The Software Similarity Problem in Malware Analysis

Andrew Walenstein and Arun Lakhotia[1]

University of Louisiana at Lafayette, Center for Advanced Computer Studies,
P.O. Box 44330, Lafayette, LA 70504-4330, U.S.A.
walenste@ieee.org,arun@louisiana.edu

Abstract. In software engineering contexts software may be compared for similarity in order to detect duplicate code that indicates poor design, and to reconstruct evolution history. Malicious software, being nothing other than a particular type of software, can also be compared for similarity in order to detect commonalities and evolution history. This paper provides a brief introduction to the issue of measuring similarity between malicious programs, and how evolution is known to occur in the area. It then uses this review to try to draw lines that connect research in software engineering (e.g., on "clone detection") to problems in anti-malware research.

Keywords. software, software evolution, commonality, program similarity, code clones, code smells, malicious software, malware, worms, Trojans, viruses, spyware, botnets, software visualization

1 Introduction

The problem of comparing software in order to determine how similar it is crops up in many contexts. Sometimes a software system is compared against itself to search for redundancies that might exist. For example, the problem of finding "scavenged" pieces of code—code that has been copied and then perhaps modified—is called "clone detection" [1]. At a more fine-grained level, programs may be compared against themselves to reduce the size of a program by squeezing out some of its redundancies at the machine-code level [2,3]. One piece of software may also be compared against another to look for evidence of copyright infringement or plagiarism (e.g., Lancaster *et. al* [4]). A software system may also be compared to previous versions of itself in an effort to reconstruct its evolution history (e.g., Zhou *et. al* [5]).

The problem of finding similarities in software systems, therefore, is a meeting point that relates several research and practice areas, among them: software engineering, education, and law. The aim of this paper is to show that another area deserves to be added to that list: the area of *malware analysis*. The term "malware" is used to refer to software that is malicious: worms, viruses, Trojans, and so on. "Malware analysis" therefore refers to the problems of understanding

Dagstuhl Seminar Proceedings 06301
Duplication, Redundancy, and Similarity in Software
http://drops.dagstuhl.de/opus/volltexte/2007/964

and managing malicious software, a task that is ordinarily performed in the context of computer security and defense.

But how is malware analysis related to these other areas? First, a brief introduction of certain problems in malware analysis is provided in Section 2. The overview makes the case that—like is the case for clone detection, compression, and plagiarism—program similarity is a key sub-problem in malware analysis. Section 3, establishes relationships to problems and research between malware analysis and other related problems such as plagiarism and evolution analysis. The paper concludes by arguing new research in the area is required.

2 The Reuse / Recognition War in Malware Analysis

The problem of finding similarities in programs is a central problem in malware analysis. A key battle between malware producer and defender is reviewed below, which is then used to explain why program similarity comparison is an important part of the defense in this battle.

2.1 WMDs in the Signature–Variation Battle

Malware writing is now a "professional" occupation [6]. There are many illegal money-making schemes that have been brought into the 21st century with the aid of malware. These include identity theft (enabling fraud), theft of valuables (such as passwords, program keys), extortion, and leasing of *botnets* (legions of remotely-controlled "zombie" computers). Malware also is a great enabler of espionage, both corporate and military. Its development has proceeded to the point where malware construction appears now like a prodigious "industry."

How prodigious? According to Symantec, 21,858 unique samples of malicious programs were discovered by it in 2005 [6]. That works out to 60 per day on average. More recently, Microsoft reported that they found 97,924 distinct variants of malware within the first half of 2006 [7]. That is over 22 per hour. If each of these malicious programs is considered to be a delivered computer product, then certainly the malware area can be considered to be quite the "productive" software construction industry.

How can malware authors be so productive? There are, most assuredly, not 100,000 malware authors, each one producing a single new program once every 6 months or so. Neither are there a smaller number of authors who produce completely new programs every few hours. Rather, most of the different malicious programs seen are modifications of some previous one. Many of these differences are minor: new data in an unused part of the executable headers, a change to a string, bug fixes, and so on. Other variations signal a relatively important change, such as the insertion of a new feature, like a new method for propagating. These more comprehensive changes are relatively rarer; a large enough change can trigger a change in the names given to the malware family: from "Bagle.AG" to "Bagle.AH", for example.

