peHash: A Novel Approach to Fast Malware Clustering

Georg Wicherski
RWTH Aachen University
gw@mwcollect.org

Abstract

Data collection is not a big issue anymore with available honeypot software and setups. However malware collections gathered from these honeypot systems often suffer from massive sample counts, data analysis systems like sandboxes cannot cope with. Sophisticated self-modifying malware is able to generate new polymorphic instances of itself with different message digest sums for each infection attempt, thus resulting in many different samples stored for the same specimen. Scaling analysis systems that are fed by databases that rely on sample uniqueness based on message digests is only feasible to a certain extent.

In this paper we introduce a non cryptographic, fast to calculate hash function for binaries in the Portable Executable format that transforms structural information about a sample into a hash value. Grouping binaries by hash values calculated with the new function allows for detection of multiple instances of the same polymorphic specimen as well as samples that are broken e.g. due to transfer errors.

Practical evaluation on different malware sets shows that the new function allows for a significant reduction of sample counts.

1. Introduction

Data collection and therefore honeypot development has been a big challenge in the past. As of today various differnt approaches to malware collection with the help of honeypots exist [1]. In addtion, there are effective tools to automate malware analysis [2]. However, current honeypot setups suffer from gathering multiple binaries with distinct message digest sums that belong to the exact same specimen and therefore pollute malware databases as well as automated analysing systems.

Current Anti-Virus solutions are too slow and inaccurate to scan each incoming sample and to blacklist based on this data, as shown later. Additionally, polymorphic malware is likely to hit the database again in the future, so a reactive blacklisting approach seems inferior to automated clustering of malware. To achieve fast clustering, a hash function that generates the same hash value for any two instances of the same specimen has been developed, thus a same hash metric can be used for clustering.

To tackle these problems, we developed a generic hash function for Portable Executable [10] files that generates a per-binary specific hash value based on structural data found in the file headers and structural information about the executable's section data. This allows clustering of binaries based on a same hash metric, resulting in clusters that group instances of the same polymorphic malware specimen.

We tested this approach on the database of the mwcollect Alliance [7] database, which at that time was heavily polluted by a huge amount of Allaple [3] instances. peHash was able to significantly reduce the pollution from an constantly increasing amount of Allaple samples to a static number of specimens.

1.1. Related Work

Hashing by itself is not a new approach to cluster malware. Experiments with the *spamsum* and *mrshash* hashes have shown promising results towards clustering of unpacked malware [4]. Both hashes can be computed in $O(n)$ complexity, however for proper clustering they rely on an edit distance comparison that has to fall below a certain threshold for a cluster to be formed. Edit distance requires dynamic programming and each hash needs to be compared with each other hash. Under the assumption that the hashes being compared have a constant length, this still leads to a complexity of $O(n^2)$. Realistic assumptions about performance, however, have to consider a significant linear factor c due to the edit distance calculations. Using binary space partitioning, clustering can however be achieved in $\Theta(n \log n)$ for most real-world input sets [12].

Another approach to malware clustering is the use of *n-grams Signatures* [5] as known from natural language processing. Calculating these signatures is linear in the sample size with a feasible linear factor c, but since n-grams do not define a distance metric, this approach cannot be solved in better than $\Theta(n^2)$.

Compared to the above mentioned approaches, the worst-case performance of peHash's hash-equality clustering using simple binary search is $O(n \log n)$. Additionally, these approaches consider the given input samples to be raw binary data without any domain specific assumptions. Polymorphic or packed samples are not clustered correctly, as compression and encryption modify the section contents significally. Opposed to this, peHash is tailored to the specific properties of windows malware and thus allows for the proper clustering of aforementioned hard cases.

A totally different approach is taken by VxClass that first needs to unpack the sample if it was packed with an executable packer and then statically examines the inter-procedural callgraph and the intra-procedural flow graphs [6]. Using annotated graph matching, similiarities in the code flow can be determined and a similiarity metric for malware specimens derived. However, the average runtime per sample is declared to be about five minutes, making this very accurate clustering approach unfeasible for any real-time filtering of polymorphic specimens.

In general, the above mentioned approaches are all either not suited for usage in the application area where peHash was designed for: near real-time clustering of specimens in high-volume malware collections.

