Signature Visualization of Software Binaries

Thomas Panas
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
{panas}@llnl.gov

Abstract

In this paper we present work on the visualization of software binaries. In particular, we utilize ROSE, an open source compiler infrastructure, to pre-process software binaries, and we apply a landscape metaphor to visualize the signature of each binary (malware). We define the signature of a binary as a metric-based layout of the functions contained in the binary. In our initial experiment, we visualize the signatures of a series of computer worms that all originate from the same line. These visualizations are useful for a number of reasons. First, the images reveal how the archetype has evolved over a series of versions of one worm. Second, one can see the distinct changes between versions. This allows the viewer to form conclusions about the development cycle of a particular worm.

Keywords: binary analysis, malware visualization

1 Introduction

Malicious code, including viruses and worms, is software that exploits vulnerabilities in computer systems with the intent to steal, modify or destroy data from that system or even damage the entire system itself. The worldwide economy is increasingly dependent on information technology. A successful vulnerability attack could cause serious consequences for major economic and industrial sectors, threaten infrastructure elements such as electric power, and disrupt response and communication capabilities of first responders. For instance, CNN reported in 2007 that researchers launched an experimental cyber attack which caused a generator to self-destruct [CNN 2007]. Such cyber attacks could damage the electric infrastructure and take months to repair. If a third of the country lost power for three months, the economic price tag would reach USD700 billion [CNN 2007], an economic blow equivalent to 40 to 50 large hurricanes striking at the same time.

The primary problem with the overwhelming quantity of malicious code stems from the fact that malware has become very simple to deploy. Developing viruses and worms no longer requires specialized knowledge of computer architectures and operating systems; the art of vulnerability exploitation is well documented in everyday literature and accessible to anyone. Furthermore, there are toolkits available today that allow novices to develop malicious code like child's play.

On the other hand, detecting malicious code is no easy game. Since malicious code typically attaches itself to legitimate software, it is essential to determine the behavior of the overall software (malware plus legitimate code) without actually executing it. Although current anti-virus software effectively detects known malware, it is relatively unsuccessful at detecting new kinds of malicious code because its primary detection technique is based on predefined malware patterns. A better approach for analyzing malware software, even in binary form, is to apply static analysis, a well-known approach in the field of compiler technology. This allows for the analysis of the malware-infected software without program execution.

2 ROSE

Our approach for software binary analysis is to parse a binary file and represent it as an abstract syntax tree (AST). Our current options to parse binaries are our own disassembler and *IDA Pro* [DATARESCUE 2007], a disassembler that supports many different platforms including Linux and Windows and a wide variety of processor architectures. For the intermediate representation (AST) of the binary we use ROSE, an open source compiler infrastructure [ROSE 2008]. ROSE is a U.S. Department of Energy (DOE) project, which currently can process million-line C, C++ and Fortran codes. Our approach of representing a binary as an AST allows us to reuse much of the infrastructure of ROSE for binary analysis. This enables us to perform static analysis on software binaries, including malware. In addition, our AST representation supports the easy creation of control-flow and data-flow graphs for both source code and binaries, allowing the development of more sophisticated malware analyses.

3 Signature Visualization

We utilize the ROSE infrastructure to access information about software binaries allowing us to uniquely represent and visualize each binary. For this purpose, we compute metrics on binaries to visualize a unique signature based on these metrics. There is no limit to the choice of metrics that may represent the signature of a binary. To conduct our experiment of visual signature comparison – described in Section 4 – we have chosen metrics that allow us to visualize a binary's signature in three dimensional space using a landscape metaphor. The right choice of metaphor is essential to improve the usability and understandability of a visualization [Panas et al. 2007]. Metaphors found in nature are intuitive and provide a graphic design that the user already understands. The metrics we have chosen to represent binary signatures are:

Number of Control Transfer Instructions represents the number of instructions within a function that cause a program to branch from its sequential execution path. Examples of control transfer instructions in x86 assembly are the *jmp* (unconditional jump), *jz* (conditional jump when zero flag=1), *loop*, *ret* and *call* instruction.

Number of Instructions represents the total number of instructions within a function.

Number of Data Transfer Instructions represents the number of instructions within a function that modify register values or memory, such as the *mov*,*push* and *bswap* instructions.

SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008.
© 2008 ACM 978-1-60558-112-5/08/0009 $5.00

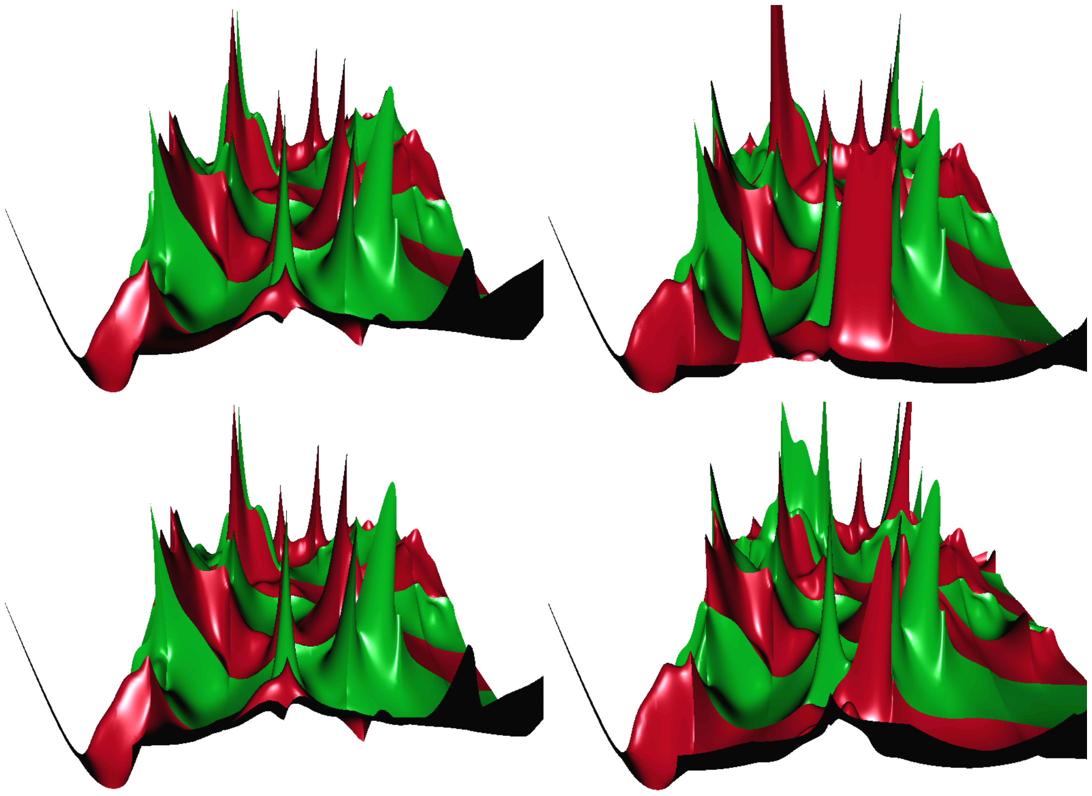

Figure 1: Binary Visualization of Malware Klez: a,b,c,d.

To represent these metrics using a three dimensional landscape metaphor, we map the number of control transfer instructions to the x-axis, number of instructions to the z-axis and the number of data transfer instructions to the y-axis. We use a landscape metaphor to represent the binary signature in order to increase the perception and natural processing of the unique visual patterns (hills) [Petre et al. 1998; Storey et al. 2000]. The problem with the suggested visualization is, however, that the image will grow with the size of the binary under investigation. Comparing different binary signature images of various sizes can therefore become a challenge. For this reason, we have applied a modulo operation on two of our metrics – namely the number of control transfer instructions (x-axis) and the number of data transfer instructions (z-axis). This approach scales the visualization to a unified size and allows for simplified comparison and reasoning between visual signatures. Figure 1 (left top) shows the pure signature representation (without host) of Klez [Wikipedia-Klez 2008], a computer worm that spreads by sending itself to other computers via email, with a modulo 64 operation performed on both the x-axis as well as z-axis. We have also added red and green color-coding to the image to additionally aid the viewer to assess and compare visual signatures. Note that we analyze a worm, i.e. a binary malware without a host.

