A Tool to Compute Approximation Matching between Windows Processes

Ricardo J. Rodríguez
Centro Universitario de la Defensa
Academia General Militar, Zaragoza, Spain
rjrodriguez@unizar.es

Miguel Martín-Pérez, Iñaki Abadía
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, Zaragoza, Spain
{miguelmartinperez, 685867}@unizar.es

***Abstract*—Finding identical digital objects (or artifacts) during a forensic analysis is commonly achieved by means of cryptographic hashing functions, such as MD5, SHA1, or SHA-256, to name a few. However, these functions suffer from the *avalanche* effect property, which guarantees that if an input is changed slightly the output changes significantly. Hence, these functions are unsuitable for typical digital forensics scenarios where a forensics memory image from a likely compromised machine shall be analyzed. This memory image file contains a snapshot of processes (instances of executable files) which were up on execution when the dumping process was done. However, processes are relocated at memory and contain dynamic data that depend on the current execution and environmental conditions. Therefore, the comparison of cryptographic hash values of different processes from the same executable file will be negative. Bytewise approximation matching algorithms may help in these scenarios, since they provide a similarity measurement in the range $[0, 1]$ between similar inputs instead of a yes/no answer (in the range $\{0, 1\}$). In this paper, we introduce `ProcessFuzzyHash`, a Volatility plugin that enables us to compute approximation hash values of processes contained in a Windows memory dump.**

***Keywords*-bytewise approximate matching, forensic memory analysis, Windows, Volatility**

I. Introduction

Computer forensic analysts usually try to find identical digital objects (or *artifacts*, defined as an arbitrary byte sequence) as a source of evidences for a posterior analysis. For instance, they can look for the presence of known malicious software (*malware*) in a forensic disk image as an evidence of a compromise computer. When some malware files are detected, they are analyzed in detail to measure the impact of the threats found.

A common approach to compute the similarity between artifacts is to use cryptographic hash functions like MD5, SHA-1, or SHA-256. A cryptographic hash function is an algorithm that takes a relatively (arbitrary) large amount of input and returns a fixed-size hexadecimal string. The output string is called the *hash value* or *digest* of the input. Hence, a hash value of an artifact serves to unequivocally identify it. Cryptographic hash functions are designed to be one-way function [1], i.e., it is easy to compute on every input, but hard to compute its inverse function. That is, given a hash values it is difficult to know the input that generates such an output.

A desirable property of cryptographic hash functions is the avalanche effect property [2], which guarantees that the hash values of two similar, but not identical, inputs (e.g., inputs in which only a single bit is flipped) produce radically different outputs. Hence, cryptographic hash functions are commonly used for data integrity and file identification of a seized device [3].

However, the avalanche effect property has some drawbacks since an active adversary may easily defeat this detection and keep their files hidden by just flipping a single bit. To overcome this situation, approximate matching has emerged in the recent years as a prominent approach that is more robust to active adversaries than traditional hashing [3]. Approximate matching identifies similarities between two digital artifacts providing a measure of similarity, in the range of $[0, 1]$. Hence, it is used to find artifacts that resemble each other or to find artifacts that are contained in another artifacts [4], providing a percentage of similarity among them.

An approximate matching method becomes especially suitable for the analysis of forensic memory images, which contains a snapshot of the set of processes up on execution when the memory image was acquired. By the intrinsic characteristics of the underlying machine's OS, different executions of the same binary file generates similar but not identical processes, residing in the addressable space memory of the machine. Hence, although cryptographic hash functions are useful to identify the binary file in a forensic disk image, they become unsuitable for the same purpose when analyzing a forensic memory image from the same machine.

An approximate matching method is classified based on what is considered to perform the similarity analysis [4]: *bytewise*, when the method relies only on the byte sequence of the digital artifact (i.e., no structures or meaning of the byte stream are considered); *syntactic*, when the method relies on internal structures present in digital artifacts under analysis; and *semantic*, when the method uses contextual attributes of the digital artifact to interpret it. In this paper, we focus on bytewise approximation matching algorithms (also known as fuzzy hashing) from a syntactic approach, since we consider process sections as input data for the algorithms.

