymorphic malware in the same family [5]. Although self-modifying polymorphic malware always changes its executable code, it rarely changes its metadata. Therefore, the pehash of a file is calculated using only metadata in the PE header. Specifically, various fields from the *IMAGE_FILE_HEADER*, *IMAGE_OPTIONAL_HEADER*, and each *IMAGE_SECTION_HEADER* are used to compute the pehash digest. Wicherski's paper on pehash showed that it is especially effective at identifying polymorphic malware samples and reducing the time wasted by repeatedly analyzing polymorphic malware samples that behave in the same way. For example, by clustering the malware samples in the mwcollect Alliance dataset and only giving a sandbox run to one malware sample per cluster, over 83.1% of dynamic analysis time was saved [5].

IV. Packed Malware

Malware packing is a very common technique used by malicious actors to inhibit static analysis and static signaturing. Packing is formally defined as the compression of an executable file and combining the compressed data with decompression code into a single executable [6]. In addition to making malware more difficult to analyze and detect, packers also have an adverse effect on the effectiveness of the aforementioned industry standard metadata hashes Imphash and pehash. When malware is

packed, the packer often modifies the *IMAGE_SECTION_HEADER*s. Pehash is affected because it uses metadata from the *IMAGE_SECTION_HEADER*s and if that data changes then the pehash digest will change as well. Packed malware often uses a technique called dynamic linking, which allows malware to leave imports unresolved until it is run. When a malware sample is packed by a packer that uses runtime linking, its IAT will change, and as a result its Imphash will also change.

V. The Rich Header

This paper proposes a new metadata hash called RichPE. The hash includes metadata from both the Rich header and the PE header. Before introducing RichPE, this paper will provide a technical overview of the Rich header, discuss how the values in it are generated during the compilation and linking process, and summarize the previous research on the Rich header with respect to malware analysis.

The Rich header of a PE file contains information about the build environment it was created in. Specifically, this information is stored in a table of 8-byte entries within the Rich header. Each entry contains a 2-byte ID corresponding to a Microsoft Product Identifier (*ProdID*), a 2-byte version of the compiler used to create the product (*mCV*), and a 4-byte count of the number of times the product was included during the linking process (*Count*) [7]. The Rich header also includes a 4-byte checksum that is calculated from portions of the MS-DOS header as well as the *ProdID*s and *mCV*s. The table of *ProdID*s, *mCV*s, and *Count*s is obfuscated in the PE file using the checksum as an XOR key [7]. An illustration of the Rich header is shown in the figure below.

FIGURE I. Rich Header Breakdown

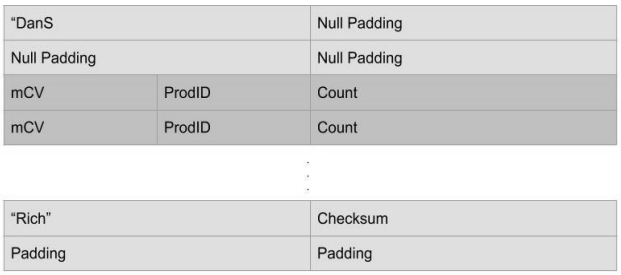

Fig. 1. The Rich header is composed of 3 sections: a header consisting of the *DanS* keyword and null padded fields, a series of 8-byte Rich header entries, and a footer with the *Rich* keyword and checksum.

This information is introduced into the Rich header through the linking process. The Microsoft Linker (*link.exe*), starting with Visual Studio 6.0 in 1998 started to emit metadata about the build environment and include it in the generated binary files. In its *IMAGE:BuildImage()* function, the linker makes a call to a function named *IMAGE:CbBuildProdIdBlock(),* which we have determined handles the creation of the Rich header. The name of the function also leads us to believe that Microsoft's internal name for the Rich header is the Product ID Block. This function is rather simple. It allocates a linked list of LIB structures, with two preset values; the linker and the resource compiler IDs. Both of those are hardcoded values in each linker release. Following this, the linker loops through the Linker Database in memory, which it read in from all the compiler-emitted OBJ and LIB files, and adds them to the linked list.

The linker does not modify or create the values in the Rich header, it just reads them in from compiler artifacts and places them into the header format. Most of the Rich header *ProdID*s are straightforward and simply correspond to a count of how many times an artifact was used in the compilation process. However, there is one interesting exception: *ProdID* 1 corresponds to the imported functions count. Surprisingly, it does not always match the number of imports listed in the IAT. After analyzing hundreds of legitimate Windows system files, it appears that no legitimate binary has fewer imports listed in *ProdID* 1 than in the IAT, but the value can be larger. To understand why this occurs, we have begun reverse-engineering the compiler itself to determine how this value is generated. It appears that *ProdID* 1 may represent the total number of imported functions referenced in all DLLs; that is, imports referenced from objects that were imported. However, more testing is needed to determine before we can conclude that this is definitely the case.