4 Signature Comparison

In this experiment, we have applied our signature visualization approach to four versions of Klez, referred to as Klez-a, Klez-b, Klez-c and Klez-d, cf. Figure 1. By examining Figure 1 one can see that the signatures of different variants of Klez are visually similar. The visual similarities of Klez signatures justifies our choice of layout metrics. Nevertheless, even though it is possible to convey the signature of a binary through imagery, it is challenging to compare the signature images of different binaries because each image contains complex patterns. It is therefore difficult to answer questions about the similarities of different binaries from the signature image alone. Despite these challenges, there is value in determining the extent to which different versions of malware are similar to each other. Thus, we have taken two approaches to measure these similarities.

First, we applied *diff*, a Linux-based application, to check for textual-based similarities among the assembly representations of each binary. Second, we created a new signature image to represent the differences between the individual signatures in order to support the viewer in reasoning about the development cycle of a particular malware.

4.1 Binary comparison with *diff*

We utilized the AST unparser in ROSE to generate assembly instructions for each malware and compared the instructions against each other using *diff*. The similarity results of different versions of Klez are presented in the following Table:

Name	**Similarity**
Klez-a, Klez-b	2%
Klez-a, Klez-c	4%
Klez-b, Klez-d	6%
Klez-c, Klez-d	2%
Klez-c, Klez-e	2%
Klez-d, Klez-e	2%
Klez-e, Klez-f	29%

Surprisingly, the data conveys that there are almost no similarities between any versions of Klez. The only exception is the comparison between the assembly representation of Klez-e and Klez-f,

Klez-a	**Klez-b**
4011c8: call DWORD PTR [4090c0 <CreateFileA>]	4011c8: call DWORD PTR [40a0c8 <CreateFileA>]
4011ce: mov edi, eax	4011ce: cmp eax, 0xffffffff
4011d0: cmp edi, 0xffffffff	4011d1: mov DWORD PTR [0xffffffe4 <hObject>+ebp], eax
4011d3: mov DWORD PTR [0xffffffe4 <hObject>+ebp], edi	4011d4: jnz 0x4011dd
4011d6: jnz 0x4011df	4011d6: xor eax, eax

Figure 2: Snippet of assembly instructions for Klez-a and Klez-b.

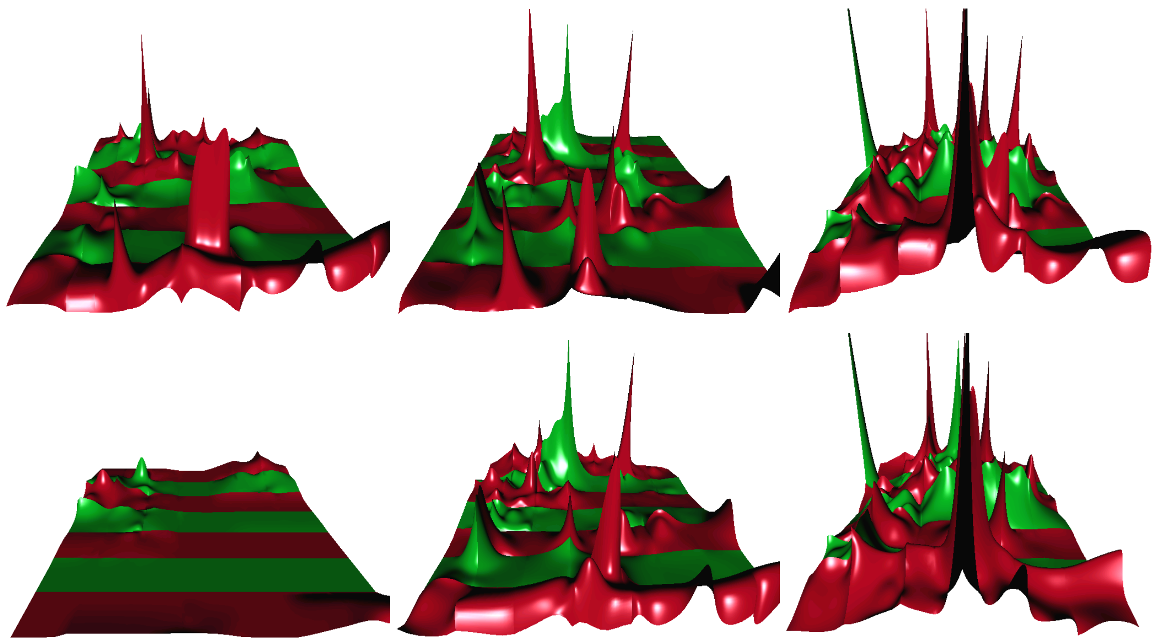

Figure 3: Delta Signature visualization of top row: Klez-a/b, Klez-b/d, Klez-d/e; bottom row: Klez-a/c, Klez-c/d, Klez-c/e.

which shows a similarity of 29%. This data is surprising as one would correctly assume that the offspring of a particular computer worm should be somewhat related to the archetype. In addition, Figure 1 at least visually reveals that some similarities among the Klez worm family should exist.