Let us illustrate the avalanche effect problem in processes by means of a running example. Consider the "hello world" program (coded in C) shown in Listing 1 that has a `getchar` function as a way to wait for user interaction before exiting, compiled with GNU `gcc` version 5.3.0 running in a Windows 7 SP1 Professional. Now, we execute the binary file three times in the same machine, and every time we dump the corresponding process to a file. The MD5 hashes of each dumped process file are, namely:

- `c5045fa95f880dc1cf09c97ae8d32e28`
- `37304f15fe397b51927ce3ba247ca397`
- `016e7557c48cb91feb487b6e0919037e`

As it can be clearly seen, there is not any match between any of the hashes of the dumped process files. However, if we compute the ssdeep hash instead (a bytewise approximate matching algorithm), we obtain similarities that range between 97% to 99%, and up to 100%, depending on the byte stream of the dumped process files that we analyze. For instance, the ssdeep hash of the byte stream that contains the binary code executed is exactly the same value, `384:fb02XGJLkOj40sqJnwCQoVrKcBdgE/daL07ticdJ5bjn+lUpNNK/7RV:fxGGOdwCJxRuYDJJraR`.

Listing 1: A "hello world" C program.

```c
#include <stdio.h>

int main(int argc, char *argv[]){
    printf("hello world!\n");
    getchar();
    return 0;
}
```

In this paper, we focus on processes executed on top of Windows OS. The main reason of the differences regarding cryptographic hashes relies on the own nature of Windows processes. While an executable file is a *static binary file*, a process of such an executable file is a *dynamic binary file*. There exist several parts in a process which strongly depend on the current time of execution, such as the content of the stack or the heap. Besides, relocation and relative addresses of import data or functions are likely to differ between subsequent executions. We provide a more technical description of the underlying reasons causing these differences in Section II-B.

In this paper, we introduce `ProcessFuzzyHash`, a plugin of the Volatility framework that allows us to compute approximation matching hashes of the processes inside a memory dump. Volatility [5], published in 2007, is a framework written in Python commonly used in computer forensics. In particular, Volatility provides a set of plugins to easily analyze memory dumps obtained from machines up on execution.

This paper is organized as follows. Section II introduces previous concepts needed to follow the rest of the paper. Related work is covered in Section III. Then, Section IV describes the architecture of `ProcessFuzzyHash` and how it works in detail. Finally, Section V concludes the paper and states future work.

II. Previous Concepts

Here, we first describe the format of Windows OS executables files and then the format of Windows processes.

A. *Portable Executable File Format*

The Portable Executable (PE) [6] file format is the file format adopted by the Windows OS executable files. Being a standard for packaging executable code, it is not only used by executable files (i.e., files with *.exe* extension) but also by other kind of (executable) files such as dynamic link libraries (*.dll* extension), screen savers (*.src* extension), or either typographic fonts files (*.fon* extension), to name a few. A PE file is divided into three main headers [7]: the DOS header, the NT header, and section headers.

In particular, the PE headers are composed as follows. First, there appears a `MS-DOS HEADER`, divided into a 64-byte size field named `IMAGE_DOS_HEADER` and an optional DOS stub content. Although this header is a reminiscent component from the MS-DOS era, it is still mandatory for any Windows executable file. It mainly contains the magic number "MZ" plus a pointer to the relative address where the next header starts. The optional DOS stub content contains the minimal code to report an error message when executing a Windows executable file in a MS-DOS OS.

The second PE header, `IMAGE_NT_HEADERS`, is divided into two parts: the `IMAGE_FILE_HEADER`, which contains metadata regarding the executable file, such as the particular machine for which it was compiled (Intel x86, IBM PowerPC, ARM, etc.) or the compilation timestamp (automatically set by the compiler), as well as other important data to allow Windows to correctly parse and execute the file, such as the number of sections or its characteristics; and the `IMAGE_OPTIONAL_HEADERS`, which additional information required for the correct execution of the file, such as the entry point address (first instruction to execute when the executable file is loaded), the stack size, or the heap size, among other.

Finally, the section headers are conformed by multiple `IMAGE_SECTION_HEADER` structures. Every one of these structures defines data regarding a specific section of the executable file, such as the names, the virtual(raw) addresses and sizes, and its characteristics (readable, writable, and/or executable). Data contained in each section appears after the section headers. For instance, an executable file can be composed of a *.text* section, which contains the binary code to be executed; a *.data* section, which contains the read-only data of the binary (e.g., string and numerical constant values); and a *.rsrc* section, which contains other read-only data such as icons, configuration files, or images.