Why are there *so many* variations? One of the key reasons is that malware authors are engaged with an ongoing war against malware detector producers. The authors do not wish to write new code from scratch, but anti-malware companies create “signatures” to match their programs, reducing the number of hosts that are vulnerable to its attack. In order to defeat the signatures the malware authors can modify their programs until the most recent signatures no longer correctly match, and the malware is not detected. Over the years, the anti-malware companies have reduced the time needed to produce new signatures. But, as the time needed for new signatures decreases, one might reasonably expect the rate of variation production to increase proportionately [8]. In the war between malware author and malware detectors, one of the major battles is between signature and variation construction.

In this ongoing battle, malware authors need to make changes that work to undermine recognition and hence detection. Malware authors are known to vet their code against malware detectors. They keep versions of many malware detectors and check to see whether (and which) signatures catch their newest versions. If their new versions are being caught, they can attempt to make changes until they find that the signatures no longer match. That is, their battle position is to try to make changes that *destroy* the ability of detectors to *match* them to knowledge about previously released versions. These program modifications, therefore, may be termed “Weapons of Match Destruction” (WMDs). A few of these are reviewed below.

WMD: Encryption and Packing

Many malware executables are either encrypted or “packed.” In this context, it means that the programs are no longer in a format that is directly executable by the machine.[1] Rather, what happens when a packed or encrypted program is run is that a small decompression or decryption procedure is executed which reconstructs the full version of the original program from the packed or encrypted data. In either the packing or encryption case, if it is done well the result will be indistinguishable from random data. If so, then there is effectively no possibility of finding matches to previous versions without first unpacking or decrypting.

WMD: Byte- or Container-Level Changes

All machine-executable programs are *packaged* into some type of executable container format. This container typically contains a variety of information in the “header,” including the program’s identity, dynamic linking information, and a list of program parts or “segments”, such as the segments holding the data and code. By treating a program at the level of a package or string of bytes, multiple variations can be forced on the program executables without making changes to the actual code. These changes include: modifications of strings or values in the data segment and change in headers.

[1] The term “packed” in malware is generally distinct from the notion of “program compression” in the sense frequently implied in the program compression literature [3].

WMD: Ordinary Evolution

Like any other used software product, malware evolves. Malware evolves for the same reasons that ordinary software evolves. A given malware family can change over time simply because bugs are fixed. It can also evolve in response to changing requirements. For example, changes to system defenses may require a worm be changed to use a different exploit during propagation.

Malware also evolves by borrowing or copying code between families. In part this is due to the voluntary sharing of ideas and code within the malware production community. In the past, this was done using a variety of means, including bulletin boards, magazines, and Internet-relay chat. Recently, the code to even "professional" malware has become widely circulated; some is even published as *open source*. [9] For example, one Bagle variant included its own source code. Because of this practice, it is common for one malware family to be related to another family via the sharing of some code.

WMD: Obfuscation

The goal of program obfuscation is to make a program more difficult to understand or analyze. In the context of malware, obfuscations can be separated into two categories: extraction hindrance and provenance hiding. Extraction hindrance seeks to defeat the ability to extract accurate information about the program. For example, it is possible to insert "junk" bytes into a program in such a way that certain types of program disassembly produce incorrect disassembly. Once the disassembly is rendered incorrect, follow-on detection methods may be defeated. Provenance hiding seeks to obscure the fact that the program is derived from a previous one. So even if program information can be extracted correctly, the relationship to previous programs can still be obscured. For example, if code blocks are permuted there is no effective change to the program, yet any detection method that depends on the sequencing of blocks may be defeated.

WMD: Metamorphism

A program is called "metamorphic" if it is able to generate offspring that are different from it due to the fact that it transforms its own code [10]. Using metamorphism it is theoretically possible to ensure the program for each variant is somehow different. For example, it has been conservatively estimated that the `Win32.Evol` virus could conceivably create $10^{1,339}$ variations in just two generations [11].