2. peHash Function Design

Since malware is often packed [8] and specifically polymorphic malware uses changing decryptor stubs, no assumptions about the actual code but the stub can be statically made. Additionally, polymorphic malware or packers often insert random instructions in the stubs, which would again result in a unique hash value for each instance of a specimen. As even data sections could be encrypted or contain code that is executed at runtime, looking at the contents of the sections in a Portable Executable and deriving information from it is not feasible in this context.

As this hash function should be of low computational complexity for the average malware binary, only a limited set of quickly gathered or calculated information can be used. Specifically, even only partially emulating the malware's code is not an option as even in small decryption stubs loops can occur, of which the runtime length cannot be easily predicted. Even if emulation is bounded by a time or instruction count limit, it would be easy to prepend a relatively long idle time to any meaningful actions, thus that emulation results are not useful for clustering.

However, every space linear, information preserving function used to modify section contents by definition results in approximately the same Kolmogorov Complexity [9] of this section. This is specifically true for static key encryption routines like a simple XOR and metamorphic code rewrites. Compression of data results in a maximal Kolmogorov Complexity, but multiple instances of the same specimen are likely to use the same compression algorithm which should then result in the same complexity for distinct instances. Since Kolmogorov Complexity is a construct of theoretical informatics and cannot generically be calculated, bzip2 compression ratio is used as an approximation [9]:

$$\frac{\text{Length}(\text{bzip2}(Data))}{\text{Length}(Data)} \in (0..1] \subset \mathbb{R}$$

Because this value is directly included in the hash value for each section in the Portable Executable and we want to simply group by exact hash matches, this value needs to be trimmed. Our experiments with different ranges using different malware sets (see below) have shown that scaling the result the bzip2 compression ratio to $[0..7] \subset \mathbb{N}$ leads to the best matches. Smaller ranges result in too big clusters that contain different specimens and bigger ranges do not improve accuracy of clusters but sometimes resulted in valid clusters being further broken up.

2.1. Structural Properties

Apart from this data specific information, additional data needs to be taken into account to have sufficient distinction between binaries. Today's polymorphic malware specimens share the same structural Portable Executable properties, as the code that generates new instances does not perform any re-linking but only generation of a new decryptor stub and re-encryption with a new key. Thus, the following Portable Executable properties [10] are taken into account as well:

- Image Characteristics: General flags for the Portable Executable, e.g. whether the given file is a DLL or can only be run on a single processor machine.

- Subsystem: Indicates the Windows Subsystem this binary is to be run in, such as GUI, CLI or device driver.

- Stack Commit Size: The initial size of program stack to be allocated in bytes. This value is rounded up to a value divisible by 4096, Windows' page boundary, before inclusion in the hash as the Windows Portable Executable Loader does the same.

- Heap Commit Size: Initial size of program heap to be allocated, also rounded up to page boundary size.

For each section in the Portable Executable, the following structural information is included:

- Virtual Address: The address, the section's content is going to be loaded to or memory is to be allocated at for a `.bss` section.

- Raw Size: Size of the section in the Portable Executable file itself; can be smaller than the actual size occupied in memory after loading due to rounding to page boundaries.

- Section Characteristics: Section flags describing initial privileges for the allocated memory, such as reading, writing and execution of code. Also contains information about alignment and whether the section contains unitialized data as a `.bss` section.

Since not all bits of all values are actually used or contain useful information, only selected parts of the values are included in the hash. Modifications of these significant bit locations indicate significant changes in the malware binary, reducing the risk that the usefullness of this hash function for a same hash metric is destroyed by single bits easily changing. The upper 8 bit for the 32 bit stack and heap commit sizes are almost always zero. Furthermore, some values might differ in the lower bits due to small size changes in polymorphic malware and must be discarded to allow for exact matching of hash values. The following pseudocode describes the exact generation for the hash value from the global properties, where $v[8..24]$ means bits 8 to 24 of value v and $\oplus$ means XOR:

$$\begin{aligned}
hash[0] &:= characteristics[0..7] \\
&\quad \oplus characteristics[8..15] \\
hash[1] &:= subsystem[0..7] \\
&\quad \oplus subsystem[8..15] \\
hash[2] &:= stackcommit[8..15] \\
&\quad \oplus stackcommit[16..23] \\
&\quad \oplus stackcommit[24..31] \\
hash[3] &:= heapcommit[8..15] \\
&\quad \oplus heapcommit[16..23] \\
&\quad \oplus heapcommit[24..31]
\end{aligned}$$