B. Structure of Windows Processes

In essence, a Windows process is the representation of a executable file (or program) when the program is upon execution. The main difference between a program and process relies on the static and dynamic nature of the binary code: while a program is a static sequence of instructions (located in any device), a process is a container for all resources required (e.g., other files, peripheral devices, or sockets, among others) for the execution of the program. Analogously to object-oriented programming, the program is a "class" while the process is an instance of such a class, that is, an "object" of the "class".

In particular, a Windows process is seen as a thread-container [8]. At a high-level abstraction, a Windows process consists of a (private) virtual address space, the executable program that generated the process (copied into the address space of the process by the Windows PE loader), a list of all required OS resources, access tokens (they define the security context of the process), a unique process identifier and, at least, one execution thread.

As we highlighted in previous section, two Windows processes of the same executable file show substantial differences when comparing their memory content. This is mainly motivated because of how the Windows PE loader works. The Windows PE loader is the mechanism that, among other things, maps into memory an executable file that resides in disk prior its execution. When loaded, any relative address to import data or functions are solved and hence, the binary content associated to those data may differ. For instance, any dynamic link library associated to the executable file is almost surely mapped into different memory addresses than previous executions, specially when executing after booting. This issue is specially performed by Address Space Layout Randomization (ASLR) [9], [10], a defense mechanism to prevent control-flow hijacking attacks [11]. In this kind of attacks, the attacker diverts the control flow of a vulnerable process. In order to control the execution flow, the attacker must know certain memory locations that contains pieces of code of interest for the attacker's goal. Hence, memory addresses of libraries are randomized to make more difficult a successful attack when ASLR takes in place.

Besides ASLR, the Windows image loader may relocate the image file of the executable file when mapped to memory prior execution. Therefore, code and data in the program are adjusted to reflect the assigned addresses.

Furthermore, note that some memory zones contain dynamic data, such as the stack or the heap of the process. These structures are strongly dependent on the execution of the process since their data vary very often when the process executes. Hence, two processes will almost surely present substantial differences in the stack and heap contents unless it is guaranteed that the capture of these contents is done at the same moment of time after the executable file was launched and under the same environmental conditions (which is very unlikely to occur).

III. Related Work

In [12], the authors studied the similarity of binaries using bytewise approximation matching. Namely, the algorithms evaluated were `ssdeep` and other tools such as *peHash* and *Imphash*, explained later. Unlike our approach, again, the authors only considered binaries in its static form. Our tool also allows us to compute other bytewise approximation matching algorithms instead of *ssdeep*.

Another work that used approximation matching algorithms is [13]. There, the authors first converted binary files into image format files and then used a bytewise approximation matching algorithm to detect similarities between known malware binaries from different malware families. Hence, fuzzy hashing techniques were used as a cluster mechanism. Since that approach is complementary to ours, as further research we aim at exploring it, but using dumped process files instead.

It is worth also mentioning other tools outside the Volatility framework that make use of bytewise approximation matching techniques. In this regard, *peHash* [14] allows to obtain hash values from Windows binaries. In particular, it uses several properties of the Windows PE format (such as section flags, virtual address, or initial stack size, among others) as input data. Similarly, *Imphash* [15] generates a hash considering as input the import functions of the Windows binary, under the assumption that similar binaries are likely to import the same functions – which is false of the import functions are obfuscated. Note that our tool also considers the PE properties and import data, since we can compute hash values of the sections into the dumped process file that contains such data. Again, these tools are focused on binary files and not on dumped process files, as our approach does.

Malfunction [16] is a set of tools for cataloguing and comparing malware at a function level. Based on Radare2 [17], it computes hash values of every function detected into the binary and stores them to later comparison. Although focused on binary files, this approach is complementary to ours and deserves further research when applied to dumped process files. Finally, *Binwally* [18] is a tool that enables us to compare binary files using `ssdeep` algorithm. Again, it considers only binary files. Besides, our tool also supports more approximate matching algorithms as well as `ssdeep`.

In summary, as far as we are aware, no other work or tool was found regarding the computation of fuzzy hashes in dumped process files. As shown, all related works focused on computing fuzzy hashing of binaries in static form, i.e., binary files prior execution. We believe our tool can specifically help forensic analyst to discriminate between legitimate versus malicious processes that attempt to appear legitimate when analyzing a forensic memory image in a fast and easy manner.