VI. Using the Rich Header for Malware Analysis

Use of the Rich header in the field of malware analysis is surprisingly low given its high potential for classification and attribution. Webster et. al's paper titled *Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage* [7] is the only published academic paper that we have found that incorporates the topics of the Rich header and malware analysis. The Rich header has also received mentions in a few malware analysis blogs, but is only featured prominently in Kaspersky's report *The devil's in the Rich header* [8].

Webster et. al surveyed 964,816 PE files from the VirusShare dataset in order to gather statistics about the contents of the Rich header. They observed that the vast majority of malware authors seem to be unaware of the Rich header and that most malware samples are compiled using the Microsoft toolchain: 71% of the surveyed malware samples contain a Rich header [7]. Webster et. al also discovered that some of the most frequently used malware packers, such as UPX, ASPack, and NSIS, do not corrupt the Rich header [7]. Interestingly, the authors found that the Rich header could be used to identify tampering of the DOS header, because the Rich header's checksum is calculated based on the contents of certain fields in the DOS header as well as the *ProdID*s and *mCV*s in the Rich header. Webster et. al found that of the 683,238 samples with a Rich header, 137,965 had an invalid Rich header checksum [7]. Finally, using a stacked autoencoder to map each malware sample's *ProdID*s, *mCV*s, and *Count*s into a 50-dimensional feature vector and a ball tree to store the feature vectors, the authors showed that the contents of the Rich header could be used to cluster malware [7].

In March 2018, Kaspersky reported their findings on the cyber attack that shut down and destroyed infrastructure at

the Pyeongchang Winter Olympics. The malware family used in this attack was named OlympicDestroyer and had a large variety of malicious capabilities, including disk wiping, worming, and credential stealing [9]. OlympicDestroyer also contained numerous false flags designed to confuse attribution. For example, OlympicDestroyer shared significant code similarities to a variety of malware used by other notable APT groups, including a disk wiper used by the Lazarus group, a credential stealer used by APT3, an AES key generation function used by APT10, and lateral movement methods used by Sandworm [9]. Kaspersky found that the Lazarus group wiper's Rich header was identical to OlympicDestroyer's. However, upon further investigation, Kaspersky noticed that the metadata within OlympicDestroyer's Rich header suggested that it was compiled using Visual Studio 6, but other information in the file proved that it was actually compiled with Visual Studio 2010 [8]. Kaspersky concluded that OlympicDestroyer's Rich header was purposely replaced with that of the Lazarus group wiper's as a false flag [8].

VII. Packing and the Rich Header

In order to further test Webster et. al's conclusion about packers, we were able to obtain copies of most of the most common malware packers used today: ASPack, FSG, PECompact, Petite, RLPack, Themida, Upack, UPX and VMProtect. We ran these packers on malware samples from the APT1 malware data set and examined how each packer modified the files. We were able to conclude that ASPack, PECompact, Petite, Themida and UPX ignore the Rich header entirely, whereas FSG, UPack, VMProtect and RLPack modified the Rich header in some way. Upon packing malware samples with FSG, UPack, and VMProtect, the Rich header was found to be entirely corrupted and unusable. We believe that the corruption of the Rich header with these packers was not in a way to intentionally remove the Rich header; rather, it seems that the packer rewrites most of the area around the MS-DOS header with other information. RLPack seems to be the only packer that we tested that is aware of the Rich header. In each case, it replaced the file's Rich header with a different one. The byte sequence for the replacement Rich header is shown in the figure below.

FIGURE II. RLPack Rich Header Hexdump

```
5d 65 fd c8 19 04 93 9b  |]e......|
19 04 93 9b 19 04 93 9b  |........|
97 1b 80 9b 11 04 93 9b  |........|
e5 24 81 9b 18 04 93 9b  |.$......|
52 69 63 68 19 04 93 9b  |Rich....|
```

Fig. 2. Rich header hexdump of a malware sample packed with RLPack. Its original Rich header has been replaced.