Additionally, for each section, the following subhash is appended to the hash:

$$\begin{aligned}
shash[0] &:= virtaddress[9..31] \\
shash[2] &:= rawsize[8..31] \\
shash[4] &:= characteristics[16..23] \\
&\quad \oplus characteristics[24..31] \\
shash[5] &:= kolmogorov \in [0..7] \subset \mathbb{N}
\end{aligned}$$

The part of the hash that relies on the properties of the Portable Executable can be calculated in $O(1)$ as only a fixed amount of data is fetched (there is a constant upper limit of sixteen sections) and the Kolmogorov Complexity approximation can be done in bzip2 runtime.

Obviously, given only the hash value it is trivial to manually craft collisions for this hash function. To prevent this from happening for a published hash value, the last step is to calculate the SHA1 value of the above hash buffer and use this as the final hash value. Thus, a peHash value can be safely published without anyone being able to create collissions unless a bruteforce approach over the peHash value space is chosen. This would involve trying about 2^{40} possibilities [1] for a two section binary, resulting in a search time of 680 years if one SHA1 calculation takes 20ms. However, given a Portable Executalbe, it is still trivial to craft a collission. Additionally, running SHA1 on the resulting hash value provides constant length hashes, no matter how many sections are contained in the executables, which is desirable for storage in database and faster comparisons.

2.2. Entry Point and Imports

Both Entry Point information and imports are intentionally not included in the hash function. As most executables use 40000h as Image Base [2], only the lower 16 bit of the entry point are likely to differ for any two given executables. However, most polymorphic or packed malware uses position independent decryptor stubs that can easily be relocated by changing the virtual address of a section without any noteworthy linking efforts. Thus, this value can too easily be changed for each instance of a polymorphic specimen and hence should not be included in the hash function.

Additionally, most packers specify a misleading Import Address Table (IAT) with random imports to confuse malware analysis software. The APIs really used are then manually located from the export tables of manually loaded libraries and a new IAT is generated for the unpacked part of the malware in memory. Thus, this information can also be easily changed without any noteworthy efforts and should also not be included in the hash function.

3. Evaluation

To validate the performance and usefulness of this hash function, we implemented it in C and

[1]Not the whole space of 2^{64} possibilities for the 8 bytes needs to be searched since some bits for the global header are reserved and thus constant.

[2]Memory address all other virtual addresses are relative to.

Cluster Size	mwcollect	Arbor Networks
1	7109	16543
2-9	3165	4104
10-99	549	611
100-499	70	71
500-999	19	4
1000-4999	18	8
5000+	7	2

Table 1. Cluster Sizes for different Datasets

tested it on three private malware sets that only contained samples known to be malicious:

1. The mwcollect Alliance Database, which at the time of testing contained 184538 unique samples by SHA512 for which a peHash could be calculated. The malware in this data set is collected by Nepenthes [1] sensors and therefore only autonomously spreading malware that exploits network vulnerabilities or file infector malware that infected such malware.

2. A selection of 90105 unique samples by MD5 from the Arbor Networks malware database. This set included all kinds of malware found on the internet, including malware that spread through spam and also manual submissions.

For these given data sets, the samples fall into 10937 and 21343 clusters respectively. For verification, all samples were scanned with the ClamAV antivirus software and a cluster was considered a possibly broken cluster if at least two different signatures where triggered by malware in a cluster. For the mwcollect Alliance dataset, 2.58% of the clusters were possibly broken; for the Arbor Networks dataset 1.51% of the clusters were possibly broken. However manually investigating all these 282 and 322 possibly broken clusters revealed that signatures for the same malware which had different names, such as 'W32.Virut.Gen.C' and 'W32.Virut.Gen.D' were

File	MD5	Size
diantz.exe	48734e9b45dca36e8a...	85504
makecab.exe	2740dc2fbefaddb891f...	85504
find.exe	09b4e22c86f7e9f1e5...	9216
print.exe	76b96ed5304319f208...	9216
subst.exe	77847ef3cec784b137...	9216
bootvrfy.exe	c2ab77d9dc66447dc1...	5120
comrereg.exe	908f0eda6a49625f98...	5120
dcomcnfg.exe	1178cd20b90936837d...	5120

Table 2. Broken Clusters for Known Good System Binaries

triggered. Thus, *none* of the candidates for broken clusters was infact broken and the false positive rate after manual correction of the ClamAV analysis is 0.