IV. Tool Architecture and Description

In this section, we introduce the architecture of `ProcessFuzzyHash` and how it operates in a high-level description, providing as well examples of its execution.

During the initial analysis phase, we considered two (non-functional) requirements that `ProcessFuzzyHash` shall fulfil: (1) `ProcessFuzzyHash` shall make use of Windows process structures contained in a dumped memory image file; and (2) the tool shall be portable, so it can be executed in as many operating systems as possible. Considering these requirements, we opt for developing a tool with Python language that makes use of the Volatility framework, given that Volatility is already able to handle Windows process structures contained in a dumped memory image file. Besides, we decided to develop `ProcessFuzzyHash` as a Volatility plugin instead of a stand-alone application to extend its capabilities and therefore contribute to the community of users (and developers) of Volatility.

`ProcessFuzzyHash` operates in two different manners: *hash generation* and *hash comparison*. A high-level description of `ProcessFuzzyHash` is depicted in Figure 1. As input, our tool needs a dumped memory image file for a machine running any Windows OS accepted by Volatility. Since our tool is a plugin of Volatility, it is executed as a command of Volatility framework, i.e., by means of the Python script `vol.py`. The output of our tool depends on the operation being performed. In the sequel, we briefly describe both operations.

`ProcessFuzzyHash` accepts a set of parameters to customize its operative. For instance, the specific process (or processes) for which approximate matching hash values shall be generated can be provided by process name (full or partial match is supported) or by process ID. Similarly, the approximation matching algorithms to use are provided as a comma-separated argument. Other parameters regarding performance, such as multithreading hash generation or specification of temporal folder, or regarding readability, such as giving results as human-readable values, can be as well provided.

Our tool currently supports four approximate matching hashing methods, namely:

- `dcfldd` [19] is a *Block-Based Hashing* algorithm that splits input data in a fixed number of segments and concatenates cryptographic hash values of each segment. The similarity measurement is the ratio of equal features.
- `ssdeep` [20] is a *Context Triggered Piecewise Hashing* algorithm that splits input data in contexts by a *rolling hash*. When the *rolling hash* yields a specific value, it triggers a change of context. Each context is hashed by traditional hash methods and recorded in a *digest*. The similarity measurement is based on a weighted edit distance between the digests.
- `sdhash` [21] is a *Statistically-Improbable Features* algorithm, which calculates the entropy of fixed-size *features* of the input data. Then, the most populate features, with more than a minimum entropy threshold, are hashed and added to a Bloom Filter (BF). The similarity between two digests is based on the number of features that belong to both BF.
- `TLSH` [22] is a *Local-Sensitive Hashing* algorithm that yields an id for each input subarray and counting it in buckets. Then, the output digest is the concatenation of length, ratio of quartiles, and the quartile of each bucket. The similarity measurement is calculated by an own weighted edit distance algorithm.

Moreover, we used a modular architecture to facilitate the inclusion of other approximation matching algorithms.

A. Hash Generation

In hash generation operation, `ProcessFuzzyHash` relies on the Volatility plugins `procdump` to obtain the processes inside the memory dump given as input, and on `memdump` to obtain all memory pages related to a given process (which can be given by parameter also). Besides the process (or processes) for which approximation matching hash (or hashes) shall be computed, in this case `ProcessFuzzyHash` also needs the user to provide as argument which part of the processes shall be considered. In this regard, `ProcessFuzzyHash` allows us to choose between the following: the full memory address space of the process, the executable file as mapped into memory (its image file), the PE headers, or a concrete PE section. Furthermore, the user can also choose either to consider binary data or just string-valued data (i.e., the arrays of printable characters in the process).

As output, `ProcessFuzzyHash` provides one line per each generated hash, indicating the process name, process ID, process creation timestamp, the part of the process considered for the hashing operation, the algorithm used, and the computed hash value.

An example of hash generation is illustrated in Listing 2, where the `ssdeep` algorithm is computed of three different processes. For the sake of readability, we omit part of the create timestamps and of the hash values.