2.2 Countering the WMDs

The fact is that most malicious programs found in the wild are variants of some previous one [8]. Because of this, the problem of comparing unknown files to previously-known malicious samples becomes important. In particular, from the viewpoint of the battle analogy, for every WMD, some defensive measure must

be constructed so that the potency of the weapons are reduced. In the ideal case, the defense would be able to see past any WMD in order to establish the similarity of an unknown program to known malicious programs. If such a powerful similarity mechanism was available then a new variant with only minor changes would not easily slip past the defenses. This capability could turn reuse from an advantage for the malware author into a *dis*advantage.

3 Malware Similarity

The defense side in the signature–variation battle needs to have methods for countering the WMDs. This means, that some way must be found for "seeing through" the variations introduced by the WMDs. Once variations are accounted for the new version may be matched against old ones. In this light, the problem is one of matching a recurring pattern with variations. This can be turned into a problem of program matching and similarity comparison. Similarity comparisons between programs can be useful for:

1. Detecting new variants as they are released by comparing them to known related variants.
2. Constructing new signatures to match the variant based on a similarity analysis between it and known previous variations.
3. Determining commonalities and relationships between different malware strains. This is important for leveraging past knowledge about threats. For example, if one knows that a new program is related to the well-known `Win32.Abogot` family, then one may have a good initial idea at the sorts of malice and problems it creates.

Thus program similarity evaluations can and do play a key role in malware analysis. The purpose of this section is to draw out some of the relations between malware analysis and other related works in code similarity comparison. These relations are, naturally, drawn out by noting similarities between malware analysis and these other fields. The review seeks to highlight similarities in: problems that need solving, method used, and commonalities in research or research direction.

3.1 Similarity of Problems

The WMDs generate the key problems for comparing programs in malware analysis. The problems they generate have some clear similarities with the essential problems in other areas. These include:

– **Plagiarism and Copyright Infringement Detection.**

  Relating one malicious software family to another involves, in part, comparing two or more software systems to find evidence where parts of one

software system match parts of another. The same problem is shared in detecting plagiarism and copyright infringement. Any of the WMDs may be used to help hide provenance. The WMD of obfuscation provides an example of how the problems are similar. For instance a new variant might differ only in the fact that the registers have been consistently renamed (e.g. swapping `eax` and `ebx`), and the order of basic blocks in the program has been permuted. These sorts of provenance obfuscations are, of course, common in plagiarism and copyright infringement when the perpetrator wishes to cover their tracks and hide the derivation relationship.

– **Software Evolution and Refactoring.**

  Malware evolves because the authors make fixes and improvements; they also refactor their code and insert foreign from other malware. These evolutionary changes present challenges to knowing how the members of a malware family relate to one another, and how different families are related via sharing of code. A similar problem is found in ordinary software evolution wherein changes make it difficult to track evolution, compare versions, and relate multiple programs. To illustrate the similarities, consider the dendogram shown in Figure 1. It shows a tree of proposed relationships between malware programs as generated by a clustering operation. Similar trees are found in phylogeny work for malware [12] as well as for ordinary software projects [13].

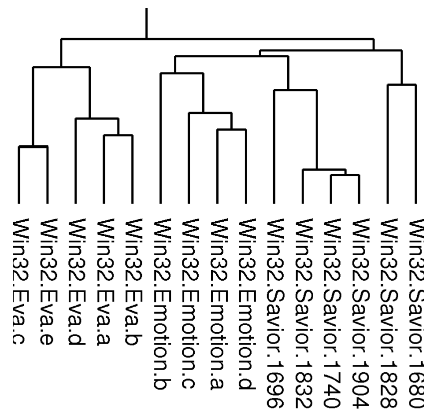

Fig. 1. Reconstructed evolution history for malware

– **Compression.**

  At the machine code level a program can exhibit similarities that are not present at the source code level and, conversely, similarities present at the source code level may not be clearly expressed at the machine code level.

Since malware is normally encountered in machine code format, malware similarity analysis must account for the irrelevant similarities expressed at the machine code level as well as the relevant similarities that are hidden at the machine code level. Machine code-level program compression research has to contend with similar problems.

Using the above analysis, a rough map of problem commonalities is produced in Table 1. Each row indicates the WMD that causes a particular research problem or issue; each column indicates the other area that is related by problem type.