Figure 2 shows a snippet of two regions of assembly code taken from Klez-a and Klez-b. This figure reveals why *diff* can not successfully compare software binaries or their assembly representations. The reasons are mainly:

Renamed Registers. The register names seem to differ between various versions of Klez. In Figure 2 the register *edi* (left) was renamed to register *eax* (right). This change could have resulted from an automated obfuscation technique.

Optimization/Permutation. Figure 2 shows that the order of the instructions has changed from one binary to the next. This phenomenon applies throughout the entire *diff* result and can be attributed to either compiler optimizations during source code compilation or to permutations of the assembly instructions (without the program semantics being changed).

One reason why *diff* fails is that renamed registers as well as compiler optimizations shuffle assembly instructions enough for *diff* not to be able to determine any similarities among the assembly codes. Furthermore, because most instructions are reorganized, the destination of conditional transfer instructions, such as *jmp* and *call*, is adjusted - causing even fewer matches. Finally, because the opcodes of instructions have different byte sizes, a reorganization of instructions leads to the change of addresses. This is why in Figure 2 there is no equivalent of Klez-a at address 4011d0 in Klez-b. For these reasons, the similarity results of Klez versions using *diff* are poor. Nevertheless, we believe that the similarity results could be improved by using a comparison algorithm that is better than line by line comparison. For instance, a clone detection algorithm should overcome obstacles such as register renaming and compiler optimizations. We will pursue this approach in the near future.

4.2 Binary Comparison using Visualization

To calculate the difference between two visual signatures, we subtract the size of one binary's function (y-axis) from the other. To avoid negative y-axis values, our implementation is based on the subtraction of the smaller value from the larger. In this way, the new signature – referred to as delta signature image – represents extensions as well as removals of instructions between two binary versions.

Figure 3 shows the delta signatures for the family of Klez worms. Left-top illustrates the delta image from Klez-a and Klez-b and the image below shows the delta from Klez-a and Klez-c. It appears that Klez-a/Klez-c are almost entirely the same, while the visual difference between Klez-a/Klez-b is larger. One may want to conclude that more functionality has been added or changed from version Klez-a to Klez-b. The next column in Figure 3 (top-middle) shows the delta signature from Klez-b and Klez-d. Below is an image for the delta of Klez-c and Klez-d. From the images one can conclude that Klez-d is probably an offspring of Klez-b. This is

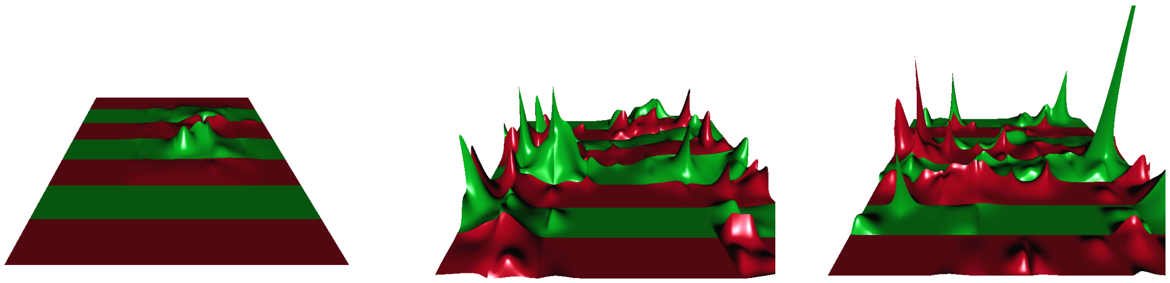
Figure 4: Delta Signature Visualization of Klez-e/Klez-f (left); using a different metric: Klez-a/Klez-b (middle), Klez-a/Klez-c (right).

because the Klez-c/Klez-d image contains a layer in the far front that is equivalent to that in the image of Klez-a/Klez-b. This means that the two have something distinctive in common. In addition, the visual delta between Klez-b/Klez-d appears to be less than that for Klez-c/Klez-d, strengthening our assumption of a transition between Klez-b and Klez-d.