B. Hash Comparison

In the hash comparison operation, `ProcessFuzzyHash` requires, at least, one (or more) hash(es) and the approximation matching algorithm used as base comparison. `ProcessFuzzyHash` supports two modes of comparison of the given hash (or hashes): either with a file containing a bunch of hash values separated by line, or with the hash values of processes contained in a dumped memory image file. Furthermore, a concrete process name or process ID can

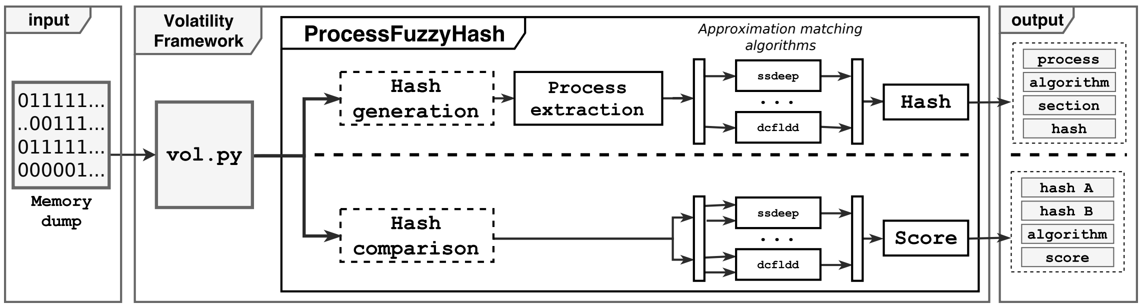

Figure 1: System overview diagram of `ProcessFuzzyHash`.

Listing 2: `ProcessFuzzyHash` execution example: hash generation using `ssdeep` algorithm.

```
$ python vol.py --plugins=processfuzzyhash/ -f vmcore.elf processfuzzyhash \
> --profile=win10x86_15063 -a ssdeep  -s pe -n vboxservice,winlogon,services

volatility foundation volatility framework 2.6
name           pid  create time  section algorithm hash
winlogon.exe   500  1(...)0      pe      ssdeep    6144:pzp/qv...8cijqdsyjqj
services.exe   544  1(...)3      pe      ssdeep    6144:q/6kxe...jxd5
vboxservice    1060 1(...)9      pe      ssdeep    12288:k/odr...cxuexsq
```

also be used as filtering to compare with a specific process inside the memory dump.

As output, in this case `ProcessFuzzyHash` provides one line per each hash compared with the given input hash, indicating both hashes, the algorithm used to compute and/or compare, and the similarity measurement (termed as *score*).

An example of hash comparison operation is illustrated in Listing 3, where a hash value is compared using `ssdeep` algorithm, considering processes inside the memory dump that match the name "svchost". As before, for the sake of readability we deliberately omit part of the hash values.

C. Tool Availability

For the sake of reproducibility and to foster research in the area of forensic memory analysis, we have released `ProcessFuzzyHash` as a plug-in included in the Volatility Framework under GNU GPLv3 license, available at https://github.com/volatilityfoundation/community/tree/master/ProcessFuzzyHash

D. Tool Performance

To evaluate our tool, we obtained 10 memory images from a Windows 7 32-bit bits with 2GB RAM, running in a VirtualBox hypervisor on top of an Intel Xeon E5606 2.13 GHz with 72GB RAM executing a Debian 9. Dumps were obtained after fresh reboot without any user interaction. Every memory image has a final size of 2.18 GB.

We selected 20 different system processes and hashed them. On average, the hash generation took around 8 seconds (s). Then, we selected the `winlogon.exe` process from every memory image and compared its code section hash with every other process in the image. The average time for comparison of each algorithm is 6.85 s for `dcfldd`, 6.9 s for `ssdeep`, 7.02 s for `sdhash`, and 6.94 s for `TLSH`. Further research is needed to exhaustively evaluate those algorithms, in terms of efficiency, precision, and recall.

V. Conclusions

As part of digital forensic triage, computer forensic analysts must find identical digital artifacts to determine, for instance, the presence of malicious software. Cryptographic hash functions are commonly used to this goal. However, these functions are unsuitable for some forensic data, such as forensic memory images, due to the dynamic nature of the underlying data. For instance, processes are relocated and its memory mapping addresses will likely change at every execution. Bytewise approximation matching algorithms become more suitable to find similar artifacts.

In this paper, we introduced `ProcessFuzzyHash`, a tool to compute bytewise approximation matching hash values of processes sections which are contained in a Windows memory image. We have explained our tool in detail and provide execution examples. Our tool has been released under GPLv3 within the Volatility Framework.