Issues	*Plagiarism*	*Copyright*	*Refactoring*	*Evolution*	*Compression*	*Malware*
evolution	■	■	■	■		■
obfuscation	■	■				■
machine code		■			■	■
redundancy			■		■	■
metamorphism						■

Table 1. Skeletal map of relations between areas

3.2 Similarity in Methods & Research

In related problem areas, methods for clone detection, copyright infringement detection, and plagiarism detection share some common features. The overall methods can be modeled at a very high level as consisting of a pipeline of processing as illustrated in Figure 2. While we model this as a simple pipeline, in reality the steps may be interleaved and sometimes certain steps are omitted. This structure provides a way of organizing a comparison of methods and research from malware analysis and other related fields.

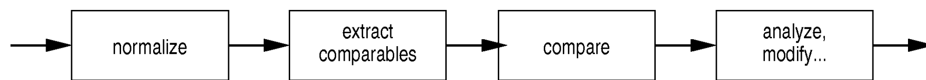

Fig. 2. Abstract program comparison pipeline

Normalize

Input programs frequently contain text or features that are considered irrelevant for the purposes of the comparisons being made. Normalization is the process of mapping the input programs into a form in such a way that irrelevant variation is removed and the input becomes more “normal”. A classic example is to remove whitespace from program texts before searching for evidence of plagiarism. Other more extensive normalizations have been proposed, including consistent replacement of identifiers with tokens [14], tokenizing various punctuation [15], and normalizing the form of syntax trees [16].

Normalization can also be an important step in malware analysis. In many respects the problem of normalization is similar for malware analysis, although in many cases the methods from benign code comparison have not carried over. For instance, the idea of consistent tokenization of identifiers may be useful in matching malware since instead of identifiers one can talk about registers at the machine code, as Baker *et. al* [17] for virtual machine code. Nonetheless, we do not know of any results in malware analysis using such techniques. Instead, the main research on normalization of malware has tended to employ relatively heavyweight semantics-preserving program transformations. For example, Kruegel *et. al* [18] investigated the use of transformations typically found in optimizing compilers—constant propagation, simplification, and so on—for normalizing malware as an aid to scanners; Walenstein *et. al* [11] used a term rewriting system to “undo” the transformations automatically performed by a self-rewriting virus. The more frequent use of more heavyweight semantics-based transformations in malware normalization may be due, perhaps, to the fact that the changes that are used to hide similarity in malware have been more complicated than those typically found in, say, student plagiarism. Alternatively, there may be fewer simple normalizations available in machine code as compared to source code. Whatever the reasons, one can expect that similar types of normalizing transformations might be used for plagiarism or clone detection.

Comparison

Once relevant items have been normalized and extract, some type of comparison framework must be employed to perform the comparisons. Several of such comparison frameworks are listed by Anderson *et. al* [19]. These are organized, in part, according to the “implementation techniques” used to detect clones, which may be interpreted as indicating also the models of comparison being used. From this list, one can see that many of these techniques are also used in malware analysis, including: graph matching [18] and distance metrics [20].

3.3 Similarities in Research or Research Direction

In malware analysis there are many important research problems which these align with problems found in our target related areas of clone detection, refactoring, plagiarism detection, and code compression. These include the following (related important problems are placed in parentheses):

– How to deal with variations created by compiler or compiler option differences (compression)
– Removing the influence of library code linked into the program (compression)
– Tracing evolution lines (clone detection/refactoring)
– Matching against large corpora (clone detection/refactoring, plagiarism)
– Removing obfuscations in programs (plagiarism)

This list is certainly not complete; rather, it can be used as a starting point for researchers from different problem worlds in their discussion of how to collaborate or use results from one area in a new one.

4 Conclusions

Malicious programs evolve, and there is much borrowing of code between different families of malware. Recognizing and modeling how these programs evolve and are related is an important problem in the area of malware analysis. For this reason the problem of matching and relating programs by their common parts is a key problem within malware analysis. Since this problem shares many parallels in other fields, many useful links to other research areas can be made.