TABLE I. Malware Packers and Their Impact on The Rich Header

Packers	Effects on the Rich Header		
	Rich Header Not Modified	***Rich Header Modified***	***Rich Header Purposefully Modified***
ASPack	**X**		
FSG		**X**	
PECompact	**X**		
Petite	**X**		
RLPack			**X**
Themida	**X**		
UPack		**X**	
UPX	**X**		
VMProtect		**X**	

VIII. RichPE Hash

As packing is becoming more commonplace, the effectiveness of Imphash and pehash are lessening. Webster et. al's research showed that most packers do not modify the Rich header and demonstrated that it is possible to cluster malware using the metadata contained within the Rich header. Therefore, the Rich header is an excellent source of features that can be used in a new metadata hash.

The simplest metadata hash that can be computed from the Rich header, which we call the Rich hash, is computed by using the the *ProdID*s, *mCV*s, and *Counts* within the Rich header as input to a cryptographic hash function. Our implementation of Rich hash uses the MD5 hashing algorithm. Unfortunately, we discovered that the *Count* fields of malware samples belonging to the same family are often very similar but not identical. If the Rich hashes of two malware samples with this property were computed, their digests would be different, which is not ideal. A different metadata hash that is not dependent upon exactly matching *Count* fields would have a much higher clustering capacity.

The next metadata hash attempt was identical to Rich hash, but all of the metadata was input into the ssdeep fuzzy hash rather than MD5. A fuzzy hash function is similar to a cryptographic hash function, with the notable difference that the digest of the hash can be used as a similarity metric. If two inputs to the fuzzy hash are similar, their digest will be as well. In the case of the Rich header, if two malware samples in the same family have very similar but not identical *Count* fields, their fuzzy hash digests will reflect that. Although the Rich fuzzy hash was effective at clustering malware by the metadata within their Rich headers, we found that our final metadata hash attempt was superior to it.

Rather than using a fuzzy hash as a workaround to the issue of the *Count* fields, we decided to take a different approach. Because the *Count* fields of similar malware samples are often similar but not identical, we decided to apply a transformation called *bitLen* to them before using them as input to the cryptographic hash. The *bitLen* transformation simply turns an integer into the minimum

number of bits needed to represent that integer. For example, suppose a malware sample has an entry in the Rich header with a *Count* of 22. The integer 22 is 10110 in binary, requiring 5 bits to represent it. Suppose that another malware sample of the same family has an identical Rich header, but the entry has a *Count* of 30 instead. The integer 30 is 11110 in binary, and it can also be represented in 5 bits. Therefore, the cryptographic hashes of the *ProdID*s, *mCV*s, and the *bitLen* transformed *Count*s of the two malware samples would be identical.

Although applying the *bitLen* transformation to each *Count* field increases the clustering capacity of Rich hash, it also decreases its precision. To offset the loss in precision, we decided to incorporate additional features from the PE header. We named the resulting metadata hash the RichPE hash. We previously noted that packing a malware sample often results in the metadata from the *IMAGE_SECTION_HEADER*s changing. Therefore, we limited the PE metadata features to ones from the *IMAGE_FILE_HEADER* and *IMAGE_OPTIONAL_HEADER*. After testing over a dozen different versions of the RichPE hash, we identified a set of metadata features that maximized its clustering capacity while keeping its false positive rate acceptable. The metadata features used in our current implementation of RichPE are shown in the figure below.

TABLE II. RICHPE HASH METADATA FEATURES

Header Name	Metadata Name
Rich Header	**ProdID of each entry**
	mCV of each entry
	bitLen(Count) of each entry
IMAGE_FILE_HEADER	**Machine**
	Characteristics
IMAGE_OPTIONAL_HEADER	**DLLCharacteristics**
	Magic
	Subsystem
	MajorLinkerVersion
	MinorLinkerVersion
	MajorOperatingSystemVersion
	MinorOperatingSystemVersion
	MajorImageVersion
	MinorImageVersion
	MajorSubsystemVersion
	MinorSubsystemVersion
	bitLen(ImageBase)
	bitLen(SizeOfImage)
	bitLen(SizeOfStackCommit)
	bitLen(SizeOfHeapCommit)

IX. RICHPE RESULTS

To evaluate the clustering effectiveness of RichPE, we chose to use a metric called Normalized Mutual Information, or NMI. NMI can be used to compare multiple clusterings of the same dataset. The NMI score of a set of clusters is a value between 0 and 1, with numbers closer to 1 representing the ideal clusters. We computed the NMI score for Imphash, pehash, Rich hash, and RichPE across a dataset of 1,000 malware samples attributed to 10 different APT groups. The NMI score for each metadata hash is shown in the table below.