As future work, we aim at evaluating bytewise approximation matching algorithms in more detail. Preliminary results show that processes from different executable files score zero similarity, while processes from the same executable file score a low to medium similarity (in the range of 30 to 50%). Furthermore, we also aim at analyzing how the parameters of these algorithms affect to the similarity score.

Listing 3: `ProcessFuzzyHash` execution example: hash comparison using `ssdeep` algorithm.

```
$ python vol.py --plugins=processfuzzyhash/ -f vmcore.elf processfuzzyhash \
> --profile=win10x86_15063  -a ssdeep -s pe -n svchost > -c '768:9n3ss...quvkp5/zm'

volatility foundation volatility framework 2.6
hash a                hash b                   algorithm score
768:9n3ss...quvkp5/zm 768:9n3sss...qdvkp5/0m   ssdeep    94
768:9n3ss...quvkp5/zm 768:9n3sss...quvkp5/zm   ssdeep    100
(more output deliberately omitted)
```

Acknowledgements

This work was supported by Spanish National Cybersecurity Institute (INCIBE) "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad", grant numbers INCIBEC-2015-02486 and INCIBEI-2015-27300.

References

[1] O. Goldreich, *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press, 2006.

[2] A. F. Webster and S. E. Tavares, "On the Design of S-Boxes," in *Advances in Cryptology — CRYPTO '85 Proceedings*. Springer Berlin Heidelberg, 1986, pp. 523–534.

[3] V. S. Harichandran, F. Breitinger, and I. Baggili, "Bytewise Approximate Matching: the Good, the Bad, and the Unknown," *Journal of Digital Forensics, Security and Law*, vol. 11, no. 2, 2016.

[4] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate Matching: Definition and Terminology," National Institute of Standards and Technology, techreport NIST Special Publication 800-168, May 2014.

[5] A. Walters and N. Petroni, "Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process," in *BlackHat DC*, 2007.

[6] Microsoft MSDN, "PE Format," [Online; https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx], accessed August 20, 2017.

[7] M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," [Online; https://msdn.microsoft.com/en-us/library/ms809762.aspx], Mar. 1994, accessed on August 20, 2017.

[8] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*, 6th ed. Microsoft Press, Apr. 2012, ch. Chapter 5: Processes, Threads, and Jobs, pp. 359–486.

[9] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 105–120.

[10] PaX Team, "PaX address space layout randomization (ASLR)," https://pax.grsecurity.net/docs/aslr.txt.

[11] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62.

[12] A. P. Namanya, Q. K. A. Mirza, H. Al-Mohannadi, I. U. Awan, and J. F. P. Disso, "Detection of malicious portable executables using evidence combinational theory with fuzzy hashing," in *Proceedings of the 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2016, pp. 91–98.

[13] M. Arefkhani and M. Soryani, "Malware Clustering using Image Processing Hashes," in *Proceedings of the 2015 9th Iranian Conference on Machine Vision and Image Processing (MVIP)*. IEEE, 2015, pp. 214–218.

[14] G. Wicherski, "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*, ser. LEET'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–8.

[15] MANDIANT, "Tracking Malware with Import Hashing," [Online; https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html], 2014, accessed on July 27, 2017.

[16] Dynetics, "Malware Analysis Tool using Function Level Fuzzy Hashing," [Online; https://github.com/Dynetics/Malfunction], Aug. 2017, accessed on August 28, 2017.

[17] "Radare2: RAw DAta REcovery," [Online; http://rada.re/], accessed on August 20, 2017.

[18] B. Rodrigues, "Binwally: Binary and Directory tree comparison tool using Fuzzy Hashing," [Online; https://github.com/bmaia/binwally], Aug. 2017, accessed on August 18, 2017.

[19] N. Harbour, "dcfldd," [Online; http://dcfldd.sourceforge.net/], Mar. 2002.

[20] J. Kornblum, "Identifying Almost Identical Files Using Context Triggered Piecewise Hashing," *Digital Investigation*, vol. 3, no. Supplement, pp. 91–97, 2006.

[21] V. Roussev, "Data Fingerprinting with Similarity Digests," in *Advances in Digital Forensics VI*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 207–226.

[22] J. Oliver, C. Cheng, and Y. Chen, "TLSH – A Locality Sensitive Hash," in *Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop*, Nov. 2013, pp. 7–13.