In drawing out some of these linkages, this paper seeks to improve the amount of cross-fertilization between research areas. In particular, we note that there are several techniques or approaches that appear to be primarily explored in one area or another. For example, relatively few techniques from clone detection and plagiarism have their parallels in malware analysis (however it should be noted that this problem in malware analysis is just now starting to get any significant attention). Conversely, the normalization approach using semantics-based transformations have rarely been seen in other areas such as plagiarism and clone detection. In the future we hope that finding a common set of core problems and approaches will serve to pool research together and result in advances in all of the related fields.

References

1. Koschke, R.: Survey of research on software clones. [21] ISSN 1682–4405.
2. Evans, W.: Program compression. [21] ISSN 1682–4405.
3. Beszédes, Á., Ferenc, R., Gyimóthy, T., Dolenc, A., Karsisto, K.: Survey of code-size reduction methods. ACM Computing Surveys **35** (2003) 223–267
4. Lancaster, T., Culwin, F.: A comparison of source code plagiarism detection engines. Computer Science Education **14** (2004) 101–112
5. Zou, L., Godfrey, M.W.: Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering **31** (2005) 166–181
6. Ször, P.: The Art of Computer Virus Research and Defense. Symantec Press (2005)
7. Braverman, M., Williams, J., Mador, Z.: Microsoft security intelligence report: January–June 2006 (2006) http://microsoft.com/downloads/details.aspx?FamilyId=1C443104-5B3F-4C3A-868E-36A553FE2A02.

8. Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware: "Vilo" method for comparing and searching binary programs. In: Proceedings of BlackHat DC 2007. (2007) https://blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf.
9. Lyda, R., Hamrock, J.: Exploring investigative methods for identifying and profiling serial bots. Journal of Digital Forensic Practice **1** (2006) 165–177
10. Ször, P., Ferrie, P.: Hunting for metamorphic. In: Proceedings of the 11th International Virus Bulletin Conference. (2001)
11. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Normalizing metamorphic malware using term-rewriting. In: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), Philadelphia, PA (2006) 75–84 http://doi.ieeecomputersociety.org/10.1109/SCAM.2006.20.
12. Karim, M.E., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. European Research Journal of Computer Virology **1** (2005) 13–23 http://dx.doi.org/10.1007/s11416-005-0002-9.
13. Yamamoto, T., Matsushita, M., Kamiya, T., Inoue, K.: Measuring similarity of large software systems based on source code correspondence. In: Product Focused Software Process Improvement. Volume 3547 of Lecture Notes in Computer Studies. Springer Berlin / Heidelberg (2005) 530–544 http://www.springerlink.com/content/h6m2vg5c3ejk3814.
14. Baker, B.S.: Parameterized duplication in strings: Algorithms and an application to software maintenance. SIAM Journal on Computing **26** (1997) 1343–1362 http://dx.doi.org/10.1137/S0097539793246707.
15. Kamiya, T., Kusumoto, S., Inoue, K.: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **8** (2002) 654–670 http://www.cs.drexel.edu/~spiros/teaching/CS675/papers/clone-kamiya.pdf.
16. Baxter, I.D., Yahin, A., Moura, L.M.D., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: 14th IEEE International Conference on Software Maintenance (ICSM'98), Washington, DC, USA, IEEE Computer Society (1998) 368–377 http://doi.ieeecomputersociety.org/10.1109/ICSM.1998.738528.
17. Baker, B.S., Manber, U.: Deducing similarities in java sources from bytecodes. In: Proceedings of the USENIX Annual Technical Conference (NO'98), 2560 Ninth Street, Suite 215, Berkeley, CA, 94710, USENIX (1998) 179–190 http://www.usenix.org/publications/library/proceedings/usenix98/baker.html.
18. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Proceedings of the 8th Symposium on Recent Advances in Intrusion Detection (RAID'2005). Lecture Notes in Computer Science, Springer-Verlag (2005) http://www.infosys.tuwien.ac.at/staff/ek/papers/raid05_polyworm.pdf.
19. Anderson, P., Frenzel, P., Koschke, R., Rieger, M.: Source code clone detectors: Overview and open problems. [21] ISSN 1682–4405.
20. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proceedings of International Symposium on Secure Software Engineering, IEEE (March, 2006) http://homes.dico.unimi.it/~monga/listpub.html.
21. Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, Dagstuhl, Germany, Dagstuhl (2006) ISSN 1682–4405.