The right top corner of Figure 3 represents the delta signature from Klez-d and Klez-e and below it is a delta signature from Klez-c and Klez-e. Here our visual judgment fails. The images are too similar for us to judge whether Klez-e is an offspring of Klez-c or Klez-d or possibly both. The final delta image in the Klez series is on the left side of Figure 4. This is a delta signature from Klez-e and Klez-f. It seems that these two malwares have much in common.

Figure 3 confirms our assumption that the visualization of binary signatures is important and can help viewers to hypothesize about the evolution of software binaries. Furthermore, signature visualization appears to overcome the problems of renamed registers and at least some compiler optimization. In this experiment, we have visualized the delta signatures from different Klez worms, and this has allowed us to form conclusions about the developmental history of this worm family. In particular, our archeological rollback allows us to affirm that the Klez worm evolved from A→B→D→E→F and A→C→E→F., cf. Figure 5.

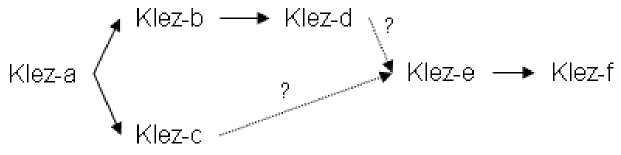

Figure 5: Evolution of Klez worm.

We have also experimented with applying other metric values to represent binary signatures. Figure 4 shows the result of applying a metric based on the name of the function to the x- and z-axis. In this case, we have converted each character in a function's name to an integer value and summed the entire string. Figure 4 (middle) shows the delta signature from Klez-a and Klez-b using this approach; the figure on the right represents the delta signature from Klez-a and Klez-c.

5 Related Work

We are aware of little related work within binary analysis and visualization. CodeSurfer/x86 [Balakrishnan et al. 2005] is a platform for analyzing x86 executables; similar to ROSE. The binary analysis extension to ROSE is relatively new and we therefore believe the analysis capabilities of CodeSurfer/x86 are more sophisticated. Malwarez [Alex Dragulescu 2008] is a visualization tool for malware. It is not inteded as a binary analysis infrastructure, but rather a piece of art. It represents disassembled code, API calls, memory addresses and subroutines in various 3D visual forms.

6 Conclusion and Future Work

In this paper we have presented one approach for visualizing the signature of software binaries. Our goal is to enable the viewer to make observations about binaries, in particular observations about the evolution of a malicious code. In our experiment we have shown how a visual approach could be more effective for determining binary code similarities than applying tools such as *diff*. We believe that clone detection could aid this process. In the future, we will explore binary clone detection and develop additional types of analysis for software binaries.

References

ALEX DRAGULESCU, 2008. Malwarez. http://www.sq.ro/malwarez.php.

BALAKRISHNAN, G., GRUIAN, R., REPS, T., AND TEITELBAUM, T. 2005. *CodeSurfer/x86 - A Platform for Analyzing x86 Executables*, vol. 3443. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, March, 250–254.

CNN, 2007. Staged cyber attack reveals vulnerability in power grid. http://www.cnn.com/2007/US/09/26/power.at.risk/.

DATARESCUE, 2007. IDA Interactive Disassembler. www.datarescue.com.

PANAS, T., EPPERLY, T., QUINLAN, D., SÆBJØRNSEN, A., AND VUDUC, R. 2007. Communicating Software Architecture using a Unified Single-View Visualization. In *Proceedings of Int. Conf. on Complex Computer Systems*.

PETRE, M., BLACKWELL, A., AND GREEN, T. 1998. Cognitive questions in software visualization. *Software Visualization: Programming as a Multimedia Experience* (January), 453–480.

ROSE, 2008. Rose compiler. www.rosecompiler.org/.

STOREY, M.-A. D., WONG, K., AND MÜLLER, H. A. 2000. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming 36*, 2–3, 183–207.

WIKIPEDIA-KLEZ, 2008. Klez (computer worm). http://en.wikipedia.org/wiki/Klez.