TABLE III. METADATA HASHES AND NMI SCORES

Metadata Hash	NMI Ratings	
	Total Clusters	***NMI***
Imphash	**419**	**0.597**
pehash	**470**	**0.589**
Rich hash	**419**	**0.598**
RichPe	**378**	**0.604**

A metadata hash is useless if it clusters unrelated malware samples too frequently. We computed the Imphashes, pehashes, Rich hashes, and RichPEs of approximately 315,000 benign PE files and 58,000 malicious PE files. For the purpose of this test, we defined a false positive as any malicious file that has the same metadata hash with a benign file. The false positive rates of the four metadata hashes on this dataset are shown in the table below.

TABLE IV. METADATA HASH FALSE POSITIVE RATES

Metadata Hash	False Positive Rates
Imphash	**2.113%**
pehash	**2.039%**
Rich hash	**3.245%**
RichPE	**0.603%**

We have shown that RichPE is very effective at clustering malware and that it has a lower false positive rate than the industry standard metadata hashes Imphash and pehash. One of the major reasons for the success of RichPE is that it uses features that are less frequently affected by packing. By clustering malware using the RichPE metadata hash, analysts can more easily classify and attribute malware samples.

X. FUTURE WORK

There is a lot of remaining research to be done in regard to the Rich header. Although we have developed a proof-of-concept implementation of RichPE, there are further improvements that can be made to it. The current RichPE implementation relies on the pefile Python library, which is slow and can fail if a malware sample has invalid metadata in its PE header. We plan to implement a more efficient version of RichPE that does not use the pefile

library. We have also considered reducing the number of metadata features used in the RichPE hash, which would increase its NMI but also increase its false positive rate.

We also plan to get access to more malware for the purposes of evaluating the NMI and false positive rate of the four metadata hashes. In addition, we wish to survey more packers to determine which do or do not modify the Rich header. We also plan to track which other metadata fields in the *IMAGE_FILE_HEADER*, *IMAGE_OPTIONAL_HEADER*, and *IMAGE_SECTION_HEADER*s are the most and least frequently modified by packers. This information will inform us about any other possible improvements we can make to the RichPE hash.

Finally, we are still in the process of reverse-engineering the compiler in order to discover more about how certain *ProdID* values in the Rich header are generated and what they mean. We also plan to research methods for spoofing the Rich header without it being able to be detected. This research may allow us to identify malware samples that are spoofing their Rich headers.

XI. Conclusion

We have shown a viable method for metadata comparison based on the Rich header across large datasets assisting with the problem of malware attribution. Our metadata hash leverages the added value from the Rich header to achieve more accurate results than industry standard hashing algorithms such as Imphash or pehash. We believe that as a result of the Rich header's ability to fingerprint a specific computer a binary was compiled on, our hash can be used with high confidence to tie disparate malware samples to each other and to their authors. We believe that our approach is a powerful tool for helping to solve the attribution problem, and that much more future work is possible in this area.

Acknowledgment

We would like to extend our thanks to Matt Elder, Bill La Cholter, Haibin Zhang, Dr. Charles Nicholas, Johns Hopkins Applied Physics Laboratory, and the Shadowserver Foundation for their continued support.

References

[1] "File Statistics during Last 7 Days." VirusTotal, www.virustotal.com/en/statistics/.

[2] Maloney, Sarah. "Attack Attribution: It's Complicated." Cybereason, Cybereason, 30 Mar. 2017, www.cybereason.com/blog/attack-attribution-its-complicated.

[3] Plachy, Johannes. "Portable Executable File Format." Krzysztof Kowalczyk, 27 July 2018, blog.kowalczyk.info/articles/pefileformat.html.

[4] Mandiant. "Tracking Malware with Import Hashing." FireEye Threat Research Blog, FireEye, 23 Jan. 2014, www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[5] Wicherski, Georg. 2009. peHash: a Novel Approach to Fast Malware Clustering. In Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more (LEET'09). USENIX Association, Berkeley, CA, USA, 1-1.

[6] Vigna, Giovanni. "When Malware Is Packing Heat." Lastline, 17 Oct. 2017, www.lastline.com/labsblog/malware-packing/.

[7] Webster G.D. et al. (2017) Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage. In: Polychronakis M., Meier M. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2017. Lecture Notes in Computer Science, vol 10327. Springer, Cham.

[8] "The Devil's in the Rich Header." Securelist - Kaspersky Lab's Cyberthreat Research and Reports, Kaspersky Labs, 8 Mar. 2018, securelist.com/the-devils-in-the-rich-header/84348/.

[9] Rascagneres, Paul, and Martin Lee. "Who Wasn't Responsible for Olympic Destroyer?" Cisco's Talos Intelligence Group Blog, Cisco Talos, 26 Feb. 2018, blog.talosintelligence.com/2018/02/who-wasnt-responsible-for-olympic.html.