Apparently, only minimal modifications to a malware binary can cause a different A/V signature to be triggered, further underlining that clustering by A/V names is not an option. Additionally, A/V names seem to use arbitary suffixes that do not seem to be related to the actual malware generation itself. Specficially, clusters of malware that were additionally infected with the Virut file infector malware [11], seemed to trigger the original malware's signature and Virut signatures at random (although deterministically per file). As such a file is both infected with Virut and still the original malware, both signatures are correct and the cluster is intact. Clearly, just triggering is a much more important goal for A/V software than providing specific information about the prelevant threat. peHash is a solution that can help by revealing relations between new samples and already known threats.

Although twice the binary count, the mwcollect Alliance database has only about half of the Arbor Network's cluster count. This meets our expectations as the mwcollect Alliance database was heavily polluted with Allaple instances, being responsible for the seven 5000+ samples clusters in the mwcollect Alliance dataset.

Interestingly, there is also a big amount of 2-9 samples clusters as depicted in Table 1. The 2-9 samples cluster's for the mwcollect Alliance dataset holds 3.49 samples on average. This is due to the fact that sometimes transfer errors occur during infection that result in some missing or changed bytes which have direct impact on a binary's message digest sum but do not change the peHash of a binary. Therefore, peHash does not only help in clustering polymorphic malware but also in detecting broken copies of already known threats and therefore further reduce analysis overhead.

To measure the clustering on known good binaries as well, we also hashed 322 executables from a vanilla Windows XP installation. The hashing resulted in 311 different clusters most of them holding a single binary and seven false positive clusters holding two to three binaries, each. Some exemplary clusters are shown in Table 2. Files in broken clusters all share the same size and manual analysis revealed that they also have identical global and per section characteristics. Differentiating between them is only possbile by looking at the actual code or the imports, which is generally not possible for peHash. This implies that it is best to use peHash only for supposedly bad executables, as it is the case with the test datasets. Promising approaches to tackle these problems by identifying safe to use imports are described under Future Work.

3.1. Performance

Since for the mwcollect Alliance, most automatic sample analysis is only carried out for one sample per peHash cluster, only 5.92% of the samples had to be analyzed saving 83.1% of analysis time. The average time to calculate a peHash for any binary from the mwcollect Alliance dataset was 511.56640625 ms with the bzip2 compression taking the majority of the time, which is a considereable improvement to the

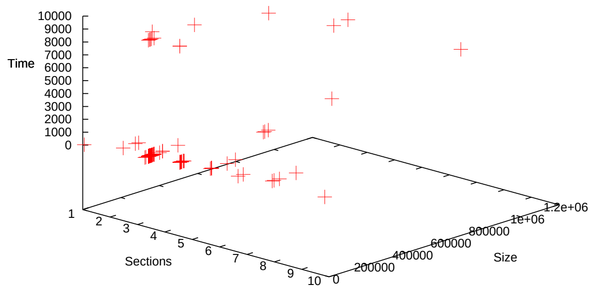

Figure 1. peHash performance in ms

saved 120-180 seconds, a sandbox run for each sample would usually have taken.

As can be seen in Figure 1, performance is not related to binary size or section count. Instead, most of the time, the peHash calculation just takes about 33-35 ms. There are some outliers however, where the bzip2 compression of a single section took up to 10 s which raise the average computation time significantly. The fact that there is no computation times between 2 s and 9 s is due to the implementation of bzip2 used.

These measurements were taken on a single Xeon CPU 2.80GHz core with a branch prediction optimized implementation compiled using GNU g++ 4.1 with the `-O2` compiler flag. The time to fetch the samples from the database was not taken into account.

4. Conclusions

The vast amount of new seemingly unique malware samples each day is a huge problem even to automatic analysis systems. However, with peHash there is a performant solution to tackle this problem for polymorphic and slightly broken samples. We have shown that peHash can accomplish correct clustering for large test sets even though only using very basic information from Portable Executables and making almost no assumptions about the sections' contents themselves but only about their information richness. A posteriori analysis with A/V signatures shows that peHash provides similar precision without static signatures and could succesfully be used for monitoring new threats.

It is, however, obvious that peHash cannot be used to cluster variants of malware families as for that the code structure itself would have to be analyzed which due to usage of packers is totally invisible to peHash. Time to unpack or even emulate binaries is however saved.

Surprisingly, we could not find any clusters that contained samples packed with the same packer but containing different payload. Some additional evaluation based on sandbox analysis for all samples in a cluster should be done in the future to verify that such clusters really do not exist.

4.1. Future Work

peHash can already successfully cluster the current in-the-wild set of malware. However, with more awareness about this research on the black-hat side, collisions might occur. Packers might for example be modified to always result in the same peHash due to usage of the same executable structure and a strong compression algorithm in combination with filling the gaps with random data, resulting in the maximum Kolmogorov Complexity for every binary.

One way to cope with this might be to actually integrate disassembly analysis around the entry point in the hash. However, randomly inserted instructions might cause problems here. As these instructions often only operate on registers, doing statistics on instructions working on memory operands or just the memory operands relative to the image base themselves could be a step in the right direction. Memory operands that reference data in the import section of a binary can indicate valid imports that are actually used, which can then also be used in the hash, whereas they would have been left out to avoid misclustering due to misleading imports as described above.

Another way to cluster samples based on the characteristics above would be to interpret these features as a vector, multiply this vector with a weighting vector and then measure distance between all vectors. If the distance goes below a certain threshold, two samples belong to the same cluster. However, this requires a comparison of all vectors with each other, resulting in $O(n^2)$ runtime for n samples whereas a direct match comparison can be done in amortized $O(n)$ runtime using a hashtable. Nebula [12] is such a tool that allows for easy clustering based on weighted vectors as a metric, extending it to work on Portable Executables should be an easy task. Similiar vector based approaches on disassembly of unpacked malware have already been discussed by Walenstein [13].

Acknowledgements

The author would like to thank Niklas Schiffler and ShadowServer for testing and providing valuable feedback to the C implementation of peHash, Jose Nazario for testing and providing evaluation data for the Arbor Networks dataset. Also he would like to thank Felix Leder, Markus Kötter, Ulrike Meyer, Mark Schlößer and Tillmann Werner for valuable suggestions while writing this paper.

References

[1] Thorsten Holz, Maximillian Dornseif, Felix Freiling, Paul Baecher, Markus Koetter: *The Nepenthes Platform*, 9th International Symposium on Advances in Intrusion Detection (RAID 2006)

[2] Carsten Willems, Thorsten Holz, Felix Freiling: *Toward Automated Dynamic Malware Analysis Using CWSandbox*, IEEE Security & Privacy, 1540-7993/07

[3] F-Secure: *Allaple.A*, `http://www.f-secure.com/v-descs/allaple_a.shtml`

[4] Vassil Roussev, Golden G. Richard III, Lodovico Marziale: *Multi-resolution similarity hashing*, Digital Investigation Journal

[5] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, Ray Sweidan: *N-gram-based Detection of New Malicious Code*, Proceedings of the 28th Annual International Computer Software and Applications Conference

[6] Zynamics: *VxClass*, `http://www.zynamics.com/index.php?page=vxclass`

[7] *The mwcollect Alliance*, `https://alliance.mwcollect.org/`

[8] ShadowServer: *Packer Statistics*, `http://www.shadowserver.org/wiki/pmwiki.php?n=Stats.PackerStatistics`

[9] NIST: *Kolmogorov Complexity*, `http://www.nist.gov/dads/HTML/kolmogorov.html`

[10] Microsoft: *Portable Executable Format Specification*, `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`

[11] Symantec: *W32.Virut.A Technical Details*, `http://www.symantec.com/security_response/writeup.jsp?docid=2006-051402-1930-99&tabid=2`

[12] Tillmann Werner: *Nebula*, `http://nebula.mwcollect.org/`

[13] Andrew Walenstein: *Exploiting Similarity Between Variants to Defeat Malware*, Blackhat DC 2007